PROGRAMMER AVEC SCHEME

DE LA PRATIQUE À LA THÉORIE

Jacques Chazarain

INTERNATIONAL THOMSON PUBLISHING

Jacques CHAZARAIN

Programmer avec Scheme

De la pratique à la théorie

INTERNATIONAL THOMSON PUBLISHING FRANCE
An International Thomson Publishing Company
e-mail : *contact@itp.fr*

Paris • Albany • Belmont • Bonn • Boston • Cincinnati • Detroit • Johannesburg
Londres • Madrid • Melbourne • Mexico • New York • Singapour • Toronto • Tokyo

Programmer avec Scheme – De la pratique à la théorie
Jacques CHAZARAIN

Couverture conçue par Jean Widmer

Les programmes figurant dans ce livre ont pour but d'illustrer les sujets traités. Il n'est donné aucune garantie quant à leur fonctionnement une fois compilés, assemblés ou interprétés dans le cadre d'une utilisation professionnelle ou commerciale.

© INTERNATIONAL THOMSON PUBLISHING FRANCE, Paris, 1996
ISBN 2-84180-131-4

Toute représentation ou reproduction, intégrale ou partielle, faite sans le consentement de l'auteur, ou de ses ayants droit, ou ayants cause, est illicite (loi du 11 mars 1957, alinéa 1er de l'article 40). Cette représentation ou reproduction, par quelque procédé que ce soit, constituerait une contrefaçon sanctionnée par les articles 425 et suivants du Code pénal. La loi du 11 mars 1957 n'autorise, aux termes des alinéas 2 et 3 de l'article 41, que les copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective d'une part, et d'autre part, que les analyses et les courtes citations dans un but d'exemple et d'illustration.

Table des matières

Avant-propos **xi**

I Pratique de la programmation avec Scheme 1

1 Introduction au langage Scheme **3**
1.1 La boucle d'interaction 3
1.2 Expressions parenthésées et préfixées 4
1.3 Esquisse de la syntaxe de Scheme 5
1.4 Les définitions en Scheme 7
1.5 Booléens, prédicats et contrôles 11
1.6 Variables locales et structure de bloc 15
1.7 Définition récursive d'une fonction 17
1.8 Entrée/sortie 21
1.9 De la lecture 23

2 Expressions symboliques **25**
2.1 Symboles et citations 25
2.2 Les listes 27
2.3 Programmation récursive avec les listes plates 34
2.4 Programmation récursive avec les listes quelconques 38
2.5 Aspects méthodologiques et preuve de fonctions récursives 40
2.6 Evaluation des expressions Scheme 45
2.7 Les doublets 46
2.8 Les a-listes 48
2.9 Représentation externe/interne des données 50
2.10 Représentation interne des listes et notions d'égalité 53
2.11 Quasi-citation 57
2.12 Compléments : polynômes 58
2.13 De la lecture 64

3 Programmation fonctionnelle **65**
3.1 Lambda expression 65
3.2 Portée statique et fermeture 68
3.3 Les fonctions en résultat 71
3.4 Les fonctions en argument 72

3.5 **Letrec** et définitions locales de fonctions 74
3.6 La fonction **map** et ses dérivés 77
3.7 Extension de la syntaxe des lambda 83
3.8 Représentation des ensembles 85
3.9 Appel récursif terminal 88
3.10 Compléments : retour-arrière et problème des huit reines 93
3.11 De la lecture 99

4 Programmation impérative 101
4.1 L'affectation en Scheme 101
4.2 Itération et récursivité terminale 104
4.3 La liste comme structure mutable 106
4.4 Caractères et chaînes 113
4.5 Vecteurs 117
4.6 Compléments : une représentation pleine des polynômes 119
4.7 De la lecture 123

5 Environnements, fermetures et prototypes 125
5.1 Portée et durée de vie d'une liaison 125
5.2 Variable, environnement et mémoire 130
5.3 Définitions récursives et **letrec** 133
5.4 Programmation modulaire 134
5.5 Mutation de variables 136
5.6 Variables mutables et prototypes 138
5.7 Fermetures et générateurs 141
5.8 Comptage et mémorisation des appels d'une fonction 144
5.9 Compléments : système de maintien de contraintes et CAO 146
5.10 De la lecture 159

6 Calcul numérique avec Scheme 161
6.1 Quelques fonctions arithmétiques 161
6.2 Ecriture d'un entier dans une base 164
6.3 Calculer avec les rationnels et les réels 167
6.4 Calculer avec les nombres complexes 171
6.5 Fractals et ensemble de Mandelbrot 172
6.6 Calculer avec les vecteurs 174
6.7 Calculer avec les matrices 175
6.8 Compléments : visualisation d'un cube en rotation 177
6.9 De la lecture 184

7 Données structurées et algorithmes 185
7.1 Structures de données et types abstraits 185
7.2 Piles fonctionnelles 187
7.3 Piles mutables 190
7.4 Evaluation des expressions arithmétiques postfixées 193
7.5 Structures de file 195
7.6 Structures de table 199
7.7 Adressage dispersé 201

7.8 Structure d'arbre 204
7.9 Arbres binaires 207
7.10 Structure générale d'arbre 217
7.11 Méthodes de recherche et de tris 219
7.12 Notions sur les graphes 224
7.13 Compléments : jeux à deux joueurs et coupures alpha-bêta 229
7.14 De la lecture 234

8 Programmation par continuation 235
8.1 Continuation d'un calcul 235
8.2 Programmation par passage de continuation 237
8.3 Continuation et valeurs multiples 239
8.4 Continuation et échappement 242
8.5 Générateur de solutions (I) 246
8.6 La fonction `call/cc` 248
8.7 `Call/cc` et échappements 251
8.8 Générateur de solutions (II) 254
8.9 Une fonction `*erreur*` 255
8.10 Coroutines 257
8.11 Compléments : simulation ferroviaire 261
8.12 De la lecture 268

9 Macros et extensions syntaxiques 269
9.1 Introduction à la notion de macro 269
9.2 Utilisations des macros 272
9.3 Définitions de nouvelles formes spéciales 275
9.4 Problèmes de capture avec les macros 279
9.5 Définitions récursives de macros 283
9.6 Formes spéciales primitives et dérivées 284
9.7 Liaisons destructurantes 288
9.8 Extensions syntaxiques définies par filtrage 291
9.9 Exemples de définitions par `define-syntax` 292
9.10 Redéfinition de certaines formes spéciales 296
9.11 Compléments : trace et comptage des appels d'une fonction 299
9.12 De la lecture 304

10 Fichiers, flux et flots 305
10.1 Fichiers et flux 305
10.2 Lecture de fichiers 306
10.3 Ecriture de fichiers 308
10.4 Le couple `force/delay` 311
10.5 Flots et listes infinies 313
10.6 Générateur de solutions (III) 319
10.7 Flots et séries formelles 320
10.8 Compléments : indentation des S-expressions 323
10.9 De la lecture 329

11 Programmation par objets **331**

11.1 Principe de la programmation par objets 331
11.2 Délégation et prototypes 334
11.3 Langage Logo 337
11.4 Application aux systèmes de Lindenmayer 342
11.5 Une extension objet de Scheme : SchemO 345
11.6 Jeu donjon et dragon 349
11.7 Implantation de SchemO 358
11.8 Compléments : vie artificielle 362
11.9 De la lecture 378

II Outils formels pour le programmeur **379**

12 Automates, langages et ordinateurs **383**

12.1 Automates finis 383
12.2 Notion de langage formel 385
12.3 Reconnaissance d'un langage par un automate fini 387
12.4 Machines de Turing 390
12.5 Automates cellulaires 393
12.6 Machines à pile 397
12.7 Structure d'un ordinateur 399
12.8 Calculabilité 402
12.9 De la lecture 404

13 Analyse lexicale et syntaxique **405**

13.1 Expressions régulières et langages associés 405
13.2 Recherche d'une expression régulière 407
13.3 Expressions régulières et automates finis 412
13.4 Analyse lexicale 415
13.5 Grammaires formelles 419
13.6 Analyse syntaxique descendante 425
13.7 Grammaires `LL(1)` 426
13.8 Un analyseur d'expressions arithmétiques 429
13.9 Syntaxe abstraite et grammaires attribuées 431
13.10 De la lecture 437

14 Affichage et formatage **439**

14.1 Mise sous forme infixe des expressions arithmétiques 439
14.2 Formatage de texte : centrage, justification... 441
14.3 Dessins d'arbres binaires et grammaires attribuées 451
14.4 Formatage de formules : MiniTeX 458
14.5 De la lecture 473

15 Calcul propositionnel **475**
15.1 Langage du calcul propositionnel 475
15.2 Valeur sémantique d'une formule 479
15.3 Tautologies 481
15.4 Déduction sémantique 484
15.5 Forme clausale 485
15.6 Calcul booléen et circuits numériques 488
15.7 Méthode de Shannon 490
15.8 De la lecture 494

16 Déduction naturelle et calcul des séquents **495**
16.1 Notion de systèmes formels 495
16.2 Déduction naturelle en logique propositionnelle 499
16.3 Calcul des séquents propositionnels 503
16.4 Implantation du calcul des séquents propositionnels 505
16.5 De la lecture 514

17 Filtrage et réécriture **515**
17.1 Filtrage de listes plates 515
17.2 Termes et syntaxe abstraite 518
17.3 Substitutions dans les termes 521
17.4 Filtrage de termes 525
17.5 Réécriture de termes 529
17.6 Applications de la réécriture 532
17.7 De la lecture 536

18 Des systèmes experts à la programmation logique **537**
18.1 Systèmes experts d'ordre 0 et chaînage arrière 537
18.2 Systèmes de règles avec variables 544
18.3 Unification dans les termes 545
18.4 Implantation d'un moteur d'inférence d'ordre 1 549
18.5 Programmation logique et Prolog 555
18.6 De la lecture 560

19 Logique du premier ordre **561**
19.1 Langages du premier ordre 561
19.2 Une représentation en Scheme des formules du premier ordre . . . 564
19.3 Occurrences libres ou liées de variables 565
19.4 Substitutions en logique du premier ordre 567
19.5 Déduction naturelle en logique du 1er ordre 570
19.6 Calcul des séquents du premier ordre 573
19.7 Implantation du calcul des séquents classiques 577
19.8 Logiques égalitaires 586
19.9 De la lecture 589

20 Introduction au lambda calcul **591**
20.1 Définition du lambda calcul 591
20.2 Bêta réduction 594
20.3 Stratégies de réduction 599
20.4 Représentation des données en lambda calcul 604
20.5 Combinateur Y, récursion et domaines 607
20.6 Lambda calcul typé 613
20.7 Inférence de type 616
20.8 Implantation de l'inférence de type 620
20.9 Isomorphisme de Curry Howard 623
20.10 De la lecture 626

21 Sémantique des langages de programmation **627**
21.1 Interprète pour un MiniLisp en Scheme 628
21.2 Interprète pour MiniScheme 635
21.3 Interprète Scheme par continuation 639
21.4 Sémantique naturelle 646
21.5 Sémantique naturelle de MiniML 648
21.6 Interprète pour MiniML en Scheme 653
21.7 Un langage fonctionnel paresseux : MiniLML 657
21.8 Sémantique naturelle d'un langage impératif 663
21.9 Notion de sémantique dénotationnelle 669
21.10 Interprète pour Blaise 671
21.11 De la lecture 678

22 Introduction à la compilation **679**
22.1 Calculette numérique et machine SC 679
22.2 Calculette à mémoire et machine SEC 683
22.3 Compilateur MiniScheme et machine SECD 690
22.4 Appel terminal et optimisation JSR-RTS 698
22.5 Machine avec zone d'activation 700
22.6 Assemblage et machine avec compteur ordinal 706
22.7 Représentation des doublets et machine Lisp 712
22.8 Récupération des doublets inutilisés : GC 725
22.9 Machine avec lien statique 732
22.10 De la lecture 737

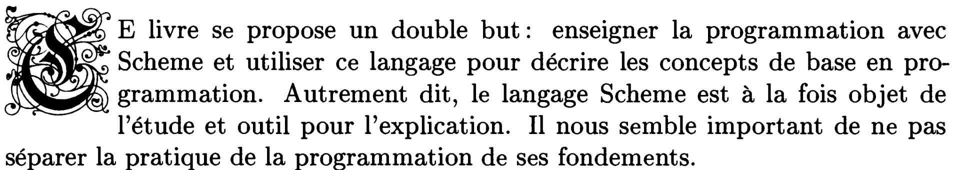
A Codes ASCII **739**

B Interprètes et compilateurs Scheme **741**

Bibliographie **743**

Index **749**

Avant-propos



CE livre se propose un double but : enseigner la programmation avec Scheme et utiliser ce langage pour décrire les concepts de base en programmation. Autrement dit, le langage Scheme est à la fois objet de l'étude et outil pour l'explication. Il nous semble important de ne pas séparer la pratique de la programmation de ses fondements.

Le choix de Scheme

Il fallait un langage adapté à la description des concepts et outils de la programmation, c'est-à-dire un langage de manipulations symboliques. C'est précisément l'une des spécificités des langages de la famille Lisp. Pourquoi avoir choisi Scheme dans cette famille ? La conception même de Scheme nous le désignait comme support idéal pour l'enseignement. C'est un langage réduit à l'essentiel et pourtant fortement expressif. Le mieux est de laisser la parole à ceux[1] qui ont défini la norme de ce langage

« ... Scheme demonstrate that a very small number of rules for forming expressions, with no restrictions on how they are composed, suffice to form a pratical and efficient programming language that is flexible enough to support most of the major programming paradigms in use today. .... ».

Signalons que ce choix n'est pas original, il a été fait vers 1980 par des enseignants du MIT et depuis il n'a fait que gagner des adeptes aux Etats-Unis. Ce mouvement a maintenant atteint les premiers et deuxièmes cycles des universités françaises. En utilisant Scheme, on reste lisible par les adeptes des différents dialectes Lisp. Dans ce but, on a utilisé de préférence une syntaxe proche du style Lisp tout en employant de temps en temps la syntaxe favorite des «purs Schemeurs».

Ajoutons un avantage pratique, non négligeable pour l'utilisateur, c'est un langage disponible sur les principaux systèmes d'exploitation et la plupart des machines. Il y a un grand nombre d'implantations gratuites de très bonne qualité. De plus, sa petite taille permet de l'utiliser sur des machines avec des configurations modestes.

---

[1] *Revised⁴ Report on the Algorithmic Language Scheme*, édité par W. Clinger and J. Rees.

Au début, la niche écologique de Scheme était l'enseignement et la recherche en informatique, son utilisation déborde maintenant de ce cadre. Il tend de plus en plus à être aussi utilisé comme langage de commande dans des systèmes interactifs (éditeurs de texte, générateurs d'interface homme-machine, systèmes graphiques, ...).

Structure du livre

Dans la **première partie**, on se propose de faire goûter aux joies de la programmation avec Scheme. On y présuppose pas de connaissance spéciale en informatique afin que ce livre puisse être lu avec la culture d'un bachelier. De nombreux exercices permettent au lecteur de vérifier immédiatement sa compréhension d'un nouveau concept.
Presque tous les chapitres se terminent par des compléments que le lecteur pourra se contenter de parcourir dans une première lecture : fractals, visualisation 3D, jeux, contraintes et CAO, calcul formel, simulations, vie artificielle, ... Ils ont un double but : donner une ouverture sur des domaines variés de l'informatique et présenter des programmes de taille plus importante. On peut les considérer comme des thèmes de projets dont on présente une solution possible.
Le langage Scheme permet d'illustrer les principaux styles de programmation : la programmation fonctionnelle bien sûr, mais aussi la programmation impérative, la programmation par prototypes, la programmation par continuation, la programmation par composition de flots, la programmation par objets, ...

La **deuxième partie** présente une variété de domaines qui sont importants pour la culture d'un programmeur : machines et automates finis, analyse lexicale et syntaxique, formatage de texte ou de formules, systèmes experts et Prolog, systèmes logiques, lambda calcul, sémantique, interprétation, compilation,... Pour démystifier certains sujets, jugés un peu théoriques, on a utilisé Scheme comme langage d'explication. La traduction d'un principe abstrait en un programme Scheme est souvent une excellente façon de le comprendre. Aussi, s'est-on imposé de donner une implantation de pratiquement toutes les théories présentées. Non seulement cela oblige à approfondir sa compréhension d'une théorie, mais la possibilité de procéder à des exécutions permet aussi d'expérimenter des variantes. En bref, notre approche pédagogique de l'informatique se prévaut du slogan : *implanter pour mieux comprendre.* En application de notre slogan nous n'avons pas déposé sur le réseau Internet les sources Scheme de ce livre. Nous espérons qu'ainsi le lecteur sera conduit à réécrire selon son style et ses besoins les programmes de cet ouvrage.
La lecture de la deuxième partie ne suppose pas la connaissance des compléments mais une bonne assimilation des principales notions de la première partie et plus précisément des chapitres 1 à 5 qui en constituent le cœur. On a aussi parfois utilisé la petite extension objet de Scheme développée au chapitre 11.

On peut décomposer la deuxième partie en trois blocs relativement indépendants :
chapitres 12 à 14 : analyse syntaxique et formatage,
chapitres 15 à 19 : outils logiques pour l'informatique,
chapitres 20 à 22 : sémantique, interprétation et compilation des langages.

Un livre est le résultat de choix ...

On s'efforce d'éviter les paragraphes de plus de dix lignes de texte car les étudiants ont tendance à les sauter et le plus possible la parole est donnée à Scheme. De façon générale, on s'est efforcé d'éviter les digressions, souvent tentantes, pour privilégier l'explication des programmes.

Les programmes plus importants (plusieurs pages) font en plus l'objet d'une description structurelle où l'on essaye de préciser l'interdépendance des fonctions afin de donner une vue de l'architecture globale.

On se conforme à la tradition des livres d'enseignement qui est de privilégier la clarté au détriment de la robustesse du code.

On n'a pas cherché à faire œuvre d'historien, aussi a-t-on préféré se restreindre à la citation de livres ou d'exposés de synthèse.

Il existe déjà quelques bons livres, principalement en anglais, sur Scheme ou sur Lisp, aussi, exceptés les exemples classiques et incontournables, on a essayé de présenter de nouveaux thèmes d'application.

Remerciements

Mon enseignement de la programmation a bénéficié des remarques et réactions de mes étudiants.

La discussion avec mes collègues est aussi une source d'enrichissement constant. En particulier, je tiens à remercier mes amis Schemeurs : Dan Friedman, Erick Gallésio et Jean-Paul Roy. Dans le domaine de la sémantique, on notera l'influence des idées de l'équipe de Gilles Kahn avec qui j'ai eu le plaisir de collaborer.
Je remercie tout particulièrement Erick pour sa relecture attentive de la plupart des chapitres ; je n'indique pas lesquels pour bien préciser que les erreurs qui restent inévitablement sont de la responsablilité de l'auteur. Comme ce livre comporte beaucoup de code, la probabilité d'erreur est encore plus grande. Nous sommes reconnaissants aux lecteurs de nous les signaler[2]. J'ai aussi bénéficié des remarques constructives de Christian Queinnec et de Manuel Serrano sur certains aspects de ce livre.

On a utilisé divers outils logiciels pour réaliser cet ouvrage mais aucun logiciel commercial. Pour Scheme on a la chance de disposer de l'excellente implantation de Marc Feeley et Doug Currie pour Macintosh appelée Gambit. Pour les systèmes Unix, il y a de nombreuses versions de Scheme, j'ai préféré utiliser le système STk de Erick Gallésio, il a l'avantage d'être couplé avec une boîte à outils graphique.

De nos jours, l'auteur d'un livre joue aussi le rôle de typographe grâce au système TEX de Donald Knuth et aux macros LATEX de Leslie Lamport. A ce propos je remercie mon collègue Olivier Lecarme pour sa TeXpertise. Enfin, je tire un coup de chapeau à Yannis Haralambous pour sa magnifique police de lettrines qui est un clin d'œil à l'art des anciens.

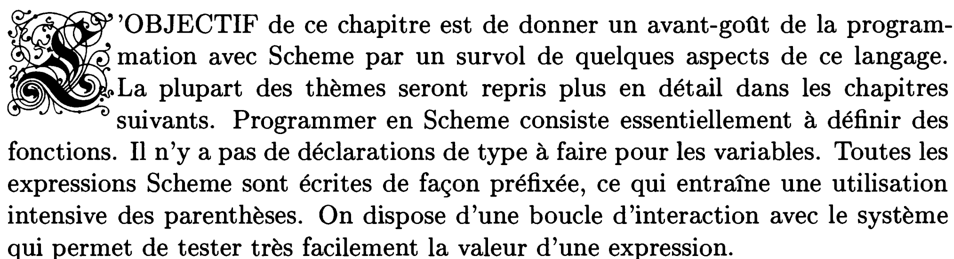
[2]On peut joindre l'auteur par courrier électronique à l'adresse jmch@unice.fr.

Partie I

Pratique de la programmation avec Scheme

Chapitre 1

Introduction au langage Scheme



L'OBJECTIF de ce chapitre est de donner un avant-goût de la programmation avec Scheme par un survol de quelques aspects de ce langage. La plupart des thèmes seront repris plus en détail dans les chapitres suivants. Programmer en Scheme consiste essentiellement à définir des fonctions. Il n'y a pas de déclarations de type à faire pour les variables. Toutes les expressions Scheme sont écrites de façon préfixée, ce qui entraîne une utilisation intensive des parenthèses. On dispose d'une boucle d'interaction avec le système qui permet de tester très facilement la valeur d'une expression.

1.1 La boucle d'interaction

Scheme est un langage fonctionnel ; cela signifie que le concept de base est la *fonction* : on définit des fonctions et on les compose. Un programme fonctionnel est une suite de définitions de fonctions qui concourent à définir une ou plusieurs fonctions plus complexes. Dans un langage impératif, comme Pascal, le concept de base est *l'instruction* dont le rôle est de provoquer un changement d'état mémoire de la machine ; l'instruction élémentaire est l'affectation. Un programme impératif est une suite d'instructions.
L'activité essentielle en programmation fonctionnelle est l'évaluation d'expressions. Pour faciliter l'interaction avec l'utilisateur, les langages de cette famille disposent d'une *boucle d'interaction* (on utilisera aussi souvent la dénomination anglaise de *top level*). C'est une boucle sans fin de la forme :

- l'utilisateur entre une expression,
- le système affiche sa valeur et passe à la ligne,
- le système affiche un caractère d'invite et attend l'entrée de l'expression suivante ...

Une suite d'évaluations sous le top level s'appelle une *session*.
Le choix du caractère d'invite est affaire de convention, on choisira dans cet ouvrage le caractère `?`.
Si on demande la valeur de l'entier `421`, on écrira `421` puis, après la frappe du caractère retour chariot (en abrégé CR), le système affichera à la ligne suivante sa valeur c'est-à-dire encore `421` puis de nouveau à la ligne suivante l'invite `?`.

```
? 421
421

?
```

Pour gagner de la place, on écrira souvent le résultat de l'évaluation sur la même ligne avec le signe `->` entre l'expression et la valeur calculée.

```
? 421  -> 421
```

Bien entendu, on peut aussi utiliser un éditeur de texte pour créer un fichier d'expressions puis faire évaluer ce fichier. La plupart des implantations de Scheme permettent même d'évaluer directement une expression située dans un fichier. Pour donner des exemples plus intéressants, il faut définir la syntaxe des expressions admissibles.

1.2 Expressions parenthésées et préfixées

Expressions préfixées

Il y a de nombreuses façons d'écrire une expression arithmétique. Scheme utilise systématiquement la notation préfixée parenthésée. Elle consiste à écrire d'abord l'opérateur puis ses opérandes et le tout entre parenthèses. Par exemple, l'expression arithmétique $4 + 76$ s'écrira `(+ 4 76)`. De façon plus générale, si *op* désigne un opérateur binaire, l'écriture préfixée d'une expression de la forme $exp1$ *op* $exp2$ est `(op ~exp1 ~exp2)` où `~expj` est la forme préfixée de la sous-expression $expj$.

Pour les opérateurs qui admettent un nombre arbitraire d'opérandes, il suffit d'écrire l'opérateur en tête de ses opérandes. Par exemple, l'expression arithmétique $4 + 76 + 19$ s'écrira `(+ 4 76 19)`. Noter qu'il y a au moins un espace entre l'opérateur et chacun des opérandes. On peut mélanger les opérateurs :

$34 * 21 - 5 * 18 * 7$ s'écrit en préfixe `(- (* 34 21) (* 5 18 7))`,
$-(2 * 3 * 4)$ s'écrit en préfixe `(- (* 2 3 4))`.

Un des avantages du parenthésage est de ne pas avoir à se soucier des priorités entre les opérateurs car une expression complètement parenthésée est toujours non ambiguë.
Cette écriture préfixée est utilisée pour *toutes* les expressions. Par exemple, l'application d'une fonction f à des arguments `2, 0, 18` s' écrit en mathématiques $f(2, 0, 18)$ et est représentée en Scheme par `(f 2 0 18)`, noter aussi la disparition des virgules car les espaces suffisent comme séparateurs.

Ce type d'écriture a pour effet de multiplier le nombre des parenthèses et peut dérouter au premier contact. Mais on s'habitue vite, d'autant que les éditeurs pour Scheme offrent en général un mécanisme pour visualiser l'appariement des parenthèses.

Exercice 1 *1. Donner la forme préfixée des expressions :*

$$(p-a)(p-b)(p-c) \ , \ 1+2x^2+3x^3 \ , \ \frac{sin(x+y)cos(x-y)}{1+cos(x+y)}$$

2. Donner la forme infixe usuelle de l'expression Scheme :

(/ ((f i)(f (- n i)))*

Indentation

Une autre façon pour éviter les erreurs de parenthésage est de recourir à des conventions de présentation ou *d'indentation* des programmes. Il y a tout un style d'écriture des expressions. Une convention, souvent utilisée, consiste à aligner verticalement les opérandes d'une fonction, cet alignement étant en retrait par rapport au nom de la fonction ; ainsi l'expression `(f a b c)` peut aussi s'écrire :

```
(f a
   b
   c)
```

Mais en fait, ce n'est pas aussi systématique ; c'est surtout une affaire de goût. Quand les opérandes sont des expressions un peu compliquées, on les aligne verticalement ; mais quand ils sont simples, on conserve la présentation horizontale. Bien entendu, la valeur d'une expression est indépendante de la convention utilisée car les espaces et les passages à la ligne ont la même signification pour Scheme. Voici une indentation possible pour l'expression mathématique $\frac{sin(a)+sin(b)}{\sqrt{1+a^2+b^2}}$:

```
(/  (+ (sin a) (sin b))
    (sqrt (+ 1 (* a a)(* b b))))
```

La forme préfixée met en évidence la structure d'une expression et facilite le travail de la fonction de lecture de Scheme. Elle présente d'autres avantages qui seront expliqués plus tard. Il est fréquent que les langages qui ont besoin d'effectuer une analyse rapide des expressions, demandent à l'utilisateur de fournir directement une forme préfixée ou parfois postfixée.

1.3 Esquisse de la syntaxe de Scheme

Commentaires

Les commentaires sont tout ce qui suit un ; sur une ligne.

Nombres

- Les entiers ne sont limités en taille que par la taille mémoire de la machine.
- Les nombres décimaux s'écrivent avec un point comme cette approximation de π, `3.14159` ou en notation exponentielle `0.314159e1`.
- Les nombres rationnels $\frac{a}{b}$ exacts s'écrivent `a/b`.

```
? (* 2/3  5/2) ; ceci est un commentaire
10/9
```

On dispose des opérations mathématiques usuelles : $+, -, *, /, max, min, \sqrt{}$.
Elles admettent un nombre quelconque d'arguments :
`(+ x1 ...xN)` $\rightarrow x_1 + \cdots + x_N$
`(- x1 ...xN)` $\rightarrow x_1 - \cdots - x_N$
`(* x1 ...xN)` $\rightarrow x_1 * \cdots * x_N$
`(/ x1 ...xN)` $\rightarrow x_1/(x_2 * \cdots * x_N)$
`(max x1 ...xN)` $\rightarrow max(x_1, \cdots, x_N)$
`(min x1 ...xN)` $\rightarrow max(x_1, \cdots, x_N)$
`(sqrt x)` $\rightarrow \sqrt{x}$
...
`? (* 1 2 3 4 5 6 7 8 9);` calcul de la factorielle de 9
`362880`

L'expression du volume de la sphère de rayon R s'écrit `(* 4/3 pi R R R)`. La température en degrés Fahrenheit s'exprime en fonction de la température C en degrés Celsius par `(+ 32 (* 9/5 C))`.
Un traitement plus systématique des aspects numériques sera donné au chapitre 6, on y verra aussi que Scheme prévoit l'implantation des nombres complexes.

Exercice 2 *Les racines du trinôme* $x^2 + 2bx + c = 0$ *sont réelles quand le discriminant* $delta = b^2 - c$ *est positif ou nul. Elles s'écrivent* $x_1 = -b + \sqrt{delta}$, $x_2 = -b + \sqrt{delta}$. *Donner l'expression Scheme de* x_1 *et* x_2.

Chaînes

Une chaîne (*string* en anglais) est une suite de caractères. Elle s'écrit avec ses caractères encadrés par des guillemets, sa valeur est ell- même :

```
? "bonjour tout le monde !"
"bonjour tout le monde !"
```

Identificateurs

On utilise des noms appelés *identificateurs* pour désigner des objets. Dans un identificateur on peut mélanger des caractères alphabétiques, numériques, des caractères spéciaux à l'exclusion de[1] # ( ) [ ]; et du caractère espace qui sert de séparateur.

[1]On renvoie à la norme de Scheme pour la définition formelle des identificateurs et pour la liste complète des caractères spéciaux à éviter.

Scheme ne fait pas de différence entre minuscule et majuscule ; on en profite pour écrire des identificateurs comme `rayonSphere` qui est identique à `rayonsphere` mais plus lisible. Attention, ce qui peut s'interpréter comme un nombre ne peut être utilisé comme identificateur : `1e-2` représente le nombre `0.01`.

S-expressions

On appelle *S-expression* toute expression syntaxiquement correcte en Scheme. En première approximation, une S-expression c'est :

soit un nombre,

soit une chaîne,

soit un identificateur,

soit une succession de S-expressions encadrées par deux parenthèses :

```
(S-expression ...S-expression)
```

Cette dernière catégorie d'expression s'appelle une *liste.*

Toute S-expression n'est pas toujours sémantiquement correcte, c'est-à-dire qu'elle ne possède pas toujours une valeur. Pour cela, il nous faut expliquer comment Scheme évalue une expression, cela occupera pratiquement toute la première partie de ce livre ! Afin de distinguer une S-expression de sa valeur, il sera parfois commode de désigner par $\overline{s}$ la valeur de l'expression s.
L'évaluation d'une expression est déclenchée par la frappe du premier retour chariot qui suit une expression syntaxiquement correcte :

```
? (+  7
      (* 3 4))
```

maintenant la touche CR déclenche l'évaluation, car l'expression est syntaxiquement correcte. Le résultat s'affiche à la ligne suivante :

```
19
```

Exercice 3 *Les expressions*
f(x y z), (f)(x y z), (f), ((f f)), ff, ((a) (b) (c)), ()
sont-elles syntaxiquement correctes ?

1.4 Les définitions en Scheme

Syntaxe d'une définition

Quand on pose `pi = 3.14`, on veut dire que dorénavant l'identificateur `pi` aura pour valeur `3.14`. Le fait de désigner une valeur par un nom est le principe même de l'*abstraction.* On nomme un concept compliqué par un identificateur : le mot Scheme[2] est un nom pour désigner un fascinant langage de programmation.
Pour nommer une valeur par un identificateur, on dispose de la forme :

[2] Ce langage faisait suite à une famille de langages appelés Conniver, Planner, ... et s'appelait en réalité Schemer mais le système d'exploitation utilisé par ses auteurs G. Steele et J. Sussman n'acceptait que les six premières lettres.

```
(define  ident  exp)
```

Son évaluation a pour effet que dorénavant l'identificateur représente la *valeur* de l'expression exp. La valeur rendue est sans importance et dépend des implantations de Scheme. On dira qu'elle est indéfinie et on ne l'affichera pas. On note que l'on n'a fait aucune déclaration sur le type de l'identificateur.
Après évaluation de

```
? (define pi   3.14159)
```

on a

```
? pi
3.14159
```

Si l'on veut donner le nom rac2 au nombre $\sqrt{2}$, on évalue :

```
? (define  rac2  (sqrt 2))
```

ensuite

```
? rac2
1.4142135623730951
```

Et évidemment cela peut être utilisé dans une expression :

```
? (* rac2 rac2)
2.000000000000000
```

En revanche, si on demande la valeur d'un identificateur qui n'a pas été défini, on provoque une erreur Scheme :

```
?  toto
*** ERROR -- unbound variable: toto
```

Il y a au départ un certain nombre d'identificateurs prédéfinis, ne serait-ce que les noms des fonctions de base.

```
?  sqrt              ; quelle est la valeur de l'identificateur sqrt ?
#[procedure sqrt]   ; c'est la fonction racine carrée.
```

Variables et liaisons

Un identificateur auquel on a associé une valeur s'appelle une *variable*. Une variable désigne en réalité un emplacement mémoire dans lequel est rangée cette valeur. Un identificateur comme define s'appelle un *mot clé* du langage car il désigne une construction prédéfinie.
La correspondance entre une variable et sa valeur s'appelle une *liaison*. L'ensemble des liaisons connues en un point du programme s'appelle l'*environnement*. Ici l'environnement est constitué des liaisons entre pi et sa valeur, et de rac2 et sa valeur. Il faudrait aussi y ajouter toutes les liaisons prédéfinies comme celle de sqrt ; on reviendra sur la notion d'environnement dans des chapitres suivants et notamment aux chapitres 2 §10 et 5 §2.
On peut changer la valeur d'une variable en la redéfinissant. L'ordre des définitions est important quand elles mettent en jeu des expressions contenant des variables, car il faut que les variables à évaluer aient déjà une valeur.

```
? (define un 1)

? (define deux 2)

? (define  trois (+ un deux))
```

On peut permuter l'ordre des définitions des variables `un` et `deux`, mais la définition de la variable `trois` doit être faite en dernier.
Si l'on décide ensuite de changer la valeur de la variable `un` par `(define un 0)`, la valeur associée à `trois` est inchangée jusqu'au moment où l'on réévalue sa définition.

Définition de fonctions

Quand on veut nommer une expression contenant des paramètres, on arrive à la notion de fonction. Définissons la fonction qui calcule le volume de la sphère de rayon R. On utilise encore la forme `define`[3] avec la syntaxe suivante :

```
(define (VolumeSphere R)
  (* 4/3 pi R R R))
```

Elle utilise le paramètre `R`, en revanche l'identificateur `pi` n'est pas un paramètre car il a été défini plus haut.
De façon plus générale, une définition de fonction est de la forme :

(define *(nom-fonction x1 x2 . . . xk)*
  corps)

Où *x1 x2 ...xk* s'appellent les *paramètres formels*, ou en bref les paramètres. En Scheme, les paramètres et la valeur d'une fonction ne font pas l'objet de déclaration de type. On dit que c'est un langage dont les variables ne sont pas typées.

Le nom d'un paramètre est sans importance, on définit la même fonction `VolumeSphere` en écrivant :

```
(define  (VolumeSphere  x)
  (* 4/3 pi x x x))
```

Appel d'une fonction utilisateur

L'appel d'une fonction définie par l'utilisateur se fait exactement comme pour une fonction prédéfinie, c'est une expression préfixée par le nom de la fonction et suivie de ses arguments :

(nom-fonction arg1 arg2 ... argk)

La valeur rendue s'obtient en évaluant le corps de la fonction quand on attribue à chaque paramètre xj la valeur de l'argument correspondant argj.

Un argument peut être une expression Scheme quelconque pourvu que sa valeur soit du type prévu dans le corps de la fonction (les valeurs ont un type).

[3] On verra au chapitre 3 §1 une autre syntaxe, dite essentielle, pour définir une fonction.

Ainsi l'appel :

```
? (VolumeSphere (+ 8 4)) -> 7234.56
```

provoque l'évaluation du corps `(* 4/3 pi R R R)` sachant que `R` a pour valeur la valeur de l'argument `(+ 8 4)` soit `12`. Si l'on décide de changer la valeur de `pi` pour augmenter la précision :

```
? (define pi 3.14159)
```

est-ce que cela va influer sur la fonction `VolumeSphere`? Oui! c'est la dernière valeur de `pi` qui sera utilisée au prochain appel de cette fonction.

```
? (VolumeSphere 12) -> 7238.22
```

Utilisation de fonctions auxiliaires

Le calcul du volume nécessite d'élever un nombre au cube. Comme c'est une opération courante, on peut décider de se doter d'une fonction cube,

```
? (define (Cube nb) (* nb nb nb))
```

et redéfinir la fonction volume par :

```
(define (VolumeSphere x)
  (* 4/3 pi (Cube x)))
```

On voit ainsi apparaître l'utilisation d'une fonction auxiliaire pour participer à la définition de la fonction principale.
Il n'y a pas de règle pour déterminer l'ordre dans lequel on définit les fonctions. Un adepte de la programmation dite descendante commencera par écrire la ou les fonctions principales, puis les fonctions auxiliaires qui en simplifient l'écriture puis les fonctions auxiliaires des fonctions auxiliaires ..., ou, à l'inverse, on peut commencer par les fonctions auxiliaires pour remonter vers les fonctions principales. C'est seulement une question de style ; l'essentiel est qu'au moment où l'on demande la valeur d'une expression, toutes les fonctions qui concourent à son calcul aient été définies.
Par exemple, la fonction suivante calcule l'hypoténuse d'un triangle rectangle de côtés `a` et `b` :

```
(define (hypotenuse a b)
  (sqrt (+ (carre a) (carre b))))
```

Elle utilise la fonction auxiliaire `carre` que l'on peut définir après :

```
(define (carre x)
  (* x x))
```

Exercice 4 *Ecrire une fonction qui calcule le prix TTC à partir du prix HT sachant que le coefficient de TVA est 18.6 %.*

Exercice 5 *L'aire d'un triangle de côté a, b, c est donnée par $\sqrt{p(p-a)(p-b)(p-c)}$ où p désigne le demi-périmètre. Ecrire une fonction Scheme donnant l'aire à partir des côtés ; on pourra introduire une fonction auxiliaire pour calculer p.*

Langages non typés versus langages typés

Contrairement à beaucoup de langages de programmation, la définition d'une fonction se fait sans déclaration du type des paramètres, ni celui du résultat. Les incohérences de type ne seront détectées qu'au moment de l'exécution. Autrement dit : les variables ne sont pas typées mais les valeurs le sont. C'est au programmeur de veiller à la cohérence de l'emploi des valeurs des expressions. Cela présente des avantages et des inconvénients :

- avantages : cette souplesse (ce laxisme, diront certains !) permet d'utiliser une fonction pour une grande variété de données : toutes celles pour lesquelles les opérateurs utilisés dans le corps ont un sens. Ainsi, on n'a pas à préciser si un argument est entier ou décimal ou même un nombre complexe. Une même variable peut représenter successivement un nombre, puis une chaîne ..., en revanche, les valeurs sont typées (cf chapitre 22 §7). Cette souplesse explique aussi le succès de Scheme dans le domaine de l'étude et l'implémentation de nouveaux langages. L'absence de déclaration de type diminue la verbosité du langage ; mais cet avantage est discutable si on s'impose d'exprimer en commentaire la spécification d'une fonction ;

- inconvénients : de façon évidente on perd en sécurité en laissant le contrôle de type à la charge du programmeur et non à celle d'un compilateur. On peut perdre aussi en efficacité, car la connaissance a priori du type des objets permet au compilateur d'utiliser directement les opérateurs appropriés.

  Un langage comme ML a essayé —et en grande partie réussi— à garder les avantages du typage sans les inconvénients.

Convention
Ce livre se voulant avant tout une présentation pédagogique, on utilisera souvent des conventions d'écriture des paramètres pour rappeler implicitement leurs types. Par exemple, si on attend un entier, on utilisera des paramètres comme : `n, i, j, n1, n2` ..., si on attend un nombre, on écrira `r, r1, nb, nb1, ...`, si on peut utiliser une expression quelconque, on écrira `s, s1, e, exp, ...` Ces conventions seront complétées au fur et à mesure de l'introduction de nouveaux types de données.

1.5 Booléens, prédicats et contrôles

Les booléens

Il n'y a pas de langage de programmation sans structure de contrôle et donc sans représentation du vrai ou du faux. Les constantes prédéfinies `#t` et `#f` servent à représenter les valeurs logiques vrai et faux. Elles s'évaluent en elles-mêmes et constituent le type *booléen*.

Quelques prédicats

Un prédicat est une fonction ayant pour valeur un booléen. Par convention, ils se terminent par un ?. On dispose de nombreux prédicats prédéfinis, en voici quelque-uns qui servent à tester le type de la valeur d'une expression.

`(number? s) -> #t` si s est un nombre et `#f` sinon

`(integer? s) -> #t` si s est un entier et `#f` sinon

`(string? s) -> #t` si s est une chaîne et `#f` sinon

`(booleen? s) -> #t` si s est un booléen et `#f` sinon

`(procedure? s) -> #t` si s est une fonction et `#f` sinon

```
? (string? 10)   -> #f

? (procedure? +) -> #t
```

On dispose aussi du prédicat `equal?` à deux arguments pour comparer des valeurs :

```
(equal? s1 s2) -> #t si s1 et s2 sont égales
```

Pour les nombres, on dispose d'un prédicat d'égalité plus spécialisé `=`, ainsi que des opérations de comparaison classiques sur les nombres : `<, <=, >, >=, zero?` :

```
?  (< nb1 nb2) -> #t si nb1 < nb2 et #f sinon
```

L'opération de négation s'appelle `not` :

```
(not s) -> #t si s vaut #f et #f sinon
```

Elle permet, par exemple, de définir la non-nullité d'un nombre par :

```
(define (not-zero? nb)
  (not (zero? nb)))
```

L'alternative `if`

La structure de contrôle la plus fondamentale est le `if`. Dans un langage applicatif, le `if` est une fonction ou plus exactement une forme spéciale, sa syntaxe est :

(if *exp-test exp-alors exp-sinon*)

Pour calculer la valeur de cette forme, il y a d'abord évaluation de l'expression en position de test ; si la valeur trouvée est `#f`, alors on retourne la valeur de ***exp-sinon***. Dans les autres cas, on retourne la valeur de ***exp-alors***. On voit donc que *#f joue le rôle du faux mais que toute autre valeur joue le rôle du vrai dans un test.*
Définissons la fonction valeur absolue d'un nombre qui est ce nombre quand il est positif et son opposé quand il est négatif :

```
(define (valeurAbsolue  nb)
  (if (<= 0 nb)
    nb
    (- nb)))
```

Notons au passage le style d'indentation utilisé pour le `if` : on dispose verticalement ses trois arguments, les parties `exp-alors, exp-sinon` commencent sous le `f` de `if`.

Exercice 6 *Définir (sans utiliser max) la fonction max2 qui calcule le maximum de deux nombres.*

Il est important de remarquer que la forme `if` n'évalue toujours qu'une des deux expressions *exp-alors, exp-sinon*. C'est en ce sens que l'on dit que `if` est une *forme spéciale* et non une fonction. Cette remarque permet d'écrire une fonction de division avec une protection contre la division par 0 :

```
(define (division-sure  r1 r2)
  (if (zero? r2)
    "on ne peut pas diviser par 0"
    (/ r1  r2)))

? (division-sure 15 0)
"on ne peut pas diviser par 0"
```

On peut aussi utiliser le `if` sans la branche `exp-sinon`. Dans ce cas, quand le test a pour valeur `#f`, la valeur rendue est indéfinie :

```
?  (if (zero?  pi) "c'est une valeur inexacte pour pi")
```

On peut évidemment emboîter des `if` pour faire des choix multiples. Par exemple, la fonction qui donne le signe d'un nombre, c'est-à-dire `1` s'il est positif, `-1` s'il est négatif et `0` s'il est nul, sera définie par :

```
(define (signe nb)
  (if (< 0 nb)
    1
    (if (zero? nb)
      0
      -1)))
```

La conditionnelle `cond`

L'utilisation de `if` emboîtés est rapidement illisible ; aussi dispose-t-on d'une autre forme spéciale, appelée `cond`, très commode pour les choix multiples :

```
(cond (exp-test1  corps1)
      (exp-test2  corps2)
      ...
      (exp-testN  corpsN))
```

Sa valeur s'obtient en calculant successivement *exp-test1, exp-test2*, ... jusqu'à en rencontrer un de vrai, disons *exp-testj*, alors on retourne la valeur de *corpsj*.
Pour être assuré qu'au moins un test réussit, on remplace souvent le dernier test par le mot clé `else` qui force le succès.
Noter l'indentation utilisée pour mettre en évidence les différentes alternatives.
Ecrivons une fonction qui retourne la mention obtenue en fonction de la note sur 20 :

```
(define (mention note)
  (cond ((<= 16 note) "TB")
        ((<= 14 note) "B")
        ((<= 12 note) "AB")
        ((<= 10 note) "P")
        (else  "echec")))
```

Notons que l'évaluation en séquence des tests dispense d'avoir à tester des doubles inégalités comme `14 <= note < 16` car, quand on arrive au test `(<= 14 note)`, on sait que le test `(<= 16 note)` est faux.

Exercice 7 *Comparer avec la définition de cette fonction à l'aide de formes* `if`*. Ecrire la fonction* `signe` *avec un* `cond`*.*

Les connecteurs logiques and et or

On dispose de connecteurs logiques pour combiner des expressions Scheme, ce sont `and` et `or`. Ces formes spéciales correspondent à l'évaluation dite avec «court-circuit
La valeur de

```
(and s1 s2 ...sN)
```

s'obtient en évaluant en séquence les expressions *s1, s2, ...* et l'on s'arrête dès que l'une vaut `#f` en rendant `#f`, sinon on rend la valeur de la dernière expression. Par exemple, les nombres a, b, c peuvent représenter les côtés d'un vrai triangle si l'expression `(and (< a (+ b c)) (< b (+ c a))(< c (+ a b))` est vraie. L'évaluation «court-circuit» permet d'évaluer l'expression suivante sans que la division par 0 puisse déclencher une erreur :

```
?  (and (<= 2 8) (number?  "oui")(= 1 (/ 2 0))) -> #f
```

De façon analogue, la valeur de

```
(or s1 s2 ...sN)
```

s'obtient en évaluant en séquence *s1, s2, ...* et l'on s'arrête dès que l'une d'elles est vraie et on rend sa valeur, sinon on rend la valeur `#f`,

```
?  (or (= 2 3) 10 (number?  "oui")) -> 10
```

car la valeur 10 est la première valeur distincte de `#f`.

Exercice 8 *Définir une fonction* `sqrt-sure` *qui calcule la racine carrée de x après s'être assurée que x est un nombre non négatif, et sinon renvoie* `#f`*.*

1.6 Variables locales et structure de bloc

Définitions locales : `let`

Dans le cours d'un calcul, il est souvent commode de ranger le résultat d'un calcul auxiliaire dans une variable de façon à ne pas avoir à le refaire plusieurs fois. Revenons au calcul de l'expression $\sqrt{p(p-a)(p-b)(p-c)}$ où p désigne le demi-périmètre $(a+b+c)/2$. Il n'est ni élégant ni efficace de refaire le calcul de p pour chaque facteur. La méthode, suggérée précédemment, qui consiste à définir une fonction auxiliaire p, est aussi critiquable. Si cette fonction n'est utilisée que pour ce calcul, elle ne mérite pas d'être connue tout le temps de la session.
Pour répondre à ce besoin, il existe la forme spéciale `let` qui traduit exactement l'idée de nommer temporairement des valeurs le temps d'un calcul. La structure du `let` est constituée d'une partie liaisons et d'un corps :

```
(let liaisons
  corps)
```

La partie liaisons est de la forme (*(var1 exp1)* ... *(varK expK)*) et signifie que, durant l'évaluation du corps, l'identificateur *varj* désigne la valeur de *expj*.

Reprenons le calcul de la surface du triangle avec un `let`, on calcule une fois le demi-périmètre que l'on représente par une variable locale `p`, alors l'aire du triangle de côtés a, b, c s'exprime par :

```
(let ((p  (/ (+ a b c) 2)))
  (sqrt  (* p (- p a)(- p b)(- p c)))).
```

Comme on peut avoir besoin de définir plusieurs liaisons (en parallèle) on a rangé l'ensemble des liaisons *(var1 exp1) ... (varK expK)* entre deux parenthèses de façon à les distinguer de la partie *corps*.
En général, l'indentation utilisée pour le `let` consiste à placer les liaisons en colonne et le corps décalé à gauche :

```
(let ((var1  exp1)
       ...
      (varK  expK))
  corps)
```

Par exemple,

```
? (let ((a 20)
        (b (* 4 8))
        (c 10))
    (* c (- a b)))
-120
```

Exercice 9 *Réécrire avec un* `let` *la fonction qui calcule l'hypoténuse d'un triangle rectangle.*

Portée du `let`

Qu'est-ce qui se passerait si on avait déjà défini certaines des variables introduites dans un `let` ? Considérons la situation suivante, on pose :

```
?  (define a 10)
```

et l'on calcule

```
? (let ((a 5)
        (b (* 2 a)))
    (+ a b))
25
```

La valeur globale de `a` est visible dans l'expression qui définit `b` et donc `b = 20`, mais dans le corps `(+ a b)` c'est la valeur `a = 5` qui est visible. Par ailleurs, la valeur de la variable locale `b` n'est pas utilisable en dehors du `let` :

```
? b
*** ERROR -- unbound variable: b
```

De façon générale, les variables définies par les liaisons du `let` cachent les autres définitions *seulement* le temps de l'évaluation du corps du `let`.

Exercice 10 *Dans les mêmes conditions, donner la valeur de :*

```
(let ((a (* a a))
      (b (* 4 5))
      (c (* a 5)))
  (+ a b c))
```

Liaisons en séquence : `let*`

Si l'on a besoin de faire les liaisons en *séquence*, on dispose de la forme spéciale `let*` :

```
(let* ((var1  exp1)
        ...
       (varK  expK))
  corps)
```

Elle permet d'utiliser dans le calcul de ***expj*** la valeur donnée à ***vari*** si $i < j$. Par exemple, après la définition :

```
?  (define a 2)
```

on a

```
? (let* ((a (* a a)
         (b (* 3 a))
         (c (* a b)))
    (+ a b c))
64
```

En effet, dans le corps de ce `let*`, la variable `a` est liée à 4, `b` à 12 et `c` à 48.

La forme `let*` n'est pas indispensable car elle s'exprime aussi par des `let` emboîtés. L'exemple précédent s'écrit encore :

```
(let ((a (* a a)))
  (let ((b (* 3 a)))
    (let ((c (* a b)))
      (+ a b c)))).
```

Il est parfois préférable d'utiliser la formulation avec des `let` emboîtés quand on veut mettre en évidence les dépendances entre les expressions.

Exercice 11 *Donner la valeur des expressions suivantes :*

```
(let ((x 5))
  (let* ((y (+ x 10))
         (z (* x y)))
    (+ x y z)))

(let ((x 4))
  (if (= x 0)
    1
    (let ((x 10))
      (* x x))))
```

Exercice 12 *Donner une expression Scheme pour calculer :*
$\frac{\sqrt{x^2+y^2}-\sqrt{x^2-y^2}}{1+\sqrt{x^2+y^2}+\sqrt{x^2-y^2}}$ *connaissant les valeurs de x et y.*

Nous allons maintenant effleurer le cœur de la programmation applicative : la définition récursive de fonctions. Un cas particulier, connu de tous ceux qui ont fait un peu d'arithmétique, est la notion de récurrence.

1.7 Définition récursive d'une fonction

Considérons la fonction `sommeCarres` qui associe à un entier `n` la somme des carrés des entiers de 1 à n :

$$sommeCarres(n) = 1 + 2^2 + 3^2 + \cdots n^2$$

En regroupant les $n-1$ premiers carrés de droite, on en déduit que cette fonction vérifie la relation de récurrence :

$$sommeCarres(n) = sommeCarres(n-1) + n^2$$

avec la convention que pour $n = 0$ on a $sommeCarres(0) = 0$.

Scheme permet d'écrire la définition de `sommeCarres` pratiquement sous cette forme :

```
(define  (sommeCarres n)
  (if (zero? n)
    0
    (+ (sommeCarres (- n 1))
       (* n n))))

? (sommeCarres 3)
14
```

Une telle définition de fonction est dite *récursive* car la fonction que l'on définit fait appel à elle-même dans l'expression qui la définit. La sous-expression `(sommeCarres (- n 1))` s'appelle un *appel récursif.* Il n'y a rien de mystérieux dans une telle définition. Elle met simplement en œuvre le procédé de calcul suivant :

- le calcul de la somme des carrés jusqu'à n se ramène au calcul de la somme des carrés jusqu'à $n - 1$ auquel on ajoute n^2,
- pour calculer la somme des carrés jusqu'à $n - 1$, on se ramène au calcul de la somme jusqu'à $n - 2$ ...
- qui, en fin de compte, se ramène au calcul de `(sommeCarres 0)` qui lui est connu.

La trace des appels à une fonction

La plupart des implantations de Scheme donnent un moyen pour visualiser cette succession d'appels à `sommeCarres`. On demande à Scheme d'afficher tous les appels à une fonction `f` en évaluant la forme `(trace`[4] `f)`.
Pour tracer la fonction `sommeCarres`, on commence par évaluer :

```
?  (trace sommeCarres)
```

Alors le calcul de `(sommeCarres 3)` va déclencher l'affichage de tous les appels récursifs de `sommeCarres` et des valeurs correspondantes. L'indentation permet de voir plus facilement la valeur retournée par chaque appel :

```
? (sommeCarres 3)
| Entry (sommecarres 3)
|  Entry (sommecarres 2)
|  |Entry (sommecarres 1)
|  | Entry (sommecarres 0)
|  | ==> 0
|  |==> 1
|  ==> 5
| ==> 14
14
```

On peut tracer en même temps plusieurs fonctions, pour suivre encore plus finement le déroulement des calculs. Par exemple, on demande aussi la trace de la

[4] La syntaxe de cette fonction peut varier selon les implantations, voir le manuel de l'utilisateur.

fonction +[5] :

```
? (trace +)
```

Alors

```
? (sommeCarres 3)
| Entry (sommecarres 3)
|  Entry (sommecarres 2)
|   |Entry (sommecarres 1)
|   | Entry (sommecarres 0)
|   | ==> 0
|   | Entry (+ 0 1)
|   | ==> 1
|   |==> 1
|   |Entry (+ 1 4)
|   |==> 5
|  ==> 5
|  Entry (+ 5 9)
|  ==> 14
| ==> 14
14
```

Pour faire cesser les traces d'une fonction, on utilise la fonction : `untrace`

```
? (untrace +)
```

Et pour faire cesser toutes les traces, on appelle simplement :

```
? (untrace)
```

La forme précise des affichages des appels d'une fonction tracée dépend du Scheme utilisé, mais l'allure générale reste la même.

La fonction factorielle

Un autre exemple typique de définition récursive est la définition de la fonction factorielle. On rappelle que $n!$ désigne le produit des n premiers entiers :

$$n! = 1 * 2 * \cdots * n$$

et on pose par convention $0! = 1$.
On vérifie que pour tout entier $n \geq 1$ on a l'égalité $n! = n * (n-1)!$, d'où une définition récursive de la fonction factorielle :

```
(define (factorielle n)
  (if (zero? n)
    1
    (* n (factorielle (- n 1)))))

? (factorielle 5)
120
```

[5]Certaines implantations ne permettent pas de tracer les fonctions prédéfinies.

Terminaison d'un appel récursif

Une définition récursive doit toujours comporter au moins un cas, dit de base, où le résultat s'obtient sans appel récursif, car sinon la fonction se rappelle indéfiniment, on dit qu'elle *boucle.* Mais ce n'est pas suffisant pour assurer la terminaison des appels. Il faut s'assurer que chaque appel récursif concerne des arguments «plus petits» que les arguments initiaux. La méthode la plus courante pour s'assurer de la terminaison d'une fonction récursive est d'exhiber une fonction à valeurs entières et positives qui prenne une valeur strictement plus petite sur l'argument de chaque appel récursif. Dans le cas très simple de la fonction `sommeCarres` il suffit de prendre l'argument `n`.
Dans le cas de factorielle, on peut aussi raisonner sur la décroissance de `n` en supposant que c'est un entier ≥ 0. En effet, le calcul de :

```
? (factorielle -2)
```

déclenche un bouclage infini qui se termine en général par un message du type
`*** ERROR Stack overflow`

Si l'on suit les appels, on constate que `(factorielle -2)` appelle `(factorielle -3)` qui appelle ... On a bien une suite décroissante d'entiers mais elle n'est pas minorée.

La fonction de Fibonacci

Autre grand classique des définitions récursives : la fonction de Fibonacci. Elle illustre le cas où l'on utilise plusieurs appels récursifs dans le corps de la fonction. Elle est définie par la relation de récurrence :

$$\textit{pour } n \geq 0 \quad f(n+2) = f(n+1) + f(n) \quad \textit{avec} \quad f(0) = 0 \quad \textit{et} \quad f(1) = 1$$

D'où immédiatement sa définition en Scheme :

```
(define (fib n)
  (if (< n 2)
    n
    (+ (fib (- n 1)) (fib (- n 2)))))
```

La terminaison est évidente puisque pour $n \geq 2$, les entiers $n-1$ et $n-2$ sont strictement plus petits que n et sont ≥ 0.

Méthodologie de la programmation récursive

L'utilisation de la programmation récursive n'est pas réservée aux fonctions définies par une relation de récurrence. C'est une méthode générale et très puissante de résolution de problèmes qui va être abondamment développée et illustrée dans ce livre.
Pour résoudre un problème, il faut toujours se poser les deux questions suivantes :

- est-ce que je peux obtenir la solution de mon problème à partir de la solution de ce problème sur des objets plus petits en un certain sens?
- est-ce qu'il y a des cas particuliers du problème où je peux obtenir une solution directement?

Si l'on répond oui, la fonction récursive qui calcule la solution du problème est presque écrite.

Exercice 13 *1. Ecrire une fonction qui associe à un entier $n > 0$ la somme $1 + 1/2 + ... + 1/n$.*

2. Ecrire une fonction **puissance** *prenant en argument un nombre x et un entier $n \geq 0$ et retourne la valeur de x^n. On notera que $x^n = x * x^{n-1}$. Etendre au cas où n peut être négatif.*

1.8 Entrée/sortie

Il est intéressant de noter que l'on a déjà pu programmer un grand nombre de fonctions sans avoir à appeler explicitement des primitives d'entrée / sortie. En effet, on a profité de la boucle d'interaction:

`...-> lire-> évaluer-> afficher -> ...`

C'est d'ailleurs un des principes de la programmation en Scheme: *on évite de mélanger dans une fonction les affichages avec les calculs*; on réalise des fonctions pour chaque tâche. Mais, dans certains cas, il est utile de pouvoir contrôler les entrées/sorties à l'intérieur d'une fonction. Dans ce chapitre d'introduction à Scheme, on se contente d'indiquer les principales fonctions de lecture et d'affichage.

Lecture: `read`

Pour la lecture, on dispose d'une puissante fonction `(read)`, qui permet de lire n'importe quelle donnée Scheme; cette donnée est retournée non évaluée.

```
? (read) ; après évaluation de cette expression
 (+   2   3) ; on entre une expression avec  des espaces superflus
(+ 2 3)  ; il s'affiche en retour l'expression non évaluée
```

Affichage: `write` et `display`

Pour l'affichage on dispose d'une fonction qui affiche la valeur de n'importe quelle S-expression `(write s)`:

```
? (write "Bonjour tout le monde")
"Bonjour tout le monde"
```

La valeur rendue par une fonction d'affichage est sans importance. On dit qu'elle est indéfinie et varie selon les implantations, ce qui compte c'est l'effet visuel d'affichage sur l'écran. Dans la suite, on n'indiquera que l'affichage et non la

valeur indéfinie retournée.

La valeur affichée par `write` est une donnée Scheme et donc lisible par `read`, mais on peut préférer un affichage plus agréable pour le lecteur humain. Par exemple, afficher les chaînes sans leurs guillemets. Pour cela, on dispose de la fonction `display` qui se comporte comme `write` sauf pour certaines valeurs comme les chaînes :

```
? (display "Bonjour  tout le monde")
Bonjour  tout le monde
```

On utilise la fonction sans paramètre `newline` pour passer à la ligne suivante.

Une boucle avec un menu

Illustrons l'utilisation de ces fonctions en écrivant une boucle qui affiche un menu et, en réponse au nombre tapé par l'utilisateur, exécute une action fictive à moins que l'utilisateur ait choisi de taper 0 pour quitter cette boucle. En cas de frappe erronée, on repose la question :

```
(define (menu)
  (display "taper 0 pour quitter, 1 pour l'action1, 2 pour l'action2 : ")
  (let ((lu (read)))
    (cond ((equal? lu 0)(display "au revoir"))
          ((equal? lu 1)(display "j'execute l'action 1")
                     (newline)(menu))
          ((equal? lu 2)(display "j'execute l'action 2")
                     (newline)(menu))
          (else (display "recommencer  commande inconnue ")
                (newline)(menu)))))
```

Dans la fonction `menu`, on voit pour la première fois l'utilisation d'effets de bord dans le corps d'une fonction, c'est-à-dire d'expressions qui ne sont pas utilisées pour leurs valeurs mais pour l'effet produit (ici un affichage). A cette fin, la syntaxe du corps d'une fonction peut être une succession d'expressions (deux dans le cas de la fonction `menu`). Pour la même raison, les parties `corpsj` des branches de la forme spéciale `cond` peuvent admettre aussi une succession d'expressions. Noter également l'utilisation de la récursivité pour réaliser cette boucle (sans fin, si l'utilisateur ne répond jamais par 0).

Exercice 14 *1. Ecrire une fonction qui demande le prix hors taxe et le taux de TVA et affiche une facture de la forme :*

```
Prix HT     : xxxx
TVA         : xxxx
-------------------
Prix TTC    : xxxx
```

2. *Ecrire une fonction qui demande à l'utilisateur de lui donner un nombre compris entre 0 et 1000 et en réponse lui retourne sa racine carrée. Intégrer cette fonction au sein d'une boucle avec un menu.*

Exercice 15 *On dispose de trois piquets : A , B , C; sur le piquet A sont empilés n anneaux de tailles décroissantes. Il s'agit de transférer ces anneaux sur le piquet C en respectant la règle suivante : on ne déplace qu'un anneau à la fois et on ne peut poser un anneau sur un anneau plus petit. Le piquet B peut être utilisé comme intermédiaire. Ecrire une fonction* `(Hanoi n source intermediaire but)` *qui affiche une suite de déplacements à effectuer. Par exemple :*

```
 ? (Hanoi 2 'A  'B  'C)
a  vers  b
a  vers  c
b  vers  c
```

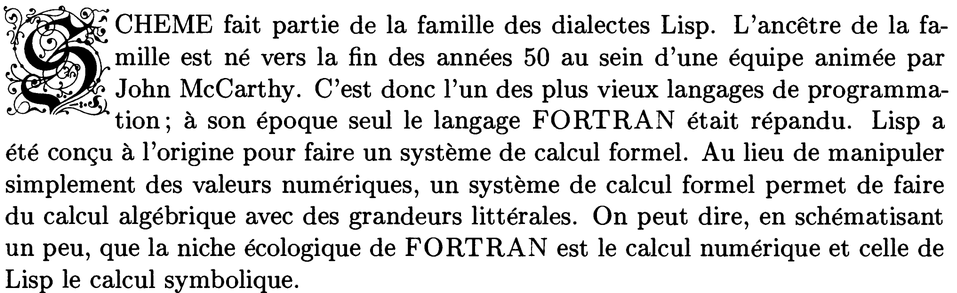
Indication : exprimer le transfert de n anneaux à partir du transfert de $n-1$ anneaux entre des piquets bien choisis. Une solution se trouve dans la plupart des livres sur Lisp.

1.9 De la lecture

L'histoire et l'évolution des dialectes Lisp est présentée dans [SG93].
Pour une initiation humoristique à Scheme [FF91].
Présentation des bases de la programmation en Scheme [Dyb87, SF90, HW94, ML95].

Chapitre 2

Expressions symboliques



SCHEME fait partie de la famille des dialectes Lisp. L'ancêtre de la famille est né vers la fin des années 50 au sein d'une équipe animée par John McCarthy. C'est donc l'un des plus vieux langages de programmation ; à son époque seul le langage FORTRAN était répandu. Lisp a été conçu à l'origine pour faire un système de calcul formel. Au lieu de manipuler simplement des valeurs numériques, un système de calcul formel permet de faire du calcul algébrique avec des grandeurs littérales. On peut dire, en schématisant un peu, que la niche écologique de FORTRAN est le calcul numérique et celle de Lisp le calcul symbolique.

Les expressions symboliques se rencontrent dans d'autres domaines que les mathématiques, comme par exemple celui de la représentation et du traitement des connaissances, appelé aussi «intelligence artificielle» ou celui de la théorie des langages de programmation. En disposant des listes d'objets symboliques comme structure fondamentale, Lisp offre un moyen très souple pour modéliser les autres données. Le nom Lisp provient de la contraction des mots «List» et «processing», ce qui résume bien une de ses principales caractéristiques.

Depuis sa première version historique, appelée Lisp 1.5, ce langage a beaucoup évolué et ses nombreux descendants constituent la famille des dialectes Lisp. Le langage Scheme est actuellement l'un des plus connus avec Common Lisp. Quand on voudra parler d'une propriété de Scheme commune aux divers dialectes, on utilisera parfois le nom Lisp au lieu de Scheme.

2.1 Symboles et citations

La citation

Pour le moment, les seules valeurs que nous ayons manipulées sont les nombres, les booléens et les chaînes. Mais Scheme est un langage de manipulation symbolique et, à ce titre, autorise comme valeur possible un symbole, c'est-à-dire un identificateur auquel on n'attache aucune autre valeur.

Mais si on veut attribuer une valeur symbole à un identificateur, on rencontre une difficulté. Par exemple, essayons d'associer à la variable `son-prenom` la valeur symbolique `Pierre` :

```
? (define  son-prenom  Pierre)
*** ERROR -- unbound variable: pierre
```

Il y a déclenchement d'une erreur car Scheme évalue l'identificateur `Pierre` qui a priori n'a pas de valeur. Pour indiquer à Scheme que sa valeur c'est le symbole lui-même, on fait précéder `Pierre` d'une *apostrophe* : `'Pierre`. C'est le mécanisme dit de la *citation* (*quotation* en anglais)

```
? (define  son-prenom  'Pierre)
```

Maintenant, si on demande la valeur de `son-prenom`, on obtient :

```
? son-prenom
pierre
```

La majuscule de `Pierre` a disparu car l'imprimeur Scheme affiche tous les symboles en minuscules.
Le mécanisme de la citation se rencontre aussi dans le langage courant. Si vous dites à quelqu'un «`écrivez votre nom`», l'interlocuteur peut réagir de deux manières :

- soit en écrivant : `Durand`, s'il s'appelle Durand
- soit en écrivant : `votre nom`.

Dans la langue écrite, la distinction est également faite aussi à l'aide d'apostrophes pour spécifier la deuxième réponse : «`écrivez 'votre age'`».

Exercice 1 *Deviner la réponse de Scheme à*
? ''Pierre

La forme `quote`

En fait le caractère ' est une abréviation. L'expression `'s` est lue par le lecteur Scheme comme étant `(quote s)`, où `quote` est une forme spéciale prédéfinie qui a la propriété de retourner son argument tel quel. Ainsi on ne sort donc pas de la classe des expressions Scheme décrites au chapitre 1 §3.

```
? (quote  Pierre)
pierre
```

Le prédicat `symbol?`

Pour tester si une expression a pour valeur un symbole, on dispose du prédicat `symbol?` :

```
? (symbol? 'Pierre)    -> #t
? (symbol? #t)         -> #f
? (symbol? "Pierre")  -> #f
```

Le dernier cas montre qu'il ne faut pas confondre symbole et chaîne bien que, dans la plupart des langages, les chaînes soient le seul moyen pour modéliser des identificateurs.
En représentant les types des valeurs que nous connaissons par leurs noms, voici à titre d'exemple une fonction qui nous donne le type d'une expression Scheme :

```
(define (type-de s)
  (cond ((symbol? s)  'symbole)
    ((number? s)  'nombre)
    ((boolean? s) 'booleen)
    ((string? s)  'chaine)
    (else 'typeNonPrevu)))
```

Identificateur, variable, mot-clé et symbole

Revenons sur la distinction entre identificateur, variable, mot-clé et symbole. Un identificateur sert à nommer un objet, cet objet peut être :

- une variable, c'est-à-dire un nom pour désigner une valeur Scheme,
- un mot clé, c'est-à-dire le nom donné à une forme spéciale (il est déconseillé de les redéfinir) : `define, let, cond, quote`, ...
- un symbole, c'est-à-dire une valeur Scheme.

Par exemple, après évaluation de :

```
(define son-prenom  'Pierre)
```

L'identificateur `son-prenom` est une *variable* qui désigne le *symbole* `Pierre`.

2.2 Les listes

Expressions symboliques structurées

On a souvent besoin de regrouper des données en une donnée composée unique. Ainsi, la liste des jours de la semaine est composée de lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche ; mais comment amalgamer ces symboles en une donnée unique ? La solution est de réunir ces symboles dans une *liste* :

```
 (lundi mardi mercredi jeudi vendredi samedi dimanche )
```

ce qui en fait une expression syntaxiquement admissible. Mais alors on a le même problème que pour les symboles, la définition :

```
? (define  Lsemaine
    (lundi  mardi mercredi jeudi vendredi samedi dimanche))
*** ERROR -- unbound variable: lundi
```

déclenche une erreur. En effet, Scheme considère cette liste comme un appel de la fonction `lundi` avec les arguments `mardi, ..., dimanche` et, selon toute vraisemblance, nous n'avons pas de fonction `lundi`.
Comment signifier à l'évaluateur Scheme qu'il ne s'agit pas de l'appel de la fonction `lundi` mais textuellement de la liste des jours? Comme pour les symboles, la solution est d'utiliser la citation:

```
? (define  Lsemaine
    '(lundi  mardi mercredi jeudi vendredi samedi dimanche))
```

Ensuite, on a bien

```
? Lsemaine
(lundi  mardi mercredi jeudi vendredi samedi dimanche)
```

Maintenant `Lsemaine` est une variable qui a pour valeur une liste de symboles. On s'efforcera de suivre la convention suivante: une variable désignant une liste sera préfixée par la lettre L.
On a déjà utilisé des listes plus complexes pour représenter les expressions arithmétiques. On peut aussi décrire des données symboliques avec des listes imbriquées. Par exemple, si l'on désire représenter les informations d'une page d'agenda, on peut utiliser une liste de la forme:

```
? (define L-agenda-10-octobre
    '(lundi
       (matin
         (9h    lire-courrier)
         (9h-30 preparer-cours)
         (10h   cours-Scheme)
         (12h   dejeuner))
       (apres-midi
         (14h seminaire)
         (16h rendez-vous))))
```

Exercice 2 *Donner la valeur des expressions: '(+ 4 7) , ''(a b) , '5.*

Expression littérale

Une expression quotée s'appelle une *expression littérale* car sa valeur est précisément son texte. Elle joue le rôle de constante:

```
? '(* 4 5)       -> (* 4 5)
? '(un + un = deux) -> (un + un = deux)
```

Opérations de base sur les listes: cons, car, cdr

Pour désigner la liste ne contenant rien, on dispose de la *liste vide* `()`.
Attention, la forme `()` n'est *pas* une expression Scheme admissible, c'est une *valeur*.
Pour utiliser la liste vide, comme expression, il faut la quoter:

```
? ()
*** ERROR -- Ill-formed expression: ()

? '()
()
```

Pour construire une liste, on dispose du *cons*tructeur **cons** :
(cons s L) retourne une nouvelle liste obtenue en insérant la valeur de s en tête de la liste L.

```
? (cons  'a   '())       -> (a)
? (cons (+ 5 8) '(a b)) -> (13 a b)
```

On peut résumer la définition de la structure LISTE par l'égalité :

```
 LISTE = ()  ou  (cons  S-expression  LISTE)
```

Cette présentation met en évidence la structure *récursive* du type LISTE.

Les accesseurs **car** et **cdr** permettent d'accéder aux éléments d'une liste *non vide.*
(car L) retourne le premier élément de L.
(cdr L) retourne la liste L privée de son premier élément.

```
? (car   '(a b c))   -> a
? (cdr   '(a b c))   -> (b c)
```

De sorte que l'on a les égalités :

(car (cons s L) = $\bar{s}$ (cdr (cons s L) = $\bar{L}$

avec notre convention de désigner par un surlignement la valeur d'une expression.

Remarque 1 En Scheme, le **car** ou le **cdr** d'une liste *vide* déclenche une erreur alors que dans la plupart des dialectes Lisp, il est d'usage de retourner la liste vide.

Remarque 2 Les noms **car** et **cdr** proviennent de la machine sur laquelle on a réalisé la première implantation de Lisp, la tradition les a gardés.

On prononce **car** comme le pronom et **cdr** en l'épelant. Si l'on veut changer le nom de ces fonctions, il suffit de définir des fonctions synonymes **tete** et **reste** par :

```
(define tete  car)
(define reste cdr)
```

On ne le conseille pas, car l'apprentissage d'un langage ne consiste pas seulement à savoir programmer dans ce langage mais aussi à pouvoir lire le code des autres.

Exercice 3 *Deviner la valeur des expressions :*
(cons 'a '((b c))) , (cdr '(a)) , (cdr Lsemaine)
(car '((+ 4 1))) , (cdr '((+ 4 1))) , (car '((a) b)
(cdr '((a) (b)) , (cdr ''(a b))

En composant les fonctions **car** ou **cdr**, on peut accéder à n'importe quel élément d'une liste :

```
? (car (cdr  Lsemaine))               -> mardi
? (car (cdr  (cdr Lsemaine)))         -> mercredi
? (cdr (cdr  (cdr (cdr Lsemaine)))) -> (vendredi samedi dimanche)
```

Ce type d'expression devient vite assez lourd à écrire, aussi dispose-t-on d'un système d'abréviation : pour désigner la composée d'au plus quatre **car** ou **cdr**, on écrit `c x1 ...xk r` où `xj` est un **a** ou un **d**. Par exemple, la fonction composée `car o cdr o cdr` s'écrit aussi **caddr**.

```
? (caadr  '(a (b) c d e) -> b
? (cddddr '(a (b) c d e) -> (e)
```

Exercice 4 *1. Ecrire l'expression permettant d'extraire le week-end:* *(samedi dimanche) de Lsemaine*

2. Ecrire des fonctions donnant le deuxième, le troisième et le quatrième élément d'une liste.

3. Donner la combinaison de car et de cdr pour obtenir en valeur le symbole a à partir des listes: ((b a) (c d)), (( ) (a d)), (((a))).

4. Quelle est la valeur de (car ''a) et de (cdr ''a) ?

Les fonctions list et append

Pour construire une liste, on dispose d'autres fonctions prédéfinies.
La fonction `list` prend un nombre quelconque d'arguments et construit une nouvelle liste composée des valeurs de ses arguments :

```
? (list '(a b) '(c d) '() '(( e )) )  ->  ((a b) (c d) () (e))
```

La fonction **append** prend un nombre quelconque d'arguments ayant chacun pour valeur une liste et construit une nouvelle liste résultant de la concaténation du contenu de ces listes :

```
? (append '(a b) '(c d)  '() '(( e ))) -> (a b c d (e) )
```

Il est important de ne pas confondre les utilisations de **cons**, `list` et **append**:

```
? (cons '(a b) '(c d))
((a b) c d))

? (list '(a b) '(c d))
((a b) (c d))

? (append '(a b) '(c d))
(a b c d)
```

Pour accéder plus directement aux éléments d'une liste, ce qui revient à les traiter comme des tableaux dynamiques, on dispose de la fonction `(list-ref L n)`. Elle retourne le nième élément de la liste L, sachant que dans une liste les éléments sont numérotés à partir de 0.
On a aussi la fonction `(list-tail L n)` qui rend ce qui reste de la liste quand on a supprimé les n premiers éléments :

```
? (define   Week-end   (list-tail Ljours 5))
```

Atomes, listes et listes plates

Parmi les S-expressions, on fait parfois la distinction entre les listes et les autres types (symboles, nombres, chaînes, ...) On englobe ces autres types sous le nom d'*expressions atomiques.* On peut alors donner une définition récursive du type S-expression :

```
S-expression  =   ATOME   ou   LISTE
LISTE = ( {S-expression} )
ATOME = symbole  ou  nombre ou  booleen ou chaine
```

La mise entre accolades de `S-expression` est une convention pour désigner la succession d'un nombre fini (éventuellement zéro) de S-expressions.

Remarque 3 On voit maintenant plus clairement une des spécificités de Lisp : les programmes sont de même nature que les données : des S-expressions. Par exemple, la définition d'une fonction par `(define ...)` est une liste. C'est un aspect fondamental de Lisp, il sera à la base du principe des extensions syntaxiques décrit au chapitre 9.

On appelle *sous-liste* d'une liste, un élément d'une liste qui est une liste ou qui est sous-liste d'une sous-liste. Par exemple, la liste
`(a (b c) () ((c)) d)` possède les sous-listes `(b c), (), ((c)), (c).`
Une liste n'ayant pas de sous-liste est dite une *liste plate.* On dit qu'un élément est au *premier niveau* d'une liste s'il n'est pas dans une sous-liste. Les éléments au premier niveau de la liste précédente sont : `a , (b c) , () , ((c)) , d .`
On appelle éléments au nième niveau ceux qui sont au premier niveau d'une sous liste située au niveau $n-1$. La *profondeur* d'une liste est le niveau d'un élément de niveau maximum ; la liste précédente est de profondeur 3.

Exercice 5 *Donner pour le type* `LISTE-PLATE` *une définition analogue à celle de* `LISTE`.

La *longueur* d'une liste est le nombre d'éléments au premier niveau, elle se calcule à l'aide de la fonction prédéfinie `length` :

```
? (length  '((a b) (c d) () (e) )) -> 4
```

Quelques prédicats sur les listes

Pour comparer des listes, on utilise le prédicat `equal?`

```
? (equal? '(a b 3)  (list 'a 'b  (+ 1 2))) -> #t
? (equal? '(())  '()) -> #f
```

Pour décider de la nature d'une S-expression, on dispose des prédicats suivants :

`(list?  s) -> #t` si la valeur de l'expression `s` est une liste .
`(null?  s) -> #t` si la valeur de l'expression `s` est la liste vide .
`(pair?  s) -> #t` si la valeur de l'expression `s` est une liste non vide .

On aura parfois besoin d'un prédicat atom? pour tester si une S-expression n'est pas une liste et aussi d'un prédicat singleton? pour tester si une liste est réduite à un seul élément. Comme ces prédicats ne sont pas des fonctions prédéfinies, on les ajoute à notre bibliothèque d'utilitaires :

```
(define (atom? s)
  (not (pair? s)))

(define (singleton? L)
  (null? (cdr L)))
```

On est souvent amené à tester si un objet est élément au premier niveau d'une liste. Pour cela, on peut utiliser la fonction prédéfinie (member s L). Elle retourne le reste de la liste L contenant la première occurrence de la valeur de s ou bien #f si elle n'en a pas. La comparaison se fait avec le test equal? :

```
? (member '(b) '(a b c d))         -> #f
? (member '(b) '(a b c (b) d))     -> ((b) d)
```

Exercice 6 *Définir à partir de la fonction member un prédicat member?.*

L'égalité de valeurs Scheme peut être définie à plusieurs niveaux de précision. Le prédicat equal? compare les structures mais non l'identité des objets. L'identité des objets se teste avec eq? et il y a aussi le prédicat un peu moins strict eqv?. Pour le moment, il suffit de savoir que pour comparer des listes ou des chaînes, on utilise equal?, que pour comparer des nombres on utilise = et que pour comparer des symboles, on utilise eq?. Ces distinctions seront expliquées au §9.
Corrélativement à ces tests, on dispose de trois fonctions d'appartenance :

member qui utilise le test equal?.

memv qui utilise le test eqv?.

memq qui utilise le test eq?.

```
? (memq '(b) '(a b c (b) d))                  -> #f
? (member "blanc" '("bleu" "blanc" "rouge")) -> ("blanc" "rouge")
```

La forme case

Ces précisions sur l'égalité nous permettent d'introduire une nouvelle forme spéciale case. Sa syntaxe est voisine de celle de cond qu'elle remplace souvent quand on veut faire un raisonnement par cas :

```
(case s
  (L1 corps1)
  (L2 corps2)
    ...
  (else corpsN))
```

Son évaluation consiste à évaluer la S-expression *s* et tester si sa valeur appartient (par `memv`) à la liste *L1*. Si oui, on retourne la valeur de ***corps1***, sinon on recommence avec la liste *L2*, ... Si on arrive à la branche commençant par le mot-clé `else` alors on rend la valeur de ***corpsN***.

Remarque 4 Comme pour le `cond`, les corps peuvent être des suites de S-expressions et le `else` est optionnel.

Comme illustration, voici une petite fonction pour traduire un pronom personnel du français vers l'anglais :

```
(define (traduit-pronom pronomF)
  (case pronomF
    ((je)       'I)
    ((tu vous) 'you)
    ((il)       'he)
    ((elle)    'she)
    ((nous)    'we)
    ((ils)     'they)
    (else 'pronom-inconnu)))

? (traduit-pronom 'vous) -> you
```

Exercice 7 *L'écriture habituelle des expressions arithmétiques utilise des règles de priorité afin déviter les ambiguïtés. On utilise en général l'ordre de priorité suivant : on affecte la priorité 3 à l'opération de puissance, la priorité 2 aux opérations* `*` *et* `/` *et la priorité 1 aux opérations* `+` *et* `-`*. Ecrire une fonction* ***priorite*** *qui donne la priorité d'un opérateur arithmétique.*

La structure de liste est très générale, c'est la bonne à tout faire en Lisp car elle permet de modéliser toutes sortes de données symboliques ou non.

Représentation des racines du trinôme

A titre d'exemple, reprenons l'exercice 2 du chapitre 1 concernant le calcul des racines de l'équation du deuxième degré $x^2 + 2bx + c = 0$ à coefficients réels.
La discussion se fait selon le signe du discriminant $delta = b^2 - c$:

- si $delta > 0$, on a deux racines réelles $x_1 = -b + \sqrt{delta}$ et $x_2 = -b + \sqrt{delta}$,
- si $delta = 0$, on a une racine réelle double $-b$,
- si $delta < 0$, on a deux racines complexes conjuguées de partie réelle $-b$ et de partie imaginaire $\pm\sqrt{-delta}$.

Pour représenter en Scheme ces différentes solutions, on convient que :

- si $delta > 0$, on représente les racines par une liste de trois éléments,

  `(reelles x1 x2)`

- si *delta* $= 0$, on représente les racines par une liste à deux éléments,
  `(double x)`
- si *delta* < 0, on représente les racines par une liste à trois éléments.
  `(complexes partieReelle partieImaginaire)`

Avec cette représentation, le calcul des racines est donné par la fonction:

```
(define (RacinesTrinome b c)
  (let ((delta (- (* b b) c)))
    (cond ((< 0 delta)
           (let ((Rac2delta (sqrt delta)))
             (list 'reelles  (+ (- b) Rac2delta)  (- (- b) Rac2delta))))
          ((zero? delta)(list 'double  (- b)))
          (else  (let ((Rac2delta (sqrt (- delta))))
                   (list 'complexes  (- b) Rac2delta))))))
```

Comme on l'a déjà indiqué, il faut éviter de mélanger les calculs et les affichages. Voici une fonction qui affiche de façon plus lisible la solution calculée par `RacinesTrinome`. On utilise la forme `case` pour discuter selon la nature de la solution à afficher:

```
(define (affiche-solution solution)
  (case (car solution)
    ((reelles) (display " 2 racines reelles : ")
               (display (cadr solution))
               (display " et " )
               (display (caddr solution)))
    ((double)  (display " 1 racine reelle double : ")
               (display (cadr solution)))
    (else      (display " 2 racines complexes conjuguees : ")
               (display (cadr solution))
               (display " +/- i " )
               (display (caddr solution)))))


? (affiche-solution (RacinesTrinome  -3 8))
2 racines reelles : 4 et 2
```

2.3 Programmation récursive avec les listes plates

La structure récursive des listes encourage la programmation récursive des fonctions qui les manipulent. Quand on écrit une fonction prenant en paramètre une liste, il faut avoir le réflexe de la programmation récursive: est-ce que je peux ramener le calcul de cette fonction au calcul de la même fonction sur le `cdr` de la liste?
On va donner quelques grands classiques de la programmation avec des listes. On peut les diviser en deux classes: les fonctions qui ne considèrent les listes qu'au premier niveau et celles qui travaillent à tous les niveaux.
On commence par le plus simple, l'utilisation des listes au premier niveau.

Somme des éléments d'une liste

Soit `Lnb` une liste de nombres, comment calculer la somme de ces nombres avec la convention que si la liste est vide alors sa somme vaut 0?
Si `Lnb = (n1 n2 ... nj)` , on utilise l'égalité:
`n1 + n2 + ... + nj = n1 + (n2 + ... nj)`
qui prouve que cette fonction vérifie la relation:
`(sommeListe '(n1 n2 ... nj)) = n1 + (sommeListe '(n2 ... nj))`
D'où l'écriture de cette fonction:

```
(define (sommeListe Lnb)
  (if (null? L)
    0
    (+ (car Lnb)
       (sommeListe (cdr Lnb)))))
```

Exercice 8 *Ecrire la fonction analogue pour le produit, avec la convention que pour la liste vide le produit vaut 1.*

La liste des entiers de 0 à n

Pour générer la liste `(0 1 ... n-1 n)`, on remarque qu'elle s'obtient en ajoutant `n` à la fin de la liste `(0 1 ... n-1)`. Pour ajouter la valeur de `s` à la fin d'une liste L, on introduit la fonction auxiliaire:

```
(define (append1  L  s)
  (append L (list s)))
```

On appelle traditionnellement `iota` la fonction qui rend la liste des entiers de 0 à `n`:

```
(define (iota n)
  (if (zero? n)
    '(0)
    (append1  (iota (- n 1)) n)))
```

Renverser une liste

Comme son nom l'indique, la fonction `renverse`[1] rend une liste où l'on a renversé l'ordre des éléments au premier niveau. Son principe est analogue à celui de la fonction `iota`: on renverse le reste de la liste et l'on y ajoute à la fin le premier élément de la liste:

```
(define (renverse L)
  (if (null? L)
    '()
    (append1  (renverse (cdr L)) (car L))))

? (renverse '(a  (b c d)  e) -> (e (b c d) a)
```

[1] Il existe une fonction prédéfinie `reverse`.

Une fonction append à deux arguments

Si l'on ne disposait pas de la fonction **append**, on pourrait la fabriquer grâce à la fonction **cons**. Définissons une fonction **append2** qui concatène deux listes `L1` et `L2`, il y a deux cas :

- si la liste `L1` est vide, le résultat c'est `L2`,
- sinon, on ajoute le **car** de `L1` en tête de la concaténation du reste de `L1` avec `L2`.

```
(define (append2 L1 L2)
  (if (null? L1)
    L2
    (cons (car L1)
          (append2 (cdr L1) L2))))

? (append2 '(a b c) '((c d))) -> (a b c (c d))
```

Notons ici que l'appel récursif ne diminue que le premier argument `L1`, mais c'est suffisant pour assurer la terminaison.
On a noté cette fonction **append2** pour ne pas écraser la définition de la fonction prédéfinie **append**. Même s'il est possible de redéfinir des fonctions prédéfinies, cela n'est pas souhaitable pour la lisibilité des programmes. Après les définitions :

```
? (define  car +)
? (define  * cdr)
```

il devient délicat de programmer, mais il s'agit ici d'un cas d'école.

Suppression d'éléments dans une liste

Soient une expression `s` et une liste `L`, on désire écrire une fonction **remove** qui rend la liste `L` dans laquelle on a supprimé toutes les occurrences au premier niveau de la valeur de `s`. Il y trois cas à considérer :

- si la liste est vide, on rend la liste vide ;
- si elle commence par un élément égal à `s`, il suffit de supprimer les `s` dans le reste de la liste ;
- si elle ne commence pas par un élément égal à `s`, on garde cet élément et on supprime `s` dans le reste.

```
(define (remove s L)
  (cond ((null? L) '())
        ((equal? s (car L))(remove s (cdr L)))
        (else (cons (car L)(remove s (cdr L))))))

? (remove 'a '(a b c a d)) -> (b c d)
```

Exercice 9 1. *Ecrire une fonction* `remove1` *qui ne supprime que la première occurrence de* `s` *dans* `L`.

2. *Ecrire une fonction* `dernier` *qui retourne le dernier élément d'une liste.*

3. *Ecrire une fonction* `sauf-dernier` *qui retourne une liste privée de son dernier élément.*

4. *Définir un prédicat* `Liste-plate?` *pour tester si une expression est une liste plate.*

5. *Ecrire une fonction qui donne le nombre d'occurrences au premier niveau d'un symbole dans une liste.*

   `? (nb-occurrence 'a '(e a c a (b c a) ((a a)) d)) -> 2`

6. *Ecrire une fonction pour supprimer le bégaiement. Etant donnée une liste, on rend une liste sans les répétitions d'éléments successifs.*

   `? (supprime-begaiement '(je je ne ne ne be be gaie plus))`

   `(je ne be gaie plus))`

7. *Etant donné un entier positif* `k` *et une liste* `L`, *écrire une fonction qui retourne les* `k` *premiers éléments de* `L`.

8. *Soit* `L` *une liste strictement croissante de nombres, écrire une fonction qui calcule le nombre d'éléments* `x` *de* `L` *tels que* `x` *et* `x + 10` *soient dans* `L`.

Le compte est bon

Voici un exemple plus délicat, c'est une forme simplifiée du jeu «le compte est bon». On se donne un entier `N` et une liste croissante `Lnb` d'entiers > 0. Il s'agit de trouver un ensemble d'éléments de `Lnb` dont la somme est égale à `N`. Par exemple, pour totaliser le nombre $N = 21$ avec des éléments de la liste
`(1 1 3 5 7 10 12 15)`, on peut utiliser les éléments `(1 1 7 12)`.
Si la liste est vide, on convient que sa somme est 0, si elle est non vide on discute sur la valeur $x0$ de son premier (et donc plus petit) élément :

1. Si $x0$ est supérieur à N il n'y aura pas de solution on rend `#f`.
2. Si $x0$ est égal à N, il suffit de rendre la liste réduite à $x0$.
3. Si $x0$ est inférieur à N, on regarde si $x0$ peut faire partir d'une solution. Pour cela on considère le même problème avec les autres éléments et pour la somme $N - x0$:
   (a) si on obtient ainsi une solution partielle, il suffira d'y ajouter $x0$;
   (b) sinon, il n'y pas de solution avec $x0$ et l'on cherche à résoudre le même problème sans utiliser $x0$.

```
(define (compte-bon N Lnb)
  (cond ((null? Lnb) (if (zero? N) '() #f))
        ((< N (car Lnb)) #f)
        ((= N (car Lnb)) (list N))
        (else (let ((sol-partielle (compte-bon (- N (car Lnb))
                                               (cdr Lnb))))
                (if  sol-partielle
                  (cons (car Lnb) sol-partielle)
                  (compte-bon N (cdr Lnb))  )))))

? (compte-bon 21  '( 1 1  3 5 7 10 12  15)) -> (1 1 7 12 )
```

2.4 Programmation récursive avec les listes quelconques

Quand on doit écrire une fonction prenant en paramètre une liste générale, il faut se demander : est-ce que je peux ramener le calcul de cette fonction au calcul de la même fonction sur le `cdr` et/ou sur le `car` de la liste ? On illustre ce principe avec quelques exemples classiques.

Aplatir une liste

On veut construire une nouvelle liste par suppression de toutes les parenthèses intérieures :

```
? (aplatir '(a (b c) () ((d)) e)) -> (a b c d e)
```

- Si le `car` de la liste est une liste, on l'aplatit et on concatène le résultat avec l'aplatissement du `cdr`,
- Si le `car` n'est pas une liste on l'ajoute en tête de l'aplatissement du `cdr`.

```
(define (aplatir L)
  (cond ((null? L) '())
        ((list? (car L))(append (aplatir (car L))(aplatir (cdr L))))
        (else (cons (car L)(aplatir (cdr L))))))
```

Somme de tous les nombres apparaissant dans une liste

La discussion comporte quatre cas :

- si la liste est vide, par convention le résultat est 0.

  Ensuite on discute selon la nature du premier élément de la liste :
- si le `car` est un nombre, on l'ajoute au résultat de la somme du reste de la liste,
- si le `car` est une liste non vide, c'est ici la nouveauté, on «s'enfonce» dans le `car`. Plus précisément, il faut sommer tous les nombres du `car` et ajouter le résultat à la somme des nombres du `cdr` ,

- sinon c'est une liste vide ou un objet qui n'est pas un nombre, on peut donc se contenter de faire la somme des nombres du `cdr`.

```
(define (sommeListe* L)
  (cond ((null? L) 0)
        ((number? (car L))(+ (car L)(sommeListe* (cdr L))))
        ((pair? (car L))(+ (sommeListe* (car L))(sommeListe* (cdr L))))
        (else  (sommeListe* (cdr L)))))

? (sommeListe*  '(a (4) 5 ((6 b)) 8)) -> 23
```

Remarque 5 On désigne par le même nom suivit de * l'extension aux listes quelconques de fonctions déjà vues pour les listes plates.

Suppression d'un élément à tous les niveaux dans une liste

Etant donné une expression `s` et une liste `L`, on demande de rendre la liste `L` dans laquelle on a supprimé toutes les occurrences de la valeur de `s`.
Le principe est pratiquement le même que pour `sommeListe*`, voici la fonction :

```
(define (remove* s L)
  (cond ((null? L) '())
        ((equal? s (car L))(remove* s (cdr L)))
        ((list? (car L))
         (cons (remove* s (car  L))
               (remove* s (cdr L))))
        (else (cons (car L)
                    (remove* s (cdr L))))))


? (remove* 'a  '( b a ((a) b) c (a) d)) -> (b (() b) c () d)
```

La fonction `renverse*`

Renversons l'ordre des éléments à tous les niveaux d'une liste :

```
? (renverse* '(a (b c d) e) -> (e (d c b) a)
```

On a aussi renversé les éléments de la sous-liste `(b c d)`.
Il y a trois cas :

- le cas de base : si la liste est vide, on retourne `()`,
- si le premier élément est une liste, on renverse le reste et l'on ajoute à la fin le premier élément renversé,
- sinon, on renverse le reste et l'on ajoute à la fin le premier élément.

```
(define (renverse* L)
  (cond ((null? L) '())
        ((list? (car L))
         (append1 (renverse* (cdr L)) (renverse* (car L))))
        (else (append1 (renverse* (cdr L))(car L)))))
```

Exercice 10 *1. Calculer la profondeur d'une liste.*

```
? (profondeur '(a (b (c d)) e)) -> 3
```

2. Calculer le nombre d'occurrences d'un symbole donné dans une liste.

```
? (nb-occurrence* 'a '(e a c a (b c a) ((a a)) d)) -> 5
```

3. Donner votre version de la fonction **equal?** *en fonction de* **eq?**. *On se contentera de considérer les listes construites à partir de nombres ou de symboles.*

2.5 Aspects méthodologiques et preuve de fonctions récursives

Après cette cure de programmation récursive, il est bon de faire une pause pour se poser des questions sur la méthodologie de la programmation. L'idéal du programmeur, c'est d'écrire des programmes *corrects, efficaces et lisibles.* Quand il y parvient, il en éprouve une satisfaction esthétique; c'est cet aspect de la programmation qui conduit souvent à parler de «l'art de la programmation». Pour des logiciels commerciaux, la *robustesse* est aussi une qualité majeure: les mauvaises utilisations du programme doivent déclencher un message d'erreur pertinent. Mais, dans ce livre à but pédagogique, la robustesse n'est pas recherchée car les traitements d'erreurs tendent à obscurcir le code.
La correction est la moindre des choses, bien que chacun connaisse le cycle infini des versions successives d'un logiciel. On corrige un certain nombre de «bogues» (un euphémisme pour désigner une erreur) et on en introduit de nouvelles! Nous allons indiquer quelques outils pour s'assurer de la correction des fonctions définies par l'utilisateur.

Lisibilité et documentation

La lisibilité facilite évidemment la correction et est indispensable pour permettre à l'auteur, ou a fortiori à une autre personne, de modifier plus tard certaines fonctionnalités du programme. L'indentation, les commentaires et le choix des identificateurs concourent à la lisibilité. L'utilisation des commentaires doit suivre certaines règles pour ne pas tomber dans des excès. Au moment de la définition d'une fonction, on conseille de donner en commentaire une documentation concernant:

- les types des paramètres formels et celui de la valeur rendue par la fonction. Pour cela, il est commode de choisir des identificateurs qui évoquent non seulement leurs éventuelles significations mais aussi leurs types. On a déjà conseillé les conventions suivantes: `s` pour désigner une S-expression quelconque, `nb` pour une valeur numérique, on préfixe par `L` le nom d'une valeur de type liste, par `LL` le nom d'une liste de listes, ...

- la spécification, c'est-à-dire une description de ce que doit calculer la fonction (mais sans indiquer ici la manière de le faire). On n'insistera jamais assez

sur le fait qu'il est indispensable de savoir précisément ce que l'on cherche à calculer si l'on veut en donner une programmation récursive. Dans le corps de la fonction on pourra aussi placer des commentaires aux points stratégiques.

Ces conseils concernent l'écriture des programmes, mais on ne peut les suivre dans un livre. En effet, les commentaires feraient double emploi avec les explications qui accompagnent les fonctions. Cependant, pour les fonctions dont la programmation s'étend sur plusieurs pages, on fournira en plus une vue structurée du programme en essayant de mettre en évidence la hiérarchie des fonctions auxiliaires.

On adaptera cette méthodologie avec souplesse selon que la fonction est triviale ou met en œuvre un algorithme complexe. De toute façon, une fonction Scheme devrait rarement dépasser une vingtaine de lignes, l'art du programmeur Scheme c'est aussi de savoir décomposer un problème en sous-problèmes simples.

Efficacité

L'efficacité dépend en premier lieu des algorithmes utilisés, mais il y a aussi des possibilités au niveau de l'utilisation du langage, on en décrira certaines dans les prochains chapitres.

Correction

Reste le problème crucial de la correction. On sait que la satisfaction de quelques tests est une condition nécessaire mais non suffisante de la validité d'une fonction. La plupart des environnements pour Scheme fournissent au programmeur des outils pour analyser l'évaluation des expressions ; on a vu la trace mais il y en a d'autres : pas à pas, débogueur ...
La trace permet souvent de mieux comprendre le processus d'évaluation, mais ne résout pas le problème de la preuve de correction d'une fonction. Commençons par un exemple, revenons à la fonction `sommeCarres` du chapitre 1 §7.

;; étant donné un entier positif n, on retourne la somme $1 + 2^2 + \cdots + n^2$
;; ou 0 si $n = 0$.

```
(define (sommeCarres n)
  (if (zero? n)
    0
    (+ (sommeCarres (- n 1))
       (* n n))))
```

Il s'agit de prouver que cette fonction vérifie la *spécification* :

$$(sommeCarres\ n) = 1 + 2^2 + ... + n^2 \quad \textit{pour tout entier} \quad n \geq 0 \qquad (*)$$

où, par convention, cette somme vaut 0 quand $n = 0$.

La preuve de cette spécification se décompose en deux questions : terminaison de la fonction et correction ?

Terminaison

Il faut commencer par s'assurer que la fonction `sommeCarres` est bien définie pour tous les entiers $n \geq 0$. La méthode de preuve est très simple et a déjà été suggérée. Elle consiste à définir sur le domaine de l'argument, ici les entiers positifs, une mesure à valeur entière positive, telle que sa valeur sur les arguments de chaque appel récursif soit strictement plus petite que sur les arguments de la fonction.
Dans cet exemple, une mesure s'impose, c'est l'entier n lui-même. A l'exécution, les appels successifs de la fonction `sommeCarres` porteront sur des entiers de plus en plus petits et ne peuvent donc pas se répéter indéfiniment. On finit par aboutir au cas de l'entier 0 pour lequel on retourne une valeur sans appel récursif.
Notons au passage que dans une définition récursive, un appel récursif doit toujours être situé dans une fonction de contrôle et qu'il doit être prévu au moins un cas où la valeur de la fonction est calculée sans appel récursif.
Dans certains cas plus complexes, on peut être amené à considérer une grandeur strictement décroissante qui ne soit pas à valeur entière mais soit à valeur dans un ensemble ordonné. L'argument sera tout aussi valable pourvu que, pour l'ordre considéré, toute suite strictement décroissante soit finie. Un tel ordre est dit *bien fondé.*

Exercice 11 *Que pensez-vous de cette définition du quotient entier de l'entier $a \geq 0$ par l'entier $b > 0$?*

```
(define (division a b)
  (if (= b 1)
    a
    (+ 1 (division (- a b) b)))) ?
```

Exercice 12 *Calcul des coefficients binomiaux. On note par $\binom{n}{k}$ le nombre de façons de choisir k objets parmi n avec $0 \leq k \leq n$.*
On démontre que $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ et en particulier : $\binom{n}{0} = \binom{n}{n} = 1$.
On peut aussi établir la relation de récurrence :

$$\binom{n+1}{k} = \binom{n}{k} + \binom{n}{k-1} \quad \textit{pour} \quad 0 \leq k \leq n$$

Utiliser cette relation pour définir la fonction `(CoeffBinome n k)` qui étant donnés des entiers $0 \leq k \leq n$, retourne la valeur de $\binom{n}{k}$. On fera attention à bien choisir le (ou les) cas de base.

Preuve de correction

Après s'être assuré de la terminaison, il reste à prouver la correction, c'est-à-dire l'égalité (*). Pour cela, il est naturel de chercher à faire un raisonnement par récurrence sur l'entier n.

- Pour $n = 0$, la fonction `sommeCarres` rend la valeur 0, ce qui est bien conforme à la convention.

- On suppose que l'égalité (*) est vérifiée pour les valeurs de n strictement inférieures à un entier k et on va prouver qu'elle l'est encore pour $n = k$.

  Par définition de la fonction `sommeCarres`, on a pour $k > 0$:

$$(sommeCarres\ k) = (+\ (sommeCarres\ (-\ k\ 1))\ (*\ k\ k))$$

  Mais l'hypothèse de récurrence nous assure que :

$$(sommeCarres\ (-\ k\ 1)) = 1 + 2^2 + \cdots + (k-1)^2$$

  par conséquent, on trouve en remplaçant $(sommeCarres\ (-\ k\ 1))$ par cette valeur que :

$$(sommeCarres\ k) = 1 + 2^2 + \cdots + (k-1)^2 + k^2$$

Exercice 13 *Faire la preuve de votre fonction* `CoeffBinome`.

Schéma général d'une preuve de fonction

Décrivons en général la marche à suivre pour prouver qu'une fonction `(f x1 ...xN)` définie par un programme Scheme vérifie une *spécification* de la forme :

$$\textit{pour tout} \quad (x_1, \ldots, x_N) \in D \quad \textit{on a} \quad R(x_1, \ldots, x_N,\ f(x_1, \ldots, x_N)) \qquad (**)$$

où les x_j sont les paramètres de la fonction f, D est son domaine de définition et R est la relation à prouver entre la valeur de $f(x_1, \ldots, x_N)$ et celle de ses paramètres $(x_1, \ldots, x_N)$.

On commence par prouver que la fonction f est bien définie sur le domaine D. Pour cela, il faut s'assurer que les fonctions utilisées pour exprimer la valeur de f se composent correctement pour tout $(x_1, \ldots, x_N) \in D$ De plus, s'il y a appel récursif de f, on doit prouver la terminaison en exhibant une mesure $m(x1, \ldots, xN)$ (le plus souvent à valeur entière positive) sur l'ensemble D. Cette mesure doit avoir une valeur strictement plus petite sur les arguments de chaque appel récursif.
Ensuite, il s'agit de prouver que l'on a bien (**), c'est-à-dire que la relation R est satisfaite sur tout le domaine. Pour cela, on combine les spécifications des fonctions qui servent à définir f. Si l'on a un appel récursif de f, on utilise un raisonnement par récurrence. En général, on raisonne par récurrence sur la valeur de la mesure $m(x_1, \cdots, x_N)$ qui a servi à prouver la terminaison ; c'est pour cette raison que l'on commence toujours par prouver la terminaison. Donnons un autre exemple avec une fonction sur les listes.

Preuve de remove

Considérons la preuve de la fonction `remove`. On rappelle qu'elle rend une liste où l'on a supprimé toutes les occurrences (au premier niveau) de la valeur d'une expression `s` dans une liste `L`.

```
(define (remove s L)
  (cond ((null? L) '())
        ((equal? s (car L))(remove s (cdr L)))
        (else (cons (car L)(remove s (cdr L))))))
```

La preuve de la terminaison est immédiate en raisonnant sur la longueur de `L` puisque les appels récursifs ne concernent que le `cdr` de `L`.
Pour la correction, on prouve par récurrence sur la longueur de `L` que la fonction `remove` fait bien son travail. Pour la longueur 0, c'est-à-dire la liste vide, il n'y a plus rien à retirer de la liste vide.
On suppose que la fonction `remove` est correcte pour les listes de longueur $\leq n$ et on le prouve pour les listes de longueur $n + 1$. Il y a deux cas de figure :

- ou bien le premier élément de `L` est égal à `s`, auquel cas il faut le supprimer, et il faut aussi supprimer `s` du reste de la liste. Mais, par hypothèse de récurrence, on sait que `(remove s (cdr L))` fait ce qu'il faut.
- ou bien le premier élément de `L` n'est pas égal à `s`, auquel cas il faut le conserver et retirer les occurrences de `s` dans le reste de `L`. Comme, par hypothèse de récurrence, on sait que `(remove s (cdr L))` fait ce qu'il faut, on doit rendre dans ce cas `(cons (car L) (remove s (cdr L)))`

Exercice 14 *Donner les preuves des fonctions* `renverse`, `renverse*` *et* `remove*`.

Lorsque le code d'une fonction n'est pas absolument évident, on conseille au programmeur d'essayer de s'en donner une preuve détaillée. La recherche de cette preuve conduit souvent à suggérer une amélioration du code. Bien entendu, il faut être réaliste : la preuve d'un gros programme a toutes les chances d'être très compliquée et donc de comporter à son tour des lacunes et des erreurs. Dans ce cas, on a besoin d'outils logiciels et logiques pour assister le programmeur ; ces outils sont pour le moment du domaine de la recherche. De nombreux travaux ont été entrepris pour essayer de construire des systèmes de démonstration automatique. Une autre voie de recherche plus récente consiste à dire qu'il ne faut pas programmer mais donner une démonstration logique de l'existence d'un objet satisfaisant à la spécification. C'est le système qui générera automatiquement un programme à partir de la preuve. L'approche est séduisante, l'avenir dira si elle viable. Comme souvent en informatique, ce sont des sous-produits de recherches fondamentales qui ont parfois plus d'applications que le but primitif.

Equipé de ces outils méthodologiques, on peut reprendre les délices de la programmation récursive[2] !

[2] On peut utiliser le même genre de raisonnement pour prouver des propriétés vérifiées par une fonction (voir un exemple à la fin du chapitre 17).

Exercice 15 *La fonction, dite 91, de McCarthy est définie par :*

```
(define (f91 x)
  (if (<= x 100)
    (f91 (f91 (+ x 11)))
    (- x 10)))
```

Démontrer sa terminaison (distinguer les cas $x \geq 100$ *et* $x < 100$*) et prouver que cette fonction vaut :* $x - 10$ *si* $x > 100$ *et 91 si* $x < 100$.

2.6 Evaluation des expressions Scheme

On est maintenant en mesure de donner une définition plus précise (mais encore incomplète) du mécanisme d'évaluation utilisé par Scheme. L'opération essentielle dans la boucle d'interaction de Scheme est l'évaluation d'une expression. Voici le principe général du calcul de la valeur d'une S-expression.

Valeur d'un atome

- si c'est un *nombre*, il est sa propre valeur :

  ```
  ?  -3.14 -> -3.14
  ```

- si c'est un *booléen*, il est sa propre valeur,
- si c'est une *chaîne*, elle est aussi sa propre valeur,
- si c'est une *variable*, sa valeur sera la donnée qu'elle représente ou une erreur si elle ne désigne rien.

Valeur d'une liste non vide

Une liste correspond toujours à un appel de fonction ou à une forme spéciale Scheme.
La fonction est indiquée par le premier élément de la liste et les arguments sont les éléments suivants, ainsi la liste `(f e1 ...eN)` correspond à ce que l'on note $f(e1, \ldots, eN)$ en mathématiques. Il y a évaluation de tous les arguments `e1` ... `eN`, puis on évalue le corps de la fonction sachant que les paramètres formels ont pour valeurs celles des arguments `ej`. Ce mécanisme de passage de paramètres est dit par *valeurs*, on dit aussi parfois que l'on utilise l'ordre applicatif.

Dans le cas des *formes spéciales* comme `define, if, quote, cond, let, let*, case` ... on dispose d'un mécanisme particulier d'évaluation pour chaque forme.

Cas de la liste vide

La liste vide n'est pas une S-expression admissible de la syntaxe Scheme, il n'y a donc pas lieu de demander sa valeur. En revanche, c'est une valeur possible :

```
? ()
*** ERREOR -- Ill-formed expression () ,

? (cdr '(a))
()
```

Valeur «indéfinie»

Les formes qui font un effet de bord, comme define, display, ... rendent une valeur *indéfinie*, car seul l'effet de bord est à considérer. On a choisi de ne rien afficher dans ce cas, car la valeur est non spécifiée et dépend donc de l'implantation utilisée.

La fonction eval

Dans beaucoup d'implantations on a accès à la fonction d'évaluation, elle s'appelle eval. Comme toute fonction Scheme, elle évalue d'abord ses arguments, par conséquent l'appel (eval s) provoque une double évaluation de s :

```
? (eval ''a) -> a
? (eval (cons '+ '( 2 3))) -> 5
```

Mais son inclusion dans la norme Scheme fait encore l'objet de discussions[3], certains souhaitent une fonction eval prenant, en deuxième argument, l'environnement à utiliser pour faire cette évaluation.

2.7 Les doublets

Définition d'un doublet

La structure de liste non vide est un cas particulier d'une structure plus générale appelée *doublet*. Un doublet[4] c'est simplement un couple de valeurs de S-expressions. C'est donc le moyen le plus simple pour créer un objet composé. Le constructeur de doublet est encore désigné par cons. L'affichage d'un doublet se fait au moyen d'une paire pointée. Le point sépare les deux valeurs stockées dans le doublet et il faut au moins un espace entre le point et les expressions qui l'encadrent :

```
? (cons  s1 s2)
(s̄1 . s̄2)
```

Voici un doublet constitué d'un symbole et d'un nombre

```
? (cons 'an  1995)
(an . 1995)
```

[3] Au moment où l'on écrit ces lignes, c'est-à-dire en 1995.

[4] On utilise aussi les termes paire pointée et paire tout court.

Les accesseurs aux composantes d'un doublet sont encore les fonctions **car** et **cdr** qui jouent maintenant un rôle symétrique : la fonction **car** rend la première composante d'un doublet et la fonction **cdr** rend la seconde. On a la relation générale :

(car (cons x y)) = $\bar{x}$ (cdr (cons x y)) = $\bar{y}$

Les composantes d'un doublet peuvent être des doublets :

```
? (cons  '(a . b)  'c) -> ((a . b) . c)
```

Autrement dit, on étend la définition des S-expressions aux doublets :

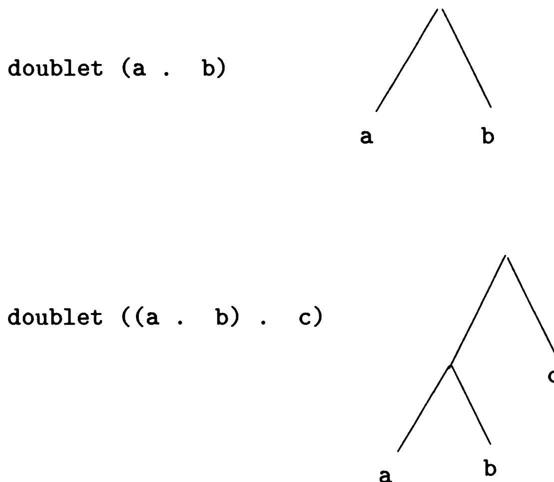
```
S-expression -->  (S-expression . S-expression )
```

Le prédicat **pair?** teste en fait si une S-expression est un doublet :

```
(pair? '())       -> #f  ;; la liste vide n'est pas un doublet
(pair? '(a b))    -> #t  ;; on va voir qu'une liste non vide est un doublet
(pair? '(a . b)) -> #t
```

Remarque 6 Un programme robuste devrait toujours s'assurer que l'on a affaire à un doublet avant d'en prendre le **car** ou le **cdr**.

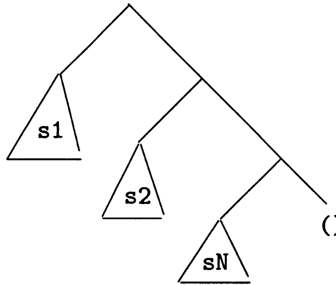
On peut donner une visualisation d'un doublet sous forme d'une arborescence :



Pour ceux qui connaissent déjà la notion d'arbre (voir chapitre 7), ceci est une représentation des doublets par des arbres binaires dont les feuilles sont étiquetées par des atomes.

Doublets et listes

En réalité, une liste non vide est un *cas particulier de doublet*: la liste `(a)` est en fait le doublet `(a . ())` ; en effet ces deux objets ont même `car` et même `cdr`. Plus généralement, la liste `(s1 s2 ... sN)` est en fait représentée par le doublet `(s1 . (s2 . (...(sN . () )) ...)`.
Les listes correspondent aux arbres en forme de «râteau»



De façon plus précise, une liste est soit la liste vide, soit un doublet dont le `cdr` est une liste.

Exercice 16 *Définir le prédicat* `list?` *à partir des prédicats* `null?` *et* `pair?`.

Pour simplifier les affichages, chaque fois qu'une paire est en fait une liste, l'imprimeur Scheme l'affiche comme une liste, sinon il utilise un affichage hybride qui minimise l'utilisation du point :

```
?  '(a . (1 2))  -> (a 1 2)

?  '(a . (b . (c . d))) -> (a b c . d)
```

Exercice 17 *Comparer* `(() . ())` *et* `(())`. *Donner une représentation graphique des doublets suivants* `((a . b) . c)`, `(((a)) . ((b)))`.

2.8 Les a-listes

Définition des a-listes

Les doublets servent aussi à définir les listes d'association ou *a-listes*, ce sont des listes de doublets :

```
( (c1 . v1) ... (cn . vn) )
```

Elles sont souvent utilisées pour représenter des données comme les tables, les dictionnaires, les fonctions,... Si on veut associer, à une clé `cj`, une valeur `vj`, on place la paire `(cj . vj)` dans une telle liste. Pour récupérer la valeur associée à une clé, on dispose de la fonction prédéfinie `assq` de syntaxe :

(assq *s* ***aliste***)

Elle retourne le *premier doublet* de la a-liste ***aliste*** dont la clé est eq? à s ; s'il n'y en a pas, la valeur rendue est #f.

```
? (assq 'a '((b . 2)(a . 1)(c . 3)(a . 0))) -> (a . 1)
? (assq 'a '((b  2)(a  1)(c  3)(a  0))) -> (a 1)
     ;; car une liste est un cas particulier de paire
? (assq '(a) '((b . 2)(a . 1)(c . 3)((a) . 0))) -> #f
```

Remarque 7 On a aussi les fonctions analogues assv et assoc qui utilisent respectivement les tests eqv? et equal?.

```
? (assoc '(a) '((b . 2)(a . 1)(c . 3)((a) . 0))) -> ((a) . 0)
```

Opérations sur les a-listes

Comme la fonction assq rend un doublet, écrivons une fonction valeur-de pour calculer la valeur associée à une clé :

```
(define (valeur-cle cle aliste)
  (let ((doublet (assq cle aliste)))
    (if doublet
      (cdr doublet)
      #f)))

? (valeur-cle 'Rameau
    '((Gesualdo . 1560)(Monteverdi . 1567) (Rameau . 1683)(Bach . 1685)))
1683

? (valeur-cle 'Dufay
    '((Gesualdo . 1560)(Monteverdi . 1567) (Rameau . 1683)(Bach . 1685)))
#f
```

Ecrivons une fonction supprime-cle pour supprimer d'une a-liste toutes les occurrences des doublets de clé donnée :

```
(define (supprime-cle cle aliste)
  (cond ((null? aliste) '())
        ((eq? cle (caar aliste))(supprime-cle cle (cdr aliste)) )
        (else (cons (car aliste)(supprime-cle cle (cdr aliste))))))

? (supprime-cle 'b '((a . 1)(b . 2) (c . 3)(b . 4)))
((a . 1)(c . 3))

? (supprime-cle 'd '((a . 1)(b . 2) (c . 3)(b . 4)))
((a . 1)(b . 2) (c . 3)(b . 4))
```

On peut associer à une a-liste une fonction sublis de substitution qui remplace toutes les occurrences des clés dans une expression par les valeurs correspondantes.

```
(define (sublis aliste s)
  (cond ((pair? s)(cons (sublis aliste (car s))(sublis aliste (cdr s))))
        ((null? s) '())
        (else (let ((paire (assoc s aliste)))
                (if paire
                  (cdr paire)
                  s)))))

? (sublis '((un . one) (deux . two) (trois . three) ) '(un + deux = trois ))
(one + two = three)
```

Pour augmenter une a-liste, il est commode de disposer d'une fonction (acons *s1 s2 aliste*) qui rend une liste obtenue en ajoutant en tête d'une a-liste le doublet construit à partir des expressions *s1* et *s2* :

```
(define (acons s1 s2 aliste)
  (cons (cons s1 s2) aliste))
```

Exercice 18 *1. Définir une fonction (*pairlis L1 L2 aliste*) qui construit une a-liste en ajoutant en tête de la a-liste* aliste *la liste des paires obtenues en appariant les éléments successifs de la liste*L1 *avec les éléments successifs de la liste* L2 *(on suppose ces listes de même longueur).*

```
? (pairlis '(un deux) '(one two) '((trois . three)(quatre . four)))
((un . one) (deux . two) (trois . three) (quatre . four))
```

2. Ecrire une fonction change-valeur *qui change la valeur associée à une clé en une valeur donnée ou bien ajoute un nouveau doublet si la clé n'existe pas.*

2.9 Représentation externe/interne des données

Pour programmer un algorithme, une étape décisive est le choix de la représentation des données du problème à l'aide des structures disponibles dans le langage de programmation. Avec la structure de liste, et plus généralement de doublet, Scheme fournit un moyen très général et très souple pour modéliser simplement une grande variété de données. On a déjà vu comment on peut représenter les expressions arithmétiques ou bien un agenda avec des S-expressions, on verra de nombreux autres exemples dans ce livre.

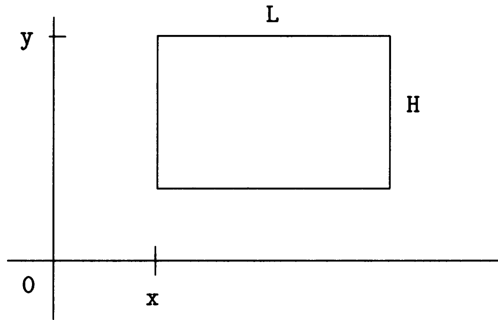
Rectangles du plan

A titre d'exemple, modélisons les rectangles du plan $x0y$ ayant leurs côtés parallèles aux axes. Il y a de nombreuses façons de se donner un tel rectangle :

- par les coordonnées de deux sommets opposés,
- par les coordonnées du centre, et les longueurs de chaque côté,

- par les coordonnées du point le plus haut à gauche et les longueurs de chaque côté.

Choisissons la dernière méthode : notons x, y les coordonnées du sommet, L la longueur du côté parallèle à $0x$ et H la longueur du côté parallèle à $0y$.



Il reste à définir une fonction qui à un rectangle associe de façon bijective un objet Scheme, appelé *représentation interne* du rectangle. Il y a évidemment de nombreux choix naturels :

- la liste de ces nombres (x y L H),
- le doublet ((x y) . (L H)),
- le doublet ((x . y) . (L . H)) ...

Pour ne pas préjuger du choix retenu, on appelle `cons-rectangle` la fonction (appelée *constructeur*) qui construit la représentation interne d'un rectangle dont on fournit les grandeurs x, y, L, H. Pour manipuler les rectangles donnés par leurs représentations internes, on a besoin d'accéder aux valeurs x, y, L, H. Aussi, on se donne les quatre fonctions suivantes (appelées *accesseurs*) qui associent ces valeurs à la représentation interne : `valeur-x, valeur-y, valeur-L, valeur-H`.

Opérations sur ces rectangles

Ces fonctions permettent de définir des opérations sur les rectangles, de façon indépendante de la représentation interne adoptée.
Définissons le prédicat `dans?` qui teste si un point `x0 , y0` se trouve à l'intérieur (ou sur le bord) d'un rectangle `r`. Le point `x , y` doit avoir ses coordonnées supérieures au sommet haut gauche du rectangle et inférieures à celles de son sommet bas droit :

```
(define (dans?  x0 y0 r)
  (let ((x1 (valeur-x r))
        (y1 (valeur-y r))
        (L  (valeur-L r))
        (H  (valeur-H r)))
    (and (<= x1 x0)(<= y1 y0)(<= x0 (+ x1 L))(<= y0 (+ y1 H)))))
```

Généralisons au cas de l'inclusion d'un rectangle dans un autre avec le prédicat (inclus? r1 r2). Il rend #t si le rectangle r1 est entièrement inclus dans le rectangle r2. Pour cela, il faut et il suffit que r2 contienne deux sommets opposés de r1 :

```
(define (inclus? r1 r2)
  (let ((x1 (valeur-x r1))
        (y1 (valeur-y r1))
        (L1  (valeur-L r1))
        (H1  (valeur-H r1)))
    (and (dans? x1 y1 r2)(dans?  (+ x1 L1) (+ y1 H1) r2))))
```

Exercice 19 *Définir le prédicat* **disjoint?** *qui vaut #t si deux rectangles n'ont pas de point commun.*

Pour translater un rectangle r selon un vecteur de composantes a , b, il suffit de translater son sommet haut gauche, d'où la fonction :

```
(define (translate  r a b)
  (let ((x (valeur-x r))
        (y (valeur-y r))
        (L  (valeur-L r))
        (H  (valeur-H r)))
    (cons-rectangle (+ x a) (+ y b) >L H)))
```

Exercice 20 *Il serait aussi utile définir une fonction de dessin d'un rectangle, mais cela nécessite des primitives graphiques. Ecrire à la place une fonction* **affiche-rectangle** *qui affiche les coordonnées x , y du sommet et les dimensions L, H d'un rectangle.*

Pour pouvoir utiliser ces fonctions, il faut faire un choix de représentation interne : prenons simplement la liste des nombres x y L H. D'où le constructeur et les quatre accesseurs :

```
(define cons-rectangle list)
(define valeur-x car)
(define valeur-y cadr)
(define valeur-L caddr)
(define valeur-H cadddr)
```

On peut alors tester nos fonctions :

```
? (define r1 (cons-rectangle 5 10 40 30))
? (define r2 (cons-rectangle  12  20  20 15))
? (inclus?  r1 r2) -> #f
```

Si l'on décide plus tard de changer de représentation, il suffira simplement de changer les cinq fonctions : cons-rectangle, valeur-x, valeur-y, valeur-L, valeur-H. Par exemple, si l'on choisit de représenter un rectangle x , y , L , H par le doublet ((x . y) . (L . H)), alors donner les nouvelles expressions de ces fonctions.
Mais il n'y rien à changer à nos fonctions dans?, inclus?, translater? ...

2.10 Représentation interne des listes et notions d'égalité

Les doublets[5] sont eux-mêmes des données Scheme qui possèdent une certaine représentation interne dans la mémoire de la machine, mais, jusqu'à maintenant, on ignorait tout de cette représentation. Il nous suffisait de savoir qu'il y avait un constructeur `cons`, des accesseurs `car`, `cdr`, un prédicat `pair?` et une fonction d'affichage sous forme de paires pointées, avec les relations :

```
(car (cons x y)) -> x̄                    (cdr (cons x y)) -> ȳ
(pair?  (cons x y)) -> #t
(write (cons x y)) -> (x̄ .  ȳ)
```

On dit que ces relations définissent le *type abstrait* doublet. On développera la notion de type abstrait au chapitre 7.
Pour satisfaire la légitime curiosité du lecteur, on va donner un aperçu (simplifié) de la représentation en mémoire de certaines valeurs Scheme, ce point sera complété aux chapitres 4, 5 et 22.

Toute valeur Scheme doit pouvoir être stockée dans la mémoire de l'ordinateur. La *mémoire* est constituée d'une suite de cases numérotées, le numéro d'une case s'appelle son *adresse*.
Détaillons ce point en se limitant aux valeurs suivantes : petits entiers, symboles, liste vide et doublets définis à partir de ces valeurs.

Variable à valeur atomique

Quand on donne une valeur à une variable, on définit une *liaison* entre la variable et sa valeur par l'intermédiaire d'une case mémoire. L'ensemble des liaisons utilisables en un point s'appelle l'*environnement* courant. Par exemple, quand on évalue la définition :

```
? (define a  10)
```

la *variable* de nom l'identificateur `a` désigne une adresse mémoire où est rangée sa valeur 10. Dorénavant la variable `a` est connue de Scheme, et on peut l'évaluer.

```
? a
10
```

L'évaluation consiste à consulter l'environnement pour lire la valeur rangée à cette adresse.

Considérons maintenant le cas d'une valeur symbolique

```
? (define  reponse  'oui)
```

C'est le même mécanisme mais la variable `reponse` a pour valeur le symbole `oui`. En gros, un symbole est représenté en mémoire par la suite de ses caractères, sa valeur est l'adresse de son premier caractère. La variable `reponse` sera donc liée à l'adresse du symbole `oui`. Ces notions seront détaillées au chapitre 5 §2 et 22 §7.

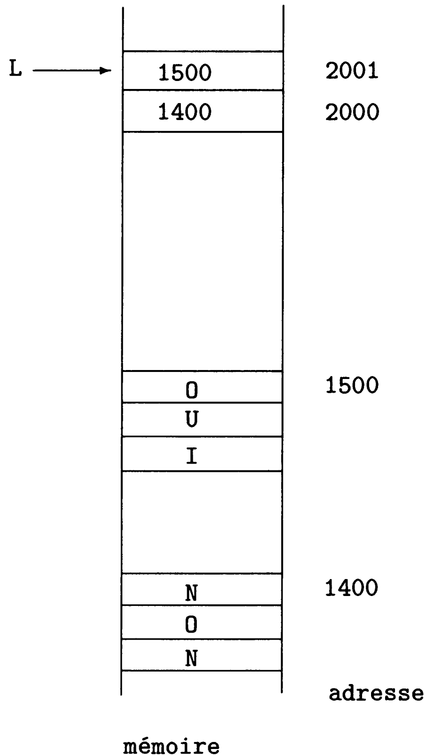
[5] On peut sauter ce paragraphe en première lecture.

Variable à valeur doublet

Considérons enfin le cas d'une valeur doublet, c'est-à-dire d'un couple de valeurs. On peut utiliser deux cases mémoires consécutives, l'une pour la valeur du `car` et l'autre (d'adresse 1 de moins) pour la valeur du `cdr`. Avec cette convention, la valeur doublet sera l'adresse de la case de son `car`. Par exemple, posons :

```
? (define  L (cons  'oui  'non))
```

Supposons que le symbole `oui` soit rangé à l'adresse 1500 et le symbole `non` à l'adresse 1400, et que l'on range le doublet `L` à l'adresse 2001, alors on aura en mémoire la disposition suivante :

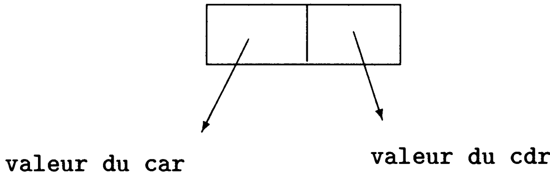


Bien entendu, ceci est très schématique, car selon la taille des cases mémoires on peut être conduit à utiliser une seule case pour ranger les deux composantes d'un doublet. Par ailleurs, les cases ne comportent que des 0 et des 1, on ne pourra pas savoir si une case contient un nombre, une adresses ou bien le code d'un caractère, ... Pour lever cette ambiguïté, une méthode consiste à réserver dans chaque case une petite zone pour indiquer la nature de la valeur qu'elle contient : nombre, symbole, doublet,... Une autre méthode consiste à spécialiser des zones mémoires pour le stockage de chaque type de valeurs. Pour plus de détails voir le chapitre 22 §7.

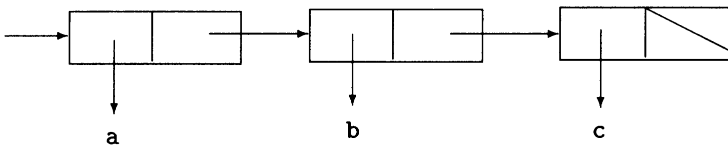
Diagrammes de boîtes

La connaissance de ces détails d'implantation n'est pas indispensable, mais elle éclaire le lecteur sur la nature des opérations qui sont faites quand on calcule le `car` ou le `cdr` d'une liste : on se contente de suivre une adresse.

Pour ne garder que la quintessence de cette implantation, il est d'usage d'utiliser une représentation des doublets par des boîtes avec deux cases : l'une avec une flèche pointant sur la valeur du `car` et l'autre sur le `cdr` :

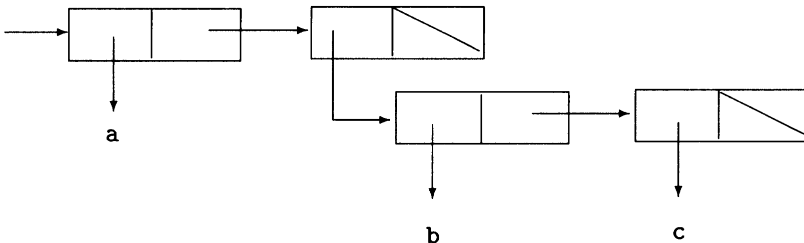


Ainsi la liste `(a b c)`, qui est en réalité le doublet `(a . (b . (c . () ))`, sera représentée par le diagramme :



où la flèche pointant sur la liste vide est symbolisée par une case barrée.

Ce type de diagramme permet de représenter non seulement les listes plates mais aussi les listes générales. Par exemple la liste `(a (b c) )`, c'est-à-dire le doublet `(a . ((b . (c . ())) . ()))`, est représentée par :



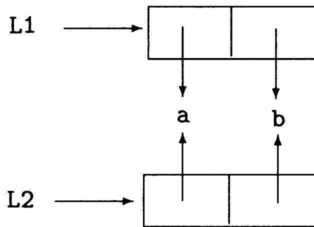
Exercice 21 *Dessiner un diagramme de boîtes pour représenter les expressions :* *((a)), ((a b) (c d)), ((a . 1) (b . 2) (c . 3))*

Les égalités eq?, eqv?, equal?

La connaissance des principes de l'implantation en mémoire des valeurs Scheme permet de préciser les distinctions entre les égalités eq?, eqv?, equal?.
Le test eq? se contente de vérifier que les valeurs des deux arguments ont même adresse, c'est-à-dire qu'ils sont *physiquement égaux*. Ce qui en fait un test très rapide. Il est utile pour les valeurs symboles car un symbole donné n'a qu'une adresse, mais ce n'est pas le test à utiliser pour les listes :

```
? (define  L1 (cons 'a 'b))
? (define  L2 (cons 'a 'b))

? (eq? L1 L2) -> #f
```

En effet, les valeurs de L1 et L2 sont physiquement distinctes car le constructeur de doublets cons, *construit un nouveau doublet à chaque appel*. Autrement dit, il demande[6] une autre case mémoire pour stocker le doublet associé à la valeur de L2. On a le diagramme :



En revanche, le test equal? sert à tester l'égalité structurelle. Il effectue un parcours des structures à comparer en vérifiant qu'elles sont construites de la même façon à partir d'éléments de base identiques. C'est donc un test plus général mais beaucoup moins rapide que eq?.

```
? (equal? L1  L2) -> #t
```

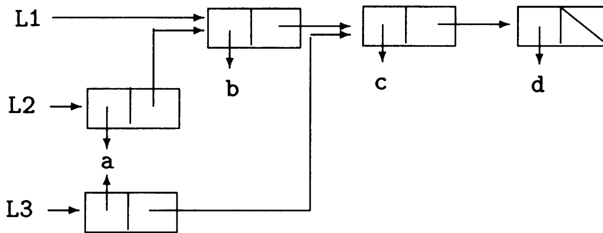
Les combinaisons de car, cdr , cons , list ... conduisent souvent à créer des S-expressions qui *partagent* des doublets. Ainsi, on vérifie qu'après les définitions :

```
(define L1 (list 'b 'c 'd))
(define L2 (cons 'a L1))
(define L3 (cons 'a (cddr L2)))
```

Le test avec eq? détecte le partage entre L1 et L3 qui est aussi évident sur le diagramme :

```
(eq? (cdr L1) (cdr L3)) -> #t
```

[6]Cette demande de case mémoire libre ne peut être satisfaite indéfiniment ; quand il n'y plus de place libre, on met œuvre un récupérateur de cases pouvant être réutilisées sans dommage. Cette technique de récupération de la mémoire sera étudiée au chapitre 22.



L'égalité eqv? a une signification très voisine de celle de eq? mais est légèrement plus générale. Par exemple, deux «grands» entiers mathématiquement égaux peuvent être considérés comme différents par eq?. En effet, la représentation interne des grands entiers est complexe et ils peuvent donc être physiquement distincts. En revanche eqv? les considérera bien comme égaux. Ajoutons que pour les nombres, on utilise de préférence le test =.

2.11 Quasi-citation

L'abréviation de quote par ' n'est pas la seule abréviation utilisée en Scheme [7].

Quasiquote et virgule

Le caractère ` représente la forme spéciale quasiquote :
`s = (quasiquote s)` et il se comporte presque comme quote :

```
? `(a b)  -> (a b)
```

La différence concerne les listes et plus exactement les éléments d'une liste précédés d'une virgule. Ces éléments seront *évalués* :

```
? (define  x  1)
? (define y '(u v))

? `(x  y  ,x    ,y)  -> (x y 1 (u v))
```

Bien entendu, il n'est pas indispensable d'utiliser ` pour obtenir ce résultat, c'est aussi la valeur de :

```
? (list 'x  'y  x  y)  -> (x y 1 (u v))
```

Mais, au lieu d'avoir à quoter ce que l'on n'évalue pas, on fait précder d'une virgule les endroits à évaluer.

Exercice 22 *Deviner la valeur des expressions suivantes :*
`(1 + 2 = ,(+ 1 2))
`(le car de la liste (a b) est ,(car '(a b)))
`(cons ,(+ 2 5) ,(list 'a 'b))

[7] Ce paragraphe peut être sauté, il ne sera vraiment utilisé qu'au chapitre 9.

Supposons que nous voulions rendre la valeur quotée d'une expression `s` -> `'s̄`. Il suffit d'évaluer l'expression `(list 'quote s)`, ce qui s'écrit encore `‘(quote ,s)` et donc tout simplement `‘',s` ! D'où la fonction `kwote` qui réalise ceci :

```
(define (kwote  s)
 ‘',s)

? (kwote  y)  -> '(u v)
```

Quasiquote et le couple virgule arobasque

Pour compléter la panoplie, il y encore le cas où l'on fait précéder un élément des deux caractères `,@` : cet élément sera évalué mais sa valeur doit être une liste et c'est le *contenu* de la liste qui sera mis à sa place :

```
? ‘(x  y  ,x  ,@y)  -> (x y 1 u v)
```

Le caractère arobasque `@` rappelle le a de **append** car on peut obtenir le même résultat avec l'expression :

```
? (append ‘(x y ,x)   y )  -> (x y 1 u v)
```

Ces techniques d'écriture sont souvent commode pour générer des expressions qui représentent le code d'une fonction, on verra des applications au chapitre 9 à propos des macros.

Exercice 23 1. *Deviner la valeur de l'expression suivante :*

```
(let ((L '(1 2 3)))
  ‘((+ ,@L) = ,(+ 1 2 3)))
```

2. *Ecrire une fonction* `make-let` *qui fabrique une expression* `let` *dont on donne les liaisons et la liste des expressions constituant le corps.*

```
? (make-let  '((x 1) (y 2))  '( (display x) (display y) ))
(let ((x 1) (y 2)) (display x) (display y))
```

2.12 Compléments : polynômes

Définition des polynômes

Pour revenir aux sources de Lisp, on va écrire un embryon de système de calcul formel pour les polynômes.
On considère des polynômes d'une variable X à coefficients réels. Un polynôme est une somme finie de monômes. Un monôme est 0 ou une expression symbolique aX^n, où a est appelé le coefficient du monôme et n est un entier positif appelé degré du monôme (quand a est $\neq 0$).
Quand le degré n est nul, on identifie le monôme aX^0 avec la constante a. Le monôme 0 n'a pas de degré, on dit parfois que son degré est $-\infty$.

Par exemple $3X^4+7X^5+-10+-7X^5+20X$ est un polynôme en X. On identifie un monôme avec le polynôme réduit à ce monôme.
Pour le moment, le symbole + joue un rôle de séparateur car on n'a pas encore défini de notion d'addition. Mais intuitivement, ce polynôme est égal au polynôme $3X^4 + -10 + 20X$. Cette multiplicité des représentations est une source de complications. Aussi, pour manipuler les polynômes il est important de pouvoir en donner une représentation unique.

Addition

On donne un sens opératoire au signe + en définissant l'addition des monômes de même degré :

$$aX^n + bX^n = 0 \quad si \quad a+b=0 \quad et \quad (a+b)X^n \quad sinon$$

En additionnant les monômes de même degré d'un polynôme et en les rangeant par ordre de degré décroissant, on obtient une représentation canonique de la forme $a_dX^d + \cdots + a_iX^i$ pour les polynômes. Pour un polynôme non nul, le monôme de plus haut degré s'appelle le monôme dominant, le degré de ce monôme s'appelle le degré du polynôme et son coefficient le coefficient directeur. Notre exemple initial s'écrit $3X^4 + 20X + -10$ et est de degré 4.

On définit ensuite l'addition de deux polynômes :

$$(a_dX^d + \cdots + a_iX^i) + (a_qX^q + \cdots + a_jX^j)$$

comme étant le polynôme obtenu après addition des monômes de même degré et mis sous forme canonique.

Représentations des polynômes

Comment allons-nous représenter les polynômes en Scheme? Il y a beaucoup de possibilités, on distingue deux classes de représentation :

représentation pleine : on place dans une liste tous les coefficients (nuls ou non) en partant du coefficient dominant. On trouve `(3 0 0 20 -10)` pour le polynôme $3X^4 + 20X + -10$,

représentation creuse : on donne la liste des monômes non nuls par degré décroissant,

On présentera une représentation pleine au chapitre 4, on fait ici le choix d'une représentation creuse. Mais on ne précise pas tout de suite la nature exacte de cette représentation. En appliquant la philosophie d'abstraction introduite au §8, on va se donner des constructeurs et des accesseurs et bâtir les opérations sur ces fonctions. Ce n'est qu'à la fin que nous préciserons l'implantation des constructeurs et accesseurs.

Tout polynôme P, non nul, se décompose de façon unique en son monôme dominant et en un polynôme de degré inférieur. On suit ce principe pour définir les

polynômes comme une structure de donnée que l'on baptise `POLYNOME`. On suppose dans toute la suite qu'il s'agit de polynôme en la même variable `X` qu'il sera donc inutile de préciser.

Constructeurs

On représente par l'identificateur `zero-polynome` le polynôme nul. On appelle `cons-polynome` le constructeur de polynômes non nuls ; c'est une fonction qui prend en paramètre un monôme défini par son degré et son coefficient (supposé non nul) et un polynôme de degré inférieur :

`POLYNOME = zero-polynome ou (cons-polynome` *`coeff degre`* `POLYNOME)`
avec *`coeff`* $\neq 0$.

Accesseurs

Les accesseurs `degre-polynome`, `coeff-dominant`, `reste-polynome` doivent vérifier les relations :

```
(degre-polynome   (cons-polynome  coeff degre polynome)) =  degre
(coeff-dominant   (cons-polynome  coeff degre polynome)) =  coeff
(reste-polynome   (cons-polynome  coeff degre polynome)) =  polynome
```

On distingue le polynôme nul des autres par le prédicat `zero-poly?` ; il vérifie :

```
(zero-polynome?  zero-polynome) = #t
(zero-polynome?  (cons-polynome  coeff degre  polynome)) = #f
```

Opérations sur les polynômes

On est alors en mesure de programmer les principales opérations sur les polynômes. La multiplication d'un polynôme par une constante consiste à multiplier le coefficient dominant puis à multiplier le reste du polynôme par cette constante :

```
(define (mul-scalaire-polynome a P)
  (cond ((zero? a) zero-polynome)
        ((zero-polynome? P) zero-polynome)
        (else (let ((d (degre-polynome   P))
                    (c (coeff-dominant   P))
                    (Q (reste-polynome   P)))
                (cons-polynome d (* c a)  (mul-scalaire-polynome a Q))))))
```

L'égalité de deux polynômes consiste à comparer les monômes dominants et les restes des polynômes :

```
(define (equal-polynome? P1 P2)
  (cond ((zero-polynome? P1) (zero-polynome? P2))
        ((zero-polynome? P2) (zero-polynome? P1))
```

```
      (else (let ((d1 (degre-polynome   P1))
                  (d2 (degre-polynome   P2))
                  (c1 (coeff-dominant   P1))
                  (c2 (coeff-dominant   P2))
                  (Q1 (reste-polynome   P1))
                  (Q2 (reste-polynome   P2)))
              (and (= d1 d2)(= c1 c2)(equal-polynome? Q1 Q2))))))
```

L'addition de deux polynômes oblige à distinguer deux cas :

- ils ont même degré : on additionne les monômes dominants en considérant à part le cas où ils s'annihilent,
- ils ont des degrés distincts : on construit un polynôme avec le monôme de plus haut degré et la somme des autres parties.

```
(define (add-polynome P1 P2)
  (cond ((zero-polynome? P1) P2)
        ((zero-polynome? P2) P1)
        (else (let ((d1 (degre-polynome   P1))
                    (d2 (degre-polynome   P2))
                    (c1 (coeff-dominant   P1))
                    (c2 (coeff-dominant   P2))
                    (Q1 (reste-polynome   P1))
                    (Q2 (reste-polynome   P2)))
                (cond ((= d1 d2)
                       (let ((c (+ c1 c2)))
                         (if (zero? c)
                           (add-polynome Q1 Q2)
                           (cons-polynome d1 c (add-polynome Q1 Q2)))))
                      ((< d1 d2)
                       (cons-polynome  d2 c2  (add-polynome P1 Q2)))
                      (else (cons-polynome d1 c1 (add-polynome Q1 P2))))))))
```

Exercice 24 *Définir la soustraction de deux polynômes.*

La multiplication de deux polynômes non nuls est basée sur l'égalité :

$$\begin{aligned}(c_1X^{d_1} + reste_1) * (c_2X^{d_2} + reste_2) &= c_1c_2X^{(d_1+d_2)}\\ &+ (c_1X^{d_1} + reste_1) * reste_2\\ &+ reste_1 * (c_2X^{d_2} + reste_2)\end{aligned}$$

```
(define (mul-polynome P1 P2)
  (cond ((zero-polynome? P1) zero-polynome)
        ((zero-polynome? P2) zero-polynome)
        (else (let ((d1 (degre-polynome   P1))
                    (d2 (degre-polynome   P2))
                    (c1 (coeff-dominant   P1))
                    (c2 (coeff-dominant   P2))
                    (Q1 (reste-polynome   P1))
                    (Q2 (reste-polynome   P2)))
                (if (zero-polynome? Q1)
                  (cons-polynome  (+ d1 d2) (* c1 c2)
```

```
                    (mul-polynome P1 Q2))
      (add-polynome (mul-polynome (cons-polynome d1 c1
                                                     zero-polynome)
                                  P2)
                    (mul-polynome Q1 P2))))))
```

Exercice 25 *Définir la puissance nième d'un polynôme. Définir la dérivée d'un polynôme.*

Exercice 26 *On associe de façon naturelle une valeur à un polynôme en X quand on donne une valeur à l'indéterminée X. Ecrire la fonction* **valeur-polynome** *qui donne la valeur d'un polynôme pour une valeur de X.*

Choix d'une représentation interne

Si l'on veut exécuter ces fonctions, il reste à choisir une représentation des polynômes, c'est-à-dire définir les fonctions Scheme associées aux constructeurs et accesseurs.
On représente un polynôme par la liste de ses monômes non nuls par degrés décroissants, un monôme étant représenté par un doublet `(degre . coefficient)`.
Le polynôme $3X^4 + 20X + -10$ est représenté par la a-liste
`((4 . 3) (1 . 20) (0 . -10))`.

```
(define (cons-polynome degre coeff  poly)
  (cond ((zero? coeff)(display "erreur : le coeff doit etre non nul"))
        ((zero-polynome? poly)(list (cons degre coeff)))
        (else (cons (cons degre coeff) poly))))

(define degre-polynome   caar)
(define coeff-dominant   cdar)

(define (reste-polynome pol)
  (if (null? (cdr pol))
    zero-polynome
    (cdr pol)))
```

La représentation du polynôme nul est totalement arbitraire, prenons par exemple le nombre 0 :

```
(define zero-polynome 0)

(define (zero-polynome? pol)
  (and (number? pol)(zero? pol)))
```

Affichage d'un polynôme

Pour afficher de façon lisible un polynôme, on définit une fonction qui fait passer de la représentation interne à une forme agréable à l'oeil.
Pour représenter un polynôme avec des caractères usuels, on convient d'afficher le monôme aX^n sous la forme `aX^n`.

On appelle `imprime-polynome` la fonction qui affiche un polynôme avec cette convention. Elle consiste à afficher les monômes successifs avec la fonction `affiche-monome`[8]. Si le degré d'un monôme vaut 1, on remplace `aX^1` par `X`.

```
(define (affiche-polynome pol)
  (if (zero-polynome? pol)
    zero-polynome
    (let ((coeff (coeff-dominant Pol))
          (degre (degre-polynome  Pol))
          (reste (reste-polynome pol)))
      (affiche-monome degre  coeff)
      (if (not (zero-polynome? reste))
        (begin (display " + ")
               (affiche-polynome reste))))))

(define (affiche-monome degre coeff)
  (cond ((zero? degre)(display coeff))
        ((= 1 degre) (display coeff)(display "X"))
        (else (display coeff)(display "X")(display "^")(display degre))))
```

On essaye notre système, appelons p1 le polynôme $3X^2 + 20X + -10$.

```
(define p1  (cons-polynome 2 3 (cons-polynome 1 20 (cons-polynome  0 -10 0))

? (affiche-polynome  p1)
3X^2 + 20X + -10

? (affiche-polynome (mul-polynome p1 p1))
9X^4 + 120X^3 + 340X^2 + -400X + 100
```

Le calcul symbolique sur les polynômes est à la base de tout système de calcul formel. Pour réaliser un «vrai» système, on doit également considérer les polynômes à plusieurs variables. On peut les voir comme des polynômes à une variable dont les coefficients sont des polynômes en les autres variables. Puis on passe aux fractions rationnelles comme on passe des entiers aux rationnels. Puis on ajoute des fonctions spéciales ($cons, sin, exp$, ...) avec des règles de simplification pour réduire les résultats des calculs. Ces systèmes posent de nombreux problèmes mathématiques et informatiques. Les problèmes mathématiques concernent surtout la recherche d'algorithmes efficaces (factorisation des polynômes, primitives des fractions rationnelles, ...). Les problèmes informatiques sont aussi très variés et difficiles :
- problème de l'équivalence : reconnaître un même objet mathématique sous diverses représentations,
- paramétrage des structures de données par d'autres structures : le type des coefficients des polynômes est un paramètre de la structure des polynômes ...

Architecture de ce calcul formel sur les polynômes

Voici l'ensemble des fonctions utilisées pour cet embryon de système de calcul formel sur les polynômes. On en donne une liste commentée et indentée pour

[8] On décrira un véritable afficheur de formules mathématiques au chapitre 14 § 4.

visualiser la hiérarchie entre les fonctions principales et les fonctions auxiliaires. On inaugure cette présentation par un cas simple mais on verra des cas où ce n'est pas superflu.

`(mul-scalaire-polynome a P)`
Multiplication du polynôme `P` par la constante `a`.

`(equal-polynome? P1 P2)`
Prédicat pour tester l'égalité de deux polynômes.

`(mul-polynome P1 P2)`
Calcul du produit de deux polynômes.

> `(add-polynome P1 P2)`
> Somme de deux polynômes.
>
> > Les accesseurs :
> > `(degre-polynome polynome)`
> > Degré d'un polynôme non nul.
> >
> > `(coeff-dominant polynome)`
> > Coefficient du monôme de plus haut degré d'un polynôme non nul.
> >
> > `(reste-polynome polynome)`
> > Polynôme non nul dont on a supprimé le terme dominant.
> >
> > `(zero-polynome? polynome )`
> > Pour distinguer le polynôme nul.
> >
> > `zero-polynome`
> > La valeur qui représente le polynôme nul.
> >
> > `(affiche-polynome pol)`
> > Pour afficher un polynôme.
> >
> > > `(affiche-monome degre coeff)`
> > > Pour afficher un monôme.

2.13 De la lecture

L'ancêtre des langages Lisp [MAE+62].
Des livres sur la programmation avec Scheme [Dyb87, SF90, ML95].
Sur la programmation de petits systèmes de calcul formel [Nor92, Jaf].

Chapitre 3

Programmation fonctionnelle

ES valeurs fonctionnelles sont au cœur du langage Scheme et l'appel de fonction est la principale structure de contrôle. On verra en effet comment certaines formes spéciales peuvent en fait se ramener à des appels de fonctions. La fonction est l'une des rares, voire la seule, unité sémantique qui se compose ; c'est cette propriété qui justifie l'écriture d'une fonction complexe par la composition de fonctions auxiliaires plus simples. Dans ce chapitre, on introduit la notion fondamentale de lambda expression, puis on développe la programmation d'ordre supérieur, c'est-à-dire la considération des valeurs fonctionnelles comme des valeurs ordinaires : on peut passer une fonction en paramètre, on peut rendre une fonction comme valeur d'une autre... On continue l'apprentissage de la programmation récursive et l'on termine avec la technique de programmation par retour-arrière.

3.1 Lambda expression

Définition d'une lambda expression

En mathématique, on utilise parfois une notation de la forme $(x, y) \to \sqrt{x+y}$, pour désigner la fonction qui, prenant deux paramètres x et y, calcule la racine carrée de leur somme. Le langage Scheme permet de faire exactement la même chose, c'est la notion de lambda expression. La fonction précédente pourra être désignée en Scheme par l'expression :

```
(lambda (x y) (sqrt (+ x y)))
```

On dit que c'est une lambda[1] expression, car c'est une forme spéciale qui commence par le mot-clé `lambda` qui est suivi de la liste des paramètres puis des expressions formant le corps de la fonction :

(**lambda** *Liste-parametres corps*)

[1] Ce nom bizarre provient du lambda calcul (cf chapitre 20).

La valeur d'une lambda expression est une fonction dite *anonyme*, car il n'y pas d'identificateur pour la désigner. Cette valeur étant une fonction Scheme, on peut l'appliquer à des arguments comme toute autre fonction :

```
? ((lambda (x y) (sqrt (+ x y)))  3  13)
4

? ((lambda (x y L) (cons x (cons y L))) 'a 'b '(c d))
(a b c d)
```

Lambda expressions sans paramètre

Pour définir une fonction constante, on utilise une lambda expression sans paramètre[2], par exemple la fonction suivante ne fait qu'un affichage :

```
(lambda () (display 1)(display " <= ")(display 2))
```

Pour l'appeler, on ne l'applique à aucun argument !

```
? ((lambda () (display 1)(display " <= ")(display 2)))
1 <= 2
```

On note que l'on obtient le même effet avec la forme spéciale `begin` :

```
? (begin  (display 1)(display " <= ")(display 2) )
1 <= 2
```

Exercice 1 *Donner la valeur des expressions :*
`((lambda (x y z) (+ x y z)) 1 (+ 2 5) 3)`
`((lambda (L) ((lambda (x) (append L (list x))) 0)) '(a) )`

Syntaxe principale pour la définition d'une fonction

Si l'on donne un nom à une lambda expression :

```
? (define  f (lambda (x y) (sqrt (+ x y))))
```

alors, la variable `f` ayant pour valeur une fonction, peut s'appliquer à des arguments :

```
? (f  3  13)
4
```

Cette forme de définition d'une fonction est même le *principal* moyen pour définir une fonction. De façon générale, une définition de fonction dans le pur style Scheme est de la forme :

```
(define  nomFonction
  (lambda (x1 ... xn)
    corps))
```

[2]Une telle lambda expression s'appelle parfois une suspension (cf chapitre 10 §4).

La forme que nous utilisons n'est qu'une commodité syntaxique. En fait, Scheme la traduit sous la forme précédente. Certains auteurs n'utilisent que la forme de définition avec une lambda expression car elle a l'avantage de ne pas faire de distinction entre une définition de fonction et une définition d'une autre valeur Scheme. Ceci étant dit, nous continuerons le plus souvent à utiliser l'ancienne forme car elle est plus concise et moins intimidante pour le lecteur. Mais surtout elle est analogue à celle utilisée dans les autres dialectes Lisp. Cette syntaxe correspond au même abus de langage que l'on fait constamment en mathématiques lorsque l'on écrit : $f(x,y) = \sqrt{x+y}$ au lieu de $f = (x,y) \rightarrow \sqrt{x+y}$.
Mais, si le lecteur souhaite remplacer une définition dans l'ancienne forme comme :

```
(define (member?  s L)
  (if (null? L)
    #f
    (or (equal? s (car L))
        (member? s (cdr L)))))
```

par une définition avec la syntaxe principale,

```
(define member?
  (lambda (s L)
    (if (null? L)
        #f
        (or (equal? s (car L))
            (member? s (cdr L))))))
```

il peut utiliser la fonction suivante. Elle construit la nouvelle forme à partir de l'ancienne ; c'est un bon exemple d'utilisation de la quasi-citation.

```
(define (ancienneForme->nlleForme  def-sans-lambda)
  (let ((nom-fct (caadr def-sansLambda))
        (Liste-parametre (cdadr def-sansLambda))
        (Lcorps (cddr def-sansLambda)))
    `(define ,nom-fct (lambda ,Liste-parametre ,@Lcorps))))

? (ancienneForme->nlleForme
   '(define (member?  s L)
      (if (null? L)
        #f
        (or (equal? s (car L))
            (member? s (cdr L))))))

(define member? (lambda (s l)
  (if (null? l) #f (or (equal? s (car l)) (member? s (cdr l))))))
```

Cette traduction donne un avant-goût des possibilités de Scheme comme méta-langage, c'est-à-dire comme langage de manipulation de langages.

Let et lambda expression

Il est intéressant de remarquer que la forme spéciale `let` n'est pas indispensable, on peut l'obtenir par application d'une lambda expression. Comparer :

```
? (let ((x (*  4 5))
       (y 3))
    (+ x y))
23
```

avec

```
? ( (lambda (x y)(+ x y)) (* 4 5)  3)
23
```

Plus généralement, vérifions que la forme :

(let ((*x1 e1*) ... (*xk ek*)) *corps*))

est équivalente à l'application :

((lambda (*x1* ...*xk*) *corps*) *e1* ... *ek*)

En effet, dans les deux cas il faut évaluer le ***corps*** dans l'environnement augmenté des liaisons entre les ***xi*** et les valeurs des ***ei***.

3.2 Portée statique et fermeture

Portée statique

La valeur de l'expression :

```
(let ((a 5))
  (let ((f (lambda (x)(+ x a)))
        (a 0))
    (f 10)))
```

pose un problème : est-ce 15 ou 10? En effet, (f 10) est l'ajout à 10 de la valeur de a ! Mais laquelle?

- celle au moment de la définition de f, donc a = 5,
- celle au moment de l'appel de f, donc a = 0?

En Scheme, on utilise la valeur de a visible au moment de la définition de f. Ce choix s'appelle la *portée statique*, alors que l'utilisation de la valeur au moment de l'appel correspond à la *portée dynamique*. La réponse est donc 15.
On a la même règle de portée avec les définitions au top level ; considérons la session :

```
? (define b 10)

? (define (h x)
    (* b x))

? (let ((b 0))
    (h 5))
50
```

C'est bien la valeur de b au moment de la définition de h qui a été utilisée dans le corps de h.

Fermeture = lambda expression + environnement

Pour préciser la portée statique, on doit introduire la notion d'*occurrence liée*. Une occurrence d'une variable dans une expression est dite liée si c'est un paramètre d'une lambda englobante ou si elle est introduite par un lieur de la famille de `let`, sinon on dit que c'est une *occurrence libre*[3].
Par exemple, les occurences des variables `a` et `+` sont libres et celles de `x` et `y` sont liées dans l'expression :

`(lambda (x)(lambda (y)(+ x y a))).`

Une même variable peut avoir des occurrrences libres et des occurrences liées. Par exemple, dans l'expression suivante `x` a une occurrence liée et une libre :

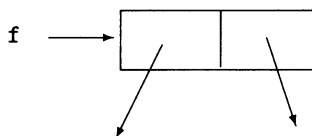
`((lambda (x)(+ x 1)) x)`

La portée statique introduit immédiatement un problème : comment se rappeler, au moment de l'appel, les valeurs qu'avaient les occurrences libres au moment de la définition ?
La réponse est simple : il suffit d'associer à une variable fonctionnelle non seulement sa lambda expression mais également un pointeur sur l'environnement au moment de sa définition. Ce couple constitué d'une lambda expression et d'un pointeur sur un environnement s'appelle une *fermeture*. Ainsi, à l'appel, les paramètres formels seront liés aux valeurs des arguments et les valeurs des variables libres seront recherchées dans l'environnement de définition désigné par le pointeur.

Exercice 2 *Deviner la valeur de :*

```
((lambda (a)
   (let ((a 1)
         (f (lambda (x)(+ a x))))
     (f a))) 5)
```

Du point de vue de la représentation interne, on peut utiliser le modèle suivant. La case mémoire associée à une valeur fonctionnelle contient un couple d'adresses : l'adresse de sa lambda expression et l'adresse de son environnement de définition.



Représentation d'une fermeture

Ceci explique que, lorsque l'on demande la valeur d'une fonction, Scheme n'affiche pas sa lambda expression mais un affichage conventionnel (et dépendant de l'implantation) pour représenter une fermeture :

[3]Ceci sera formalisé aux chapitres 19 §3 et 20 §1.

```
? h   -> #[procedure h]
```

Pour reconnaître les valeurs fonctionnelles, on dispose du prédicat `procedure?` :

```
? (procedure? car)                -> #t
? (procedure? (lambda (x)(* a x))) -> #t
? (procedure? h )                 -> #t
? (procedure? (cons 'a 'b))       -> #f
```

Attention ! La comparaison des fonctions par `eq?` ne teste que l'identité physique des objets Scheme :

```
? (eq? car  (lambda (L)(car L)))        -> #f
? (eq? (lambda (x) x)   (lambda (x) x)) -> #f
? (eq? cadr  cadr)                      -> #t
```

Deux fonctions `f` et `g` sont dites *extentionnellement* égales si elles ont le même domaine de définition et prennent les mêmes valeurs pour tous les éléments de ce domaine. Ce type d'égalité mathématique n'est pas vérifiable de façon algorithmique dès que les domaines sont infinis. Aussi, il n'est pas étonnant que le test `equal?` ne puisse être utilisé pour comparer des fonctions :

```
? (equal? car (lambda (L)(car L)))        -> #f
? (equal? (lambda (x) x)   (lambda (x) x)) -> #f
```

Portée dynamique

La liaison statique ne permet pas d'utiliser le style suivant. Définissons des fonctions `f` et `g` en supposant que `a` ne soit pas déjà une variable globale.

```
(define  (f a)
  (+ a (g 2)))

(define (g x)
  (+ a x))
```

L'appel de `(f 1)` déclenche
`*** ERROR in g - unbound variable :  a`

Au moment de l'appel `(f 1)`, `a` est lié à 1, mais cette liaison n'est pas connue de `g`, donc l'appel `(g 2)` déclenche l'erreur. Au moment de la définition de `g`, la variable `a` est libre aussi la fermeture associée à `g` contient bien un pointeur sur l'environnement global mais `a` est inconnu dans cet environnement.
Quelle aurait été la valeur de `(f 1)` si Scheme utilisait la portée dynamique ?
Les notions introduites dans ce paragraphe seront complétées au chapitre 5 et approfondies à l'occasion des descriptions plus formelles données au chapitre 21.

3.3 Les fonctions en résultat

Dans la plupart des langages impératifs, les fonctions doivent être définies par l'utilisateur dans le texte du programme ; on ne peut pas en créer de nouvelle à l'*exécution* du programme. Il ne s'agit pas ici de générer du code à l'exécution mais de créer dynamiquement une fonction grâce à la notion de lambda expression. Par exemple, étant donné un entier `n`, on peut lui associer la fonction de multiplication par `n` : `(lambda (x) (* x n))`. L'association entre `n` et cette lambda expression est réalisée par la fonction `multiplication-par-n` :

```
(define (multiplication-par-n n)
  (lambda (x)(* n x)))
```

Chaque appel de `multiplication-par-n` créera une fonction :

```
? (define multiplication-par-9 (multiplication-par-n 9))

? (multiplication-par-9 7) -> 63
```

Cette possibilité a de nombreuses conséquences, on en présente quelques-unes.

Composition de fonctions

Dans les langages qui ne disposent pas du concept de lambda expression, on ne peut pas créer à *l'exécution* la composée de deux fonctions ; on doit avoir préalablement défini la fonction composée. Autrement dit, pour composer deux fonctions d'une variable `f` et `g`, on doit écrire une nouvelle définition du style :

```
(define (compose-g-f x)
  (g (f x))
```

En Scheme, on peut écrire une fonction générale de composition qui associe à deux fonctions quelconques `g` et `f` la fonction composée x $\rightarrow$ `(g (f x))` (notée $g \circ f$ en mathématiques[4]). On peut créer des nouvelles fonctions à l'exécution :

```
(define (compose g f)
  (lambda (x)(g (f x))))

(define cinquieme (compose car cddddr))

? (cinquieme '(a b c d e f g)) -> e

? ((compose (multiplication-par-n 2)(multiplication-par-n 3)) 5) -> 30
```

Exercice 3 *Etant donné un entier positif et une fonction f, écrire la fonction* `iterer-n-fois` *qui retourne la fonction f itérée n fois :* $f \circ \cdots \circ f$.

[4]On suppose qu'il n'y pas de problème de domaine pour composer ces fonctions.

Curryfication d'une fonction

On peut toujours, d'une certaine façon, se ramener au cas des fonctions d'*une* variable. En effet, une fonction de deux variables $(x, y) \to f(x, y)$ peut être considérée comme une fonction de x à valeur une fonction de y en utilisant la décomposition : $x \to (y \to f(x, y))$.
Cette transformation s'appelle la *curryfication*[5]. Elle s'étend aux fonctions de n variables, la fonction $f(x_1, \ldots, x_n)$ a pour version curryfiée la fonction $x_1 \to (x_2 \to (x_3 \to \ldots (x_n \to f(x_1, \ldots, x_n)) \ldots))$
Pour les fonctions de deux variables, on peut définir cette transformation par :

```
(define (Curryfication f)
  (lambda (x) (lambda (y)( f x y))))
```

Un des intérêts de la curryfication est de pouvoir spécialiser une fonction dès que l'on connaît la valeur de son premier argument (ou des ses premiers arguments, si elle a plus de deux variables). On trouvera une illustration avec la spécialisation de `remove` au §5. Un autre avantage des fonctions d'une variable est que l'on peut les composer sans avoir à se préoccuper de la cohérence du nombre de leurs arguments.
L'opération inverse s'appelle la «decurryfication» :

```
(define (deCurryfication f)
  (lambda (x y)((f x) y)))
```

L'avantage de la curryfication est de se ramener à travailler essentiellement avec des fonctions d'une variable ; ce point de vue est utilisé par les langages fonctionnels de la famille ML.

3.4 Les fonctions en argument

En tant que grandeur de première espèce, les fonctions peuvent aussi être utilisées en argument d'une fonction. C'est ce que l'on a déjà fait, sans le signaler, dans le paragraphe précédent : les fonctions : `compose, Curryfication, deCurryfication` utilisent des arguments fonctionnels. Voyons d'autres illustrations de cette possibilité.

Filtrage d'une liste

Un exemple classique est la fonction `filtre` qui, étant donné un prédicat p? et une liste L, rend la sous-liste des éléments satisfaisant ce prédicat :

```
(define (filtre  p?  L)
  (cond ((null? L) '())
        ((p? (car L)) (cons (car L)(filtre p? (cdr L))))
        (else  (filtre p? (cdr L)))))

? (filtre  odd?  '(1 5 4 7 8 41 0 12)) -> (1 5 7 41)
```

[5]Du nom du logicien anglais Haskell Curry. Son prénom a servi à baptiser un langage fonctionnel.

Exercice 4 *Ecrire une fonction* `map*` *qui prend en paramètre une fonction f d'un paramètre et une liste, et retourne la liste obtenue en appliquant f aux éléments de* `L` *et à tous les niveaux.*

```
? (map* number? '(a (5 b) 0 c)) -> (#f (#t #f) #t #f)
```

Voici une application, au calcul numérique, de la possibilité de passer une fonction en paramètre.

Méthode des approximations successives

Supposons que nous voulions résoudre une équation de la forme $x = f(x)$, où f désigne une fonction numérique d'une variable numérique. Cette équation s'interprète géométriquement comme l'intersection du graphe de f avec la première bissectrice. On a appris, en mathématique, une technique extrêmement puissante pour trouver une approximation d'une solution de cette équation : la *méthode des approximations successives.* On part d'une valeur arbitraire x_0 et on construit la suite $x_1 = f(x_0), \ldots, x_n = f(x_{n-1}), \ldots$ On démontre, sous des hypothèses assez générales, que cette suite x_n converge vers une valeur x telle que $x = f(x)$.
Pour calculer le nième itéré x_n, on définit la fonction :

```
(define (n-ieme-iteration f n x0)
  (if (zero? n)
    x0
    (f (n-ieme-iteration f (- n 1) x0))))
```

Calculons le cinquième itéré relatif à la fonction $x \to \frac{x}{2} + \frac{1}{x}$ en partant de la valeur $x_0 = 1$:

```
? (n-ieme-iteration (lambda (x)(+ (/ x 2) (/ 1.0  x)))   5  1)
1.414213562373095
```

On reconnaît une excellente approximation de $\sqrt{2}$. Ce n'est pas étonnant car la suite x_n converge vers $x > 0$ tel que $x = \frac{x}{2} + \frac{1}{x}$ soit $x^2 = 2$.

Si l'on souhaite obtenir une bonne approximation de la solution de l'équation $x = sin(2x)$, il suffit d'utiliser de nouveau la fonction `nieme-iteration` en passant en argument la fonction $x \to sin(2x)$:

```
? (n-ieme-iteration (lambda (x)(sin (* 2 x)))  5  1)
.9418574795773446
```

La méthode des approximations successives est un outil très général dont on donnera même au chapitre 20 une application à la théorie des définitions récursives.

Exercice 5 *On désire contrôler le nombre d'itérations pour obtenir une approximation de $\sqrt{2}$ à un nombre $\epsilon > 0$ donné à l'avance. Montrer qu'il suffit pour cela d'avoir $|2 - x_n^2| \leq \epsilon$. Ecrire une fonction qui calcule $\sqrt{2}$ avec une précision donnée.*

3.5 Letrec et définitions locales de fonctions

Si une fonction ne sert que comme un calcul auxiliaire à une autre fonction et ne présente pas un intérêt général, on souhaite pouvoir l'utiliser dans la fonction principale mais on ne veut pas qu'elle soit visible ailleurs. C'est la notion de fonction locale à une autre.

Définition locale d'une fonction par un `let`

En fait, le concept n'est pas nouveau : une fonction étant une valeur comme une autre, il suffit d'utiliser un `let` pour réaliser une définition locale.
Ainsi, pour calculer la longueur $\sqrt{a^2 + b^2}$ d'un vecteur du plan de composantes a et b, on a besoin de calculer les carrés de a et de b ; on définit localement une fonction `carre` :

```
(define (longueur  a b)
  (let ((carre (lambda (x) (* x x)))
    (sqrt (+ (carre a)(carre b)))))
```

Définition récursive locale

Si l'on veut définir *récursivement* une fonction locale, on ne peut pas le faire exactement de cette façon. Définissons la fonction `iota` qui associe à un entier $n \geq 0$ la liste `(0 1 ....n)` :

```
(define (iota n)
  (if (zero? n)
    '(0)
    (append (iota  (- n 1))
            (list n))))
```

Cette définition est correcte mais fait un usage important de `append` qui, comme on le verra plus tard, est une fonction gaspilleuse de place mémoire. L'utilisation de `append` provient de l'obligation de faire l'ajoût en fin de liste. D'où l'idée d'inverser le problème : on considère une fonction qui associe à un entier `m` la liste `(m m+1 ....  n)`. Cette fonction auxiliaire `M-a-N` se définit en faisant la récurrence sur l'entier du début `m` ; cela permet de remplacer `append` par `cons` :

```
(define (M-a-N m n)
  (if (< n m)
    '()
    (cons m (M-a-N (+ m 1) n))))
```

Mais, si l'on n'utilise pas cette fonction ailleurs, il est préférable de la définir à l'intérieur de `iota`. Cela permet aussi de se dispenser du paramètre `n` qui n'est pas modifié par cette fonction auxiliaire `M-a-N` :

```
(define  (iota n)
  (let ((M-a-N
          (lambda (m)
```

```
          (if (< n m)
            '()
            (cons m (M-a-N  (+ m 1)))))))
    (M-a-N 0)))

? (iota 5)
*** ERROR -- unbound variable : M-a-N
```

L'appel récursif de `M-a-N` déclenche une erreur car la fonction `M-a-N` n'est pas connue dans son corps, c'est le problème inhérent aux définitions récursives. On doit utiliser un `let` spécialement adapté aux définitions récursives, appelé `letrec`.

La forme spéciale `letrec`

La forme spéciale `letrec` permet, comme `define` au top level, de rendre `M-a-N` visible dans le corps de sa définition :

```
(define  (iota n)
  (letrec  ((M-a-N
             (lambda (m)
               (if (< n m)
                 '()
                 (cons m (M-a-N  (+ m 1)))))))
    (M-a-N 0)))

? (iota 5) -> (0 1 2 3 4 5)
```

La syntaxe de `letrec`[6] est la même que celle du `let` :

(`letrec` *`liaisons corps`*) où *`liaisons`* `= ((`*`var1 e1`*`)...  (`*`vark ek`*`))`

Mais, contrairement au `let`, les variables locales *`var1, ...,vark`* sont visibles dans toutes les expressions *`e1,...,ek`*. Cependant, il y a une restriction : on doit pouvoir évaluer chaque expression *`ej`* sans avoir besoin d'évaluer une variable *`varj`* ; c'est bien le cas quand *`ej`* est une lambda expression.
Grâce à cette sémantique, le `letrec` permet aussi de définir des fonctions mutuellement récursives. Par exemple, les prédicats `even?` et `odd?` sont un cas typique de fonctions admettant une définition récursive mutuelle :

```
? (letrec ((even?
            (lambda (n)
              (if (zero? n)
                #t
                (odd? (- n 1)))))
           (odd?
            (lambda (n)
              (if (zero? n)
                #f
                (even? (- n 1))))))
   (odd? 45))
#t
```

[6] On dispose en Common Lisp d'une forme voisine : `labels`.

Exercice 6 *On définit par récurrence les suites* u_n *et* v_n *par* $u_n = u_{n-1} + v_{n-1}$ *et* $v_n = u_{n-1} * v_{n-1}$ *avec* $u_0 = v_0 = 1$. *Calculer à l'aide d'un* **`letrec`** *la valeur de* $u_3 * v_4$.

Version curryfiée de la fonction `remove-test`

Commençons par écrire une fonction `remove-test`, analogue à celle du chapitre 3 §3, mais qui prend comme paramètre supplémentaire le prédicat de comparaison à utiliser. On définit localement une fonction récursive auxiliaire pour faire le calcul :

```
(define (remove-test predicat? s L)
  (letrec ((aux (lambda (L)
                  (cond ((null? L) L)
                        ((predicat? s (car L))(aux (cdr L)))
                        (else (cons (car L)(aux (cdr L))))))))
    (aux L)))
```

Si l'on connaît le prédicat à utiliser, on aimerait pouvoir spécialiser cette fonction en y remplaçant le paramètre `comparable?` par sa valeur. La définition précédente s'y prête mal. Voici maintenant une forme curryfiée de la fonction `remove-test` qui permet cette particularisation. C'est une fonction ayant comme seul paramètre le prédicat et dont la valeur est une lambda expression des autres paramètres. Pour mettre en évidence la forme curryfiée, on utilise la syntaxe principale des définitions de fonction :

```
(define remove-test
  (lambda (predicat?)
    (lambda (s L)
      (letrec ((aux (lambda (L)
                      (cond ((null? L) L)
                            ((predicat? s (car L))(aux (cdr L)))
                            (else (cons (car L)(aux (cdr L))))))))
        (aux L)))))
```

On la spécialise pour divers prédicats :

```
? ((remove-test <)  2 '(5 0 1 -1  2 3)) -> (0 1 -1)
? ((remove-test =)  2 '(5 0 1 -1  2 3)) -> (5 0 1 -1 3)
```

Cette version curryfiée permet de définir trois fonctions spécialisées de suppression :

```
(define  remove  (remove-test equal?))
(define  remv    (remove-test eqv?))
(define  remq    (remove-test eq?))
```

`Letrec` et `define` en interne

On sait que l'on peut définir au top level des fonctions récursives avec la forme :

```
(define f (lambda (x1 ... xi) corps)
```

car la règle de portée de **define** est analogue à celle de **letrec**: la liaison entre f et la lambda expression est visible dans le corps de la lambda. Aussi certaines implantations de Scheme permettent de faire des définitions locales par **define**:

```
(define  (iota n)
  (define M-a-N
    (lambda (m)
      (if (< n m)
        '()
        (cons m (M-a-N (+ m 1))))))
  (M-a-N 0))
```

mais comme ce n'est pas standard, on préférera la formulation équivalente avec **letrec**:

```
(lambda (x1 ... xi)              ≡        (lambda (x1 ...xi)
  (define  f1  e1)                          (letrec ((f1  e1)
   . . .                                              . . .
  (define  fn  en)                                    (fn  en))
  corps)                                      corps))
```

3.6 La fonction map et ses dérivés

On a souvent besoin d'appliquer un même traitement à chaque élément d'une liste; la programmation d'ordre supérieur favorise ce style grâce à des fonctionnelles (on dit aussi itérateurs) prédéfinies.

La fonction map

Pour appliquer une fonction f à tous les éléments d'une liste (x1 ...xN) et renvoyer la liste des résultats ((f x1) ...(f xN)), on peut utiliser la fonction:

```
(define (mon-map f L)
  (if (null? L)
    '()
     (cons (f (car L))
           (mon-map f (cdr L)))))

? (mon-map  odd?  '(4 1 8 5 0 -5))  -> (#f #t #f #t #f #t)
```

Mais Scheme dispose d'une fonction prédéfinie map encore plus générale car on peut l'utiliser avec des fonctions à n arguments en lui passant en paramètre n listes de même longueur. Sa syntaxe est:

```
(map  f  L1 ... Ln)

? (map  odd?  '(4 1 8 5 0 -5))       -> (#f #t #f #t #f #t)
? (map  *   '(1 2 3) (10 10 10))     -> (10 20 30)
? (map  list  '(a 1 (2) ()))         -> ((a) (1) ((2)) (()))
? (map  append   '((a b)  (c)) '((1 ) (2 3))) -> ((a b 1) (c 2 3))
```

Un des intérêts de la fonction `map` est d'éviter une programmation récursive pour définir ce type d'application. On dit que c'est un itérateur de liste car on effectue la même opération sur tous les éléments d'une liste ; l'ordre des applications de `f` aux éléments de L n'est pas spécifié.
On utilise souvent `map` en combinaison avec une lambda expression :

```
? (map (lambda (x)(cons x '(a b))) '(1 2 3)) -> ((1 a b) (2 a b) (3 a b))
```

Illustrons encore l'utilisation de `map` en définissant la fonction `pairlis`. C'est une fonction qui ajoute en tête d'une a-liste les doublets obtenus en prenant les clés dans une liste de clés et les valeurs dans une liste de valeurs

```
(define (pairlis  Lcles Lvaleurs  aliste)
  (append (map cons  Lcles Lvaleurs)
          aliste))

? (pairlis  '(a b c) '(1 2 3) '((d . 4))) -> ((a . 1) (b . 2) (c . 3)(d . 4))
```

Remarque 1 Attention, les formes spéciales ne sont pas des fonctions et ne peuvent donc pas être des arguments fonctionnels :

```
?  (map or '(#f #f #t #t) '(#f #t #f #t)) -> ERROR
```

On peut parfois contourner la difficulté en enveloppant la forme dans une lambda :

```
? (map (lambda (x y)(or x y)) '(#f #f #t #t) '(#f #t #f #t)) -> (#f #t #t #t)
```

Exercice 7 *Donner la valeur des expressions :*
(map (lambda (x)(+ x 1)) (iota 4))
(map procedure? (list + '+ car 'map #t))
(map (lambda (L) (map (lambda (M) (cons L M)) '(a b c))) '(1 2))

On doit parfois appliquer une fonction de plusieurs variables en laissant fixes certaines variables. Pour cela, on utilise une lambda expression auxiliaire. Voyons un exemple de cette situation.
Soit `LListes` une liste de listes, écrivons une fonction `ajouter-en-tete` pour ajouter en tête de chaque liste la valeur d'une expression `s`. On ne peut pas écrire `(map cons s LListes)` car `s` doit être fixe ; une méthode consiste à écrire :

```
(define (ajouter-en-tete s LListes)
  (map (lambda (L)(cons s L))  LListes))

? (ajouter-en-tete 'van '((Dongen 1877 1968) (Eyck 1370 1426)(Gogh 1853 1890)
((van dongen 1877 1968) (van eyck 1370 1426)(van gogh 1853 1890))
```

Exercice 8 *En utilisant la technique expliquée à la fin du §1, écrire une fonction qui transforme une expression* `let` *en l'expression équivalente avec une lambda expression.*

La fonction for-each

Dans certains cas, on ne s'intéresse pas à la liste des valeurs des (f xj) mais aux effets de bord provoqués par l'évaluation des (f xj). Pour cela, on utilise la fonction prédéfinie **for-each** qui a la même syntaxe que **map** :

```
(for-each  fct  L1 ... Ln)
```

La valeur finale rendue est indéterminée car seuls les effets importent, en revanche l'ordre des évaluations va du début à la fin des listes *Li*.

Comme application, affichons les éléments d'une liste L avec un espace entre chaque. C'est l'objet de la fonction **display-liste** :

```
(define (display-liste  L)
  (for-each  (lambda (x)(display x)(display " ")) L))

? (display-liste  '(a b c))
a b c
```

La fonction apply

Pour appliquer une fonction f à des arguments s1,..., sN, on évalue l'expression (f s1 ...sN) mais, dans certains cas, les valeurs des arguments sont déjà données directement dans une liste (v1...vN), alors comment leur appliquer la fonction f ? C'est le rôle de la fonction prédéfinie apply, sa syntaxe est :

```
(apply  fct  Liste-valeurs-arguments)

? (apply + '(5 8 2)) -> 15
? (apply  cons  '(1 2)) -> (1 . 2)
```

Mais, attention :

```
? (apply + '( 5  (* 2 4))) -> ERROR
```

car il n'y a pas évaluation des éléments à l'intérieur de la liste.

La combinaison de apply et de map est typique du style de programmation à l'ordre supérieur. Illustrons-le avec le produit scalaire de deux listes. Etant donné deux listes L = (x1 ...xK) et M = (y1 ...yK) de nombres, on appelle produit scalaire la somme x1 y1 + ...+ xK yK. On peut écrire une fonction **produit-scalaire** en remarquant que (map * L M) rend la liste des produits (x1 y1 ...xK yK), il reste ensuite à les additionner :

```
(define  (produit-scalaire-liste  L M)
  (apply  +  (map  *  L M)))
```

Utilisation des itérateurs pour manipuler les listes générales

L'utilisation des fonctions du type `map` n'est pas restreinte aux fonctions opérant seulement au premier niveau d'une liste.
Calculons la profondeur d'une S-expression. Il suffit, pour une liste non vide, d'ajouter 1 au maximum des profondeurs de chaque élément de la liste :

```
(define (profondeur s)
  (cond ((atom? s) 0)
        ((null? s) 1)
        (else (+ 1 (apply  max (map profondeur s))))))

? (profondeur  '(a (b c) ((d)) "oui"  ())) -> 3
```

Exercice 9 *Ecrire la fonction* `sommeListe*` *du chapitre 2 §4 avec cette technique.*

Adjonction de nouveaux itérateurs

Scheme ne fournit que les itérateurs `map`, `for-each`, on se propose d'en ajouter d'autres que l'on rencontre dans certains dialectes Lisp, à savoir : `append-map`, `map-select`, `every`, `some`, `list-it`.

La fonction append-map

Quand les valeurs d'une fonction `f` sur les éléments d'une liste `L` sont des listes, on est souvent conduit à faire la concaténation de ces listes. Dans ce but, définissons l'itérateur `append-map` :

```
(define  (append-map  f L)
  (apply  append  (map  f L))
```

Appliquons ceci à une nouvelle définition de la fonction `aplatir`. Pour aplatir une liste non vide, il suffit de concaténer les aplatissements de chaque élément au premier niveau. Cela conduit à définir l'aplatissement d'un atome comme étant une liste réduite à cet atome :

```
(define (aplatir s)
  (if (list? s)
    (append-map  aplatir s)
    (list s)))
```

Un map sélectif

On a parfois besoin d'un `map` sélectif, lorsqu'on veut n'appliquer une fonction `f` qu'aux éléments de la liste qui satisfont un prédicat `p?`. L'idée est d'utiliser `append-map` et d'attribuer la valeur `()` aux éléments qui ne satisfont pas au prédicat, ainsi ils ne contribueront pas à la liste finale :

```
(define (map-select f L p?)
  (append-map (lambda (x)
                (if (p? x)
```

```
                        (list (f x))
                        '()))
                     L))

? (map-select (lambda (x)(/ 1 x))
              '(a 3 0 4 7)
              (lambda (x)(and (number? x)(not (zero? x)))))
(1/3 1/4 1/7)
```

Exercice 10 *Etant donné des listes (x0 ... xM) et (y0 ... yN) et une fonction f de deux variables, écrire une fonction* **tableau** *qui construit la liste :*
((f x0 y0) ...(f x0 yN) ...(f xM y0)...(f xM yN))

Les itérateurs every et some

On veut généraliser la forme **and** au cas où les arguments du **and** sont les valeurs d'une fonction. Plus précisément, on définit une fonction **every** telle que :

```
  (every f '(e1 ... en))  = (and (f e1) ... (f eN))
```

La définition naturelle est donc d'appliquer la conjonction **and** à la liste des valeurs de f :

```
(define (every f L)
  (apply and (map f L)))
```

Mais cette définition n'est pas acceptable car on ne peut utiliser **and** comme un argument fonctionnel de apply. En effet, **and** n'est pas une fonction mais une forme spéciale. Pour définir **every**, on revient au style habituel des définitions récursives :

```
(define (every f L)
  (if (null? L)
    #t
    (and (f (car L))
         (every f (cdr L)))))
```

Quand on prend comme fonction f un prédicat p?, l'expression (every p? L) est vraie si tous les éléments de L vérifient la propriété p? :

```
? (every number? '(1 2 a 3)) -> #f
```

De façon analogue, on définit la fonction **some** en remplaçant **and** par **or** :

```
(some f '(e1 ...en) ) = (or (f e1) ...(f eN))
```

soit

```
(define (some f L)
  (if (null? L)
    #f
    (or (f (car L))
         (some f (cdr L)))))
```

Si l'on prend comme fonction `f` un prédicat `p?`, l'expression `(some p? L)` est vraie s'il existe au moins un élément de `L` qui vérifie la propriété `p?` :

```
? (some number? '(1 2 a 3)) -> #t
```

Accumulation des résultats d'une fonction : `list-it`

On rencontre parfois une même structure de programme dans des fonctions différentes. Considérons les quatre fonctions suivantes.

On calcule la somme des valeurs d'une fonction `h` sur une liste d'arguments `L` :

```
(define (somme  h L)
  (if (null? L)
    0
    (+ (h (car L)) (somme  h (cdr L)))))
```

On calcule le produit des valeurs d'une fonction `h` sur une liste d'arguments `L`

```
(define (Produit  h L)
  (if (null? L)
    1
    (* (h (car L)) (Produit  h (cdr L)))))
```

On redéfinit append pour deux listes :

```
(define (Mon-append L M)
  (if (null? L)
    M
    (cons (car L)(Mon-append (cdr L) M))))
```

On redéfinit la fonction `map` dans le cas d'une fonction unaire `h` :

```
(define (Mon-map h L)
  (if (null? L)
    '()
    (cons (h (car L)) (Mon-map  h (cdr L)))))
```

On pressent une structure commune entre ces définitions. Pour y voir plus clair, explicitons le détail du calcul dans chaque cas ; on pose `L = (x0 x1 ...xN)`

```
(somme  h L)     = (+    (h x0) (+    (h x1) (+ ...(+ (h xN)       0) ...)))
(Produit  h L)   = (*    (h x0) (*    (h x1) (* ...(* (h xN)       0) ...)))
(Mon-append L M) = (cons x0     (cons x1     (  ...(cons xN        M) ...)))
(Mon-map h L)    = (cons (h x0) (cons (h x1) (  ...(cons (h xN)  '()) ...)))
```

La ressemblance s'explique maintenant par l'utilisation du même schéma de calcul. On accumule les résultats de l'application d'une fonction binaire `f` aux éléments de la liste. Le premier argument de `f` parcourt la liste `L` et le deuxième accumule les résultats successifs, la valeur finale est :

```
(f x0 (f x1 ( ... (f xN  b) ... )))
```

De façon plus précise, on définit la fonction `list-it` :

```
(define (list-it f  L  b)
  (if (null? L)
    b
    (f (car L)
       (list-it f  (cdr L) b))))
```

Il vient, en identifiant les schémas de calcul :

```
(somme      h L)  = (list-it  (lambda (x y)(+ (h x) y))  L 0)
(produit    h L)  = (list-it  (lambda (x y)(* (h x) y))  L 1)
(Mon-append L M)  = (list-it  cons L M)
(Mon-map    h L)  = (list-it  (lambda (x y)(cons (h x) y)  L '())
```

Si la fonction `list-it` était une primitive du langage, son utilisation à la place d'une définition récursive fournirait un gain d'efficacité. Dans le cas de Scheme, le gain est plus dans la compacité du code que dans l'efficacité.
Ce style de programmation, utilisant des combinateurs puissants pour définir les fonctions comme dans un calcul algébrique, peut conduire à un code concis mais illisible !

Exercice 11 *Définir une fonction* `it-list` *qui met en œuvre le même genre de calcul mais c'est le deuxième argument de f qui parcourt la liste :*

(f ...(f (f a y1) y2) ...yN)

Appliquer ce schéma à la définition des fonctions `renverse` *et* `factorielle`*.*

3.7 Extension de la syntaxe des lambda

Fonctions admettant un nombre quelconque d'arguments

Il y a de nombreuses fonctions prédéfinies de Scheme qui acceptent un nombre arbitraire d'arguments : `append, +, list`, ... Mais on n'a donné aucun moyen pour en définir de nouvelles. Pour cela, on dispose d'une forme plus générale pour la liste des paramètres d'une lambda expression. On écrit :

```
(lambda (x1 . . . xN . Lrest) corps)
```

pour désigner une fonction demandant `N` arguments obligatoires *x1, ..., xN* et acceptant des arguments supplémentaires facultatifs qui seront placés dans la liste *Lrest*. Dans l'exemple :

```
? ((lambda (x . L)(cons x L)) 1 2 3 4) -> (1 2 3 4)
```

on voit que l'argument obligatoire 1 est lié avec `x` et la liste des arguments facultatifs (2 3 4) est liée avec `L`.
Si l'on ne désire aucun argument obligatoire, on dispose de la syntaxe :

```
(lambda  Lrest  corps)
```

où la liste des arguments sera liée avec la liste *Lrest*, ce qui permet de n'avoir aucun argument.
L'exemple le plus simple permet de retrouver la fonction prédéfinie `list` :

```
? ((lambda  L  L)  'a 'b 'c) -> (a b c)
```

On a les formes correspondantes avec l'autre style de définition des fonctions.

- on définit une fonction avec N paramètres obligatoires et des paramètres facultatifs par :

  ```
  (define  (f  x1 ... xN . Lrest)
    corps)
  ```

- on définit une fonction n'ayant que des paramètres facultatifs par :

  ```
  (define  (f . Lrest)
    corps)
  ```

Voici deux fonctions `display-all` et `display-alln` qui utilisent ces extensions. On utilisera souvent par la suite ces fonctions d'affichage, aussi, est-il préférable de les ajouter à votre fichier d'utilitaires.
La première généralise `display` au cas d'un nombre quelconque d'arguments :

```
(define (display-all . L)
  (for-each  display L))
```

Si l'on souhaite de plus un passage à la ligne en fin d'affichage, on utilisera la fonction `display-alln` (le `n` rappelle celui de `newline`) :

```
(define (display-alln . L)
  (for-each  display L)
  (newline))
```

C'est très pratique pour faire des affichages où l'on mélange du texte et des valeurs calculées :

```
? (display-alln  "soit x = " (+ 8 5)  "  alors 2x = " (* 2 (+ 8 5)))
soit x = 13  alors 2x = 26
```

Récursion et fonctions à arguments facultatifs

Il faut prendre garde aux définitions récursives avec des fonctions admettant un nombre arbitraire d'arguments. Définissons par exemple une fonction `mcons` qui réalise un `cons` itéré de ses arguments (elle prend au moins deux arguments) :

```
(mcons s1 s2 ... sN) = (cons s1 (cons s2 ... (cons sN-1  sN) ... )))
```

Une première méthode qui vient à l'esprit consiste à écrire :

```
(define (mcons  . L)
  (cons (car L)
        (if (null? (cddr L))
          (cadr L)
          (mcons (cdr L)))))

? (mcons 1 2 3 '(a b))
*** ERROR -- PAIR expected  (cddr '((2 3 (a b))))
```

C'est incorrect ! En effet, l'appel récursif de `mcons` n'utilise qu'un seul argument `(cdr L)`. En réalité, on désire que `(cdr L)` soit la suite des arguments de `mcons`, d'où la version corrigée qui utilise `apply` :

```
(define (mcons  . L)
  (cons (car L)
        (if (null? (cddr L))
          (cadr L)
          (apply mcons (cdr L)))))

 ? (mcons 1 2 3 '(a b)) -> (1 2 3 a b)
```

On peut éviter d'utiliser `apply` en introduisant une fonction locale à un seul argument, la liste `L` :

```
(define (mcons . L)
  (letrec ((mcons-aux
             (lambda (L)
               (cons (car L)
                     (if (null? (cddr L))
                       (cadr L)
                       (mcons-aux (cdr L)))))))
    (mcons-aux L)))
```

Exercice 12 *1. Généraliser la fonction* ***append-map*** *du §6 pour permettre de l'appliquer à des fonctions à plus d'un argument et du nombre correspondant de listes.*

2. Ecrire une fonction ***compose-toutes*** *qui compose un nombre quelconque de fonctions d'une variable :*

```
? ((compose-toutes car cdr cdr) '(a b c d)) -> c
```

3. Etendre la définition de ***every*** *et* ***some*** *pour admettre des fonctions à plus d'un argument.*

3.8 Représentation des ensembles

A titre d'illustration des paragraphes précédents, on écrit une petite bibliothèque de fonctions pour manipuler les ensembles.

Quand on considère, en mathématiques, un ensemble d'objets `{x1, ..., xN}`, on fait abstraction de l'ordre des objets et l'on ne répète pas plusieurs fois un même objet. Ainsi, l'ensemble des éléments qui composent la liste `(a b c a d e c d)` est `{a b c d e}`. Pour représenter un ensemble, l'idée la plus simple[7] consiste à utiliser la liste sans répétition de ses éléments. Pour ne pas préjuger de la nature des éléments qui composeront les ensembles, on les comparera avec le prédicat `equal?` . L'ensemble vide sera représenté par la liste vide.

On convertit une liste en un ensemble en supprimant les répétitions :

```
(define (liste->ensemble L)
  (cond ((null? L) '())
        ((member (car L)(cdr L))(liste->ensemble (cdr L)))
        (else  (cons (car L)(liste->ensemble (cdr L))))))

? (liste->ensemble  '(1 2 3 2 3 4 5)) -> (1 2 3 4 5)
```

Réunion d'ensembles

La réunion de deux ensembles est constituée des éléments qui sont au moins dans un des ensembles :

```
(define (reunion2 E1 E2)
  (cond ((null? E1) E2)
        ((member (car E1) E2)(reunion2 (cdr E1) E2))
        (else (cons (car E1)(reunion2 (cdr E1) E2)))))

? (reunion2 '(1 2 3  4) '( 2 3 4 5 6)) -> (1 2 3 4 5 6)
```

On étend la fonction `reunion2` à un nombre quelconque d'ensembles en utilisant l'itérateur `list-it` du §6 :

```
(define (reunion . Lens)
  (list-it reunion2  Lens '()))

? (reunion '(1 2 3  4) '( 2 3 4 5 6) '(4 5 6 7) '(6 7 8))
(1 2 3 4 5 6 7 8)
```

Exercice 13 *Ecrire un prédicat* ***ens-inclus?*** *qui teste l'inclusion d'un ensemble dans un autre.*
Ecrire un prédicat ***ens-equal?*** *qui teste l'égalité de deux ensembles.*

Intersection d'ensembles

L'intersection de deux ensembles est constituée des éléments qui sont dans les deux ensembles :

[7] Mais pas la plus efficace !

```
(define (intersection2  E1 E2)
  (cond ((null? E1) E1)
        ((member (car E1) E2)(cons (car E1)
                                   (intersection2 (cdr E1) E2)))
        (else (intersection2 (cdr E1) E2))))

? (intersection2 '(1 2 3  4) '( 2 3 4 5 6)) -> ( 2 3 4)
```

Remarque 2 Les fonctions union2 et intersection2 font de nombreux appels à la fonction member. Si l'on avait à manipuler des ensembles de grande taille, il faudrait disposer d'un test d'appartenance très efficace. Ce qui conduirait à réviser la représentation utilisée pour les ensembles. On renvoie au chapitre 7 pour des méthodes de consultation plus efficaces.

On étend la fonction intersection à une intersection d'un nombre quelconque d'ensembles (dire pourquoi on n'a pas utilisé list-it) :

```
(define (intersection . Lens)
  (cond ((null? Lens)'())
        ((null? (cdr Lens)) (car Lens))
        (else (intersection2 (car Lens)
                             (apply intersection (cdr Lens))))))

? (intersection '(1 2 3  4) '( 2 3 4 5 6) '(4 5 6 7) ) -> (4)
```

Exercice 14 *Ecrire une fonction qui calcule la différence ensembliste E1 - E2. Elle est constituée des éléments de E1 qui ne sont pas dans E2.*

Liste des parties

La liste des parties d'un ensemble E est la liste de tous les sous-ensembles possibles de E y compris E lui-même et l'ensemble vide.
La liste des parties de (a b c) est (() (c) (b) (b c) (a) (a c) (a b) (a b c)).
Si l'ensemble est vide, sa liste des parties est réduite à (()) sinon, considérons le premier élément a de E : il y a les parties qui le contiennent et celles qui ne le contiennent pas. Les premières sont les parties de E - {a} et les deuxièmes s'obtiennent en insérant a dans toutes les premières. D'où la fonction :

```
(define (Lparties E)
  (if (null? E)
    (list '())
    (let ((Lparties-reste (Lparties (cdr E))))
      (append Lparties-reste
              (map (lambda (L)(cons (car E) L)) Lparties-reste)))))

? (Lparties '(a b c)) -> (() (c) (b) (b c) (a) (a c) (a b) (a b c)).
```

Exercice 15 *Ecrire une fonction qui calcule la liste des permutations d'un ensemble. Par exemple, l'ensemble à trois éléments (a b c) possède une liste de* 3! *permutations : ((a b c) (b a c) (b c a) (a c b) (c a b) (c b a)).*

Exercice 16 *Un* multi-ensemble *se comporte comme un ensemble mais on permet la répétition des éléments. Par exemple,* {a a b b c} *et* {a b a c b} *sont égaux comme multi-ensemble, en revanche* {a a b b c} *et* {a a b b c c} *sont distincts. Définir une implantation des multi-ensembles avec les opérations naturelles d'union et d'intersection.*

3.9 Appel récursif terminal

On va voir que certaines définitions récursives peuvent se réduire à une simple itération. Pour cela, analysons de plus près le mécanisme d'évaluation d'un appel de fonction récursive.

Factorielle non itérative

Prenons comme exemple l'incontournable fonction factorielle :

```
(define  (fact n)
  (if (zero ? n)
    1
    (* n (fact (- n 1)))))
```

Détaillons le calcul de `(fact 3)` en appliquant pas à pas la définition :

```
? (fact 3)
-> (if (zero? n) 1 (* n (fact (- n 1))))              avec la liaison n=3
-> (* n (fact (- n 1)))                               avec la liaison n=3
 ->(* 3  (fact 2))
 ->(* 3  (if (zero? n) 1 (* n (fact (- n 1))))         avec la liaison n=2
 -> (* 3 (* n (fact (- n 1)))                          avec la liaison n=2
 ->(* 3  (* 2 (fact 1)))
  ->(* 3  (* 2  (if (zero? n) 1 (* n (fact (- n 1))))  avec la liaison n=1
  -> (* 3  (* 2   (* n (fact (- n 1))))                avec la liaison n=1
  ->  (* 3  (* 2   (* 1 (fact  0))))
   -> (* 3  (* 2   (* 1 (if (zero? n) 1 (* n (fact (- n 1))))    avec  n=0
   ->  (* 3  (* 2   (* 1 1)))
  <- (* 3  (* 2   1))
 <- (* 3   2 )
<- 6
```

On voit que la multiplication `(* 3 (fact 2))` n'a pas pu être faite avant de connaître `(fact 2)` ; ensuite la multiplication `(* 2 (fact 1))` a dû attendre que l'on connaisse `(fact 1)` ; enfin la multiplication `(* 1 (fact 0))` a été faite dès la connaissance de `(fact 0)` et l'on a alors pu remonter en cascade (les flèches `<-`) les multiplications en attente.

Pour réaliser ce stockage des opérations en attente, on utilise un mécanisme de sauvegarde en mémoire des actions à faire. On les empile comme des pense-bêtes jusqu'à pouvoir en réaliser une. On détaillera cette technique au chapitre 7. Pour le moment, il nous suffit de remarquer que le nombre de pense-bêtes à utiliser est directement proportionnel à la taille de `n`, ce qui signifie une occupation mémoire croissante avec la taille de l'argument. On imagine que dans certains cas on puisse ainsi saturer toute la mémoire réservée à ce travail et donc échouer dans le calcul.

Voyons une autre définition récursive de factorielle qui n'aura pas cet inconvénient.

Factorielle itérative

On accumule dans un deuxième paramètre les valeurs des appels récursifs :

```
(define (fact-it  n   acc)
  (if (zero? n)
      acc
      (fact-it  (- n 1)  (* acc n))))
```

Admettons provisoirement que l'on ait prouvé que
`(fact-it n 1) = (fact n)` et détaillons le calcul de :

```
? (fact-it   3 1)
-> (if (zero? n) acc (fact-it  (- n 1) (* acc n)))    avec  n=3 et acc=1
 -> (fact-it  2  3)
 -> (if (zero? n) acc (fact-it  (- n 1) (* acc n)))   avec  n=2 et acc=3
  -> (fact-it  1  6)
  -> (if (zero? n) acc (fact-it  (- n 1) (* acc n)))  avec  n=1 et acc=6
   -> (fact-it  0  6)
   -> (if (zero? n) acc (fact-it (- n 1)(* acc n)))   avec  n=0 et acc=6
    -> 6
```

Donc le calcul de `(fact-it 3 1)` se ramène à celui de `(fact-it 2 3)` qui se ramène à celui de `(fact-it 0 6)` qui vaut 6. Le point nouveau est que chacun de ces calculs n'a *pas eu besoin de mémoriser* le calcul précédent. Il n'y a donc pas empilement car chaque pense-bête *remplace* le précédent. En conséquence, la place mémoire pour effectuer ces calculs a une taille constante, indépendante de l'argument `n`. Cela ne signifie pas seulement une exécution plus rapide mais surtout une exécution possible indépendante de la taille de la donnée (du moins si le résultat reste stockable en mémoire).
Comment distinguer a priori que l'on sera dans ce cas de figure? Pour être certain qu'un appel récursif ne va pas générer des calculs en attente, il suffit de s'assurer que dans la définition de la fonction les appels récursifs ne sont pas suivis par une autre opération. On dit que ce sont des *appels récursifs terminaux*[8].
On voit que dans `fact` l'appel récursif `(fact (- n 1))` est suivi de la multiplication par `n`, alors que l'appel récursif de `fact-it` est terminal.

Prouvons maintenant l'égalité `(fact-it n 1) = (fact n)` par récurrence sur `n`. Comme souvent, pour démontrer une formule par récurrence, il faut commencer par la généraliser. On va démontrer que l'on a :

```
(fact-it  n  acc)  =  (* acc  (fact n))
```

Pour `n=0` l'égalité est immédiate.
On suppose qu'elle est vraie pour `n-1` et l'on montre qu'elle est encore vraie pour `n` :
`(fact-it n acc)`

[8] Il y a d'autres cas mais ils sont plus difficiles à détecter.

=(fact-it (- n 1) (* acc n)) ; par définition de fact-it
=(* (* acc n) (fac (- n 1))) ; d'après l'hypothèse de récurrence
=(* acc (* n (fact (- n 1)) ; d'après l'associativité de la multiplication
=(* acc (fact n)) ; d'après la définition de (fact n) pour n>0

Mise sous forme itérative

L'idée d'ajouter le paramètre acc pour accumuler les résultats intermédiaires est un moyen assez général pour passer d'une définition récursive à une définition récursive terminale, dont voici deux autres exemples.
Ecrivons une fonction qui calcule la somme des carrés d'une liste de nombres :

```
(define (somme-carres L)
  (if (null? L)
    0
    (+ (* (car L)(car L))
          (somme-carres (cdr L)))))
```

Pour en donner une version récursive terminale, on introduit un paramètre d'accumulation des sommes partielles :

```
(define (somme-carres-it L  acc)
  (if (null? L)
    acc
    (somme-carres-it (cdr L)  (+  (* (car L)(car L)) acc))))
```

Et l'on démontre, comme pour fact-it, que

```
(somme-carres-it  L acc) = (+ acc  (somme-carres L))
```

ce qui conduit à définir somme-carres par :

```
(define (somme-carres L)
  (somme-carres-it L  0))
```

La fonction somme-carres-it joue un rôle auxiliaire, aussi il est préférable de la cacher à l'intérieur de somme-carres :

```
(define (somme-carres L)
  (letrec ((somme-carres-it
             (lambda (L  acc)
               (if (null? L)
                 acc
                 (somme-carres-it (cdr L)  (+  (* (car L)(car L)) acc))))))
    (somme-carres-it L 0)))
```

La norme du langage Scheme impose que les appels récursifs terminaux soient exécutés comme des itérations[9]. Pour profiter de cette caractéristique, la programmation en Scheme est souvent de la forme précédente : on définit une fonction locale auxiliaire en forme récursive terminale pour calculer la fonction principale. Cette

[9] Voir aussi le chapitre 4 §2.

fonction auxiliaire aura souvent pour nom celui de la fonction principale avec le suffixe aux, parfois on l'appellera simplement loop pour insister sur l'aspect itératif (les ouvrages en langue anglaise utilisent souvent le nom de helper). Voici la fonction factorielle écrite dans ce style :

```
(define (fact n)
  (letrec ((fact-aux
             (lambda (n acc)
               (if (zero? n)
                 acc
                 (fact-aux  (- n 1)  (* acc n))))))
     (fac-aux n 1)))
```

Ecrivons également dans ce style la fonction 1-a-N qui, étant donné un entier n, rend la liste (1 ...n) :

```
(define (1-a-N n)
  (letrec ((loop
             (lambda (k L)
               (if (zero? k)
                 L
                 (loop (- k 1) (cons k L))))))
    (loop n '())))
```

Pour comprendre cette fonction, il faut savoir exactement ce que calcule la fonction locale loop. On laisse au lecteur le soin de démontrer par récurrence sur k l'égalité :

```
(loop  k L) = (append  '(1 ... k)  L)
```

Exercice 17 *Comparer l'exécution de ces deux fonctions et expliquer.*

```
(define (boucle-iterative)
  (display "je boucle")
  (boucle-iterative))

(define (boucle-non-iterative)
  (boucle-non-iterative)
  (display "je boucle"))
```

Exercice 18 1. *Déduire de la fonction* **produit-scalaire-liste** *définie au §6, une autre forme non récursive de* **sommes-carres**.

2. *Prouver que la fonction suivante est une version récursive terminale de la fonction* **reunion2** *du paragraphe précédent.*

```
(define (union2 E1 E2)
  (letrec ((union2-aux
             (lambda (E1 acc)
               (if (null? E1)
                 acc
                 (union2-aux (cdr E1)
                             (if (member (car E1) acc)
```

```
                                          acc
                                          (cons (car E1) acc))))))
         (if (null? E2)
           E1
           (union2-aux  E1 E2)))
```

3. *Donner une version récursive terminale des fonctions* ***liste->ensemble*** *et* ***intersection2***

Fibonacci et Ackerman

Une version itérative n'est pas toujours construite en introduisant un seul accumulateur. Considérons la célèbre fonction de Fibonacci introduite au chapitre 1.

```
(define (fib n)
  (if (<= n 1)
    n
    (+ (fib (- n 1)) (fib (- n 2)))))
```

Exécutons l'appel `(f 6)` après avoir demandé à voir la trace de `fib`. On constate de nombreuses duplications du même calcul : `(fib 4)` est calculé 2 fois, `(fib 3)` 3 fois et `(fib 2)` 5 fois. Pour éviter ce «gâchis», on calcule à chaque étape les valeurs du couple `(fib n)` , `(fib (- n 1))` car à l'étape suivante le couple `(fib (+ n 1))` , `(fib n)` est égal au couple `(fib n) + (fib (- n 1))` et `(fib n)`. D'où une version récursive terminale de la fonction de Fibonacci avec les accumulateurs `x` et `y` :

```
(define (fib-it n)
  (letrec ((fib-aux
             (lambda (n x y)
               (if (zero? n)
                 y
                 (fib-aux (- n 1) (+ x y) x )))))
    (fib-aux n 1 0)))
```

Enfin, il faut savoir qu'il existe des fonctions récursives simples que l'on ne peut *pas* écrire sous forme itérative, la plus connue est due au logicien Ackerman :

```
(define (ack m n)
  (cond ((zero? m)(+ n 1))
        ((zero? n)(ack (- m 1) 1))
        (else (ack (- m 1) (ack m (- n 1))))))
```

C'est une fonction ayant une croissance très rapide, ne la tester que pour de très petites valeurs de `m` et `n`.

Exercice 19 *Démontrer que cette fonction termine quand* ***m*** *et* ***n*** *sont des entiers positifs ou nuls.*

Exercice 20 *1. Etant donné une liste plate de symboles et un symbole, on demande de définir une fonction récursive terminale* **`Liste-occurrences`** *qui calcule la liste des numéros des positions du symbole dans la liste :*

```
? (liste-occurrences 'a '(a b c a d e b a a)) -> (0 3 7 8)
```

2. Même question pour la fonction qui calcule le nombre d'éléments distincts dans une liste plate :

```
? (nb-elements-distincts '(a b c a d e b a a)) -> 5
```

Exercice 21 *On donne une version récursive terminale de la fonction* **`reverse`** *:*

```
(define (reverse-it L R)
  (if (null? L)
    R
    (reverse-it (cdr L) (cons (car L) R))))
```

Démontrer que l'on a `(reverse-it L R) = (append (reverse L) R)`.

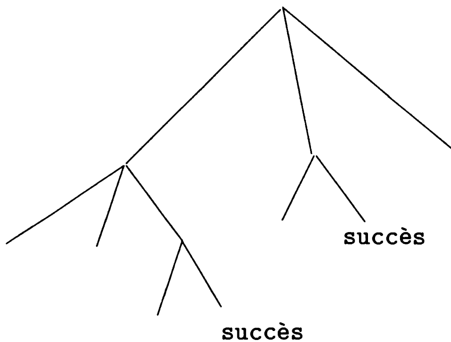
3.10 Compléments : retour-arrière et problème des huit reines

Recherche avec retour-arrière

Dans beaucoup de problèmes combinatoires ou puzzles, on ne dispose pas d'algorithme direct pour trouver une solution. On procède par essais successifs, chaque échec remet en cause des choix antérieurs. La construction d'une solution se fait progressivement par l'extension d'une solution partielle.
Un exemple classique est le problème des huits reines. Il consiste à placer huit reines sur un échiquier de sorte qu'aucune ne puisse en prendre une autre. On rappelle qu'une reine peut prendre toute pièce située sur une même rangée ou une même diagonale qu'elle. Voici un exemple d'une telle disposition où les reines sont indiquées par un R :

```
                        8    o  o  R  o  o  o  o  o

                        7    o  o  o  o  o  R  o  o

                        6    o  o  o  R  o  o  o  o

Numéro des lignes       5    o  R  o  o  o  o  o  o

                        4    o  o  o  o  o  o  o  R

                        3    o  o  o  o  R  o  o  o

                        2    o  o  o  o  o  o  R  o

                        1    R  o  o  o  o  o  o  o

                             1  2  3  4  5  6  7  8
```

La recherche d'une solution se fait progressivement : on place une reine dans la 1ère colonne puis une reine dans la deuxième colonne qui ne soit pas en prise, etc. Il se peut que certains choix ne permettent pas d'aboutir à placer la reine suivante. Alors, on remet en cause le dernier choix qui précède l'échec et l'on recommence. Si on épuise toutes les possibilités pour le dernier choix sans réussir, on remet en cause l'avant-dernier choix ... Cette technique de recherche s'appelle l'exploration avec retour-arrière (backtracking en anglais). On peut visualiser cette recherche d'une solution comme l'exploration d'un arbre dont certaines feuilles sont des solutions :



Arbre d'exploration des états

On part d'un état initial qui est la racine de l'arbre, les fils étant tous les choix possibles pour l'état suivant. Quand on arrive à un nœud d'échec, on remonte à son père et l'on repart vers un fils non encore exploré, s'il n'y en a pas on remonte au grand-père ...
La recherche d'un nœud par ce procédé s'appelle une exploration d'arbre en profondeur d'abord, ce sujet sera développé au chapitre 7.

Une solution, toutes les solutions, les solutions une à une

Pour programmer cette recherche, on définit la fonction `une-solution` qui, étant donné un état de l'exploration, retourne une solution *prolongeant* cet état ou `#f` s'il n'y en a pas. Son principe est simple :
- si l'on est dans un état final (c.-à-d. sans état suivant), soit c'est une solution et on la retourne, soit c'est un échec et on rend `#f` ;
- Si l'on est dans un état intermédiaire, on fait la liste de tous les états suivants et on lance une recherche à partir de chacun d'eux tant que l'on ne trouve pas de solution.

On suppose connue la fonction `Letats-suivants` qui donne la liste de tous les états pouvant succéder à un état donné et les prédicats `etat-final?, solution?`.

```
(define (une-solution etat)
  (if (etat-final? etat)
```

```
    (if (solution? etat) etat #f)
    (some une-solution (Letats-suivants etat))))
```

Dans certains cas on peut souhaiter obtenir la liste de *toutes* les solutions, c'est la fonction `Liste-des-solutions`. Au lieu de s'arrêter à la première solution, on concatène dans une liste les solutions correspondant à chaque état suivant possible :

```
(define (Liste-des-solutions etat)
  (if (etat-final? etat)
    (if (solution? etat) (list etat)  '())
    (append-map  Liste-des-solutions (Letats-suivants etat))))
```

Enfin, on peut préférer obtenir les solutions à la demande de l'utilisateur. C'est le rôle de la fonction `des-solutions`. Elle est basée sur le même principe que la fonction `une-solution` mais, quand on a trouvé une solution, on demande à l'utilisateur s'il veut en voir une autre. Pour forcer cette recherche, on retourne `#f` après avoir affiché la solution trouvée :

```
(define (des-solutions etat)
  (if  (etat-final? etat)
    (if (solution? etat)(autre-solution? etat) #f)
    (some   des-solutions  (Letats-suivants etat))))

(define (autre-solution? etat)
  (display-alln etat)
  (display "autre solution ? o/n : ")
  (eq? (read) 'n))
```

Pour chaque problème particulier, il suffira de définir les trois fonctions : `etat-final?`, `solution?`, `Letats-suivants`.

Application au problème des huit reines

Pour appliquer ces fonctions au puzzle des huit reines, il reste à modéliser le problème et écrire les fonctions `etat-final?`, `solution?`, `Letats-suivants`. On numérote les colonnes et les lignes de l'échiquier de 1 à 8. Comme il y aura une et une seule reine par colonne, il suffit de connaître le numéro de la ligne correspondant à chaque colonne. Aussi, un état sera une liste d'entiers `(x1 ...xk)` où `xi` est le numéro de la ligne de la ième reine située dans la ième colonne. Par exemple, l'état de la figure du début est représenté par la liste `(1 5 8 6 3 7 4)`.
On aura trouvé une solution quand l'état comportera huit reines et que chaque reine ne pourra pas être prise par une autre.

```
(define (etat-final? etat)
  (= (length etat) 8))

(define solution? etat-final?)
```

Pour calculer la liste des états suivant un état donné, on considère les huit positions possibles pour ajouter une reine dans la prochaine colonne et l'on ne garde que

les positions qui ne la mettent pas en prise avec les reines existantes dans l'état donné. On introduit le prédicat `admissible?` pour tester si une position pour une nouvelle reine est compatible avec les reines existantes. Autrement dit, si aucune des reines existantes n'est sur une diagonale ou une même rangée que la nouvelle reine. Pour écrire la fonction `admissible?` on utilise une fonction auxiliaire qui teste l'admissibilité de la nouvelle reine avec une reine à une distance donnée. Le paramètre `distance` représente la distance horizontale entre la nouvelle reine et une reine précédente.

```
(define (admissible? nlle-reine Lreines-existantes)
  (letrec ((admissible-aux?
             (lambda (Lreines-existantes distance)
               (if (null? Lreines-existantes)
                 #t
                 (let ((une-reine (car Lreines-existantes)))
                   (and (not (= une-reine  (+  nlle-reine distance)))
                        (not (= une-reine  nlle-reine))
                        (not (= une-reine  (-  nlle-reine distance)))
                        (admissible-aux? (cdr Lreines-existantes)
                                         (+ 1  distance))))))))
    (admissible-aux? Lreines-existantes 1)))
```

```
...   o   o   R   o   o

...   *   o   o   o   o

...   o   *   o   R   o

...   o   R   *   o   o

...   o   o   o   *   o

...   *   *   *   *   R   <--  nouvelle reine dans une position admissible

...   o   o   o   *   o

...   R   o   *   o   o
```

Pour construire la liste des états suivant un état donné, on considère chaque position admissible pour une nouvelle reine et on l'ajoute à l'état pour rendre finalement la liste de ces nouveaux états :

```
(define (Letats-suivants etat)
  (append-map
    (lambda (position)
      (if (admissible? position etat)
        (list (cons position etat))
        '()))
    (1-a-N  8)))
```

On a utilisé la fonction `1-a-N` définie au paragraphe précédent.

On teste nos diverses méthodes de résolution en prenant comme état initial, l'état vierge de toute reine. Voici une première solution :

```
? (une-solution '()) -> (4 2 7 3 6 8 5 1)
```

Si l'on préfère des solutions à la demande :

```
? (des-solutions '())
(4 2 7 3 6 8 5 1)
autre solution ?  o/n : o
(5 2 4 7 3 8 6 1)
autre solution ?  o/n : o
(3 5 2 8 6 4 7 1)
autre solution ?  o/n : n
#t
```

Enfin, si l'on désire connaître le nombre total de solutions à ce problème, il est donné par :

```
? (length  (liste-des-solutions '()))  -> 92
```

Remarque 3 Cette solution des huit reines, obtenue en appliquant une méthode très générale, est loin d'être la plus efficace. Comme c'est souvent le cas, on peut obtenir des solutions plus efficaces en tenant compte de la structure particulière du problème.

Exercice 22 *Ecrire une fonction qui affiche une solution avec le format de la figure du début.*

Architecture des solutions du problème des reines

`(une-solution etat)`
`(liste-des-solutions etat)`
`(des-solutions etat)`
Il y a trois fonctions principales, selon que l'on cherche une solution, la liste des solutions, les solutions à la demande. On se donne comme état initial un début de configuration possible (éventuellement réduit à une configuration vide).

> `(etat-final?  etat)`
> Prédicat pour tester si un état correspond à une position que l'on ne peut plus étendre.
>
> `(solution?  etat)`
> Prédicat pour tester si un état correspond à une solution (fonction identique à la précédente pour les reines).
>
> `(Letats-suivants etat)`
> La liste de tous les états immédiatement suivant admissibles. et possibles à partir d'un état donné.

`(1-a-N n)`
La liste des entiers de 1 à n.

`(admissible? nlle-reine Lreines-existantes)`
Prédicat pour tester si la position de la nouvelle reine n'est pas en prise par les reines placées dans les colonnes précédentes.

Heuristiques

L'efficacité de la recherche d'une solution est fortement dépendante de l'ordre dans lequel on considère les états suivants, autrement dit un rôle crucial est joué par la fonction `Letat-suivants`. Pour certains problèmes on dispose d'une *heuristique*. C'est une fonction `H` qui associe une valeur de plausibilité à un état. On a donc intérêt à effectuer la recherche en commençant par les états les plus plausibles. En conséquence, on s'arrange pour que la fonction `Letat-suivants` rende les états triés en ordre décroissant pour la relation d'ordre `etat1` $\geq$ `etat2` si `(H etat1)` $\geq$ `(H etat2)`. On a aussi intérêt à éliminer le plus tôt possible les états que l'on sait ne pouvoir conduire à une solution, cela joue directement sur le nombre de branches à explorer dans l'arbre de recherche.

Illustrons l'utilisation d'une heuristique par la recherche d'une solution au jeu du taquin. Il s'agit d'un damier de 3 sur 3 avec des cases mobiles numérotées de 1 à 8 et une case vide. Une case ne peut se déplacer que vers la case vide.

Exemple de configuration du jeu du taquin, ici seule une des cases 7 , 6 , 5 peut être déplacée sur la case vide

2	8	3
1	6	4
7		5

`Un état initial du jeu du taquin`

Le but du jeu est de transformer une configuration en la configuration suivante:

1	2	3
8		4
7	6	5

`Etat final du jeu du taquin`

A chaque étape on choisit une des cases à transférer sur la case vide. Selon la position de la case vide il y 2 ou 3 ou 4 choix possibles. Pour essayer de guider le choix vers la solution, on définit l'heuristique suivante. Etant donné une configuration `C`, on définit le nombre `(H C)` comme étant le nombre de cases mal placées ; il s'agit donc de choisir en priorité un mouvement qui amène à une configuration minimisant `H`.

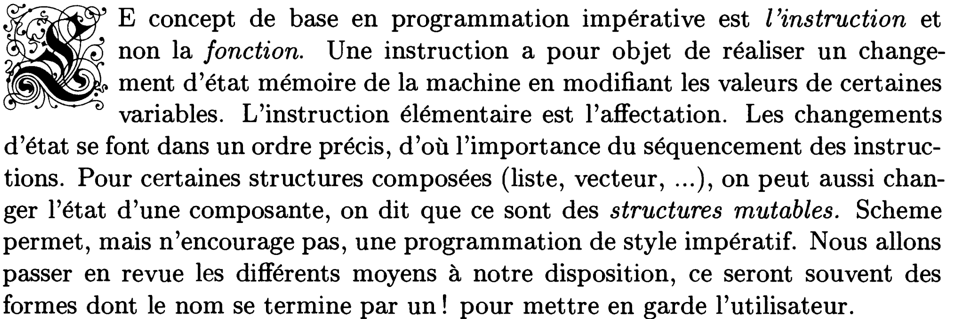
Exercice 23 *Choisir une représentation de ce jeu et écrire une fonction donnant une suite de mouvements pour atteindre le but à partir d'une configuration quelconque.*

3.11 De la lecture

Deux classiques sur la programmation fonctionnelle [Bur75, Hen80].
Pour programmer avec le langage fonctionnel ML [Pau91, WL93, CM95].
Pour la notion d'heuristique et le jeu du taquin, on peut consulter [Nil92, CM95].

Chapitre 4

Programmation impérative



LE concept de base en programmation impérative est *l'instruction* et non la *fonction*. Une instruction a pour objet de réaliser un changement d'état mémoire de la machine en modifiant les valeurs de certaines variables. L'instruction élémentaire est l'affectation. Les changements d'état se font dans un ordre précis, d'où l'importance du séquencement des instructions. Pour certaines structures composées (liste, vecteur, ...), on peut aussi changer l'état d'une composante, on dit que ce sont des *structures mutables.* Scheme permet, mais n'encourage pas, une programmation de style impératif. Nous allons passer en revue les différents moyens à notre disposition, ce seront souvent des formes dont le nom se termine par un ! pour mettre en garde l'utilisateur.

4.1 L'affectation en Scheme

La forme spéciale `set!`

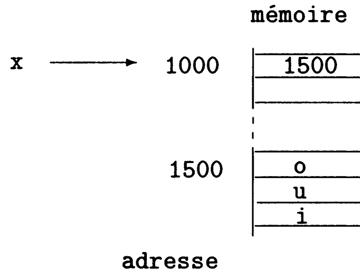
Commençons par l'effet de bord de base : c'est la forme spéciale `set!` qui réalise l'affectation en Scheme. Sa syntaxe est :

(`set!` *variable expression*)

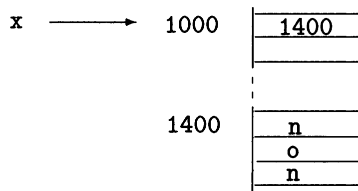
L'évaluation de cette forme a pour effet de *changer* la valeur associée à la variable dans *l'environnement courant* et de lui associer la valeur de l'expression ; la valeur retournée est indéfinie. Cela suppose donc qu'une valeur a déjà été associée à cette variable.

```
? (define x 'oui)

? x
oui
```

Pour ceux qui ont lu le chapitre 2 §10, voici la représentation (schématique) en mémoire de l'action de `set!` sur la variable `x` dont l'adresse est par exemple 1000 :




```
? (set! x  'non)
```




```
? x
non
```

Illustrons l'utilisation de la forme `set!` à la permutation des valeurs associées aux variables globales `x` et `y`. On sauvegarde la valeur de `x` en utilisant une variable auxiliaire `aux`

```
? (define  y  'ok)
? (define  aux   x)
? (set! x y)
? (set! y aux)

? x -> ok
? y -> non
```

On peut aussi utiliser `set!` pour changer la valeur d'une variable locale :

```
? (let ((a 10))
     (set! a 0)
     a)
0
```

Exercice 1 *1. On considère la session suivante :*

```
? (define u '(a b c))
? (define v  (cdr u))
? (set! u  (cons 'd u))
```

Que pensez-vous de la valeur de v ?

2. *Pouvez-vous écrire une fonction qui a pour effet de permuter les valeurs de deux variables globales ?*

On doit utiliser `set!` avec précaution et parcimonie, car cela peut faire perdre une propriété fondamentale de la programmation fonctionnelle : une même expression prend dans les mêmes conditions toujours la même valeur. Considérons les définitions

```
? (define a 1)

? (define (f x)
    (set! a (+ a x))
    (* a x))
```

alors la valeur de `(f 1)` change à chaque appel :

```
? (f 1) -> 2
? (f 1) -> 3
```

Bien que non terminée par un `!`, la forme `define` réalise aussi un effet de bord quand on redéfinit une fonction. C'est grâce à cette propriété que le programmeur Scheme a droit à l'erreur : il peut revenir sur des définitions antérieures *sans avoir à tout réévaluer.*

Une application de set! au comptage des appels

Il y a des cas où l'utilisation d'un effet de bord peut être commode. Supposons que nous voulions compter le nombre d'appels à une fonction donnée `f` (prédéfinie ou non) pendant l'évaluation d'une expression. On introduit une variable globale `*nb-appel-de-f*` initialisée à 0 et on modifie la définition `f` pour introduire dans son corps une incrémentation du compteur. Ne pas oublier de faire une sauvegarde de la définition initiale de `f`. On donnera une autre méthode plus satisfaisante au chapitre 6 §5.
Appliquons ce principe au calcul du nombre d'appels à `cons` :

```
(define *nb-appel-de-cons* 0)
(define old-cons cons)

(define (cons s1 s2)
  (set! *nb-appel-de-cons* (+ 1 *nb-appel-de-cons*))
  (old-cons s1 s2))
```

Comptons le nombre d'appels à `cons` provoqué par l'exécution de la fonction d'insertion suivante. On donne une liste croissante de nombres, `inserer` retourne la liste où l'on a inséré à sa place un nombre donné.

```
(define (inserer nb L)
  (cond ((null? L)(cons nb L))
        ((< nb (car L))(cons nb L))
        (else (cons (car L)(inserer nb (cdr L))))))
```

Après évaluation de :

```
? (inserer 5 '(2 4 7  7 11  15)) -> (2 4 5 7 7 11 15)
```

on trouve

```
? *nb-appel-de-cons* -> 3
```

A la fin on restaure la définition de cons :

```
(define cons old-cons)
```

4.2 Itération et récursivité terminale

Une boucle while

L'existence du set! permet aux amateurs de Pascal de programmer la factorielle comme si l'on avait un while en Scheme :

```
(define (fac n)
  (let ((i n)
        (f 1))
    (letrec ((while (lambda ()
                      (if (= i 0)
                        f
                        (begin (set! f (* f i))
                               (set! i (- i 1))
                               (while))))))
      (while))))
```

Les variables locales i et f servent respectivement de compteur et d'accumulateur. On peut écrire un algorithme analogue dans un pur style Scheme en passant les variables i et f en paramètre du while :

```
(define (fac n)
  (letrec ((while (lambda (i f)
                    (if (= i 0)
                      f
                      (while (- i 1) (* f i))))))
    (while n 1)))
```

L'élimination de la récursivité terminale nous assure que ce while s'exécute bien comme une itération. Le langage Scheme permet donc d'allier l'élégance de la programmation récursive avec l'efficacité de l'itération.

On verra au chapitre 9 §3 que, si on le souhaite, on peut ajouter à Scheme une boucle while de syntaxe (while s0 s1 ...sN). La valeur de cette forme est :

- si la valeur de s0 est vraie, on évalue en séquence les formes s1...sN et l'on réévalue la forme ;
- si la valeur de s0 est fausse, on ne fait rien et on retourne une valeur indéfinie.

Transformation d'une forme récursive en forme itérative

Voici un exemple de «dérécursivation» utilisant la boucle `while`. Considérons la définition récursive suivante, où l'on suppose que *f* n'apparaît pas ailleurs que dans l'appel récursif qui est donc terminal.

```
(letrec ((f (lambda (x)
              (if exp0
                exp1
                (f exp2)))))
  (f exp))
```

On sait que Scheme exécutera l'appel de *f* de façon itérative, mais dans ce cas très simple on peut *expliciter* une forme itérative équivalente en utilisant la boucle `while`. En effet, il n'est pas très difficile de se convaincre[1] que l'expression suivante calcule la même valeur :

```
(let ((x  exp))
  (while (not exp0)
         (set! x exp2))
  exp1)
```

Exercice 2 *Etendre cette transformation au cas d'une fonction f de plusieurs variables et l'appliquer à la fonction récursive terminale* `fac-aux` *du chapitre 3 §9.*

Par exemple, ce schéma nous permet de transformer la définition récursive de la fonction qui calcule le dernier élément d'une liste non vide :

```
(define (dernier L)
  (if (null? (cdr L))
    (car L)
    (dernier (cdr L))))
```

en une définition à la Pascal avec une boucle `while` !

```
(define (dernier-it L)
  (while (not (null? (cdr L)))
         (set! L (cdr L)))
  (car L))
```

La boucle do

La programmation récursive permet d'écrire n'importe quelle boucle, cependant il peut être commode d'utiliser une boucle pour alléger le code d'une fonction ou pour attirer l'attention sur l'utilisation dans le corps d'effets de bord. Pour cela, le langage Scheme met à notre disposition la forme `do` de syntaxe :

[1] On pourra raisonner par récurrence sur le nombre d'appels récursifs utilisés pour calculer `(f exp)`.

```
(do ((var1 init1  incr1)
      ...
     (varN initN  incrN))
    (test-fin  [resultat])
  corps)
```

Son évaluation consiste à initialiser en parallèle les variables locales ***var1...varN*** avec les valeurs des expressions ***init1...initN***. Puis, si l'expression ***test-fin*** a pour valeur `#f`, on exécute le ***corps*** et sinon on retourne la valeur de l'expression facultative ***resultat*** (ou indéfinie si elle n'est pas fournie). Ensuite, on recommence après avoir donné à chaque variable ***varj*** la valeur de l'expression ***incrj***. Bien entendu, cette boucle n'est pas indispensable, on réalise le même résultat avec une définition récursive terminale de la forme :

```
(letrec ((mon-do (lambda (var1 ... varN)
                (if (not test-fin)
                  (begin corps
                         (mon-do incr1 ... incrN))
                  [resultat]))))
   (mon-do init1 ... initN))
```

A titre d'exemple, écrivons une fonction qui affiche la table de multiplication par le chiffre `x` :

```
? (define (table x)
    (do ((i 1 (+ i 1)))
        ((= i 10))
      (display-alln x " x " i " = " (* i x))))


? (table 7)
7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
7 x 4 = 28
7 x 5 = 35
7 x 6 = 42
7 x 7 = 49
7 x 8 = 56
7 x 9 = 63
```

Exercice 3 *Ecrire une fonction qui calcule le maximum d'une liste de nombres et retourne le doublet constitué de ce maximum et de l'indice du premier élément qui le réalise. On comparera deux styles de programmation : un parcours de la liste avec une boucle* `do` *et un autre avec une fonction récursive terminale.*

4.3 La liste comme structure mutable

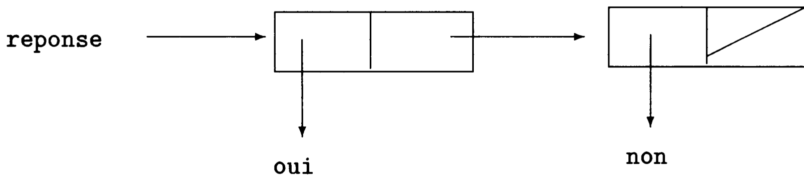
Les fonctions `set-car!` et `set-cdr!`

Quand une variable représente une structure composée, on peut vouloir modifier des composantes de cette structure. C'est possible avec la structure de doublet

grâce aux fonctions de modification `set-car!` et `set-cdr!`.
L'appel de `(set-car! exp s)` présuppose que la valeur de l'expression `exp` est un doublet, alors le `car` de ce doublet est remplacé par la valeur de l'expression `s`, la valeur rendue est indéfinie.
De même pour `(set-cdr! exp s)` qui change la valeur du `cdr`.
Illustrons ces transformations par un schéma de boîtes :

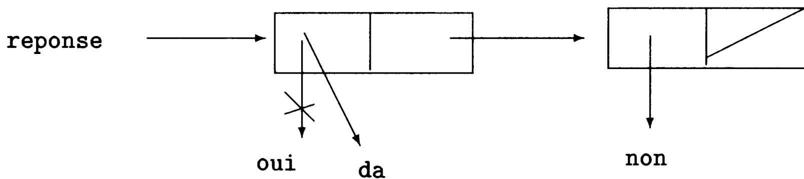
```
? (define  reponse (list 'oui 'non))
```

On n'a pas défini cette liste par une quotation `'(oui non)` car Scheme considère comme une *constante* une liste définie par quotation, aussi la norme Scheme n'autorise pas de la modifier ensuite par `set-car!` ou `set-cdr!` .



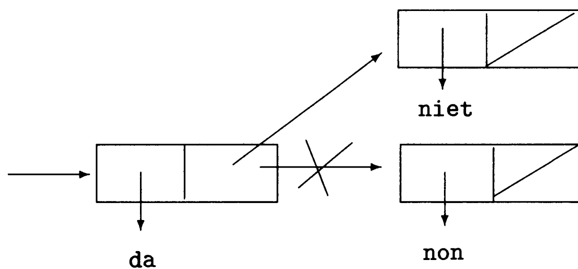

```
? (set-car! reponse 'da)
```

Après ce `set-car!`




```
? reponse
(da  non)

? (set-cdr! reponse '(niet))
```

Après ce `set-cdr!`



```
? reponse
(da  niet)
```

On peut définir une fonction qui remplace le car et le cdr d'un doublet par le car et le cdr d'un autre :

```
? (define (rplac  doublet1 doublet2)
    (set-car! doublet1 (car doublet2))
    (set-cdr! doublet1 (cdr doublet2)))
```

Pour tester, définissons le doublet :

```
? (define  answer (list 'yes  'no))  %%%
```

après

```
? (rplac reponse answer)
```

on a, malgré le passage des paramètres par valeur,

```
? reponse
(yes  no)
```

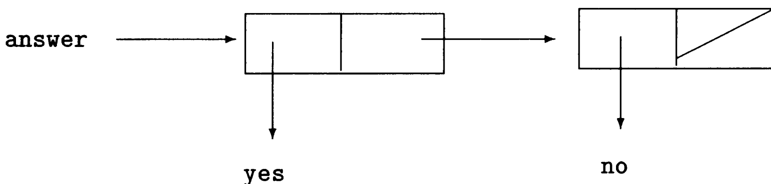
car ce sont les composantes du doublet answer qui ont été modifiées.

Quelques utilitaires de modifications physiques des listes

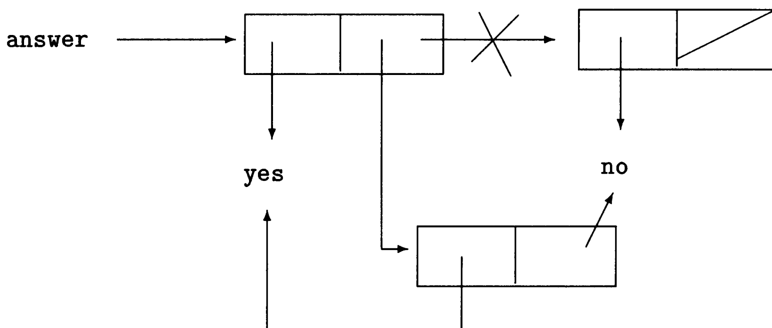
Avec les primitives set-car! et set-cdr!, on peut se fabriquer un petit arsenal de fonctions pour la modification physique des listes.
La fonction AjouterEnTete! ajoute physiquement en tête d'une liste *non vide* L la valeur de s :

```
? (define (AjouterEnTete! s L)
    (set-cdr! L (cons (car L)(cdr L)))
    (set-car! L s)
    L)

? (AjouterEnTete!  'maybe answer)
? answer
(maybe  yes  no)
```

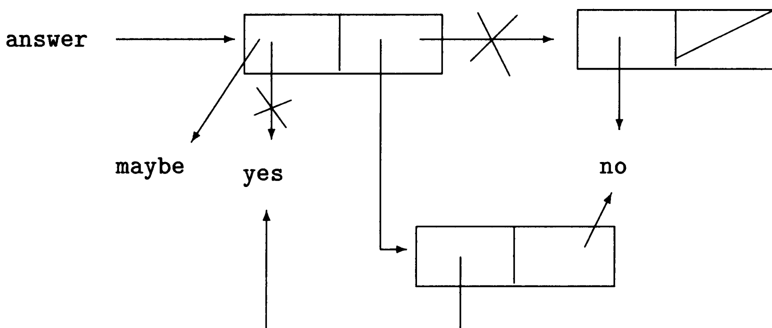
Pour bien comprendre ces modifications, le mieux est de faire les diagrammes des boîtes. Au début, la liste answer est représentée par :



Après le `set-cdr!` on a créé un doublet `(yes . no)` et fait pointer le `cdr` de `answer` dessus :



Après le `set-car!` le `car` de `answer` pointe bien sur le symbole `maybe` :




```
? answer -> (maybe yes no)
```

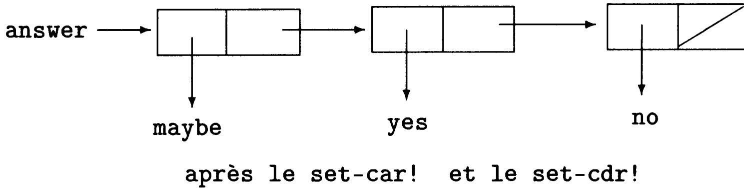
Voyons maintenant l'opération inverse, c'est la fonction `SupprimerTete!`. Elle supprime physiquement le premier élément d'une liste `L` *ayant au moins deux éléments.*

Comme on ne peut supprimer le doublet sur lequel pointe `L`, on copie le deuxième doublet dans le premier puis l'on supprime le deuxième. On doit donc faire pointer le `car` de `L` sur le `car` du `cdr` et faire pointer le `cdr` de `L` sur le `cddr`.

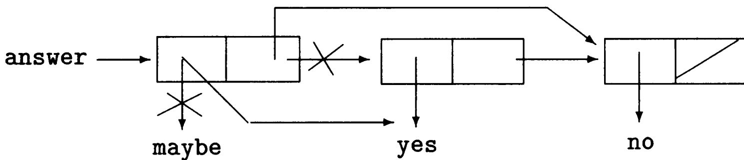
```
(define (SupprimerTete! L)
  (set-car! L (cadr L))
  (set-cdr! L (cddr L))
  L)
```

Là encore, un petit croquis vaut mieux qu'un long discours.

```
? (supprimeTete!  answer)
```



après le set-car! et le set-cdr!

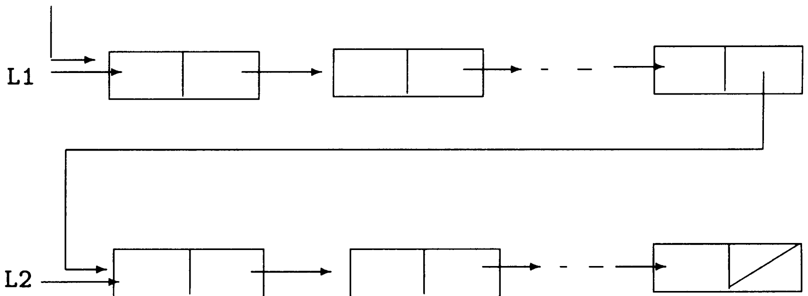



```
?  answer -> (yes no)
```

Les fonctions append et append!

Considérons la concaténation physique de deux listes L1 et L2, une méthode qui saute aux yeux consiste à faire pointer le dernier cdr de L1 vers L2; appelons append2! cette fonction.

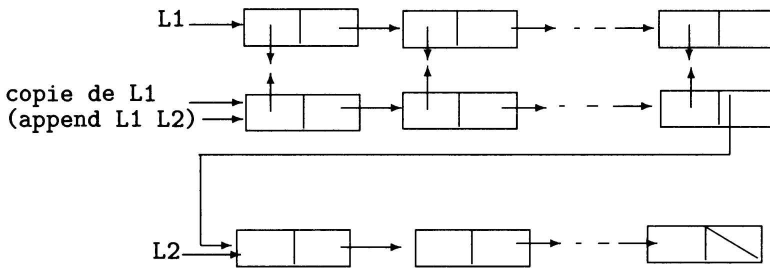
(append2! L1 L2)



Pour écrire la définition de append2!, on utilise la fonction last-pair qui retourne le dernier doublet d'une liste non vide et qui a eu l'honneur de faire partie des fonctions essentielles des premières définitions de Scheme:

```
(define (append2! L1 L2)
  (cond ((null? L1) L2)
        (else (set-cdr! (last-pair L1) L2) L1)))

(define (last-pair L)
  (if (or (null? L)(null? (cdr L)))
    L
    (last-pair (cdr L))))
```

Cette méthode permet de construire la concaténation de deux listes sans avoir à créer un seul doublet. En revanche, elle a un gros défaut : elle modifie physiquement la liste L1. Elle ne devra donc être utilisé que si l'on est certain de ne plus avoir besoin de l'ancienne valeur de L1 (mais attention d'autres listes peuvent partager cette valeur, voir chapitre 2 page 56). C'est la raison pour laquelle la fonction prédéfinie append réalise cette même transformation à partir d'une *copie* de L1. Elle doit donc créer autant de doublets qu'en comporte L1 ce qui est parfois très coûteux, et explique la réticence des programmeurs Scheme vis-à-vis de append. Voici une représentation schématique du fonctionnement de append :



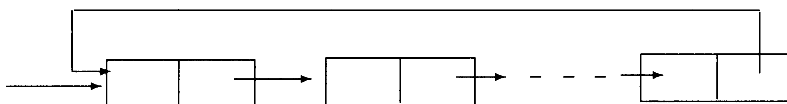
Remarque 1 Le fonctionnement de append montre que l'on obtient une *copie* d'une liste L par (append L '()).

Exercice 4 *Définir la fonction* **append!**, *analogue de* **append2!**, *mais acceptant un nombre arbitraire d'arguments.*

Voyons maintenant le cas de l'ajout physique d'un élément en *fin* de liste. Il est maintenant facile d'écrire une fonction qui ajoute physiquement un objet en fin d'une liste non vide : on fait la concaténation physique de la liste avec celle réduite à cet objet :

```
(define (append1! L s)
  (append2! L (list s)))
```

L'usage des modifications physiques peut aboutir à créer des objets non imprimables. Par exemple, on peut transformer une liste en une *liste circulaire* en l'ajoutant physiquement à elle-même :



Liste circulaire

```
? (define liste-circulaire  (append2! reponse reponse))
```

Si on demande sa valeur, l'imprimeur boucle[2] :

```
? liste-circulaire -> (yes no yes no yes no yes no yes no yes no ...
```

De façon plus générale, l'usage des modifications physiques dans les listes peut conduire à des erreurs difficiles à trouver aussi conseille-t-on de n'y recourir qu'en cas de réelle nécessité. Scheme est avant tout destiné à la programmation fonctionnelle mais la possibilité d'utiliser des traits impératifs comme l'affectation le classe dans la catégorie des langages dits *applicatifs*.

Exercice 5 1. *Ecrire une fonction qui remplace physiquement le kième élément d'une liste par un objet donné.*

2. *Ecrire une fonction qui insère physiquement un objet donné après le kième élément d'une liste.*

3. *Ecrire une fonction qui supprime physiquement le kième élément d'une liste de longueur > 1.*

Exercice 6 *Voici une fonction du folklore Lisp inspirée de la fable du lièvre et de la tortue. Elle sert à tester si une liste comporte une boucle au premier niveau :*

```
(define (liste-circulaire? L)
  (letrec ((boucle?
             (lambda(tortue lievre)
               (and (pair? tortue)
                    (pair? lievre)
                    (pair? (cdr lievre))
                    (or (eq? tortue lievre)
                        (boucle? (cdr tortue) (cddr lievre)))))))
    (and (pair? L)
         (boucle? L (cdr L)))))
```

1. *Prouver que cette fonction rend* `#t` *ssi la liste* `L` *se termine par une boucle. (Indication : le lièvre court deux fois plus vite et part en avance mais, si la liste se termine par une boucle, sa position coincidera avec celle de la tortue après un certain nombre de tours du lièvre.)*

2. *En déduire un test pour détecter si une liste comporte une boucle à un niveau quelconque.*

3. *Donner une fonction qui calcule la longueur d'une liste ou retourne le symbole* `cycle` *si la liste comporte une boucle (indication : voir [Ste90] 15.2).*

[2] Sauf si l'implantation a prévu une méthode d'affichage des listes circulaires.

4.4 Caractères et chaînes

Représentation des caractères

Les caractères sont affichés avec la notation `#\` précédant le symbole du caractère : `#\a` `#\;` `#\)`. Certains caractères possèdent un nom : `#\space` pour l'espace, `#\newline` pour le passage à la ligne, `#\tab` pour la tabulation.
Les caractères satisfont au prédicat `(char? s)`, on peut les désigner sans les quoter. On a la correspondance avec les codes ASCII[3] par les fonctions de conversion `char->integer` et `integer->char` :

```
? (char->integer  #\a) -> 97
? (integer->char  65) ->  #\A
```

Quelques fonctions de base sur les chaînes

Une chaîne est une suite finie de caractères. On accède directement à un caractère d'indice donné par la fonction `(string-ref` ***str indice*** `)`. La numérotation des caractères commence à 0. La longueur d'une chaîne est donnée par la fonction `string-length`.

```
? (define  une-chaine  "bonjour")
? (string-ref une-chaine   (+ 1 2) ) ->  #\j
? (string-length  une-chaine) -> 7
```

Les chaînes sont des structures mutables, on peut remplacer physiquement le kième caractère d'une chaîne par la valeur de l'expression `s` en utilisant la fonction `(string-set! str k s)` :

```
? (string-set!  une-chaine   2  #\u)
? une-chaine   -> "boujour"
```

On dispose de nombreuses méthodes pour construire une chaîne [4] :

- `(make-string long [carac])` → une chaîne de longueur `long` et n'utilisant que le caractère `carac` s'il est donné :

  ```
  ? (make-string 5 #\a) -> "aaaaa"
  ```

- `(string carac1 ...)` → une chaîne composée des caractères donnés :

  ```
  ? (string  #\a  #\b  #\space  #\c)
   "ab c"
  ```

- `(string-append str1 ...strN)` → la chaîne résultant de la concaténation des chaînes données.

[3] *Americain Standard Code for Information Interchange.*
[4] On renvoie à la norme Scheme pour l'intégralité des fonctions sur les chaînes.

Chaînes et symboles

On peut convertir une chaîne en un symbole et vice versa :

`(symbol->string symb)` → la chaîne des caractères constituant le symbole donné
`(string->symbol str)` → le symbole formé par les caractères de la chaîne.

Illustrons ces diverses constructions par des fonctions simples sur les chaînes. Ces fonctions effectuent un parcours de la chaîne et réalisent des effets de bord, aussi il est naturel d'employer la boucle `do`.
Ecrivons une fonction `upcase!` qui rend une *nouvelle* chaîne où tous les caractères alphabétiques sont mis en majuscule. On utilise la primitive `(char-upcase carac)` qui retourne la majuscule associée à un caractère alphabétique et laisse inchangés les autres.

```
(define (upcase! str-in)
  (let ((Long (string-length str-in)))
    (let ((str-out (make-string Long)))
      (do ((index 0 (+ 1 index)))
          ((= index Long) str-out)
        (string-set! str-out index
                     (char-upcase (string-ref str-in index)))))))
```

Ecrivons une fonction `downcase!` qui transforme tous les caractères alphabétiques en minuscules en *modifiant physiquement* la chaîne donnée. Elle utilise la primitive `char-downcase` qui retourne la minuscule associée à un caractère alphabétique et laisse inchangés les autres.

```
(define (downcase! str-in)
  (let ((Long (string-length str-in)))
    (do ((index 0 (+ 1 index)))
        ((= index Long) str-in)
      (string-set! str-in index
                   (char-downcase (string-ref str-in index))))))

? (define  ch  " a + a = 2A")
? (Upcase  ch) -> " A + A = 2A"
? (downcase! ch) -> " a + a = 2a"
? ch -> " a + a = 2a"
```

En combinant concaténation et conversion entre symbole et chaîne, on peut construire de nouveaux symboles. Voici une fonction qui ajoute un préfixe et un suffixe à un symbole :

```
? (define (ajouter-prefixe-suffixe prefixe-str symb suffixe-str)
    (string->symbol (string-append prefixe-str (symbol->string symb)
                                               suffixe-str)))

? (ajouter-prefixe-suffixe "set-" 'car "!") -> set-car!
```

L'évaluation d'une expression Scheme retourne toujours une valeur, mais pour des raisons d'affichage on a parfois besoin d'une évaluation silencieuse, c'est-à-dire qui *n'affiche rien en retour.* Pour cela, une méthode consiste à transformer la chaîne vide en un symbole, ce symbole vide ne provoquera aucun affichage :

```
? (string->symbol "") ->
```

Conversion d'une chaîne en une liste de mots

Voici une fonction plus complexe, elle retourne la liste des mots qui composent une chaîne donnée. On appelle *mot* une suite maximale de caractères sans séparateur. On convient d'appeler *séparateur* les caractères : espace, passage à la ligne, tabulation ou un caractère de ponctuation.
La variable locale `index` est partagée par les fonctions `loop`, `lireMot` et `lireSeparateur`, elle sert à décrire le parcours de la chaîne. La fonction `loop` accumule les mots dans la liste `Lmots`. Selon la nature du caractère courant, on procède à la lecture du mot suivant ou des séparateurs. Quand on a atteint le dernier caractère de la chaîne, la fonction `loop` rend la liste renversée pour les remettre dans le bon ordre.
La fonction `lireMot` concatène les caractères successifs et retourne le symbole associé dès que l'on rencontre un séparateur ; elle actualise également la valeur de l'index.
Enfin, la fonction `lireSeparateur` sert à sauter les séparateurs et actualise en même temps l'index.

```
(define (chaine->ListeMots phrase-str)
  (let ((index 0)
        (Longueur (string-length phrase-str))
        (Lseparateurs
          '(#\space #\newline  #\, #\; #\. #\: #\? #\! #\")))
    (letrec ((loop (lambda (Lmots)
                     (if (= Longueur index)
                       (reverse Lmots)
                       (let ((c (string-ref phrase-str index)))
                         (if (separateur? c)
                           (begin (lireSeparateur) (loop Lmots))
                           (loop (cons (lireMot (string c)) Lmots)))))))

             (lireMot (lambda (str)
                        (set! index (+ index 1))
                        (if (= Longueur index)
                          (string->symbol str)
                          (let ((c (string-ref phrase-str index)))
                            (if (separateur? c)
                              (string->symbol str)
                              (lireMot (string-append str (string c)))
                              )))))

             (separateur? (lambda (c)(memv c Lseparateurs)))

             (lireSeparateur
               (lambda ()
                 (set! index (+ index 1))
                 (if (= Longueur index)
                   #f
```

```
                (if (separateur? (string-ref phrase-str  index))
                  (lireSeparateur)))))
      (loop '()))))
```

```
? (chaine->ListeMots
   "Innombrables sont nos voies et nos demeures incertaines")[5]
(innombrables sont nos voies et nos demeures incertaines)
```

Une petite fonction format

On a déjà défini l'extension display-all de display, voici une autre extension qui est une version très simplifiée de la fonction **format** de Common Lisp. Elle prend en argument une chaîne dite **chaine-de-formatage** et des arguments (format chaine-de-formatage arg1 argt2 ...).
Voici un exemple d'utilisation de **format** :

```
? (format "affichage de  ~~ avec format ~% x = ~s " (+ 1 2))
 affichage de  ~ avec format
 x = 3
```

La chaîne de formatage a le comportement suivant : elle est affichée telle quelle sans les guillemets *sauf* pour les caractères précédés de ~, dans ce cas on discute selon le caractère qui suit :

- la séquence ~s indique un emplacement où sera affichée par write la valeur de l'argument correspondant ;
- la séquence ~a indique un emplacement où sera affichée par display la valeur de l'argument correspondant ;
- la séquence ~% provoquera un passage à la ligne ;
- la séquence ~~ sert à afficher le caractère ~.

La valeur finale retournée est indéfinie.
La méthode consiste à parcourir la chaîne de formatage et à afficher chaque caractère non précédé de ~. Quand on rencontre ~ , on fait un traitement au cas par cas. Le paramètre i de la fonction format-aux désigne l'indice du caractère courant :

```
(define (format  chaine . Largts)
   (let ((Longueur (string-length chaine)))
     (letrec ((format-aux
                (lambda (i Largts)
                  (if (= i Longueur )
                    'indefinie
                    (let ((c (string-ref chaine i)))
                      (if (eq? c #\~)
                        (case (string-ref chaine (+ 1 i))
```

[5] Saint-John Perse, *Pluies*.

```
                        ((#\a #\A) (display (car Largts))
                         (format-aux (+ i 2) (cdr Largts)))
                        ((#\s #\S) (write (car Larg))
                         (format-aux (+ i 2) (cdr Largts)))
                        ((#\%) (newline)(format-aux (+ i 2) Largts))
                        ((#\~) (display #\~)(format-aux (+ i 2) Largts))
                        (else (display "caractere non prevu")))
                      (begin (display c)(format-aux (+ i 1) Largts))
           ))))))
     (format-aux 0 Largts))))
```

Exercice 7 *1. Ecrire une fonction pour tester l'égalité de deux chaînes en ne faisant pas la distinction entre majuscule et minuscule.*

2. Ecrire une fonction pour renverser physiquement une chaîne.

3. Ecrire un prédicat pour comparer les chaînes dans l'ordre lexicographique (celui du dictionnaire). On se basera sur l'ordre `char<?` *défini sur les caractères. En fait, ce prédicat correspond à la fonction prédéfinie* `string<?`.

4.5 Vecteurs

La structure de vecteur généralise celle de chaîne. Un vecteur est une suite finie d'objets. Les composantes sont numérotées à partir de 0 et peuvent être de nature hétérogène.

Accesseurs et modificateurs de vecteurs

Contrairement aux listes, on accède à temps constant à une composante de numéro donné `k` d'un vecteur, c'est la fonction `(vector-ref vecteur k)`.
L'affichage d'un vecteur se fait comme pour une liste mais avec un # au début, on peut se donner explicitement un vecteur en le quotant :

```
? '#(2  a   (b c) ) -> #(2 a (b c))
? (vector-ref  '#(2  a  (b c) 1 ) -> a
```

Les vecteurs sont des structures mutables, on peut modifier physiquement la kième composante d'un vecteur en y plaçant la valeur d'une expression `s` au moyen de la procédure `(vector-set!  vecteur k s)`.
Par exemple, on peut échanger physiquement deux composantes d'indice `i  j` d'un vecteur par :

```
(define (echanger! vect i j)
  (let ((x (vector-ref vect i)))
    (vector-set! vect i (vector-ref vect j))
    (vector-set! vect j x)))

? (define  V '#(a b 1 2))
? (echanger! V 0 3)
? V -> #(2 b 1 a)
```

Remarque 2 Si l'on doit représenter des objets structurés de taille connue, il faut penser à utiliser les vecteurs, mais si la taille peut changer alors les listes s'imposent.

Constructeurs de vecteurs

On dispose de nombreuses méthodes pour construire un vecteur :

- `(make-vector n [s] )` → un vecteur de longueur n composé de n fois la valeur de s si une expression s est donnée :

```
? (make-vector 4 '(a b)) -> #((a b) (a b) (a b) (a b))
```

- L'analogue de la fonction `list` pour les vecteurs s'appelle `vector`. Elle retourne un nouveau vecteur dont les composantes sont les valeurs des expressions passées en arguments :

```
? (vector  (* 2 2) '(b c)  '#(0 1)) -> #(4 (b c) #(0 1))
```

On a des fonctions de conversion entre listes et vecteurs : `(vector->list vecteur)` et `(list->vector list)` :

```
?  (vector->list   '#(4 (b c) #(0 1))) -> (4 (b c) #(0 1))
?  (list->vector   '(4 (b c) #(0 1))) -> #(4 (b c) #(0 1))
```

Programmation avec les vecteurs

Quand un vecteur est créé, il conserve ensuite sa longueur. Elle est donnée par la fonction `vector-length`. Aussi, la programmation avec les vecteurs conduit souvent à utiliser des boucles ayant un nombre de pas connu à l'avance. Donnons quelques fonctions simples pour manipuler les vecteurs.

Construisons un sous-vecteur d'un vecteur donné ; il sera constitué par les composantes d'indice entre `debut` et `fin` d'un vecteur donné, on suppose que :
$0 \leq$ `debut` $\leq$ `fin` $<$ `(vector-length vecteur)`

```
(define (sous-vecteur v debut fin)
  (let ((v-copie (make-vector (+ 1 (- fin debut)))))
    (do ((i debut (+ 1 i)))
        ((< fin i) v-copie)
      (vector-set! v-copie (- i debut) (vector-ref v i)))))

? (sous-vecteur (vector 1 'a  "oui"  '(u v)) 1 2)  -> #(a "oui")
```

Pour renverser un vecteur, il suffit d'échanger les éléments en position symétrique. Dans le cas d'un nombre impair d'éléments, l'élément central est inchangé. Le nombre d'échanges est donc le quotient entier de la longueur par 2.

```
(define (renverser-vect! vect)
  (do ((i 0 (1+ i))
       (j (- (vector-length vect) 1) (- j 1)))
      ((<= j i) vect)
    (echanger! vect i j)))

? (renverser-vect! '#(1 2 3 4 5 6 7 8 9)) -> #(9 8 7 6 5 4 3 2 1)
```

On peut définir l'analogue de `map` pour les vecteurs : on applique une fonction à chaque composante et l'on rend le vecteur des valeurs. Si la fonction prend plusieurs arguments, on se donne autant de vecteurs de même longueur.

```
(define (map-vector f . Lvecteurs)
  (let* ((L (vector-length (car Lvecteurs)))
         (vecteur-out (make-vector L)))
    (do ((i 0 (+ 1 i)))
        ((= i L) vecteur-out)
      (vector-set! vecteur-out i (apply f (map (lambda (v)(vector-ref v i))
                                                Lvecteurs))))))

? (map-vector + '#(1 2 3) '#( 4 5 6)) -> #(5 7 9)
```

Exercice 8 *Ecrire une fonction* `for-each-vect` *qui prend les mêmes arguments que* `map-vector` *et applique la fonction f à chaque composante puis rend un résultat indéfini.*

Exercice 9 *Ecrire une fonction qui, étant donné une chaîne, retourne un vecteur à 26 composantes, la ième composante correspond au nombre d'apparitions de la ième lettre de l'alphabet dans la chaîne.*

4.6 Compléments : une représentation pleine des polynômes

On a exposé en complément au chapitre 2 une représentation des polynômes qui était particulièrement bien adaptée au cas des polynômes creux. Dans le cas des polynômes pleins, c'est-à-dire où il n'y pas trop de différence de degré entre les monômes qui les composent, on va donner une nouvelle représentation basée sur les vecteurs.

Représentation interne

On continue à ne considérer que des polynômes à une variable, disons `X`. On représente un polynôme par le vecteur de ses coefficients par ordre de degré croissant :

$2 + 5X + 4X^2 + 7X^4$ est représenté par le vecteur `#( 2 5 4 0 7)`

Le polynôme nul est représenté par le vecteur `'#()`. D'où immédiatement les fonctions :

```
(define zero-poly '#())

(define (zero-poly? p)
  (equal? p zero-poly))

(define (degre-poly p)
  (if (zero-poly? p)
    'indefinie
    (- (vector-length p) 1)))
```

On associe à un vecteur le polynôme obtenu en supprimant les composantes qui sont nulles à partir d'un certain rang.

```
(define (vect->poly vect)
  (let ((degre (degre-vect vect)))
    (if (= -1 degre)
      '#()
      (sous-vecteur vect 0  degre))))

? (vect->poly '#(1 2 0 3 0 0 0)) -> #(1 2 0 3)
```

Où la fonction degre-vect donne l'indice de la dernière composante non nulle d'un vecteur (ou -1 s'il n'y en a pas) :

```
(define (degre-vect vect)
  (let ((L (vector-length vect)))
    (do ((i (- L 1) (- 1 i)))
        ((or (< i 0)(not(zero? (vector-ref vect i))))   i)
     )))

? (degre-vect '#(1 2 0 3 0 0 0)) -> 3
```

Inversement, on associe à un polynôme un vecteur de longueur donnée, en complétant par des zéros les composantes d'indices supérieurs au degré du polynôme :

```
(define (poly->vect poly Long)
  (let ((taille (vector-length poly))
        (vect (make-vector Long 0)))
    (do ((i 0 (+ 1 i)))
        ((= i taille) vect)
      (vector-set! vect i (vector-ref poly i)))))

? (poly->vect '#(1 2 0 3)  7) -> #(1 2 0 3 0 0 0)
```

Opérations sur les polynômes

Avec ces utilitaires, on définit les opérations d'addition et de multiplication des polynômes.
L'addition s'obtient en additionnant les coefficients des vecteurs après avoir étendu le polynôme de plus petit degré en un vecteur de même longueur que l'autre.

```
(define (add-poly p1 p2)
  (let ((d1 (degre-poly p1))
        (d2 (degre-poly p2)))
    (if (< d1 d2)
      (add-poly p2 p1)
      (let ((r (poly->vect p2 (+ 1 d1))))
        (vect->poly (map-vectors + p1 r))))))

? (add-poly '#( 2 5 4 0 7) '#( 2 5 4 0 -7 )) -> #(4 10 8)
```

La multiplication par un nombre c consiste à multiplier chaque coefficient par c :

```
(define (mult-scal-poly c p)
  (if (zero? c)
    zero-pol
    (map-vectors (lambda (x)(* c x)) p)))

? (mult-scal-poly 3 '#( 2 5 4 0 7)) -> #(6 15 12 0 21)
```

Par définition de la multiplication de deux polynômes $\sum_{i=0}^{d_1} a_i X^i$ et $\sum_{j=0}^{d_2} b_j X^j$ on sait que le produit aura pour degré la somme des degrés et le coefficient de X^k est donné par l'expression :

$$c_k = \sum_{i=0}^{k} a_i b_{k-i}$$

Mais comme on doit avoir aussi $i \leq d_1$ et $0 \leq k - i \leq d_2$, on en déduit que i varie entre $max(0, k - d_2)$ et $min(k, d_1)$ dans le cas où $d_2 \leq d_1$.

```
(define (mul-poly p1 p2)
  (if (or (zero-poly? p1)(zero-poly? p2))
    zero-pol
    (let ((d1 (degre-poly p1))
          (d2 (degre-poly p2)))
      (if (< d1 d2)
        (mul-poly p2 p1)
        (let ((produit (make-vector (+ 1 d1 d2) 0))
              (k-coeff-produit
               (lambda (k)
                 (let ((i0 (max 0 (- k d2)))
                       (i1 (min d1 k))
                       (coeff 0))
                   (do ((i i0 (+ 1 i)))
                       ((< i1 i) coeff)
                     (set! coeff (+ coeff
                                    (* (vector-ref p1 i)
                                       (vector-ref p2 (- k i))
                                 )))))))
          (do ((k 0 (+ 1 k)))
              ((< (+ d1 d2) k) produit)
            (vector-set! produit k  (k-coeff-produit  k))))))))
```

```
? (mul-poly '#(1  0  1) '#(1 0 1 ) ) -> #(1 0 2 0 1)
```

Une comparaison des performances montre la plus grande rapidité de cette dernière représentation tant que les polynômes ne sont pas très creux. Pour bénéficier à la fois des avantages des représentations creuses et pleines, les systèmes de calcul formel permettent leurs utilisations conjointes. Cela conduit à des problèmes informatiques intéressants et difficiles sur l'équivalence des représentations d'un même objet abstrait.

Exercice 10 *Ecrire une fonction d'affichage des polynômes analogue à celle du chapitre 2 § 12.*

Architecture du programme de calcul avec les polynômes pleins

`(mul-poly p1 p2)`
La multiplication de deux polynômes.

`(add-poly p1 p2)`
L'addition de deux polynômes.

`(mult-scal-poly c p)`
La multiplication d'un polynôme par une constante.

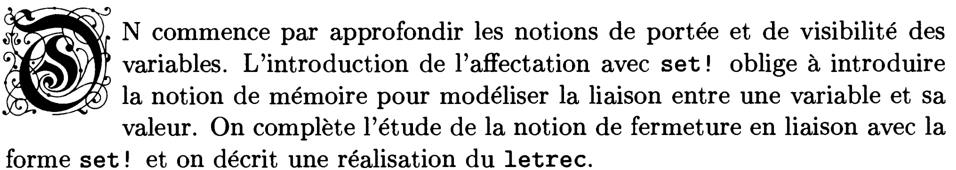
> `(map-vectors f v1 v2)`
> La création d'un vecteur dont les composantes sont les valeurs d'une fonction de deux variables sur les composantes de deux vecteurs.
>
> `(vect->poly vect)`
> La conversion d'un vecteur en un polynôme.
>
> > `(degre-vect vect)`
> > L'indice du dernier terme non nul d'un vecteur.
>
> `(poly->vect poly L)`
> La conversion d'un polynôme vecteur en un vecteur de longueur L que l'on complète, si besoin, par des zéros.
>
> > `(degre-poly p)`
> > Le degré d'un polynôme non nul.
> >
> > `zero-poly`
> > Le polynôme nul.
> >
> > `(zero-poly? p)`
> > Le prédicat pour reconnaître le polynôme nul.

4.7 De la lecture

Pour la programmation impérative en Scheme, on peut aussi consulter [SF90] ou [ML95].
Pour la programmation impérative en Pascal, un ouvrage classique est [Wir76].

Chapitre 5

Environnements, fermetures et prototypes



ON commence par approfondir les notions de portée et de visibilité des variables. L'introduction de l'affectation avec `set!` oblige à introduire la notion de mémoire pour modéliser la liaison entre une variable et sa valeur. On complète l'étude de la notion de fermeture en liaison avec la forme `set!` et on décrit une réalisation du `letrec`.

En étudiant diverses méthodes pour avoir une notion de variable mutable, on aboutit à la notion de prototype. C'est une notion qui combine les mécanismes de la fermeture et de l'affectation pour encapsuler les états d'une structure composée et les actions que l'on peut y effectuer. On en donnera diverses applications. On termine par l'étude d'un système de propagation de contraintes qui utilise les prototypes.

5.1 Portée et durée de vie d'une liaison

Maintenant que le lecteur est plus expérimenté, nous allons revenir sur les règles qui régissent les liens entre une variable et sa valeur.

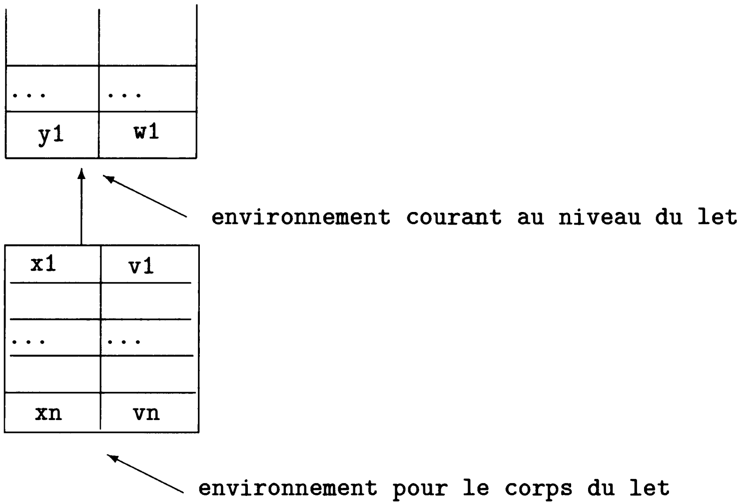
Une *variable* est un identificateur qui désigne une valeur. La liaison entre une variable et une valeur est réalisée lors de l'évaluation d'une *expression liante*, c'est-à-dire une forme comme : `let, let*, letrec, do, define` et surtout l'appel de fonction. Précisons ce dernier cas : à l'appel d'une fonction les paramètres formels de la lambda expression correspondante sont liés avec les valeurs des arguments ; on verra que les autres formes liantes peuvent se réduire à ce cas. Par ailleurs, au début d'une session on dispose de l'ensemble des liaisons relatives aux fonctions prédéfinies de Scheme.

Que représente une variable?

Au moment de l'évaluation d'une expression la question fondamentale est : *quelle est la valeur désignée par chaque variable ?* Cette valeur peut dépendre de la position de la variable dans le texte (*portée*) et de l'expression en cours d'évaluation (*durée de vie*). Pour décrire l'ensemble des liaisons actives, on utilise la notion d'*environnement*. Un environnement est constitué d'une suite de blocs de liaisons. Considérons par exemple, l'évaluation d'un `let` dans un certain contexte. A ce contexte est associé un environnement dit courant. Pour calculer la valeur de :

```
(let ((x1 e1)
       ...
      (xn en))
  corps)
```

on évalue en parallèle les expressions `e1, ..., en` dans l'environnement courant, on obtient les valeurs `v1, ..., vn`. Puis on évalue le corps dans l'environnement obtenu en ajoutant le bloc de liaisons `x1` → `v1, ..., xn` → `vn` à l'environnement courant :



Portée d'une liaison

En Scheme, la portée (*scope* en anglais) d'une liaison est dite *lexicale*. Cela signifie que l'on peut trouver la liaison qui définit une variable par simple inspection du texte du programme. On dit aussi que la portée est *statique* pour insister sur le fait qu'elle ne dépend pas de l'exécution du programme, dans le cas contraire on dit que la portée est *dynamique*[1].

La règle à utiliser pour trouver la liaison qui définit une variable dépend de la forme liante. Bien entendu, chaque règle reflète la sémantique que nous avons déjà indiquée pour ces formes.

[1] Ces notions seront formalisées au chapitre 21.

Dans le cas de l'appel `((lambda (x1...xn) corps) e1 ... en)`, la portée de la liaison entre un paramètre formel `xi` et la valeur de `ei` est le corps de la lambda. Il en est de même pour la forme équivalente
`(let ((x1 e1) ... (xn en)) corps)`.

Dans le cas de la forme `(letrec ((f1 e1) ... (fn en)) corps)`, la portée de la liaison entre `fi` et la valeur de `ei` est le corps augmenté de *toutes* les expressions `ej`.

Pour le `let` séquentiel `(let* ((x1 e1) ... (xn en)) corps)`, la portée de la liaison entre `xi` et la valeur de `ei` est le corps augmenté des expressions `ej` pour $j > i$.

Les variables globales définies par `define` ont pour portée *toute* la session. On peut dire que la portée d'une définition est rétroactive, car elle est prise en compte par les définitions précédentes de la session. Par exemple, on a déjà utilisé sans problème des définitions du style :

```
? (define add-m  (lambda (x)(+ m x)))

? (define m 10)

? (f 5)
15
```

La fonction `add-m` utilise la liaison pour `m` qui a été introduite après.
Ce fonctionnement permet une programmation descendante : on peut définir les utilitaires après les fonctions principales. Ce n'est pas le cas dans un langage comme ML qui impose un ordre sur les définitions.

Mais ces règles doivent être modulées par une notion de *visibilité*. Une liaison plus interne peut cacher une autre liaison d'une variable de même nom. Dans ce cas, c'est la liaison la plus récente qui doit être retenue. De façon pratique, on trouve la liaison en remontant les blocs de liaisons et en prenant la première liaison trouvée. Illustrons ce mécanisme avec la session suivante :

```
? (define a 1)
a

? (define (f x)
    (+ x 1))

? (let ((x  a)
        (y 20))
    (let* ((u (+ x y))
           (v (+ a u))
           (a (f 5)))
      (list a (f a) x y u v)))
(6 7 1 20 21 22)
```

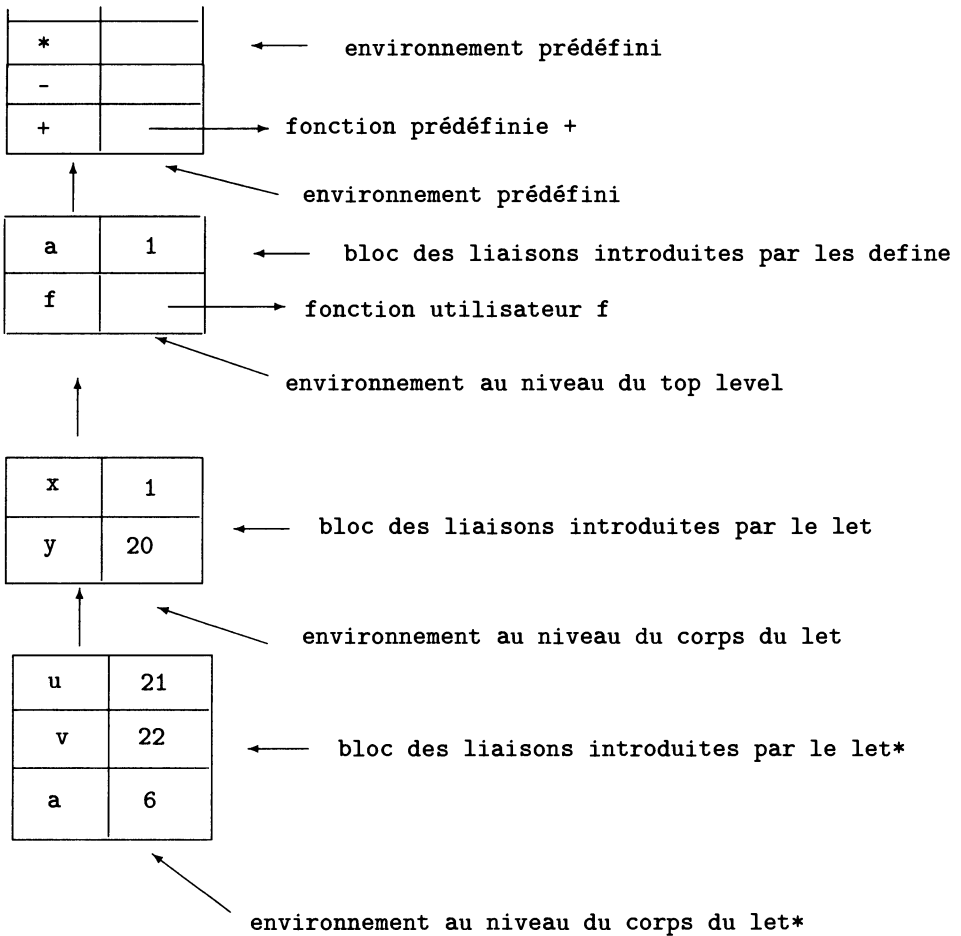


Fig 1.1 Environnements pour la session ci-dessus

Pour avoir la valeur de `f` au niveau du corps du `let*`, on doit remonter de deux blocs. En revanche, la valeur de **a** en ce point est donnée par une des liaisons du `let*` qui cache la valeur de **a** définie au top level.

Exercice 1 *Dessiner les blocs de liaisons pour l'évaluation de :*

```
(let* ((x 2)
       (y (+ x x))
       (z (* x y)))
  (let ((x y)
        (y x)
        (+ *))
    (+ x y z)))
```

Et indiquer, pour chaque variable, la liaison en vigueur dans l'expression `(+ x y z)`.

Durée de vie d'une liaison

La durée de vie (*lifetime* ou *extend* en anglais) est la période de l'exécution pendant laquelle une liaison est utilisable (même si elle n'est pas accessible).

En Scheme, la durée de vie d'une liaison est *illimitée*. C'est une spécificité de ce langage car dans beaucoup de langages, la durée de vie est réduite à l'exécution du bloc où est définie la liaison.
Voici un exemple qui illustre cet aspect, considérons la session :

```
? (define  f '?)

? (let ((a 10))
    (set! f (lambda (x) (+ x a))))

? (f 5)
15
```

La liaison `a -> 10` a pour portée le corps du `let`, mais ce corps construit une fermeture globale qui capture cette liaison dans son environnement.
Bien après l'exécution du `let`, on pourra appeler la fonction `f` qui utilisera la liaison `a -> 10` pour évaluer son corps. Cette particularité sera une source de complications pour la compilation des programmes Scheme (voir chapitre 22 §9).

Environnement global

On appelle environnement global, à un moment de la session, l'ensemble des liaisons visibles au niveau de la boucle d'interaction. Cela concerne tous les objets Scheme prédéfinis et toutes les variables introduites par un `define` au premier niveau.
La valeur associée à une variable est une notion temporelle car elle peut changer pendant une session.
Voici un exemple (à ne pas suivre). On définit `ancien-car` comme étant un synonyme de la fonction prédéfinie `car` :

```
? (define  ancien-car  car)
```

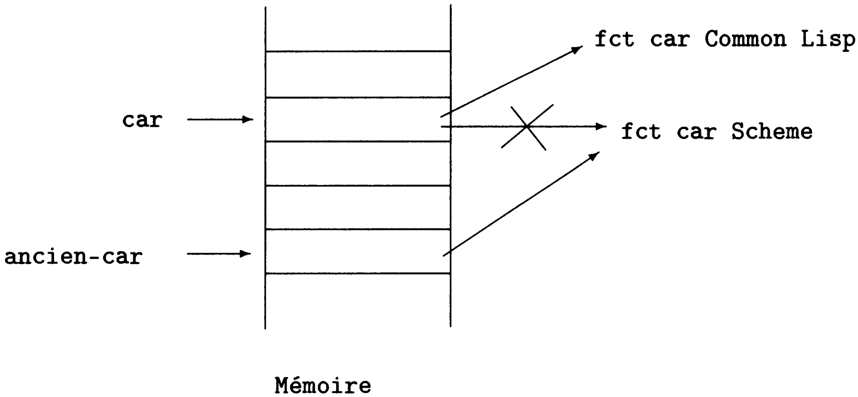
On peut souhaiter modifier la définition de la fonction `car` pour lui donner le même comportement qu'en Common Lisp. En Common Lisp le `car` d'une liste vide n'est pas une erreur mais la liste vide :

```
? (define car (lambda (x)(if (null? x) '() (ancien-car x))))

? (car '()) -> ()
```

Exercice 2 *Expliquez pourquoi on n'a pas utilisé la définition :*

```
? (define car (lambda (x)(if (null? x) '() (car x))))
```

La variable car ne désigne plus le car standard mais ce n'est pas un phénomène de visibilité qui l'explique. Voici un schéma de l'état mémoire qui précise la situation :



La définition du car lispien n'a pas modifié la liaison entre l'identificateur car et sa case mémoire mais le *contenu* de cette case.
Bien entendu, ce n'était qu'un exemple d'école, en pratique il faut éviter de redéfinir les fonctions prédéfinies sous peine d'amener la plus grande confusion dans la lecture et la maintenance du code. Heureusement, on avait sauvé la valeur originale de car, et après

```
? (define car  ancien-car)
```

tout rendre dans l'ordre.

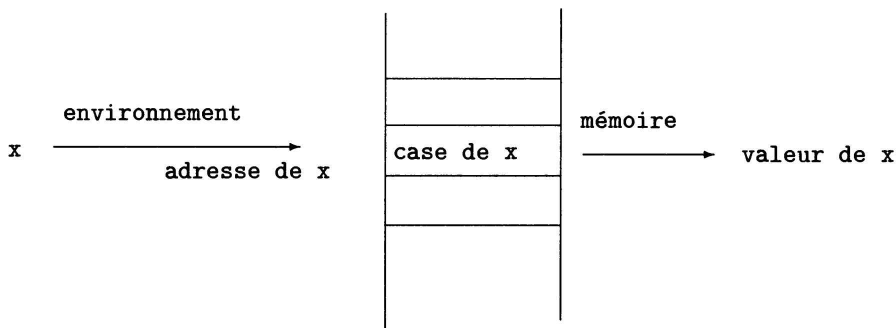
5.2 Variable, environnement et mémoire

Valeur d'une variable

La discussion précédente montre que la relation entre une variable et sa valeur ne peut pas toujours s'expliquer sans faire intervenir la mémoire. L'introduction de l'affectation nous oblige à modifier notre notion initiale d'environnement. Un environnement n'indique plus la valeur d'une variable mais son *adresse* en mémoire. De sorte que, comme on l'a entrevu aux chapitres 2 §10 et 4 §1, la valeur d'une variable dépend de deux fonctions :

- la fonction qui associe à une variable sa case mémoire ; c'est cette fonction qui s'appelle maintenant l'*environnement* ,
- la fonction qui associe à une case mémoire la valeur qu'elle contient ; cette fonction s'appelle la *mémoire.*

La valeur de la variable x est donnée par la composition de fonctions (mémoire (environnement x)). Ce que l'on représente avec le schéma suivant :



Modifications en mémoire et fermeture

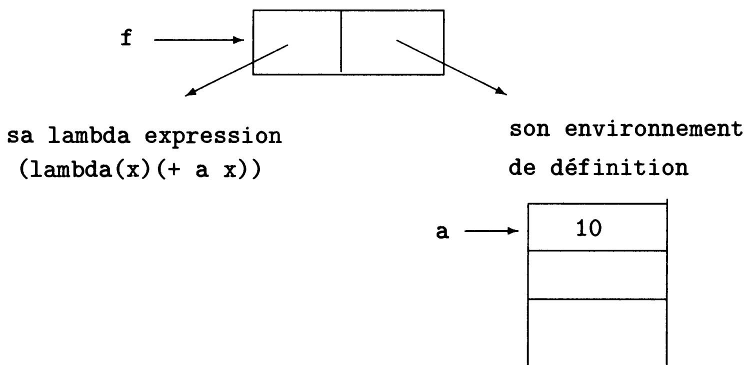
Les modifications de la *mémoire* sont réalisées par les fonctions terminées par un ! et aussi par `define`. Les changements d'*environnement* sont le fait des lieurs comme l'appel de fonction, le `let`, ... L'exemple suivant combine ces deux aspects :

```
? (let ((a 10))
    (set! a 0)
    a)
0
```

Combinons maintenant l'affectation avec la création d'une fermeture. On définit une variable locale `a`, puis une fonction `f` utilisant cette variable :

```
? (let ((a 10))
    (let ((f (lambda (x)(+ a x)))
      (f 0)))
10
```

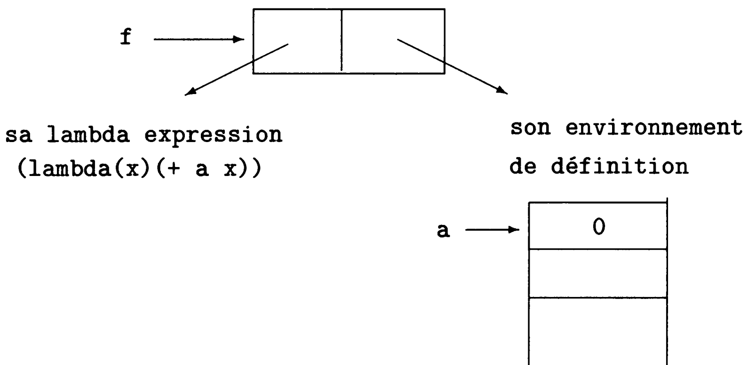
On a vu au chapitre 3 §2 que l'on peut représenter la fermeture de `f` par un couple.



Que se passe-t-il si l'on change la valeur de `a` par un `set!` ?

```
? (let ((a 10))
    (let ((f (lambda (x)(+ a x))))
      (set! a 0)
      (f 0)))
0
```

La valeur de a rangée dans la fermeture de f est modifiée.
Explication : la fermeture associée à f contient un pointeur sur cet environnement. Comme cette liaison de a est visible au moment du set!, il y a modification du *contenu* de la case mémoire de a, d'où une modification de la valeur de la fermeture associée à f. Voici le schéma après exécution du set! :



Exercice 3 *Devinez la valeur de :*

```
(let ((a 10))
  (let ((f (lambda (x)(+ a x))))
    (let ((a 5))
      (set! a 0)
      (f 0))))
```

Expliquer en comparant avec l'exemple précédent.

Protection contre la redéfinition d'une variable

On peut se prémunir contre la redéfinition de fonctions primitives avec le mécanisme de la fermeture. Illustrons ce point avec la session suivante.
Définissons une fonction cinquieme :

```
(define cinquieme
  (let ((car car))
    (lambda (L)(car (cddddr L)))))

? (cinquieme '(a b c d)) -> *** ERROR -- PAIR expected
```

Remplaçons la fonction prédéfinie car par son analogue Lisp :

```
(define car
  (lambda (L)(if (null? L) '() (car L))))
```

Cela n'a pas d'incidence sur la fonction cinquieme car elle utilise la valeur originale de car rangée dans sa fermeture :

```
? (cinquieme '(a b c d)) -> *** ERROR -- PAIR expected
```

Exercice 4 *Si l'on avait défini* cinquieme *par :*

```
(define (cinquieme L)
  (let ((car car))
    (car (cddddr L))))
```

aurait-on eu la même protection vis-à-vis de la redéfinition de car *?*

5.3 Définitions récursives et letrec

Maintenant, on peut expliquer pourquoi le let ne permet pas de définir une fonction récursive. Essayons de définir par un let une fonction qui retourne le dernier élément d'une liste non vide :

```
? (let ((dernier (lambda (L)
                   (if (null? (cdr L))
                     (car L)
                     (dernier (cdr L))))))
    (dernier  '(a b c)))
*** ERROR -- Unbound variable: dernier
```

En effet, la liaison de dernier avec sa valeur est visible dans le corps du let mais pas dans la lambda, aussi le premier appel récursif de last déclenche-t-il une erreur. Mais si dernier était déjà défini dans l'environnement global, alors c'est cette valeur qui serait utilisée par l'appel récursif, il n'y aurait pas d'erreur pour Scheme mais ce n'est probablement pas ce que le programmeur souhaitait faire :

```
? (define dernier  last-pair)

? (let ((dernier (lambda (L)
                   (if (null? (cdr L))
                     (car L)
                     (dernier (cdr L))))))
    (dernier  '(a b c)))
(c)
```

Mais on espérait obtenir le symbole c !. La forme letrec permet de résoudre ce problème :

```
? (letrec ((dernier (lambda (L)
                      (if (null? (cdr L))
                        (car L)
                        (dernier (cdr L))))))
    (dernier  '(a b c)))
c
```

Simulation du `letrec` par `let` et `set!`

Revenons sur l'évaluation de :

```
(letrec  ((f1 e1)
             ...
           (fN eN))
  corps)
```

Elle consiste à évaluer en parallèle les expressions `e1` ...`eN` dans un environnement *contenant aussi les nouvelles liaisons pour les variables xj.* Mais, comme il n'y pas encore de valeur dans les cases mémoires des `fj`, l'évaluation des `ej` ne doit pas nécessiter la connaissance de ces valeurs[2]. Ensuite, on place dans la case mémoire de `fj` la valeur `vj` ainsi calculée puis on évalue le `corps` dans cet environnement. On peut simuler ce comportement en combinant le `let` et l'affectation `set!` :

```
(let ((f1 'any)        ; on initialise les variables locales fj
          ...          ; avec des valeurs arbitraires
      (fN 'any))
  (let ((f1-aux  e1)   ; ainsi les fj sont visibles au moment de
         ...           ; de l'évaluation des ek. Les noms des variables
        (fN-aux  eN))  ; fi-aux ne doivent pas apparaître dans les ej.
    (set! f1 f1-aux)   ; on donne aux fj leurs véritables valeurs
     ...
    (set! fN fN-aux)
    corps))
```

Par exemple, voici une définition locale de `factorielle` sans utiliser `letrec` :

```
? (let ((f 'any))
    (let ((f-aux (lambda (n)(if (zero? n) 1 (* n (f (- n 1)))))))
      (set! f  f-aux)
      (f 5)))
120
```

Exercice 5 *Supprimer le* `letrec` *dans cette définition des fonctions récursives mutuelles f et g :*

```
(letrec ((f (lambda (n)(if (zero? n) 1 (+ (f (- n 1)) (g (- n 1))))))
         (g (lambda (n)(if (zero? n) 1 (* (f (- n 1)) (g (-  n 1)))))) )
  (f 3))
```

5.4 Programmation modulaire

Le `letrec` permet de définir des fonctions récursives locales à *une* fonction, mais comment partager des fonctions locales entre *plusieurs* fonctions globales? Et plus généralement, comment partager des variables et des fonctions *locales* entre plusieurs fonctions globales? C'est le principe de la *programmation modulaire*[3] : on désire cacher certaines définitions et en rendre visibles d'autres. Le langage

[2] C'est, par exemple, le cas quand `ej` est une lambda expression.

[3] On peut sauter ce paragraphe en première lecture.

Scheme ne donne pas de primitive pour définir des *modules*, mais on peut construire aisément un embryon de système de modules. On s'inspire de la méthode utilisée ci-dessus pour simuler le `letrec`.
Supposons que l'on désire définir globalement des fonctions `f1,...,fN` en partageant des variables `x1,...,xh` et des fonctions auxiliaires `aux1,...,auxk` sans avoir à définir globalement les `xi` et les `auxj`.
On procède en trois temps :

- on commence par définir globalement les variables `f1,..., fN` avec une valeur arbitraire :

```
(define  f1 'any)
...
(define  fN 'any)
```

- on définit les variables locales `xi` par un `let` puis les fonctions auxiliaires `auxj` et les fonctions `f1-loc,...,fN-loc` par un `letrec`. Les fonctions `fj-loc` sont les versions définies localement des fonctions `fj` souhaitées. On utilise pour les `fi-loc` des noms qui ne sont pas déjà utilisés :

```
(let ((x1 s1)
      ...
      (xh sh))
  (letrec ((aux1  (lambda (...) ...))
           ...
           (auxk  (lambda (...) ...))
           (f1-loc  (lambda (...) ...))
           ...
           (fN-loc  (lambda (...) ...)))
    (set! f1 f1-loc)
           ...
    (set! fN fN-loc) ))
```

- enfin, dans le corps du `letrec`, on affecte aux variables globales `fi` leurs vraies valeurs.

Le `letrec` permet de définir les fonctions `fi-loc` en partageant les variables et les fonctions auxiliaires. Ensuite, les `set!` permettent de rendre visible globalement ces définitions en les affectant aux variables `fi`. Bien entendu, s'il n'y a pas de définition récursive, un simple `let` suffit.
Illustrons ce style modulaire avec l'exemple du programme du chapitre 3 §10 pour placer huit reines sur un échiquier.
Il y a trois fonctions globales :

```
(define une-solution  'any)
(define les-solutions 'any)
(define des-solutions 'any).
```

Pour les définir on utilise des fonctions locales : `etat-final?`, `solution?`, `admissible?`, `Letats-suivants`, `1-a-N`, les deux premières n'étant pas récursives.

```
(let ((etat-final? (lambda (etat) ...))
      (solution?   (lambda (etat) ...))
  (letrec ((admissible?       (lambda (nlle-reine ...) ...))
           (Letats-suivants   (lambda (etat) ...))
           (1-a-N             (lambda (n) ...))
           (une-solution-loc  (lambda (etat) ...))
           (les-solutions-loc (lambda (etat) ...))
           (des-solutions-loc (lambda (etat) ...)))
    (set! une-solution  une-solution-loc)
    (set! les-solutions les-solutions-loc)
    (set! des-solutions des-solutions-loc)))
```

On constate que ce que l'on gagne en modularité on le perd en clarté.

Pour des raisons purement pédagogiques, on n'utilisera pas dans ce livre cette technique de programmation modulaire. L'obligation de tout regrouper au sein d'un grand `let` se concilie difficilement avec notre mélange d'explication et de code. Mais il est facile au lecteur de le faire après coup. La programmation modulaire s'impose pour un logiciel de grande taille ou développé par plusieurs auteurs. L'absence d'un mécanisme prédéfini pour gérer les modules est une faiblesse (temporaire?) de Scheme vis-à-vis de cet aspect «génie logiciel».

5.5 Mutation de variables

Modification d'une variable Scheme

Les fonctions de mutation physique permettent de modifier des structures composées mais peut-on modifier la valeur d'une variable globale par une fonction? Faisons l'expérience suivante: on considère des variables globales à valeurs numériques, on désire écrire une fonction `incrementer!` qui *modifie* leur valeur par l'ajout de 1. On peut penser réaliser cette opération avec:

```
(define (incrementer! x)
  (set!  x  (+ 1 x))
```

Pour tester, on définit une variable globale numérique `n` par:

```
? (define n 5)

? (incrementer!  n)
```

Que vaut maintenant `n`?

```
? n  -> 5
```

On constate que la valeur de `n` n'a pas été modifiée!
Ceci étonne souvent les débutants, mais l'explication est simple: il n'y a pas de passage de paramètre par variable[4] en Scheme. A l'appel de la fonction `incrementer!`, il y a liaison du paramètre formel `x` avec la *valeur* de `n`, il y a donc disparition

[4] Cette notion est expliquée au chapitre 21 §8.

du nom de la variable n au cours de l'évaluation du corps de la fonction. Et, même si on applique cette fonction à une variable globale qui s'appelle x comme le paramètre formel, il n'y a pas d'effet sur la variable x :

```
? (define  x  0)
? (incrementer! x)
? x -> 0
```

Explication : c'est le paramètre formel x qui est affecté par le set! car il cache la variable globale x.
On a échoué dans la réalisation de la fonction incrementer!, mais on verra au chapitre 9 §2 que la notion de macro permettra d'y parvenir.

Variables mutables via les vecteurs

Pour réaliser une fonction qui modifie la valeur d'une variable on utilise une nouvelle représentation pour les variables. On représente une variable par une structure mutable : un vecteur dont l'unique composante contient la valeur de la variable. On construit une variable par la fonction creer-variable :

```
(define (creer-variable  valeur)
  (vector valeur)))
```

On accède à la valeur de la variable par la fonction valeur-variable :

```
(define (valeur-variable  variable)
  (vector-ref variable   0))
```

On modifie la valeur par la fonction set-variable! :

```
(define  (set-variable! variable  valeur)
  (vector-set! variable  0 valeur))
```

Ecrivons une version de incrementer! avec cette représentation des variables :

```
(define   (incrementer!  x)
  (set-variable!  x (+ 1 (valeur-variable x))))
```

Vérifions maintenant que incrementer! réalise bien l'effet cherché :

```
? (define  n  (creer-variable  5)

? (valeur-variable  n)  -> 5
? (incrementer! n)
? (valeur-variable  n)  -> 6
```

Carte bancaire

Compliquons un peu le problème. On désire modéliser l'utilisation d'une carte bancaire pour faire des transactions sur un compte bancaire. L'utilisation de la carte est conditionnée par la connaissance d'un code secret. On utilise un vecteur à deux composantes, la deuxième servant à stocker le code secret.

```
(define (creer-carte-bancaire valeur codeSecret)
  (vector  valeur codeSecret))
```

Une consultation de la valeur d'une carte nécessite de fournir le code secret.

```
(define (valeur-compte carte)
  (display-alln "donner le code : ")
  (if (= (vector-ref carte 1)(read))
    (vector-ref carte  0)))
```

Le retrait consiste à diminuer le solde d'une quantité donnée par l'utilisateur.

```
(define (retrait-compte! carte  retrait)
  (display-alln "donner le code : ")
  (if (= (vector-ref carte 1)  (read))
    (vector-set!  carte  0 (- (vector-ref  carte 0) retrait))))
```

Créons une carte bancaire `une-carte` initialisée à 4000, de code secret 1234, puis effectuons un débit et une lecture :

```
? (define  une-carte (creer-carte-bancaire  4000 1234))

? (retrait-compte!  une-carte  500)
donner le code : 1234

? (valeur-compte une-carte)
donner le code :   1234
3500
```

Malheureusement, une personne informée de cette représentation d'une carte bancaire peut lire le montant du compte sans connaître le code secret :

```
? (vector-ref  une-carte 0)  ->  3500
```

Pour remédier à cet inconvénient, on va utiliser une autre représentation basée sur la notion de fermeture.

5.6 Variables mutables et prototypes

Carte bancaire (suite)

Pour cacher le code secret et le solde d'un compte, on utilise le mécanisme de capture d'un environnement lors de la définition d'une fermeture. Une carte bancaire sera une fermeture créée par l'évaluation d'une lambda expression dans un environnement contenant le solde et le code secret. Pour consulter ou débiter son compte, l'utilisateur devra fournir son code secret.

```
(define (creer-carte-bancaire solde codeSecret)
  (lambda ()
    (display-alln "donner le code secret : ")
    (if (= codeSecret (read))
      (begin
        (display-alln "taper <<solde>> ou donner la valeur du retrait : ")
        (let ((reponse (read)))
          (if (eq? 'solde reponse)
            solde
            (set! solde (- solde reponse))))))))
```

La création d'une carte bancaire retourne une fermeture :

```
? (define carte (creer-carte-bancaire 4000 1234))
```

Pour faire un retrait ou consulter son compte, on appelle la fonction sans paramètre `carte` :

```
? (carte)
donner le code secret : 1234
taper <<solde>> ou donner la valeur du retrait : 500

? (carte)
donner le code secret : 1234
taper <<solde>> ou donner la valeur du retrait : solde
4500
```

Les variables `solde` et `codeSecret` ne sont visibles que dans le corps de la fonction `creer-carte-bancaire` ; maintenant une personne connaissant cette implantation ne pourra pas se dispenser de connaître le code secret pour utiliser une carte bancaire. Pour éviter d'avoir à passer par un dialogue, on incorpore la réponse en paramètre de la lambda expression, d'où une nouvelle version :

```
(define (creer-carte-bancaire solde codeSecret)
  (lambda (reponse)
    (display-alln "donner le code secret : ")
    (if (= codeSecret (read))
      (if (eq? 'solde reponse)
        solde
        (set! solde (- solde reponse))))))

? (define carte1 (creer-carte-bancaire 5000 1234))
```

Pour débiter ou lire cette carte bancaire, on applique la fonction `carte1` sur l'argument correspondant :

```
? (carte1  1000)
donner le code secret : 1234

? (carte1 'solde)
donner le code secret : 1234
3500
```

Il est facile de définir une interface plus naturelle pour utiliser notre compte bancaire :

```
(define (solde  compte)
  (compte 'solde))

(define (retrait-compte!  compte  valeur)
  (compte valeur))

? (retrait-compte! carte1  500)
donner le code secret : 1234

? (solde carte1)
donner le code secret : 1234
3500
```

Bien entendu, si on définit un autre compte,

```
? (define carte2  (creer-carte-bancaire 1500 333))
```

son évolution sera complètement indépendante de celle de la `carte1`.

Exercice 6 *1. Cette implantation est peu robuste, ajouter des tests pour assurer que, dans le cas d'un débit, l'utilisateur fournisse bien un nombre et qu'il est inférieur au solde courant.*

2. Ajouter la possibilité de déposer de l'argent sur un compte sans avoir à demander le code secret, puis inclure la possibilité d'un versement d'un compte sur un autre.

Variables mutables via les fermetures et prototypes

Revenons à notre problème initial de représentation des variables. En s'inspirant de la méthode utilisée pour les cartes bancaires, on est amené à représenter une variable par une fermeture. Cette fermeture prend en paramètre deux types de réponse : `valeur` et `set-variable!` ; on interprète ces réponses comme des messages envoyés à la variable. Comme le message peut comporter un ou deux arguments, on a utilisé une lambda expression avec la possibilité d'un nombre variable d'arguments.

```
(define (creer-variable valeur)
  (lambda message
    (case  (car message)
      ((valeur) valeur)
      ((set-valeur!) (set! valeur (cadr message))))))

? (define var (creer-variable 5))

? (var 'set-valeur! 0)
? (var 'valeur) -> 0
```

La fonction de modification `incrementer!` du §2 s'écrit maintenant :

```
(define (incrementer! x)
  (x 'set-valeur! (+ 1 (x 'valeur))))

? (incrementer! var)
? (var 'valeur) -> 1
```

Exercice 7 *1. Donner une interface plus agréable pour utiliser ces variables.*

2. Ecrire une fonction qui permute les valeurs de deux variables.

Ce style de programmation permet d'encapsuler des données avec une interface obligée. Faute d'un meilleur nom, on appellera *prototype* ce type d'objet ayant un état interne et pouvant réagir à des messages. Voici un squelette de création de prototype avec des champs : *champ1 ...champK* et des fonctions *f1 ...fN* pour les manipuler :

```
(define (creer-prototype init1 init2 ....)
  (let ((champ1 init1)
        (champ2 init2)
        ....
       )
    (lambda message
      (case (car message)
        ((f1) (code de f1))
        ((f2) (code de f2))
        ...
        (else  un eventuel message d'erreur si le message n'est pas prevu))))
```

On verra au chapitre 7 de nombreux exemples de structures de données représentées par cette technique et on approfondira cette approche au chapitre 11 §2.

Exercice 8 *Définir les listes croissantes de nombres comme des prototypes répondant aux messages :* ***ajouter-element, supprimer-element, afficher, vider.***

5.7 Fermetures et générateurs

On utilise aussi les fermetures pour mémoriser des valeurs entre deux appels d'une fonction. Voici quelques variations sur ce thème.

Compteur

Ecrivons une fonction pour générer un compteur. Un compteur est une fonction sans paramètre qui à chaque appel retourne l'entier suivant ; il est initialisé avec la valeur 0. Le principe consiste à utiliser une variable locale `index` qui est visible dans le corps de la lambda mais pas à l'extérieur. On peut aussi le voir comme un prototype qui ne répond qu'au seul message (implicite) d'incrémentation.

```
(define (creer-compteur)
  (let ((index -1))
    (lambda ()
      (set! index (+ 1 index))
      index)))

? (define un-compteur (creer-compteur))

? (un-compteur) -> 0
? (un-compteur) -> 1
? (un-compteur) -> 2
```

Exercice 9 *Modifier* `creer-compteur` *pour qu'un compteur réponde aussi au message* `reset` *de remise à zéro.*

Générateur de nombres “aléatoires”

Au lieu de générer des entiers successifs, on peut avoir besoin de générer des entiers aléatoires dans un intervalle donné $[0\ N[$. La suite d'entiers ne sera pas réellement aléatoire, car calculée par un algorithme précis, mais suffira souvent à donner l'apparence d'une telle suite. Elle est basée sur une congruence arithmétique du type :

$f(x) = a * x \quad modulo\ p \quad$ où $a = 7^5 = 16807$ et $p = 2^{32} - 1 = 4294967295$.

On se donne comme valeur initiale x_0 un nombre premier (disons 1009) et l'on considère la suite, définie par la récurrence : $x_{n+1} = a * x_n \quad modulo\ p$, comme une suite de nombres «aléatoires» dans l'intervalle $[0\ N[$. On renvoie à l'ouvrage [Knu73] pour l'étude de ce type de suite. Par homothétie, on réduit cette suite à l'intervalle $[0\ 1[$.

On écrit d'abord une fonction pour créer des générateurs de nombres aléatoires :

```
(define (creer-random)
  (let* ((a (expt 7 5))
         (p (- (expt 2 32) 1))
         (f (lambda (x)(modulo (* a x) p)))
         (x0 1009))
    (lambda ()
      (let ((x1 (f x0)))
        (set! x0 x1)
       (/ x1 p)))))
```

D'où un générateur de nombres dans l'intervalle $[0\ 1[$

```
(define random (creer-random))
```

et, après homothétie et troncature, on obtient un générateur d'entiers dans l'interva $[0\ N[$

```
(define (randomN  n)
  (inexact->exact (floor (* n (random)))))

? (randomN 5) -> 0
```

```
? (randomN 5) -> 3
? (randomN 5) -> 4
```

Exercice 10 *Calculer la moyenne des valeurs des appels* (randomN 10) *pour une suite d'appels et comparer cette moyenne avec la valeur idéale 4,5.*

Générateur de symboles : genVar

Il est parfois utile de pouvoir créer à l'exécution de *nouveaux* symboles. Il existe en Lisp — mais pas en Scheme — une fonction appelée **gensym** qui rend, à chaque appel, un nouveau symbole. Pour ne pas faire de confusion avec cette fonction, on appelle **genVar** notre générateur.
Sur le même principe que le compteur, on définit un générateur de symboles dont on a fourni le préfixe. Les symboles générés sont constitués d'un préfixe et se terminent par un numéro incrémenté à chaque appel :

```
(define (creer-genVar prefixe-str)
  (let ((No 0))
    (lambda ()
      (set! No (+ 1 No))
      (string->symbol (string-append prefixe-str (number->string No))))))

(define  genVar (creer-genVar "temp"))

? (genVar) -> temp1
? (genVar) -> temp2
```

Attention, ce générateur ne garantit pas, comme **gensym**, que l'on cré toujours un symbole jamais vu. Par exemple, à cet instant des appels de **genVar**, on aura l'égalité :

```
? (eq? 'temp3 (genVar)) -> #t
```

Mélange aléatoire

On a souvent besoin de générer des jeux de données aléatoires à partir d'une donnée type. A titre d'exemple, montrons comment générer des jeux de vecteurs par permutations aléatoires des composantes d'un vecteur. Par exemple, c'est le problème du mélange des cartes.
Le principe est très simple, notons L la longueur du vecteur et $J = L - 1$ l'indice de la dernière composante du vecteur à mélanger :

- on échange la dernière composante avec une composante choisie au hasard d'indice dans $[0\ J]$,
- puis on échange l'avant-dernière composante avec une composante choisie au hasard d'indice dans $[0\ J-1]$,
- ...

- on échange la ième composante avec une composante choisie au hasard d'indice dans $[0\ i]$,
- ...

D'où une fonction qui modifie un vecteur en mélangeant l'ordre de ses composantes :

```
(define (melanger-vecteur! vect)
  (let ((J (- (vector-length vect) 1)))
    (do ((i J (- i 1)))
        ((= -1 i) vect)
      (echanger! vect i (randomN (+ 1 i))))))

? (melanger-vecteur! '#(1 2 3 4 5 6 7 8 9)) -> #(7 1 5 3 4 6 2 8 9)
? (melanger-vecteur! '#(1 2 3 4 5 6 7 8 9)) -> #(9 8 3 7 1 4 2 6 5)
```

où `echanger!` est la fonction, définie au chapitre 4 §4 qui réalise la transposition physique de deux composantes.

5.8 Comptage et mémorisation des appels d'une fonction

Nombre d'appels

Si on combine l'idée du compteur avec le calcul de la valeur d'une fonction récursive, on peut en déduire un moyen de comptabiliser le nombre d'appels récursifs nécessaires au calcul d'une valeur.
Par exemple, considérons la définition habituelle de la fonction de Fibonacci ; on désire connaître le nombre d'appels de `fib` utilisés pour calculer `(fib n)`.

```
(define   (fib n)
  (if (< n 2)
    n
    (+ (fib (- n 1)) (fib (- n 2))) ))
```

On incorpore dans le code de Fibonacci un compteur qui est incrémenté à chaque appel. Pour distinguer entre la valeur rendue et le nombre d'appels utilisés, on utilise la technique des prototypes ce qui offre également une possibilité de remise à zéro du compteur.

```
(define (creer-fib-compteur )
  (let ((compteur 0))
    (letrec ((fib
              (lambda (n)
                (set! compteur (+ 1 compteur))
                (if (< n 2)
                  n
                  (+ (fib (- n 1)) (fib (- n 2)))))))
      (lambda message
```

```
        (case (car message)
          ((fib) (fib (cadr message)))
          ((nb-appels) compteur)
          ((reset-compteur) (set! compteur 0))))))

(define  fib-avec-compteur (creer-fib-compteur))

? (fib-avec-compteur 'fib 8)     -> 21
? (fib-avec-compteur 'nb-appels) -> 67
? (fib-avec-compteur 'reset-compteur)
```

Remarque 1 On donnera au chapitre 9 §11 une méthode beaucoup plus générale pour compter les appels d'une fonction.

Mémo fonctions

La mémorisation peut servir à réaliser une notion d'apprentissage pour une fonction `f` quelconque : on mémorise dans une table interne les valeurs qu'elle a déjà calculées.
Le principe est le suivant : à chaque appel de `f` il y a consultation de la table interne. Si on trouve la valeur correspondant à l'argument, on la renvoie directement, sinon on la calcule et on enrichit la table avec cette nouvelle entrée.
On représente la table des valeurs déjà calculées par une a-liste locale. Illustrons cette méthode en reprenant l'exemple de la fonction de Fibonacci.
La mémo version de Fibonacci utilise une table initialisée par les valeurs dans le cas où n=0 ou 1.

```
(define (memo-fib n)
  (let ((table (list '(0 . 0) '(1 . 1))))
    (letrec ((memo-fib-aux
               (lambda (n)
                 (let ((valeur-memorisee (assq n table )))
                   (if valeur-memorisee
                     (cdr valeur-memorisee)
                     (let ((valeur-calculee (+ (memo-fib-aux (- n 1))
                                               (memo-fib-aux (- n 2)))))
                       (set! table (cons (cons n valeur-calculee) table ))
                       valeur-calculee))))))
      (memo-fib-aux n))))
```

On constate que `memo-fib` est plus rapide que `fib`, cependant deux appels successifs de `memo-fib` pour le même entier prennent le même temps ! On pouvait penser que le deuxième appel serait réduit à une simple consultation de la table. Il y a bien mémorisation des valeurs intermédiaires calculées par `memo-fib-aux` au cours d'un appel de `memo-fib` mais pas de mémorisation entre deux appels de `memo-fib`. En effet, à chaque appel de `memo-fib`, la variable locale `table` est réinitialisée. Pour conserver cette `table` on ne doit pas retourner
`(memo-fib-aux n)` mais la *fermeture* `memo-fib-aux` qui contient la table.

```
(define (creer-memo-fib)
  (let ((table (list '(0 . 0)  '(1 . 1))))
    (letrec ((memo-fib-aux
               (lambda (n)
                 (let ((valeur-memorisee (assoc n table )))
                   (if valeur-memorisee
                     (cdr valeur-memorisee)
                     (let ((valeur-calculee (+ (memo-fib-aux (- n 1))
                                               (memo-fib-aux (- n 2)))))
                       (set! table (cons (cons n valeur-calculee) table ))
                       valeur-calculee))))))
      memo-fib-aux)))
```

On construit ensuite une fonction de Fibonacci qui conserve la table entre deux appels par :

```
?  (define fibonacci (creer-memo-fib))
```

```
? (fibonacci  20) -> 6765   ; réponse après un court instant
? (fibonacci  20) -> 6765   ; réponse instantanée
```

5.9 Compléments : Système de maintien de contraintes et CAO

Notion de contrainte

On a souvent à manipuler des grandeurs qui doivent satisfaire à tout moment à un ensemble de relations. Une relation (on dira aussi une *contrainte*) est simplement une fonction définie sur des variables et à valeur booléenne. Si l'on est amené à modifier une des grandeurs, on désire disposer d'un système qui maintienne la cohérence en provoquant des modifications des autres variables.

Par exemple, la relation $9C = 5(F - 32)$ relie la température C exprimée en degrés centigrades à son équivalent F en degrés Fahrenheit. Ainsi, $F = 40$ et $C = 20$ constitue un couple de valeurs admissibles. Dans ce cas très simple, toute modification de C entraîne une modification de F et vice et versa.

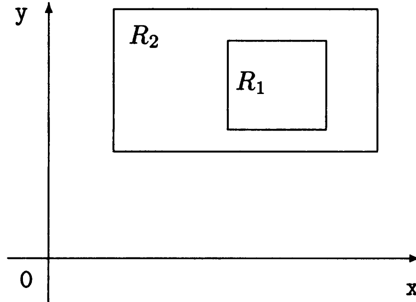
Mais on peut aussi considérer des inégalités comme $x \geq y$. Une augmentation de x ne nécessitera *pas* de modifier y. En revanche, une augmentation de y supérieure à la valeur courante de x entraînera une modification de x.

On peut également considérer des inégalités comme la condition de hors gel $C \geq 0$. Si l'on essaye de faire passer F de la valeur 40 à la valeur 30, il y aura échec car la condition de hors gel ne pourra plus être satisfaite.

Plus généralement, on considérera comme relation toute S-expression à valeur booléenne.

La conception assistée par ordinateur (CAO) est un domaine où l'on doit tenir compte d'un grand nombre de contraintes pour satisfaire aux fonctionnalités des objets à concevoir. On étudiera à la fin de ce paragraphe la conception d'une targette.

Exercice 11 *On considère dans un repère xOy des rectangles de côtés parallèles aux axes. On représente un tel rectangle par :*
- les coordonnées x , y de son sommet le plus haut à gauche,
- la longueur l du côté parallèle à Ox et la longueur h de celui parallèle à $0y$.
On se donne deux tels rectangles R_1 et R_2 ; écrire les relations pour que R_1 soit entièrement inclus dans R_2.



On va réaliser un système assez général pour maintenir[5] la cohérence d'un ensemble de variables. On souhaite que toute demande de modification d'une variable qui rompt la validité de certaines relations entraîne une cascade de modifications sur les autres variables de façon à rétablir la validité de toutes les relations. Dans le cas où le système n'arrive pas à trouver un ensemble de valeurs pour rétablir cette cohérence, la demande de modification sera rejetée et les modifications que l'on avaient commencé à effectuer seront annulées.

Représentation des variables

Dans cette section, on appelle variable une quantité susceptible de subir des contraintes. Une *variable* sera représentée par un prototype à quatre champs : son nom, sa valeur courante, sa valeur précédente et la liste des contraintes qui la concernent. Ce prototype devra répondre aux messages qui demandent à lire la valeur d'un champ mais aussi aux messages suivants :
- `add-contrainte!` : pour ajouter une contrainte à sa liste de contraintes,
- `supprimer-contrainte!` : pour enlever une contrainte à sa liste de contraintes,
- `set-valeur!` : on sauvegarde la valeur courante puis on change la valeur en la valeur donnée,
- `reset!` : pour remettre la valeur sauvegardée à la place de la valeur actuelle,
- `affiche` : pour afficher les champs de la variable.

D'où la fonction de création de tels prototypes :

```
(define (creer-variable nom valeur Lcontrainte)
```

[5]Il ne s'agit pas de trouver une solution cohérente mais de la maintenir après modification de certaines grandeurs. Pour trouver une solution, il faut faire appel à des algorithmes spécifiques à la nature des relations. Par exemple, dans le cas particulier des inéquations linéaires, on dispose du classique algorithme du simplexe.

```
(let ((ancienne-valeur valeur))
  (lambda message
    (case (car message)
      ((nom) nom)
      ((valeur) valeur)
      ((anc-valeur) ancienne-valeur)
      ((set-valeur!) (set! ancienne-valeur valeur)
                     (set! valeur (cadr message)))
      ((Lcontrainte) Lcontrainte)
      ((add-contrainte!)
       (set! Lcontrainte (cons (cadr message) Lcontrainte)))
      ((supprimer-contrainte!)
       (set! Lcontrainte (remove (cadr message) Lcontrainte)))
      ((affiche)
       (display-alln "variable: " nom "  valeur: " valeur "
                Lcontrainte: " (map (lambda (contr)(contr 'nom))
                                    Lcontrainte)))
      ((reset!) (set! valeur ancienne-valeur)))))
```

Représentation des contraintes

Avant de représenter les contraintes par des prototypes, il faut expliquer comment on choisira les variables à modifier. Considérons la relation $x \leq y + z$ entre trois variables. Si l'on modifie x, laquelle des variables y ou z devrons nous modifier et comment? C'est l'utilisateur qui doit donner à l'avance la réponse à ces questions. Bien entendu, on n'a besoin de modifier une autre variable que si l'inégalité n'est plus satisfaite. Si l'on veut qu'une modification de x soit répercutée sur y, on dit que y est *variable duale* de x et on se donne l'expression $y = x - z$ de y en fonction de x et z. Si l'on veut qu'une modification de y soit répercutée sur z, on dira que z est *variable duale* de y et on se donne l'expression de $z = x - y$. Enfin, si l'on veut qu'une modification de z soit répercutée sur x, on dira que x est *variable duale* de z et on se donne l'expression de $x = y + z$.

Dans certaines situations, il est utile d'introduire une dissymétrie entre les variables. On dit qu'une variable est *esclave* si une modification de sa valeur ne peut pas déclencher une modification d'une autre variable de la relation. Par exemple, on peut souhaiter que tout déplacement du rectangle extérieur puisse provoquer un déplacement du rectangle intérieur mais qu'à l'inverse le rectangle intérieur ne puisse pas déplacer le rectangle extérieur. Par convention, on indiquera ce cas en associant `#f` comme élément dual d'une variable esclave. C'est en particulier le cas des relations à une variable comme la condition $C \geq 0$.

Une contrainte sera représentée par un prototype à cinq champs:
- son nom,
- l'expression de la relation sous la forme d'une lambda expression,
- la liste des variables concernées (dans le même ordre que la liste des paramètres formels de la lambda expression),
- la liste `Lvar-expression` des lambda expressions exprimant chaque variable en fonction des autres; cette liste est aussi donnée dans le même ordre que la liste des paramètres formels de la relation,

- la liste des variables duales en correspondance avec la liste des variables ; on donne `#f` comme variable duale pour signifier que c'est une variable esclave.

Avec l'exemple de la relation précédente $x \leq y + z$, où z sera une variable esclave, on aura :
- expression de la relation : `(lambda (x y z)(<= x (+ y z)))`,
- la liste `Lvar` des variables concernées : `(list Vx Vy Vz)` où `Vx, Vy, Vz` sont les trois prototypes qui seront concernés par cette contrainte,
- la liste `Lvar-expression` :

```
( list  (lambda (y z)(+ y z))
        (lambda (x z)(- x z))
       '())
```

- la liste `Lvar-duale` des variables duales : `(list Vy Vz #f)`.

Le cas d'une simple inégalité comme $c \geq 0$ sera défini par la donnée de :
- expression de la relation : `(lambda (c)(<= 0 c))`,
- la liste des variables concernées : `(list Vc)` où `Vc` est la variable représentée par le paramètre `c`,
- la liste `Lvar-expression` : `(())`,
- la liste des variables duales : `(#f)`.

Pour calculer la variable duale associée à une variable donnée, il suffit de parcourir en parallèle les listes `Lvar` et `Lvar-duale` et de s'arrêter quand on arrive sur la variable donnée. D'une façon plus générale, c'est le rôle de la fonction `valeur-associee` :

```
(define  (valeur-associee ident Lident Lvaleur)
  (cond ((null? Lident)  #f)
        ((eq? (car Lident) ident)(car Lvaleur))
        (else (valeur-associee ident (cdr Lident) (cdr Lvaleur)))))
```

D'où une fonction de création de prototypes pour représenter les contraintes :

```
(define (creer-contrainte nom relation  Lvar  Lvar-expression  Lvar-duale)
  (lambda message
    (case (car message)
      ((nom) nom)
      ((Lvar) Lvar)
      ((var-duale)  (valeur-associee (cadr message)
                                     Lvar Lvar-duale))
      ((var-expression)(valeur-associee (cadr message)
                                        Lvar Lvar-expression))
      ((satisfaite?)(calculerExp relation Lvar)))))
```

Pour savoir si une relation est satisfaite, il faut calculer sa valeur ; c'est-à-dire la valeur du corps de sa lambda expression quand on donne aux paramètres formels les valeurs des variables qu'ils représentent, c'est l'objet de la fonction :

```
(define (calculerExp  lambdaExp Lvar)
  (let ((ListeValeurs (map (lambda (var) (var 'valeur)) Lvar)))
    (apply lambdaExp ListeValeurs)))
```

Interface utilisateur

Pour utiliser notre système, on définit quelques fonctions qui faciliteront l'interaction avec l'utilisateur.
On conserve dans une a-liste globale `*Lnom.var*` la liste des doublets : `(nom-de-variable . prototype-associé)` et on utilise une variable globale booléenne `*bavard?*` pour indiquer si l'on désire une exécution bavarde ou silencieuse.
Pour créer une nouvelle variable et l'ajouter au système, on utilise la fonction `ajouterVar` ; elle vérifie aussi si la variable n'est pas déjà présente :

```
(define (ajouterVar nom valeur)
  (if (assq nom *Lnom.var*)
    (display-alln "la variable " nom " est deja presente ")
    (let ((var (creer-variable nom valeur '())))
      (set! *Lnom.var* (cons (cons nom var) *Lnom.var*))
      var)))
```

En sens inverse, pour supprimer une variable du système, il faut non seulement la supprimer de la liste globale `*Lnom.var*` mais aussi supprimer toutes les contraintes qui l'utilisent, car ces contraintes perdent toute signification.

```
(define (supprimerVar var)
  (set! *Lnom.var* (remove-cle  (var 'nom) *Lnom.var*))
  (for-each (lambda (contrainte)(supprimerContrainte contrainte))
            (var 'Lcontrainte)))
```

La fonction `remove-cle` pour supprimer une entrée d'une a-liste a été définie au chapitre 2 §5., ce qui conduit à définir les fonctions analogues pour les contraintes.
Pour ajouter une contrainte, on l'ajoute à la liste des contraintes de chaque variable concernée puis on s'assure que la nouvelle contrainte est bien satisfaite, sinon on annule l'opération.

```
(define (ajouterContrainte contrainte)
  (for-each (lambda (var)(var 'add-contrainte!  contrainte))
            (contrainte 'Lvar))
  (if (contrainte 'satisfaite?)
    #t
    (begin
      (supprimerContrainte contrainte)
      (display-alln "ne peux ajouter une contrainte non satisfaite ")
      #f)))
```

Pour supprimer une contrainte, il suffit de l'enlever de la liste des contraintes de chacune de ses variables :

```
(define (supprimerContrainte contrainte)
  (for-each (lambda (var)(var 'supprimer-contrainte! contrainte))
            (contrainte 'Lvar)))
```

Il est aussi utile de pouvoir afficher l'ensemble des caractéristiques de chaque variable :

```
(define (afficherVariables)
  (for-each (lambda (nom.var)((cdr nom.var) 'affiche))
            *Lnom.var*))
```

Le moteur de propagation

On peut maintenant aborder la réalisation du *moteur de propagation* des modifications. Le principe est très simple : quand on désire changer la valeur d'une variable, on considère les contraintes où elle intervient, et pour chaque contrainte non satisfaite, on demande la modification correspondante de sa variable duale. Cette demande peut à son tour provoquer d'autres demandes de modifications ; c'est le phénomène des modifications en cascade.

Quels sont les cas d'échecs avec ce procédé ? Il y en a de deux types : variable esclave et bouclage :
- si la propagation des modifications aboutit à une demande de modification d'une variable esclave que l'on ne peut pas satisfaire, d'où un échec,
- si la propagation redemande la modification d'une variable que l'on a déjà modifiée, ceci peut être à la source de bouclage de la méthode aussi déclenche-t-on un échec[6]. Pour déceler ce type de bouclage, on ajoute dans une variable globale `*LnomVarModifie*` le nom de chaque variable modifiée. Cette variable globale est aussi utile quand on veut réaliser un «annuler», c'est-à-dire le rétablissement de l'état précédent.

Pour donner un exemple de bouclage, considérons les trois variables x , y , z liées par les deux contraintes $x = y + z$ et $z = x + y + 10$. On décide que y est duale de x, z est duale de y et x est duale de z. On part de la configuration $x = 10$, $y = -5$ et $z = 15$. On vérifie que la demande de modification de la valeur de x en 15 déclenche une suite de modifications qui boucle.
Il y a un autre cas d'échec qui se ramène au premier. Si l'on utilise une relation qui n'est pas partout définie alors il faudra rajouter des contraintes pour rester dans le domaine de définition. Par exemple, avec la contrainte $x \geq \sqrt{y-1}$ on doit ajouter la condition $y \geq 1$.

Pour modifier une variable, on appelle la fonction :

```
(define (ModifierVar var nlleValeur)
  (set! *LnomVarModifie* '())
  (let ((succes? (EssayerModifierVar var  nlleValeur)))
    (or succes?
        (annuler)
        (display 'echec)))).
```

Elle appelle la fonction principale `EssayerModifierVar` qui retournera `#t` si la méthode de propagation réussit et `#f` en cas d'échec. Son principe est le suivant, il y a trois cas :
- si la nouvelle valeur est identique à l'ancienne alors il n'y a rien à faire,
- si l'on détecte que la variable a déjà été modifiée il y a échec par bouclage,

[6] Dans certains cas, un bouclage peut converger vers une solution mais on n'entrera pas dans ces finesses.

- sinon on effectue la modification de la variable et l'on demande le rétablissement de toutes les contraintes où intervient cette variable.

```
(define (EssayerModifierVar var  nlleValeur)
  (display-alln-cond *bavard?* "essayer modifier " (var 'nom)
                              "  de:" (var 'valeur) "   en " nlleValeur )
  (let ((valeurActuelle (var 'valeur))
        (nom-var (var 'nom)))
    (cond ((= valeurActuelle nlleValeur) #t)
          ((memq nom-var *LnomVarModifie*)
           (display-alln-cond *bavard?*  "boucle pour " nom-var) #f)
          (else (var 'set-valeur! nlleValeur)
                (set! *LnomVarModifie* (cons nom-var *LnomVarModifie*))
                (every (lambda (contrainte)
                         (retablir-contrainte contrainte var))
                       (var 'Lcontrainte))))))
```

Si l'on désire une exécution bavarde, on laisse une trace de l'exécution par l'utilisation de la fonction d'affichage conditionnel `display-alln-cond` qui se comporte comme `display-alln` quand son premier argument vaut `#t` :

```
(define (display-alln-cond condition? . Larg)
  (if condition? (apply display-alln Larg)))
```

Si, après modification d'une variable, une contrainte cesse d'être satisfaite, il faut demander la modification de la variable duale. S'il n'y pas de variable duale, c'est l'échec causé par une variable esclave. Sinon, on calcule l'expression de la variable duale ainsi que la liste de ses arguments de façon à calculer la nouvelle valeur à lui donner.

```
(define (retablir-contrainte contrainte var)
  (display-alln-cond *bavard?* "satisfaire contrainte: " (contrainte 'nom))
  (or (contrainte 'satisfaite?)
      (let ((Lvar-contrainte (contrainte 'Lvar))
            (var-duale (contrainte 'var-duale var)))
        (if var-duale
          (let  ((sa-Lambda-exp (contrainte 'var-expression var-duale))
                 (ses-var (remove var-duale Lvar-contrainte)))
            (EssayerModifierVar var-duale
                                (calculerExp sa-Lambda-exp  ses-var)))
          (begin (display-alln-cond *bavard?* "var esclave :" (var 'nom)
                                  "  ne satisfait pas " (contrainte 'nom))
                 #f)))))
```

En cas d'échec, on a fait appel à la fonction `annuler` qui utilise les valeurs sauvegardées dans le champ `ancienne-valeur` pour restaurer les variables modifiées :

```
(define (annuler)
  (for-each (lambda (nomVar)
              (let ((var (cdr (assq nomVar *Lnom.var*))))
                (var 'reset)))
```

```
            *LnomVarModifie*)
  #f)
```

Remarque 2 Ce petit système de gestion de contraintes ne fait qu'effleurer le sujet. De nombreux problèmes se posent. En général il n'y pas unicité d'une solution satisfaisant les contraintes (spécialement s'il y a des inégalités). De plus, l'ordre de considération des contraintes à restaurer peut influer sur la solution trouvée ou même sur la découverte d'une solution quand il y a des variables esclaves. Pour simplifier, on a choisi de déclarer échec quand on était amené à modifier deux fois une même variable. Ce n'est pas toujours un cas de bouclage, car il existe des cas où l'on converge vers une solution après un nombre fini de telles modifications.

Architecture de ce système de gestion de contraintes

Fonctions pour l'interface avec l'utilisateur

`*Lnom.var*`
La a-liste globale des couples de nom et variable contrainte.

`(creer-variable nom valeur Lcontrainte)`
Création d'un prototype appelé variable pour représenter
une quantité susceptible de subir des contraintes.

`(ajouterVar nom valeur)`
Ajouter une nouvelle variable de valeur et de nom donnés.

`(supprimerVar var)`
Supprimer une variable du système, on supprime toutes les contraintes l'utilisant.

`(creer-contrainte nom relation Lvar Lvar-expression Lvar-duale)`
Création d'un prototype pour représenter une contrainte.

> `(valeur-associee ident Lident Lvaleur)`
> Valeur d'un identificateur d'une liste d'identificateurs
> dont on connait la liste des valeurs.
>
> `(calculerExp lambdaExp Lvar)`
> Valeur de l'appel d'une lambda expression quand les
> arguments sont les valeurs des variables.

`(supprimerContrainte contrainte)`
Supprimer une contrainte du système, on l'enlève de chaque variable concernée.

`(ajouterContrainte contrainte)`
Ajouter une nouvelle contrainte dans les champs des variables concernées ; si on s'aperçoit que cette contrainte n'est pas satisfaite, elle est retirée.

`(afficherVariables)`

Afficher les caractéristiques de chaque variable du système.

Le moteur de propagation.

`(ModifierVar var nlleValeur)`

Demande de modification de la valeur d'une variable,

en cas d'échec il y restauration.

`(annuler)`

Pour restaurer l'état antérieur.

`(EssayerModifierVar var nlleValeur)`

On change la valeur de la variable, retourne `#t`

en cas de succès et `#f` sinon.

`(retablir-contrainte contrainte var)`

Si après a une modification de la variable, la contrainte n'est plus

satisfaite, on essaye de la rétablir en modifiant sa variable duale.

`(display-alln-cond condition? . Larg)`

Affichage conditionnel d'expressions.

Application à la CAO d'une targette

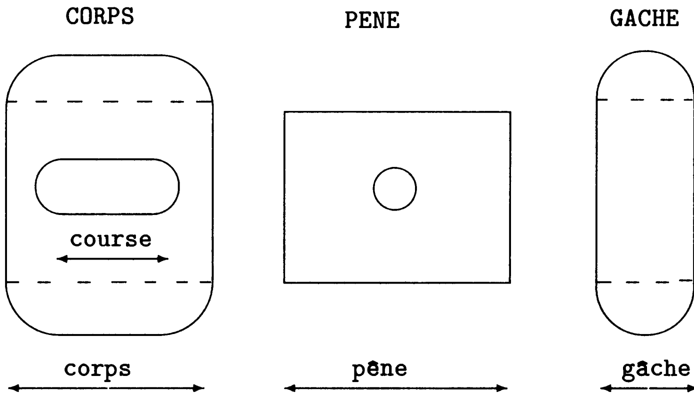
Donnons un exemple d'utilisation de ce système pour la conception d'une targette. Une targette est un mécanisme pour bloquer une porte. Une targette est constituée de trois pièces : le *corps* dans lequel coulisse le *pêne* qui vient se loger dans la *gâche.*

La targette est manœuvrée par un bouton qui coulisse dans une ouverture du corps et en limite aussi la course. La targette a deux positions :

- position ouverte : la porte est libre,

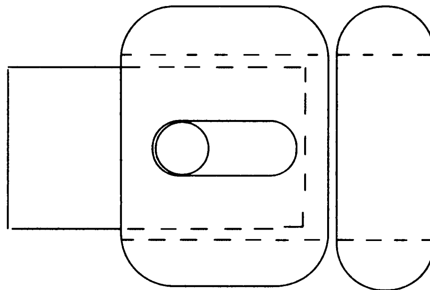
- position fermée : la porte est bloquée.

Pour assurer cette fonctionnalité, on doit avoir certaines relations entre des grandeurs longitudinales associées au corps, au pêne et à la gâche.



Les trois parties d'une targette

Les quatre variables à relier sont : les longueurs du corps, du pêne, de la gâche ainsi que la course du pêne à l'intérieur du corps.
En position ouverte, il faut que le pêne ne dépasse pas du corps.



Targette en position ouverte

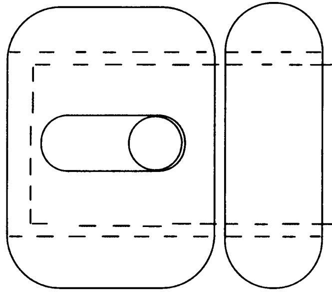
Si l'on raisonne sur les demi-longueurs, on trouve la condition :

$$\frac{pene}{2} \leq \frac{course}{2} + \frac{corps}{2}$$

d'où la relation d'ouverture

$$pene \leq corps + course$$

En position fermée, on exige que le pêne remplisse au moins toute la gâche (on suppose négligeable l'espace entre le corps et la gâche).



Targette en position fermée

En raisonnant encore sur les demi-longueurs, on trouve la condition :

$$\frac{pene}{2} \geq gache + \frac{(corps - course)}{2}$$

d'où la relation de fermeture :

$$pene \geq 2gache + (corps - course)$$

Enfin, on ajoute une condition de solidité. Pour garantir la résistance du corps, la course doit être inférieure au corps, plus précisément on impose la relation

$$(corps - course) \geq 10$$

et pour assurer la solidité de la gâche, on impose une longueur minimum de 15mm. D'où la relation de solidité :

$$(corps - course) \geq 10 \quad et \quad gache \geq 15$$

Remarque 3 En combinant les relations d'ouverture et de fermeture, on en déduit que $course \geq gache$.
En combinant la relation de fermeture avec la relation de solidité, on en déduit que $pene \geq 2gache + 10$.
En conséquence ces trois relations impliquent que les quantités : `gache, course, corps` et `pene` seront toujours supérieures à 15.

Pour modéliser ces conditions avec notre système, il faut partir d'une position admissible, car redisons-le, il ne s'agit pas de calculer une solution en partant de rien. On se donne par exemple une gâche de 20, une course de 25, un corps de 40. La condition de fermeture impose un pêne de longueur ≥ 55 et la condition d'ouverture impose un pêne de longueur ≤ 65, aussi prend-on un pêne de longueur 60. Le but est d'étudier les configurations possibles en partant de cet état.

On initialise les variables globales :

```
(define *Lnom.var* '())
(define *bavard?*   #f)
```

On définit les prototypes qui représentent le corps, la course, la gache et le pene :

```
(define  corps  (ajouterVar 'corps  40))
(define  course (ajouterVar 'course 25))
(define  gache  (ajouterVar 'gache  20))
(define  pene   (ajouterVar 'pene   60))
```

On définit les trois contraintes : `ouverture, fermeture et solidite`.
La contrainte d'ouverture met en jeu trois variables : `pene, corps, course`. On donne, pour chaque variable, son expression en fonction des autres de façon à satisfaire à l'égalité et l'on indique la liste des variables duales respectives : `course, pene, corps.` :

```
(define ouverture
  (creer-contrainte 'c-ouverture
                    (lambda (pene corps course)
                      (<= pene (+ corps course)))
                    (list  pene corps course)
                    (list
                         (lambda (corps course)(+ corps course))
                         (lambda (pene course)(- pene course))
                         (lambda (pene corps)(- pene corps)))
                    (list course pene corps)))
```

Même chose pour la contrainte de `fermeture` qui met en jeux les quatre variables : `corps, course, gache, pene` avec les variables duales respectives : `pene, corps, pene, gache`[7]

```
(define  fermeture
  (creer-contrainte 'c-fermeture
                    (lambda (corps course gache pene)
                      (<= (+ corps (- course) (* 2 gache))  pene))
                    (list corps course gache pene)
                    (list
                         (lambda (course gache pene)
                           (+ pene course (* -2 gache)))
                         (lambda (corps gache pene)
                           (+ corps (* 2 gache)(- pene)))
                         (lambda (corps course pene)
                           (/ (+  course pene (- corps)) 2))
                         (lambda (corps course gache)
                           (+ corps (* 2 gache)( - course)))
                         )
                    (list pene corps pene gache)))
```

La contrainte de solidité est une conjonction, elle met en jeu trois variables : `course, corps, gache` et on prend comme variables duales : `corps, course, #f`. Ici la variable gache est esclave à cause de la condition gache ≥ 15.

[7] Ces choix de variables duales sont arbitraires et peuvent être modifiés par le lecteur.

```
(define  solidite
  (creer-contrainte 'c-solidite
                    (lambda (course corps gache)
                      (and (<= (+ 10 course) corps)(<= 15 gache)))
                    (list course corps gache)
                    (list
                         (lambda (corps gache)(- corps 10))
                         (lambda (course gache)(+ course 10)))
                    (list corps course #f)))
```

On ajoute ces trois contraintes :

```
? (ajoutercontrainte ouverture)
? (ajoutercontrainte fermeture)
? (ajoutercontrainte solidite)
```

Puis on affiche l'état du système :

```
? (afficherVariables)
variable: pene    valeur: 60  Lcontrainte:
                              (c-fermeture c-ouverture)
variable: gache   valeur: 20  Lcontrainte:
                              (c-solidite c-fermeture)
variable: course  valeur: 25  Lcontrainte:
                              (c-solidite c-fermeture c-ouverture)
variable: corps   valeur: 40  Lcontrainte:
                              (c-solidite c-fermeture c-ouverture)
```

Si l'on souhaite avoir une gâche de 35, le système va nous indiquer les dimensions à donner aux autres éléments :

```
? (ModifierVar gache 35)  -> #t

? (afficherVariables)
variable: pene   valeur: 85  Lcontrainte:
                              (c-fermeture c-ouverture)
variable: gache  valeur: 35  Lcontrainte:
                              (c-solidite c-fermeture)
variable: course  valeur: 45  Lcontrainte:
                              (c-solidite c-fermeture c-ouverture)
variable: corps  valeur: 55  Lcontrainte:
                             (c-solidite c-fermeture c-ouverture)
```

Si l'on désire avoir une course très réduite, le système va nous répondre que c'est impossible avec les contraintes actuelles :

```
? (ModifierVar course 10)  -> echec
```

Mais si l'on supprime la contrainte de solidité :

```
? (supprimercontrainte solidite)
```

on pourra alors réduire la course à 10 :

```
? (ModifierVar course 10)   -> #t

? (afficherVariables)
variable: pene    valeur: 35  Lcontrainte: (c-fermeture c-ouverture)
variable: gache   valeur: 10  Lcontrainte: (c-fermeture)
variable: course  valeur: 10  Lcontrainte: (c-fermeture c-ouverture)
variable: corps   valeur: 25  Lcontrainte: (c-fermeture c-ouverture)
```

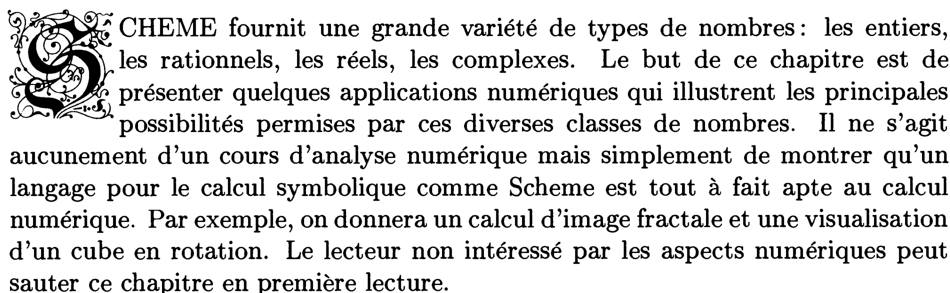
Bien entendu, un système de CAO couplerait ce moteur avec une interface graphique pour afficher les nouveaux objets après chaque demande de modification d'une grandeur. On pourrait aussi avoir un fonctionnement interactif: les demandes de modifications se feraient en tirant avec la souris sur le côté à modifier.

5.10 De la lecture

Tous les ouvrages sur Scheme traitent de la notion de fermeture et de ses applications à la notion de prototype [ASS89, SF90, ML95].
On trouve une introduction à un petit système de propagation de contraintes égalitaires dans [ASS89].

Chapitre 6

Calcul numérique avec Scheme



SCHEME fournit une grande variété de types de nombres : les entiers, les rationnels, les réels, les complexes. Le but de ce chapitre est de présenter quelques applications numériques qui illustrent les principales possibilités permises par ces diverses classes de nombres. Il ne s'agit aucunement d'un cours d'analyse numérique mais simplement de montrer qu'un langage pour le calcul symbolique comme Scheme est tout à fait apte au calcul numérique. Par exemple, on donnera un calcul d'image fractale et une visualisation d'un cube en rotation. Le lecteur non intéressé par les aspects numériques peut sauter ce chapitre en première lecture.

6.1 Quelques fonctions arithmétiques

Fonctions arithmétiques prédéfinies

Pour les entiers on dispose des principales fonctions arithmétiques :

`(quotient a b)` → le résultat de la division euclidienne de l'entier $a \geq 0$ par l'entier $b > 0$.

`(remainder a b)` → le reste de la division euclidienne de l'entier $a \geq 0$ par l'entier $b > 0$ avec l'égalité

$$a = bq + r \quad et \quad 0 \leq r < b$$

où q est le quotient et r le reste.

`(gcd n1 ...nK)` → le plus grand commun diviseur des entiers `n1 ...nK`

`(lcm n1 ...nK)` → le plus petit commun multiple des entiers `n1 ...nK`

`(expt a b)` → le nombre `a` à la puissance `b`.

Exercice 1 *Si la fonction `gcd` n'existait pas, en donner une définition récursive en utilisant l'égalité `(gcd a b) = (gcd (- a b) b)` quand $a \geq b$.*

Exercice 2 *Ecrire un prédicat* `parfait?` *qui teste si un nombre entier est égal à la somme de ses diviseurs autres que lui-même. Par exemple l'entier 6 est parfait car* $6 = 1 + 2 + 3$. *Trouver tous les nombres parfaits inférieurs à 10000.*

Puissance entière par dichotomie

Pour calculer la puissance entière d'un entier, on peut utiliser une méthode de dichotomie. Ainsi pour calculer 2^8, il suffit d'élever au carré 2^4, et 2^4 est lui-même le carré de 2^2. Cette méthode permet de réduire le nombre de multiplications de n à $\log(n)$. De façon plus précise, on se base sur les égalités :

$$x^n = \begin{cases} (x^2)^{n/2} & \text{si n est pair} \\ x(x^2)^{n/2} & \text{si n est impair} \end{cases}$$

D'où une redéfinition de la fonction `expt` dans le cas d'une puissance entière :

```
(define (puissance-entiere x n )
  (letrec ((puis-aux
             (lambda (x n acc)
               (cond ((= 0 n) acc)
                     ((even? n)
                      (puis-aux (* x x) (quotient n 2) acc))
                     (else
                      (puis-aux (* x x) (quotient n 2) (* x acc)))))))
    (puis-aux x n 1)))
```

Remarque 1 Comme l'opération de multiplication `*` s'applique à toutes les classes de nombres, on n'est pas obligé de supposer que l'argument `x` de la fonction précédente soit un entier.

Décomposition en facteurs premiers

La décomposition en facteurs premiers d'un entier peut être obtenue facilement quand le nombre n'est pas trop grand. On représente une telle décomposition par une liste. Par exemple, la décomposition de 440 est représentée par la liste `(2 2 2 5 11)`, la répétition du 2 signifiant que 440 est divisible par 2^3.
Pour connaître la multiplicité du nombre premier 2 dans la décomposition d'un nombre, on considère les quotients successifs de ce nombre par 2. Quand le quotient obtenu n'est plus divisible par 2, on recommence avec 3. Pour 4, on sait d'avance qu'il ne divisera pas le nombre restant puisque, 4 n'étant pas premier, on a déjà extrait les diviseurs premiers qui le compose, ... D'où une fonction qui retourne la liste des diviseurs d'un entier `n`. Le travail est fait par une boucle `loop`. Cette boucle accumule dans la liste `L` les facteurs premiers du nombre `n` et le paramètre `k` parcourt la liste des entiers à la recherche des diviseurs de `n` puis de ses quotients successifs. Il est inutile de chercher des valeurs de $k > n$.

```
(define (FacteursPremiers n)
  (letrec ((loop (lambda (n k L)
                   (cond ((= 1 n) L)
```

```
                         ((= 0 (remainder n k))
                          (loop (quotient n k) k (cons k L)))
                         ((< n k) L)
                         (else (loop n (+ k 1) L))))))
    (reverse (loop n 2 '()))))

? (FacteursPremiers 440) -> (2 2 2 5 11)
```

Autant il est rapide de calculer un nombre en connaissant sa décomposition en facteurs premiers, autant il est long de décomposer en facteurs premiers un grand entier ; cette remarque est à la base de certaines méthodes de cryptage.

Exercice 3 *Si au lieu d'incrémenter* `k` *de 1 on l'incrémentait de 2, la factorisation irait presque deux fois plus vite. Essayer d'introduire cette optimisation.*

Exercice 4 *Ecrire une fonction qui calcule le plus grand diviseur (autre que lui- -même) d'un nombre.*

Exercice 5 *Ecrire directement un prédicat* `premier?` *pour tester si un entier est premier.*

Identité de Bezout

Beaucoup de résultats d'arithmétique peuvent se déduire facilement d'un théorème attribué à Bezout. Le théorème de Bezout nous apprend que :
pour tout entier a et b, il existe des entiers u et v (non uniques) tels que :

$$au + bv = pgcd(a, b).$$

On se propose, étant donné deux entiers a et b, de calculer de tels nombres u et v ainsi que le pgcd de a et b.
Dans ce chapitre, on en donne une version peu élégante. Elle consiste à rendre la liste des trois nombres u, v, d tels que $d = pgcd(a, b)$ et $au + bv = d$. On verra une programmation plus élégante au chapitre 8 §3.
Le calcul est basé sur la remarque immédiate suivante : tout nombre qui divise a et b divise $b - a$ et b et réciproquement, d'où l'égalité $pgcd(a, b) = pgcd(b - a, b)$.
Cette égalité permet de déduire des valeurs de u et v pour a et b à partir des valeurs de u_1 et v_1 pour $(b - a)$ et b. En effet, si on a $(b - a)u_1 + bv_1 = d$ alors on a $a(-u_1) + b(v_1 - u_1) = d$ d'où $u = -u_1$ et $v = v_1 - u_1$.
Selon que a est inférieur ou supérieur à b, on considérera $(b - a)$ ou $(a - b)$. Enfin, si $a = 0$ il suffit de prendre $u = 1, v = 1, d = b$ et si $b = 0$ on prend $u = 1$, $v = 1$, $d = a$. Appelons `bezout` la fonction qui retourne la liste constituée de `u, v, d`.

```
(define (bezout a b)
  (cond ((= 0 a)(list 1 1 b))
        ((= 0 b)(list 1 1 a))
        ((< a b)(let ((u&v&d (bezout a (- b a))))
                  (list (- (car u&v&d)(cadr u&v&d))
```

```
                        (cadr u&v&d)
                        (caddr u&v&d))))
        (else (let ((u&v&d (bezout (- a b) b)))
                (list (car u&v&d)
                      (- (cadr u&v&d)(car u&v&d))
                      (caddr u&v&d))))))

? (bezout 99 78) --> (15 -19 3)
```

L'égalité de Bezout permet de démontrer rapidement des résultats classiques comme

- si a divise bc et est premier avec b alors a divise c.

  En effet, le pgcd de a et b étant égal à 1, l'égalité de Bezout donne l'existence de u et v tels que $au + bv = 1$ et en multipliant cette égalité par c on obtient $auc + bcv = c$ qui montre que a divise c.

- Soit p un nombre premier, alors pour tout entier $a \neq 0$ il existe un entier b tel que $ab = 1$ modulo p.

  En effet, l'égalité de Bezout donne l'existence de u et v tels que $au + pv = 1$ et il suffit de prendre $b = u$.

Exercice 6 *(pour matheux) Soit p un nombre premier, écrire une fonction qui étant donné un entier $m < p$, retourne l'entier $n < p$ tel que $m.n = 1$ modulo p.*

6.2 Ecriture d'un entier dans une base

Liste des chiffres d'un entier en base B

Comme Scheme ne fournit pas une fonction générale d'écriture d'un entier n dans une base quelconque, on définit une fonction qui calcule la liste des chiffres de n, écrit dans une base donnée.
On rappelle que l'entier n a pour liste de chiffres en base B la liste $(a_k \cdots a_0)$ s'il s'écrit :

$$n = a_k B^k + \cdots + a_1 B + a_0 \quad avec \quad 0 \leq a_i < B$$

Pour calculer les chiffres, on remarque que a_0 doit être le reste de la division euclidienne de n par B puisque :

$$n = B(a_k B^{k-1} + \cdots + a_1) + a_0 \quad avec \quad 0 \leq a_0 < B.$$

On obtient a_1 en appliquant le même raisonnement au quotient $(a_k B^{k-1} + \cdots + a_1)$ de n par B .... Cela montre au passage l'existence et l'unicité de cette décomposition de n. D'où la fonction :

```
(define (Liste-chiffres n  base)
  (letrec ((Lchiffres-aux
             (lambda (n Lchiffres)
               (if (< n  base)
                 (cons n Lchiffres)
```

```
            (Lchiffres-aux (quotient n  base)
                           (cons (remainder n base) Lchiffres)))))
    (Lchiffres-aux n '()))
```

```
? (Liste-chiffres 1023 10) -> (1 0 2 3)
? (Liste-chiffres 1023 2)  -> (1 1 1 1 1 1 1 1 1 1)
? (Liste-chiffres 1023 16) -> (3 15 15)
```

Quand la base est supérieure à 10, on utilise des lettres pour représenter les chiffres plus grands que 10 (en supposant que la base est < 36). Le `A` représente le 10, le `B` le 11 , ... Plus généralement, le caractère `(integer->char (+ 55 k))` sert à représenter le chiffre `k` $\geq$ 10 puisque le code ASCII de `A` est 65. Ce qui conduit à définir une fonction d'affichage d'un nombre donné par la liste de ses chiffres :

```
(define (affiche-entier  Lchiffres base)
  (for-each (lambda (k)
              (display (if (< k 10)
                           k
                           (integer->char (+ (char->integer  #\A) (- k 10))))))
            Lchiffres))
```

D'où un affichage plus traditionnel de 1023 en base 16 :

```
? (affiche-entier  '(3 15 15) 16) -> 3FF
```

Valeur d'une liste de chiffres

Etudions le problème inverse : calculer la valeur d'un entier dont on connaît la liste des chiffres dans une base. La formule :

$$n = a_k B^k + (a_{k-1} B^{k-1} \dots a_1 B + a_0)$$

montre que le calcul de n se ramène à celui de $(a_{k-1} B^{k-1} \dots a_1 B + a_0)$ à condition de connaître la valeur de k. Mais il n'est pas élégant de calculer en premier lieu la longueur de la liste. On peut aussi se baser sur la relation

$$n = a_0 + B(a_1 + \dots + a_k B^{k-1})$$

qui se programme immédiatement à partir de la connaissance de la liste des chiffres renversée. On obtient :

```
(define (valeurLchiffre L B)
  (let ((R (reverse L)))
    (letrec ((valeur-aux
               (lambda (R)
                 (if (null? R)
                   0
                   (+ (car R) (* B (valeur-aux (cdr R) )))))))
      (valeur-aux R))))
```

Mais, il n'est pas non plus satisfaisant de procéder d'abord à l'inversion de la liste des chiffres. Heureusement, il existe une méthode qui permet de faire le calcul directement ; elle consiste à effectuer les calculs suivants :

a_k
$Ba_k + a_{k-1}$
$B(Ba_k + a_{k-1}) + a_{k-2}$
$B(B(Ba_k + a_{k-1}) + a_{k-2}) + a_{k-3}$
...
$B(B(\cdots B(Ba_k + a_{k-1}) \cdots) + a_1) + a_0$

Cette méthode de calcul s'appelle le schéma de Horner. Elle est employée en analyse numérique pour obtenir la valeur d'un polynôme en minimisant le nombre des additions et multiplications.
On en déduit une version récursive terminale pour ce calcul :

```
(define (valeurLchiffre L B)
  (letrec ((valeur-aux
             (lambda (L acc)
               (if (null? L)
                 acc
                 (valeur-aux (cdr L) (+ (car L)(* B acc)))))))
    (valeur-aux L 0)))

? (valeurLChiffre '(4 2 1) 10) -> 421
```

Exercice 7 *Si le calcul effectué par la fonction* **`valeur-aux`** *ne vous paraît pas évident, il est instructif de prouver qu'elle satisfait à la spécification suivante :* *(`valeur-aux '(ak ...a0) acc`)* $= accB^{k+1} + a_kB^k + \cdots + a_0$.

Nombre de zéros à la fin de n!

Parfois, on peut avoir des informations sur les chiffres qui composent un nombre sans avoir à calculer ce nombre. Illustrons ceci par le calcul du nombre de zéros qui terminent l'écriture de $n!$ en base 10. Par exemple l'écriture de 50! se termine par douze zéros :

```
? (fac 50)
30414093201713378043612608166064768844377641568960512000000000000
```

Mais combien y a-t-il de zéros qui terminent l'écriture de 999999! ? Comme ce nombre est trop grand pour que nous puissions calculer sa factorielle, on remarque que le nombre de ses zéros est exactement l'exposant de la plus grande puissance de 10 qui le divise. Enfin, comme $10 = 2 \times 5$, c'est la plus petite des multiplicités de 2 ou de 5 dans sa décomposition en facteurs premiers. Par construction de $n!$, on sait d'avance que la multiplicité du nombre premier 5 sera inférieure à celle de 2. On est donc ramené au problème du calcul de la multiplicité de 5 dans la décomposition de $n!$, sans avoir à calculer $n!$. Pour cela, on dispose d'une formule due à Legendre. Elle donne, en fonction de n, la multiplicité du nombre premier p dans la décomposition en facteurs premiers de $n!$:

multiplicité de p dans $n!$ = ((somme des chiffres de l'écriture de n en base p) -1) divisée par $p-1$

La somme des chiffres de n en base p se calcule comme la liste de ses chiffres, au lieu d'accumuler dans une liste, on les ajoute dans l'accumulateur **somme** :

```
(define (Somme-Chiffres N base )
  (letrec ((loop (lambda (N somme)
                   (if (< N base)
                     (+ N somme)
                     (loop (quotient N base)
                           (+ (remainder N base) somme))))))
    (loop N 0)))
```

D'où immédiatement la formule donnant le nombre de zéros terminant l'écriture de $n!$:

```
(define (nb-zeros-fin-n!  n)
  (quotient (- n (somme-chiffres n 5)) 4))
```

On retrouve ainsi :

```
? (nb-zeros-fin-n!  50) -> 12
```

et l'on découvre que :

```
? (nb-zeros-fin-n!  999999) -> 249992
```

6.3 Calculer avec les rationnels et les réels

Les nombres rationnels

Un rationnel s'écrit sous forme d'une fraction irréductible `a/b`. Sa valeur est le nombre rationnel obtenu après simplification éventuelle de sa fraction :

```
?  12/8  -> 3/2
```

On accède au numérateur et au dénominateur d'un rationnel par les fonctions **numerator** et **denominator** qui retournent les valeurs correspondantes à la fraction irréductible équivalente :

```
? (numerator   12/8)  -> 3
? (denominator 12/8)  -> 2
```

Les opérations arithmétiques de base conservent les rationnels, ce qui permet de faire des calculs exacts sur les expressions arithmétiques :

```
? (+ 1/3 1/4) -> 7/12
? (* 2/3  9/2) -> 3
```

Fonctions sur les réels

On dispose des fonctions les plus courantes comme :
`(exp x)` $\rightarrow$ e^x
`(expt x y)` $\rightarrow$ x^y
`(sqrt x)` $\rightarrow$ $\sqrt{x}$
`(log x)` $\rightarrow$ $\ln x$;; logarithme népérien de x
`(sin x)` $\rightarrow$ $\sin x$;; où x est en radian
`(cos x)` $\rightarrow$ $\cos x$;; idem
`(tan x)` $\rightarrow$ $\tan x$;; idem
`(atan x)` $\rightarrow$ $\arctan x$;; idem
...

Nombre de chiffres de $n!$

Ne quittons pas si vite la célèbre factorielle. On peut aussi se demander le nombre de chiffres pour écrire $n!$ en base 10, *sans avoir à connaître* la valeur de $n!$. On va obtenir une estimation de ce nombre en utilisant une approximation classique de $n!$:

$$n! \quad \simeq \quad (\frac{n}{e})^n\sqrt{2\pi n}$$

Testons cet équivalent en calculant le quotient de cette approximation par $n!$. La fonction `test-equivalent` fait ce calcul, en prenant pour π la valeur de $4\arctan(1)$ et pour e la valeur de e^1 :

```
(define (test-equivalent n)
  (let  ((e (exp 1))
         (pi (* 4 (atan 1))))
    (/ (* (expt (/ n e) n) (sqrt (* 2 pi n)))
       (fac n))))
```

```
? (test-equivalent  100) -> .99916701656785
```

Le nombre de chiffres de $n!$ sera de l'ordre de grandeur de l'exposant x tel que $10^x \simeq n!$, d'où la condition :

$$10^x \quad \simeq \quad (\frac{n}{e})^n\sqrt{2\pi n}$$

En prenant le log (népérien), on arrive à l'estimation cherchée :

$$x \quad \simeq \quad \frac{1}{log(10)}(nlog(n) - n + 1/2log(2\pi) - 1/2log(n))$$

D'où la fonction :

```
(define (nb-chiffres-n!  n)
  (let ((log-n (log n))
        (pi (* 4 (atan 1))))
    (round (/ (+ (* n log-n) (- n) (/ (log (* 2 pi)) 2) (- (/ log-n 2)))
              (log 10)))))
```

Voici une estimation du nombre de chiffres de 999 999! :

```
? (nb-chiffres-n! 999999) -> 5 565 697
```

Approximation de e^x

Pour calculer e on a utilisé l'existence de la fonction exponentielle ; si cette fonction était absente de Scheme, on pourrait néanmoins calculer une bonne approximation de la constante e en utilisant sa représentation comme somme d'une série. On sait que la série

$$e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!}$$

converge rapidement, d'où l'utilisation des sommes partielles pour en avoir une approximation de e^x.
La somme partielle d'ordre $k+1$ se déduit de la somme partielle d'ordre k en ajoutant le terme $\frac{x^{k+1}}{(k+1)!}$. Comme ce terme se déduit du précédent par multiplication par $\frac{x}{k+1}$, on est conduit à utiliser une fonction auxiliaire qui prend comme variable l'entier k, le terme courant et la somme partielle :

```
(define (ex x n)
  (letrec ((loop
             (lambda (k terme somme)
               (if (< n k)
                 somme
                 (loop  (+ k 1)  (/ (* terme x)(+ 1 k)) (+ somme terme))))))
    (loop 1 1 1)))

? (ex 1 20)  -> 6613313319248080001/2432902008176640000
```

Le résultat obtenu est un nombre rationnel car toutes les opérations utilisées dans cette fonction conservent les nombres rationnels. Pour avoir l'expression décimale associée, on utilise la fonction de conversion :

```
? (exact->inexact (ex 1 20)) -> 2.718281828459045
```

qui est une approximation de e avec quinze décimales exactes.

Exercice 8 *Calculer sur le même principe une approximation de la fonction* $\sin(x)$.

Calcul numérique d'intégrale par la méthode des trapèzes

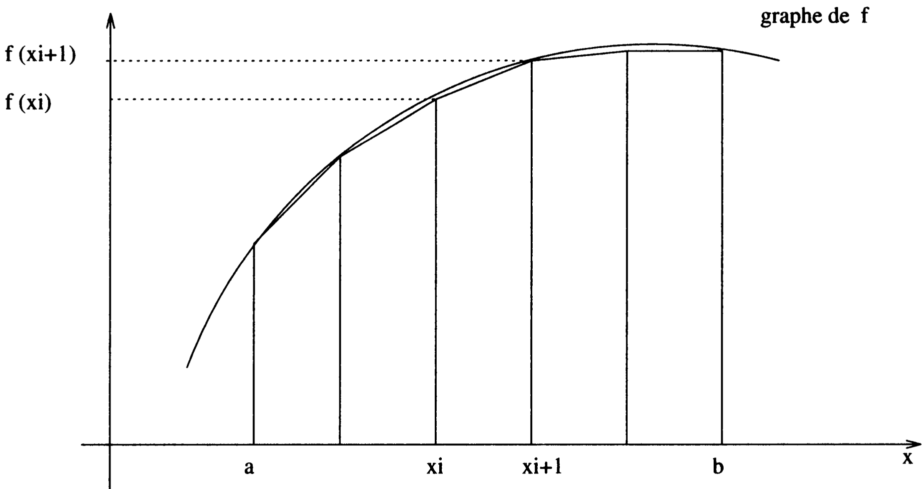
On dispose de nombreuses méthodes pour calculer π, d'ailleurs certains mathématiciens font la course à celui qui fournira le plus grand nombre de décimales. Pour illustrer le calcul d'une intégrale par la méthode des trapèzes, on va essayer d'obtenir une approximation de π en utilisant l'égalité :

$$\frac{\pi}{4} = \int_0^1 \frac{1}{1+x^2} dx$$

qui découle du fait que $\arctan(x)$ est une primitive de $\frac{1}{1+x^2}$.
On sait que l'intégrale définie $\int_a^b f(x)dx$ s'interprète comme l'aire de la zone comprise entre le graphe de la fonction f et l'axe des x. La méthode des trapèzes consiste à approcher cette aire par une somme de petits trapèzes.
Si l'on subdivise l'intervalle $[a\ b]$ en N parties, la base de chaque trapèze est de longueur $pas = \frac{b-a}{N}$ et donc l'aire au-dessus de l'intervalle $[x_i\ x_{i+1}]$ est approchée par $(f(x_i) + f(x_{i+1})) * pas/2$, d'où l'approximation :

$$aire\ des\ trapezes\ = pas * [\frac{f(a) + f(b)}{2} + \sum_{i=1}^{N-1} f(x_i)]$$

avec $x_i = a + i * pas$.



Méthode des trapèzes

Cette expression se calcule par la fonction `somme-trapeze` qui prend en paramètre la fonction à intégrer, les bornes `a`, `b` de l'intégrale et le nombre `N` de divisions :

```
(define (somme-trapeze f a b N)
  (let ((pas (/ (- b a) N)))
    (do ((i 1 (+ i 1))
         (xi (+ a pas) (+ xi pas))
         (somme (/ (+ (f a) (f b)) 2) (+ somme (f xi))))
        ((= i N) (* somme pas))
      )) ;; corps vide
```

D'où le calcul de $\int_a^b f(x)dx$ que l'on démontre être approché avec une erreur en $\frac{1}{N^2}$. On en déduit une approximation de π :

```
(define (calcul-pi N)
  (let ((f (lambda (x)(/ 1 (+ 1 (* x x))))))
```

```
    (* 4 (somme-trapeze f  0 1 N))))

? (exact->inexact (calcul-pi 100)) ->  -> 3.14157
```

ce qui est une approximation médiocre de π. En fait, la méthode des trapèzes ne converge pas vite, il existe des méthodes numériques beaucoup plus performantes.

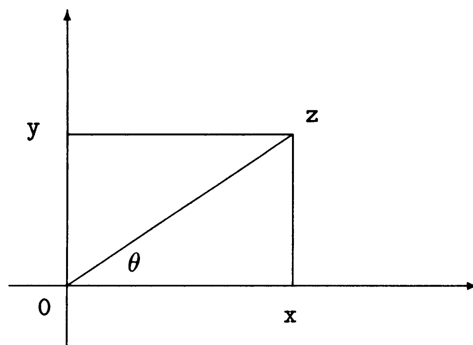
6.4 Calculer avec les nombres complexes

Fonctions sur les nombres complexes

Toutes les implantations de Scheme ne proposent pas les nombres complexes mais, quand elles les proposent, on dispose de l'extension aux nombres complexes des fonctions données en tête du paragraphe précédent. On renvoie à la norme Scheme pour le détail des déterminations à utiliser.

La représentation cartésienne d'un nombre complexe est définie par les fonctions :

`(make-rectangular x y)` $\rightarrow$ le nombre complexe `x + i y`.
`(imag-part z)` $\rightarrow$ `x`
`(real-part z)` $\rightarrow$ `y`



La représentation polaire d'un nombre complexe est définie par les fonctions :

`(make-polar r theta)` $\rightarrow$ le nombre complexe $re^{i\theta}$.
`(magnitude z)` $\rightarrow$ le module du nombre complexe `z`.
`(angle z)` $\rightarrow$ angle dans $]-\pi\ \pi]$ d'un nombre complexe $z \neq 0$.

Une constante complexe s'écrit aussi directement `x+yi` :

```
? (* 1+i 1-i) -> 2
```

La tour des nombres en Scheme

Une caractéristique importante du traitement des nombres par Scheme est de garder l'inclusion des types :

$$\text{integer} \subset \text{rational} \subset \text{real} \subset \text{complex} \subset \text{number}.$$

Par exemple, si une fonction attend un réel, on peut lui passer un entier. Bien entendu, les représentations internes de ces types de nombres sont différentes, mais Scheme cache cette distinction. Pour connaître le type d'un nombre, on dispose des prédicats suivants qui affinent le prédicat `number?`: `integer?`, `rational?`, `real?`, `complex?`.

Ces inclusions entre catégories de nombres justifient les résultats des tests suivants :

```
? (real?  1+4i) -> #f
? (real?  1+0i) -> #t
? (= 8/2  4+0i) -> #t
```

6.5 Fractals et ensemble de Mandelbrot

Pour illustrer le calcul avec les nombres complexes, on va décrire la construction de l'ensemble de Mandelbrot.

Ensembles fractals et récursivité

Les images de synthèse nous ont habitué aux paysages artificiels qui semblent refléter la diversité de la nature et qui sont en fait le résultat d'algorithmes de construction d'objets dits *fractals*. D'où la question : comment créer un apparent désordre avec quelques règles simples ? Une réponse est : utiliser des procédures graphiques récursives. Ainsi, chaque partie de l'objet reflétera l'image du tout. C'est ainsi que se développent les organismes multicellulaires : la multiplication des cellules reproduit à l'infini le même schéma de construction. D'où notre définition[1] : les fractals sont une manifestation géométrique de la récursivité. Ce point de vue sera illustré au chapitre 11 avec l'étude des systèmes de Lindenmayer.

Il y a un domaine voisin où s'introduisent les fractals : c'est le domaine des systèmes chaotiques. On dit qu'un système évolutif a un *comportement chaotique* si pour des données initiales arbitrairement voisines, il peut présenter des comportements extrêmement différents. Pour simplifier, on considère un système qui évolue à des intervalles de temps discrets et dont l'état X_{n+1} à l'instant $n+1$ se déduit de l'état X_n à l'instant n par une fonction $f : X_{n+1} = f(X_n)$.
L'état X_n est donc parfaitement déterminé par la donnée de l'état initial $X_0 = C$. On s'intéresse aux valeurs de C au voisinage desquelles une petite modification de C provoque une grande modification de X_n dès que n est "grand".

[1] Ce n'est pas sous cet angle que Mandelbrot a abordé l'étude des fractals mais sous l'angle de l'extension de la notion de dimension à des valeurs fractionnaires.

Ensemble de Mandelbrot

Considérons la suite de nombres complexes $z_{n+1} = z_n^2 + c$ avec $z_0 = c$. On démontre que pour certaines valeurs de c, la suite z_n reste bornée et que pour d'autres valeurs cette suite s'éloigne à l'infini.

L'ensemble M de Mandelbrot est défini comme l'ensemble des nombres complexes c pour lesquels la suite z_n reste bornée.

Sa frontière est donc une zone d'instabilité : pour des valeurs initiales voisines de la frontière certaines suites restent bornées et d'autres non. On démontre que M est inclus dans le rectangle $-2 \leq x \leq 0.5$, $-1.25 \leq y \leq 1.25$.

Pour construire une image de cet ensemble, on quadrille ce rectangle et, pour chaque point du quadrillage, on teste s'il est dans M. On prouve facilement que c n'est pas dans M si et seulement si il existe n tel que $|z_n| > 2$. Mais, comme on ne connaît pas a priori la valeur de n, on va se contenter de faire les tests jusqu'à $n = 100$.

On considère un quadrillage 100 x 100 de ce rectangle et l'on parcourt tous les points de ce quadrillage à l'aide de deux boucles `do` imbriquées. Quand un point c de coordonnées x, y est dans M, on le marque en noir par une procédure graphique `(draw-point x y)`.

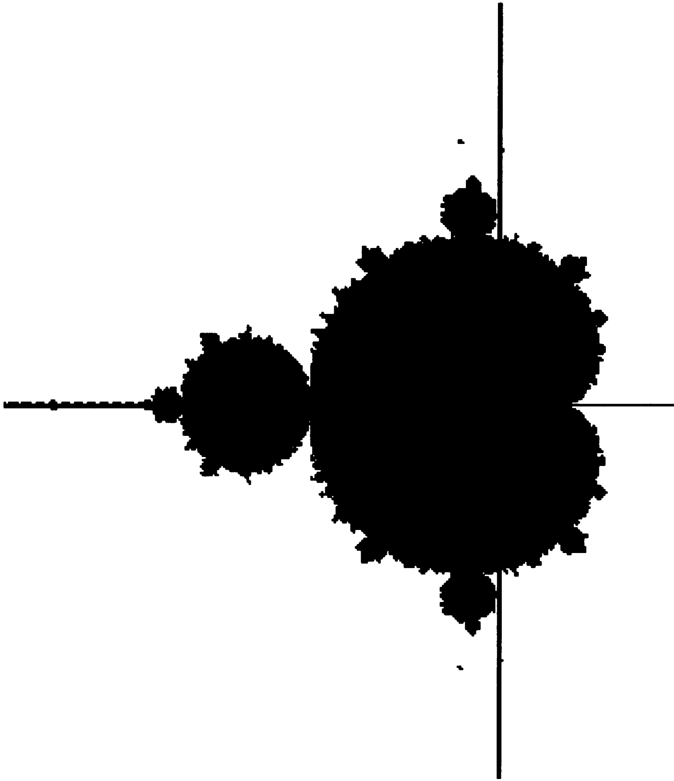
```
(define (mandelbrot)
  (do ((i -200 (+ i 1)))
      ((= i 50))
    (do ((j -125 (+ j 1)))
        ((= j 125))
      (if (appartient? (make-rectangular (/ i 100.0) (/ j 100.0)))
        (draw-point i j)))))
```

Pour tester l'appartenance à M du point représenté par le nombre complexe c, on utilise la fonction :

```
(define (appartient? c)
  (letrec ((loop
             (lambda (n  z)
               (cond ((< 100 n) #t)
                     ((< 2 (magnitude z)) #f)
                     (else (loop (+ n 1) (+ c (* z z))))))))
    (loop 0 c)))
```

qui rend `#t` si au cours de cent itérations on a toujours $|z_n| \leq 2$.

Des agrandissements de la frontière montrent que chaque partie reflète la structure du tout, mais j'ignore s'il existe une définition récursive pour cet ensemble.



Ensemble de Mandelbrot

6.6 Calculer avec les vecteurs

Grâce au type prédéfini `vector` présenté au chapitre 4, il est facile de se faire une petite bibliothèque de calcul avec les vecteurs. On y ajoute quelques fonctions plus géométriques.
On additionne les vecteurs en procédant composante par composante :

```
(define (add-vector  V W)
  (map-vector + V W))
```

La multiplication d'un vecteur `V` par un nombre `k` se fait aussi composante par composante :

```
(define (k-mul-vector  k V )
  (map-vector (lambda (x)(* k x)) V))
```

Le produit scalaire de deux vecteurs de même longueur consiste à ajouter les produits des composantes. On utilise une boucle `do` pour faire cette somme :

```
(define (produit-scalaire V W)
  (let ((m (vector-length V)))
    (do ((i 0 (+ i 1))
         (resultat 0 (+ resultat (* (vector-ref V i)(vector-ref W i)))))
        ((= i m) resultat)
      ))) ;; corps vide

? (produit-scalaire '#(1 2 3) '#(4 5 6)) -> 32
```

Exercice 9 *Définir une fonction qui calcule le produit vectoriel de deux vecteurs de l'espace* $\boldsymbol{R}^3$.

6.7 Calculer avec les matrices

Représentation des matrices

Les matrices permettent de manipuler des tableaux de nombres comme

$$\begin{pmatrix} 5 & 0 & -3 & 7 \\ 4 & 12 & 8 & 2 \\ 1 & 9 & 5 & 0 \end{pmatrix}$$

Ce qui fait leur importance provient des opérations que l'on peut effectuer sur elles. Ces opérations peuvent se comprendre en se rappelant qu'une matrice est aussi la représentation d'une application linéaire dans des bases. Les colonnes de la matrice étant les composantes, dans l'espace d'arrivée, des transformées des vecteurs de la base de l'espace de départ.
Il n'y a pas de type prédéfini matrice en Scheme, aussi fait-on un choix de représentation. On représente une matrice par un vecteur dont chaque composante est un vecteur qui représente une ligne.
Avec cette convention, la matrice précédente sera représentée par le vecteur :

```
#( #(5   0   -3   7)
   #(4  12    8   2)
   #(1   9    5   0))
```

On définit des accesseurs et des modificateurs pour la structure de matrice.

```
(define nb-lignes vector-length)

(define (nb-colonnes M)
  (vector-length (vector-ref M 0)))
```

Comme pour les vecteurs, les colonnes et les lignes d'une matrice seront numérotées à partir de 0 :

```
(define (ieme-ligne M i)
  (vector-ref M i))

(define (jeme-colonne M j)
```

```
  (map-vector (lambda (V)(vector-ref V j))
              M))
```

On accède à l'élément M_{ij} situé la ième ligne et jème colonne d'une matrice M par :

```
(define (Matrice-ref M ligne-i  colonne-j)
  (vector-ref (vector-ref M ligne-i) colonne-j))
```

On modifie la valeur de l'élément M_{ij} par :

```
(define (matrice-set! M ligne-i colonne-j  nouvelle-valeur)
  (vector-set! (vector-ref M ligne-i) colonne-j  nouvelle-valeur))
```

Opérations sur les matrices

Les opérations algébriques de base sur les matrices sont l'addition, la multiplication par un nombre, la transposition et le produit.

L'addition de deux matrices de même taille est une matrice de même taille $S = M + N$, obtenue en ajoutant les coefficients correspondants : $S_{ij} = M_{ij} + N_{ij}$.

```
(define (add-matrice M N)
  (map-vector add-vector M N))
```

La multiplication d'une matrice M par un nombre k consiste à multiplier tous les coefficients de la matrice par k :

```
(define (k-mul-matrice k M)
  (map-vector (lambda (V)(k-mul-vector  k V )) M))
```

La transposée d'une matrice M est une matrice notée ${}^tM$ telle que ${}^tM_{ij} = M_{ji}$. Autrement dit, les lignes de ${}^tM$ sont les colonnes de M. D'où la fonction :

```
(define (transpose  M)
  (let ((n (nb-lignes M)))
    (do ((i 0 (+ i 1))
         (tM (make-vector n)))
        ((= i n) tM)
      (vector-set! tM i (jeme-colonne M i)))))
```

La multiplication de deux matrices rectangulaires M , N est possible dès que le nombre des colonnes de la première est égal au nombre des lignes de la seconde. Le terme général P_{ij} de la matrice produit $P = MN$ est donné par la formule :

$$P_{ij} = \sum_{k=0}^{n_1-1} M_{ik} * N_{kj}$$

qui peut s'interpréter comme étant le produit scalaire de la ième ligne de M par la jème colonne de N, ou par la jème ligne de ${}^tN$.

```
(define (mul-matrice M N)
  (let ((tN  (transpose-matrice N)))
    (map-vector (lambda (L1)
                  (map-vector (lambda (L2)
                                (produit-scalaire L1 L2))
                              tN))
                M)))

? (mul-matrice '#(#(1 2) #(3 4) #(5 6))  '#(#(0 1 0) #(1 0 1)))
 #(#(2 1 2) #(4 3 4) #(6 5 6))
```

On identifie un vecteur V, disons `#(a b c)`, avec une matrice à `m` lignes et une colonne $\overline{V}$ = `#( #(a) #(b) #(c) )` de sorte que l'application d'une matrice A à ce vecteur revient à effectuer le produit de matrices $A\overline{V}$. Mais cette représentation des vecteurs par des matrices est assez lourde aussi, préfère-t-on définir directement l'application d'une matrice sur un vecteur :

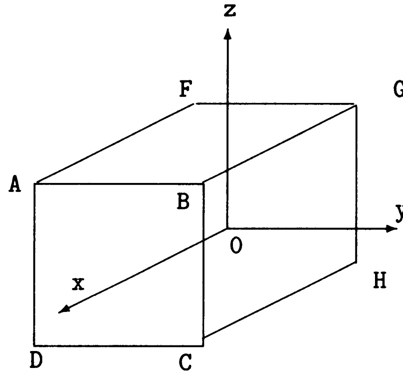
```
(define (app-matrice M V)
  (map-vector (lambda (ligne)
                (produit-scalaire ligne V))
              M))

? (app-matrice '#(#(1 2) #(3 4) #(5 6))  '#(1 1)) -> #(3 7 11)
```

Les matrices sont utilisées dans des domaines très variés. Citons l'analyse numérique où la discrétisation des problèmes transforme des équations différentielles en systèmes linéaires, les statistiques où l'on doit manipuler des tableaux de nombres, la robotique où l'on doit modéliser la composition des mouvements de solides articulés, ... A titre d'illustration, on va donner en complément une application des matrices à la visualisation par ordinateur.

6.8 Compléments : visualisation d'un cube en rotation

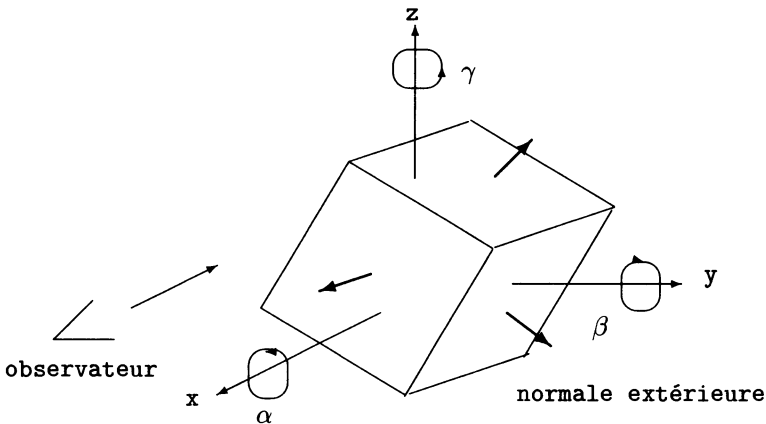
Le problème

On s'intéresse à la visualisation par ordinateur d'objets de l'espace à trois dimensions. On considère un problème très simple mais typique : on visualise un cube en rotation autour d'un axe. On sait qu'un excellent moyen pour aider à la perception d'un objet dans l'espace est de l'animer d'un mouvement de rotation autour d'un axe et d'afficher la succession des vues que le mouvement engendre.
On se donne un repère orthonormé $0xyz$ et l'on considère un cube centré en 0 et dont les faces sont parallèles ou perpendiculaires aux axes.
On commence par placer ce cube dans une position plus générique en lui faisant subir une rotation d'angle α autour de $0x$ puis d'angle β autour de $0y$. Ensuite, on lui fait faire un tour autour de l'axe $0z$. Pour visualiser cette dernière rotation, on la fractionne en n pas et l'on affiche, après chaque fraction de tour, ce que voit un observateur situé à l'infini sur l'axe $0x$.



Le cube

On suppose que le cube n'est pas transparent, aussi faut-il distinguer les faces visibles des faces cachées. Cette distinction est facile à faire ici : une face est visible quand le rayon lumineux qui va de cette face à l'œil de l'observateur fait un angle aigu avec la normale extérieure. Dans notre cas de figure, une face sera donc visible quand la composante sur $0x$ de sa normale extérieure est positive.



Cube en position générique

Représentation interne d'un cube

Voici la représentation du cube dans sa position initiale, L désigne la demi-longueur du côté du cube.

```
(define  L 50)
(define -L (- L))
```

Les huit sommets sont représentés par les vecteurs :

```
(define A (vector  L -L  L))
(define B (vector  L  L  L))
(define C (vector  L  L -L))
(define D (vector  L -L -L))
(define E (vector -L -L -L))
(define F (vector -L -L  L))
(define G (vector -L  L  L))
(define H (vector -L  L -L))
```

Les normales extérieures aux six faces sont définies par :

```
(define N-ABCD  '#(1  0  0))
(define N-DEFA  '#(0 -1  0))
(define N-ABGF  '#(0  0  1))
(define N-FEHG  '#(-1 0  0))
(define N-GBCH  '#(0  1  0))
(define N-HEDC  '#(0  0 -1))
```

Enfin, on représente un cube par la liste de ses huit sommets et de ses six normales.

```
(define cube0 (list A B C D E F G H
                    N-ABCD N-DEFA N-ABGF N-FEHG N-GBCH N-HEDC))
```

Pour recupérer la liste des sommets de chaque face d'un cube, on définit des fonctions qui construisent pour chaque face la liste de ses quatre sommets et on ajoute en tête la normale extérieure :

```
(define (faceABCD cube)
  (list (list-ref cube 8)
        (list-ref cube 0)(list-ref cube 1)(list-ref cube 2)(list-ref cube 3)

(define (faceDEFA cube)
  (list (list-ref cube 9)
        (list-ref cube 3)(list-ref cube 4)(list-ref cube 5)(list-ref cube 0)

(define (faceABGF cube)
  (list (list-ref cube 10)
        (list-ref cube 0)(list-ref cube 1)(list-ref cube 6)(list-ref cube 5)

(define (faceFEHG cube)
  (list (list-ref cube 11)
        (list-ref cube 5)(list-ref cube 4)(list-ref cube 7)(list-ref cube 6)

(define (faceGBCH cube)
  (list (list-ref cube 12)
        (list-ref cube 6)(list-ref cube 1)(list-ref cube 2)(list-ref cube 7)

(define (faceHEDC cube)
  (list (list-ref cube 13)
        (list-ref cube 7)(list-ref cube 4)(list-ref cube 3)(list-ref cube 2)
```

Il est commode de regrouper toutes ces fonctions dans une liste :

```
(define *Lfaces* (list faceABCD faceDEFA faceABGF faceFEHG faceGBCH faceHEDC)
```

On accède aux sommets et à la normale extérieure d'une face par les fonctions :

```
(define (sommet i une-face)
  (list-ref une-face i))

(define (normale-ext une-face)
  (list-ref une-face 0))
```

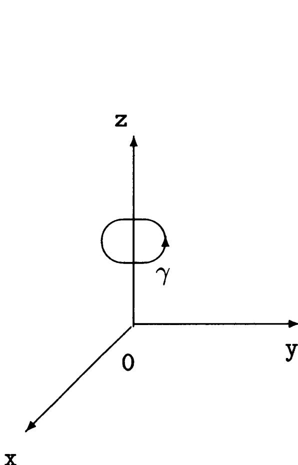
Rotation d'un cube

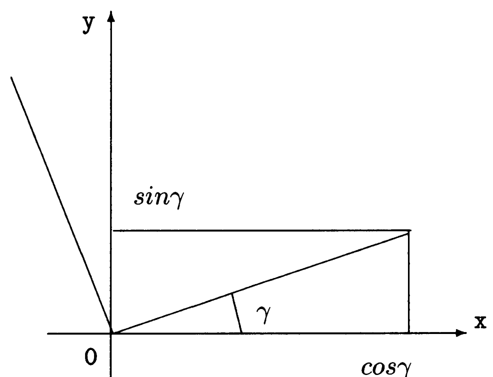
Les matrices de rotation autour d'un axe sont faciles à calculer. La matrice de la rotation d'angle γ autour de l'axe $0z$ conserve les composantes en z, il suffit donc de considérer sa restriction au plan xOy qui est une rotation plane d'angle γ. Dans le plan, le transformé du vecteur unitaire de l'axe 0x a pour composantes cos γ et sin γ et le transformé du vecteur unitaire de l'axe Oy a pour composantes $-sin(\gamma)$ et $cos(\gamma)$ d'où la matrice dans le plan (voir la figure qui suit).

$$\begin{pmatrix} cos(\gamma) & sin(\gamma) \\ -sin(\gamma) & cos(\gamma) \end{pmatrix}$$

eE finalement la matrice de la rotation d'angle γ autour de $0z$ est

$$\begin{pmatrix} cos(\gamma) & sin(\gamma) & 0 \\ -sin(\gamma) & cos(\gamma) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$





Rotation autour de 0z

Rotation dans le plan x0y

Sa représentation en Scheme est :

```
(define (rot-z gamma)
  (vector (vector (cos gamma) (- (sin gamma)) 0)
          (vector (sin gamma) (cos gamma)     0)
          (vector    0            0         1)))
```

De façon symétrique, on obtient les matrices de rotation autour de $0x$ et Oy.

```
(define (rot-x alpha)
  (vector (vector 1    0     0)
          (vector 0 (cos alpha) (- (sin alpha)))
          (vector 0 (sin alpha) (cos alpha))))

(define (rot-y beta)
  (vector (vector (cos beta) 0 (- (sin beta)))
          (vector    0       1            0)
          (vector (sin beta) 0 (  cos beta))))
```

La transformation d'un cube par une matrice consiste à appliquer la matrice à ses sommets et ses normales.

```
(define (transformer-cube matrice cube)
  (map  (lambda (v)(app-matrice matrice v))
        cube))
```

Visualisation d'un cube en rotation

On doit dessiner les projections sur le plan $y0z$ des faces visibles.
On définit l'opération de projection qui associe à un vecteur `V = #(x y z)` la liste `(y z)` de ses composantes sur $y0z$.

```
(define (Lproj/y0z V)
  (list (vector-ref V 1)(vector-ref V 2)))
```

Pour savoir si une face est visible, on introduit le prédicat :

```
(define (visible? une-face)
  (<= 0 (vector-ref (normale-ext une-face) 0)))
```

Le dessin de la projection d'une face se fait en plaçant le crayon sur le premier sommet puis en traçant les quatre segments joignant les sommets successifs.

```
(define (visualise-face  face cube)
  (let ((la-face  (face cube)))
    (if (visible? la-face)
      (begin
        (apply move-to  (Lproj/y0z (sommet 1 la-face)))
        (apply line-to  (Lproj/y0z (sommet 2 la-face)))
        (apply line-to  (Lproj/y0z (sommet 3 la-face)))
        (apply line-to  (Lproj/y0z (sommet 4 la-face)))
        (apply line-to  (Lproj/y0z (sommet 1 la-face)))))))
```

Pour cela, on a utilisé des fonctions graphiques `move-to, line-to`[2]. On suppose que l'on dispose des deux fonctions suivantes :

`(move-to x y)` : positionne le crayon graphique au point de coordonnées entières obtenues à partir des nombres réels x et y.
`(line-to x y)` : trace un segment de la position courante du crayon à la nouvelle au point de coordonnées entières obtenues à partir des nombres réels x et y.

La visualisation du cube consiste à dessiner la projection de chacune de ses faces visibles :

```
(define (visualise-cube cube  visualise-face-cache?)
  (map (visualise-face visualise-face-cache?)
       cube))
```

Finalement, l'animation de la rotation du cube autour de $0z$ se fait en deux temps :

- on place le `cube0` dans une position `cube2` après une rotation d'angle α autour de $0x$ puis d'angle β autour de $0y$,
- on discrétise la rotation autour de l'axe $0z$ en N parties d'angle $2\pi/N$. Après chaque rotation d'angle $2\pi/N$ on visualise le cube obtenu. On doit effacer la visualisation précédente avant chaque réaffichage par une troisième primitive graphique `(clear)`. Cette primitive peut, par exemple, effacer toute la fenêtre graphique.

  Il est judicieux de préparer le calcul de la nouvelle position avant de demander l'effacement de façon à pouvoir réafficher aussitôt.

```
(define (visualise-rotation-cube alpha beta Nb-pas)
  (let* ((cube1 (transformer-cube  (rot-x alpha) cube0))
         (cube2 (transformer-cube  (rot-y beta)  cube1))
         (2pi 6.28)
         (gamma (/ 2pi Nb-pas ))
         (rotation (rot-z gamma)))
    (do ((i 0 (+ i 1))
         (cube cube2 (transformer-cube rotation cube)))
        ((= i Nb-pas))
      (clear)
      (visualise-cube cube)
      )))

? (visualise-rotation-cube 0.3 0.7 15 #t)
```

Remarque 2 On peut étendre facilement ce petit système de visualisation à des polyèdres convexes généraux. Il suffit de se donner un polyèdre par la liste de ses sommets et normales et de définir les fonctions pour accéder à chacune de ses faces.
Si l'on dispose d'un écran avec des niveaux de gris, on pourra visualiser les différentes intensités d'éclairage entre les faces en utilisant, pour chaque face visible, un niveau d'éclairement proportionnel au produit scalaire du vecteur unitaire de $0x$ avec la normale extérieure à la face.

[2] La norme Scheme ne parle pas de fonctions graphiques mais beaucoup d'implantations en proposent.

Architecture du programme de visualisation du cube

`(visualise-rotation-cube alpha beta Nb-pas)`
La fonction principale qui affiche à l'écran l'aspect d'un cube tournant autour de l'axe $0z$. L'observateur est situé dans la direction de l'axe Ox. Au départ le cube est placé dans une position générique par une rotation d'angle α autour de Ox et d'angle β autour de Oy.

> `(rot-z gamma)`
> La matrice de rotation d'angle γ autour de l'axe $0z$.
>
> `(rot-x alpha)`
> La matrice de rotation d'angle α autour de l'axe $0x$.
>
> `(rot-y beta)`
> La matrice de rotation d'angle β autour de l'axe $0y$.
>
> `(transformer-cube matrice cube)`
> Le cube transformé par une matrice de rotation.
>
> `(clear)`
> Ordre graphique pour effacer l'écran graphique.
>
> `(visualise-cube cube)`
> Dessine la projection d'un cube sur le plan $y0z$.
>
>> `(visualise-face face cube)`
>> Dessine la projection d'une face si elle est visible.
>>
>>> `(proj/y0z V)`
>>> Projection d'un vecteur sur le plan $y0z$.
>>>
>>> `(move-to x y)`
>>> Ordre graphique pour placer le crayon.
>>>
>>> `(line-to x y)`
>>> Ordre graphique pour tracer un trait du point courant à un point donné.
>
> `cube0`
> Représentation du cube de sommets `A B C D E F G H` en position initiale.
>
>> `faceABCD`
>> La fonction qui rend la liste `(list N-ABCD A B C D)`.
>>
>> ...
>>
>>> `(normale-ext face)`
>>> La fonction qui donne la normale extérieure d'une face.

`(sommet i face)`
La fonction qui donne le ième sommet d'une face.
...

A
Le vecteur représentant la position initiale du point. **A**
...

6.9 De la lecture

Les principaux algorithmes sur les entiers sont analysés en détail dans [Knu73].
Pour le calcul numérique on peut se référer à [SB80].
Un ouvrage très riche sur le chaos et les fractals [PJS92].
Les problèmes de visualisation sont traités dans [FvDFH95].

Chapitre 7

Données structurées et algorithmes

A façon dont on écrit un programme dépend en grande partie de la manière dont on modélise les données du problème. C'est la fameuse égalité donnée par N. Wirth comme titre à l'un de ses livres :

`Algorithms + Data Structures = Programs`.

Une structure de données est définie par les propriétés que doivent satisfaire les opérations sur cette structure. C'est comme en mathématiques, une structure algébrique (groupe, anneau, espace vectoriel, ...) est définie par les axiomes que doivent vérifier les lois de cette structure. L'implantation d'une structure de données consiste à fournir une représentation de ses opérations à l'aide de constructions dans un langage déterminé. On y ajoute parfois un afficheur pour associer une forme visible à la représentation utilisée et un lecteur pour l'opération inverse qui consiste à passer de l'apparence externe à la structure interne.
Comme ce livre n'est pas un cours d'algorithmique, on se contentera de décrire quelques représentations en Scheme des structures de données les plus courantes : pile, file, table, arbre, graphe. Notons que ce langage nous fournit déjà des représentations pour des structures comme : nombres, listes, chaînes, vecteurs, ...

7.1 Structures de données et types abstraits

Une structure de données est construite par agrégat de structures plus élémentaires qui en forment les constituants. Les opérations qui permettent d'accéder aux constituants d'une structure s'appellent les *accesseurs*, celles qui permettent de construire une nouvelle structure à partir de constituants s'appellent des *constructeurs*. On introduit parfois des *prédicats* pour distinguer des sous-types de structure. On peut aussi considérer des opérations qui modifient les composants d'une structure, ce sont les *modificateurs*. Pour mettre en garde le lecteur, on applique aux modificateurs la convention Scheme de terminer leur nom par le caractère !.

Par exemple, la structure de doublet possède :
- les accesseurs `car` et `cdr`,
- le constructeur `cons`,
- les modificateurs :`set-car!` et `set-cdr!`,
- le prédicat `pair?`.

Pour certaines structures on peut envisager deux points de vue :

- le point de vue fonctionnel où toutes les opérations sur les structures sont des *fonctions* : les opérations créent de nouveaux objets,
- le point de vue impératif où certaines opérations réalisent des *modifications*, on dit aussi des structures *mutables*.

Dans l'exemple des doublets, les opérations de mutation `set-car!` et `set-cdr!` ne font donc pas partie de l'aspect fonctionnel.
Les propriétés que doivent vérifier les opérations relatives à une structure constituent sa définition sémantique. Ainsi, les opérations sur les aspects fonctionnels des doublets sont des fonctions dont on doit préciser :
- le domaine de définition et le domaine des valeurs, c'est la *signature* ou encore le type des opérations,
- les relations qu'elles vérifient.

Dans le cas des doublets, on a les signatures suivantes :

	domaine de définition		domaine des valeurs
`cons`	S-expression × S-expression	→	doublet
`car`	doublet	→	S-expression
`cdr`	doublet	→	S-expression
`pair?`	S-expression	→	bool

Relations sémantiques (`s1` et `s2` désignent des S-expressions quelconques) :

```
(car   (cons s1 s2)) = s̄1
(cdr   (cons s1 s2)) = s̄2
(pair? (cons s1 s2)) = #t
```

Ces relations décrivent le *type abstrait* de la structure ; le qualificatif d'abstrait signifie que ce sont des propriétés indépendantes de toute représentation.
Pour spécifier les aspects mutables, on doit ajouter :

`set-car!`	doublet x S-expression	→	`indefinie`.
`set-cdr!`	doublet x S-expression	→	`indefinie`.

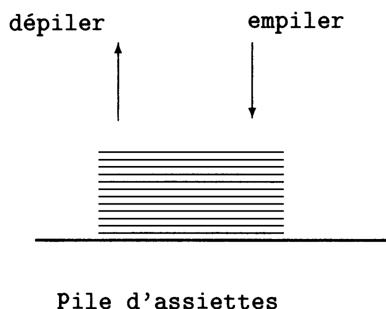
En général, les modificateurs ne retournent pas de valeurs précises, ils sont seulement utilisés pour réaliser un changement d'état de la structure. On doit donc spécifier l'état après une modification.

Soit `d` une expression à valeur doublet :
après `(set-car! d s)` on a `(car d) = s̄`
après `(set-cdr! d s)` on a `(cdr d) = s̄`

On commence par l'archétype des structures de données : la pile.

7.2 Piles fonctionnelles

Le mot pile fait partie du langage courant et la pile d'assiettes est la première image qui vient à l'esprit. Ici la notion est plus précise car on exige que les opérations d'ajout ou de retrait d'assiettes se fassent toujours par l'assiette du dessus (en réalité, on ne procède pas ainsi afin d'éviter d'utiliser toujours les mêmes assiettes !). En bref, on dit qu'une pile fonctionne selon la règle du «dernier entré premier sorti».



Pile d'assiettes

Une pile est une suite finie d'objets d'un certain type `E` permettant de faire les opérations permises par son type abstrait.

Le type abstrait pile

- `vide?` : est le prédicat pour tester s'il y a ou non des éléments dans la pile.
- `vide` : fonction pour créer une pile vide.
- `depiler` : quand la pile est non vide, on obtient une nouvelle pile en supprimant un élément appelé sommet
- `empiler` : c'est ajouter un élément à la pile, cet élément devient le sommet de la nouvelle pile.
- `sommet` : désigne l'élément que l'on peut dépiler d'une pile non vide.

Désignons par PILE l'ensemble des piles possibles et par PILE* les piles non vides. On peut axiomatiser les propriétés que doivent satisfaire les opérations précédentes. Cette axiomatisation définit le type abstrait des piles. Le point de vue fonctionnel conduit à préciser d'abord le type de ces fonctions :

pile:vide?	PILE	$\to$	BOOL
pile:vide		$\to$	PILE
pile:depiler	PILE*	$\to$	PILE
pile:empiler	E $\times$ PILE	$\to$	PILE*
pile:sommet	PILE*	$\to$	E

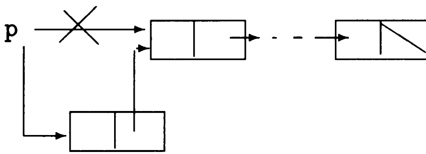
Pour éviter les confusions avec des fonctions analogues sur d'autres types de données, on a préfixé les noms par `pile:`.
On impose les relations sémantiques suivantes (`x` désigne un élément de E et `p` une pile) :

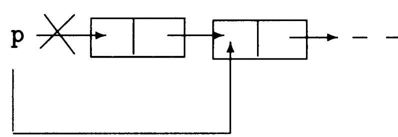
```
(pile:vide?     (pile:vide))          = #t
(pile:vide?     (pile:empiler  x p))  = #f
(pile:depiler   (pile:empiler  x p))  = p̄
(pile:sommet    (pile:empiler  x p))  = x̄
```

Pour la structure de PILE les opérations `pile:vide` et `pile:empiler` sont des constructeurs et les opérations `pile:sommet` et `pile:depiler` des accesseurs.

Une implantation des piles

Il y a beaucoup de façons d'implanter la structure de pile en Scheme ; la méthode la plus naturelle consiste à représenter le contenu d'une pile par une liste. Les opérations d'empilement et de dépilement se font en tête de liste. Chaque empilement entraîne la création d'un nouveau doublet et on ne récupère pas directement[1] les doublets dépilés.





Empilement Dépilement

```
(define (pile:vide)  '())

(define pile:vide?  null?)

(define (pile:empiler  x p)
  (cons x  p))

(define (pile:depiler p)
  (if (pile:vide? p)
    (display-alln "erreur  :  la pile est vide ")
    (cdr  p)))

(define  (pile:sommet  p)
  (if (pile:vide? p)
    (display-alln "erreur  :  la pile est vide ")
    (car  p)))
```

On vérifie immédiatement que cette implantation satisfait aux axiomes (ici `E` désigne n'importe quelle classe d'objets Scheme). On verra au chapitre 8 comment définir un échappement en cas d'erreur, il remplacerait avantageusement le simple affichage utilisé ici.

[1]On les récupère indirectement par l'opération de ramassage des doublets inutilisés, cf. chapitre 22.

Mini éditeur ligne

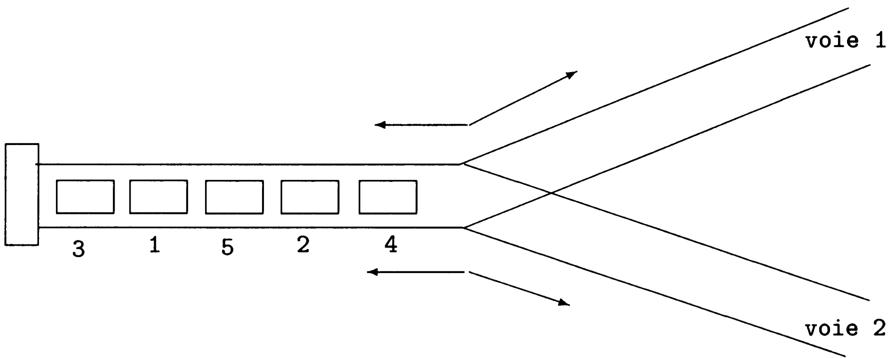
Comme illustration de la structure de pile, construisons un mini éditeur ligne. On donne une chaîne contenant les commandes d'édition. Il s'agit d'afficher la suite des caractères résultant de l'exécution de ces commandes. Ces caractères sont stockés dans un tampon. Chaque caractère est une commande ; on distingue quatre catégories de caractères :

- tout caractère différent des trois caractères # , $ et `newline` est un caractère du texte, on le stocke dans le tampon ;
- le caractère # signifie supprimer le caractère précédent dans le tampon ;
- le caractère $ signifie supprimer tous les caractères du tampon ;
- le caractère de passage à la ligne signifie la fin de la chaîne de commande et déclenche l'affichage du tampon constitué par les caractères du texte.

Pour réaliser ces opérations, on représente le tampon par une pile de caractères. La fonction locale `loop` lit les caractères successifs de la chaîne de commande, l'indice `i` sert à repérer le caractère courant.

```
(define (executer-commandes chaine-commande)
  (let ((L (string-length chaine-commande)))
    (letrec ((loop
               (lambda (i tampon)
                 (if (< i L)
                   (let ((car-lu (string-ref chaine-commande i)))
                     (case car-lu
                       ((#\#) (loop (+ i 1) (pile:depiler tampon)))
                       ((#\$) (loop  (+ i 1) (pile:Vide)))
                       ((#\newline)(map display (reverse tampon)))
                       (else (loop (+ i 1)
                                   (pile:empiler  car-lu tampon)))))
                   (display-alln "erreur la commande doit se terminer
                                            par #\newline")))))
       (loop 0 (pile:Vide)))))

? (executer-commandes "ABC$abce#de#\newline")
abcde
```

Exercice 1 *Pour trier les wagons d'un train situé sur une voie de garage, on dispose d'un aiguillage et des voies 1 et 2. En modélisant la voie de garage par une pile, indiquer la suite des opérations à effectuer pour mettre les wagons dans l'ordre 5 4 3 2 1.*



On verra de nombreuses autres applications de la structure de pile. Le plus souvent les piles servent à stocker temporairement des informations sachant que l'on aura besoin d'utiliser en premier la dernière stockée.

7.3 Piles mutables

Les piles mutables sont plus proches de notre intuition d'une pile : quand on retire ou ajoute une assiette, c'est toujours la même pile qui est concernée.

Le type abstrait des piles mutables

Les opérations d'empilement et de dépilement réalisent un changement d'état de la pile. Ces modifications se nomment `empiler!` et `depiler!`. On désigne par PILE! l'ensemble des piles mutables avec les relations suivantes :

`pile!:Vide?`	PILE!	→	BOOL
`pile!:Vide`		→	PILE!
`pile!:empiler`	E × PILE!	→	PILE!
`pile!:depiler`	PILE!	→	E
`pile!:sommet`	PILE*!	→	E

Notons que la fonction `pile!:depiler` rend l'élément dépilé.
Après exécution de `(pile!:empiler s p)`, on a `(pile!:sommet p) =` $\bar{s}$
Après exécution de `(pile!:depiler (pile!:empiler s p))`, on est revenu dans l'état de `p`

Implantations des piles mutables

On va donner deux implantations des piles mutables : la première est basée sur la mutation des listes et l'autre sur les prototypes.

Implantation avec listes mutables

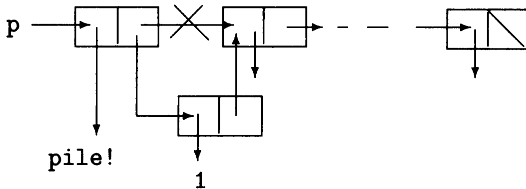
On a vu au chapitre 5 §3 pourquoi l'on ne peut espérer définir l'empilement avec une définition du genre suivant :

```
(define  (pile!:empiler  x pile)
  (set! pile (cons x pile)))
```

Pour utiliser les propriétés de mutation des listes, on représente la pile vide par une liste non vide dont on pourra modifier ensuite le cdr.

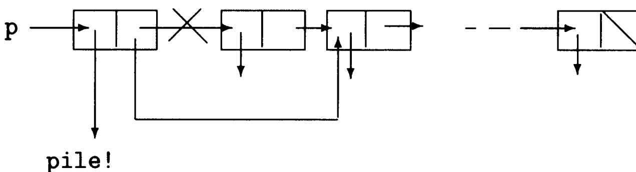
```
(define   (pile!:Vide)  (list   'pile!))

(define  (pile!:Vide?  e)
  (and (pair? e)(null? (cdr e))(eq?  'pile! (car e))))
```

Pour dépiler ou empiler une pile p, la variable p désigne toujours le même doublet. On procède à une modification physique de (cdr p) comme indiqué dans les schémas ci-dessous :



Empilement de la valeur 1

```
(define  (pile!:empiler  s p)
  (set-cdr! p (cons s (cdr p)))
   p)

(define (pile!:sommet   p)
  (if (pile!:Vide? p)
    (display-alln "erreur : pile vide ")
    (cadr p)))
```



Dépilement

```
(define (pile!:depiler p)
    (if (pile!:Vide? p)
```

```
      (display-alln "erreur : pile vide ")
      (let ((sommet (cadr p)))
        (set-cdr! p (cddr p))
        sommet)))
```

Exercice 2 *Définir et implanter la notion de pile mutable bornée, c'est-à-dire dont le nombre maximum d'éléments est borné par une constante* **nbMax**. *On pourra, par exemple, représenter une pile par un vecteur et utiliser un entier pour désigner l'indice du sommet de pile.*

Implantation avec des prototypes

Voici une autre implantation avec des prototypes selon le schéma décrit à la fin du §3 chapitre 5. Une pile est un prototype qui répond à des messages pour modifier (ou observer) son contenu.

```
(define (make-pile!:vide)
  (let ((contenu '()))
    (lambda (message . Largt)
      (case message
        ((pile!:vide?) (null? contenu))
        ((pile!:empiler)
         (set! contenu (cons (car Largt) contenu)) contenu)
        ((pile!:depiler)
         (if (null? contenu)
           (display-alln "erreur : pile vide ")
           (let ((sommet (car contenu)))
             (set! contenu (cdr contenu))
             sommet)))
        ((pile!:sommet)
         (if (null? contenu)
           (display-alln "erreur : pile vide ")
           (car contenu)))))))
```

On construit un prototype p1 pour tester :

```
(define p1 (make-pile!:vide))

? (p1 'pile!:empiler 1)
? (p1 'pile!:empiler 2)

? (p1 'pile!:sommet)  -> 2
? (p1 'pile!:depiler) -> 1
```

Exercice 3 *Ajouter la possibilité de traiter le message* **voir-contenu** *qui provoque l'affichage de "pile de contenu: " suivi de la liste des valeurs contenues dans la pile.*

Pour disposer d'une interface plus agréable, on peut ajouter les définitions :

```
(define (pile!:empiler s p)
  (p 'pile!:empiler s))

(define (pile!:depiler p)
  (p 'pile!:depiler!))

(define (pile!:sommet p)
  (p 'pile!:sommet))

(define (pile!:vide? p)
  (p 'pile!:vide?))
```

7.4 Evaluation des expressions arithmétiques postfixées

L'écriture usuelle d'une expression comme $(7 + 3) * \sqrt{16}$ est dite *infixée.* Pour évaluer à la main cette expression on commence par évaluer l'addition et la racine carrée ; on stocke ces valeurs : 10 et 4 puis on effectue la multiplication. Pour cela, notre œil fait immédiatement une analyse de l'expression complète pour distinguer les deux opérandes de la multiplication.
Pour ne parcourir qu'une seule fois l'expression, on aimerait pouvoir faire le calcul au fur et à mesure que l'on avance de la gauche vers la droite dans l'expression. A cette fin, on observe que pour calculer la valeur d'un opérateur, *on doit commencer par calculer les valeurs de ses arguments.* D'où l'idée de la *notation postfixée* : on écrit en premier les arguments et ensuite l'opérateur correspondant. Ainsi la sous-expression `7 + 3` s'écrit en postfixe `7 3 +`, l'expression précédente devient `7 3 + 16 rac2 *`. Notons que les parenthèses ont disparu car une expression postfixe n'est pas ambiguë.

Voici comment on peut l'évaluer en la lisant de gauche à droite :
On lit le nombre 7 → on le stocke.
On lit le nombre 3 → on le stocke.
On lit l'opérateur binaire + → on effectue l'opération avec les deux valeurs précédemment stockées. On peut alors les oublier et on stocke le résultat 10.
On lit le nombre 16 → on le stocke.
On lit l'opérateur unaire $\sqrt{\ }$ → on effectue l'opération avec la valeur précédemment stockée. On peut alors l'oublier et on stocke le résultat 4.
On lit l'opérateur binaire `*` → on effectue l'opération avec les deux valeurs précédemment stockées, on les oublie et on stocke le résultat 40.
Quand on a lu toute l'expression, on retourne la dernière valeur trouvée, c'est le résultat : 40.

Remarque 1 Le fait de pouvoir évaluer en une seule passe une expression postfixée a fait choisir ce mode de représentation dans de nombreuses machines réelles ou virtuelles : calculettes, langage de programmation Forth, langage de description de page PostScript.

Le stockage des valeurs des opérandes puis leur effacement dès que l'on a effectué l'opération, conduit naturellement à stocker ces valeurs temporaires dans une pile. D'où une méthode systématique pour évaluer une expression postfixée. On parcourt l'expression de gauche à droite :
- quand on lit un nombre n → on l'empile,
- quand on lit un opérateur à p arguments → on calcule sa valeur avec les valeurs des sommets obtenus en dépilant p fois et on empile ce résultat,
- quand on a fini de lire l'expression, sa valeur est en sommet de la pile.

Voici une fonction qui applique cette méthode, on se borne à considérer les opérateurs : `+ , - , * , / , sqrt`.
On utilise une pile mutable pour faire les sauvegardes.

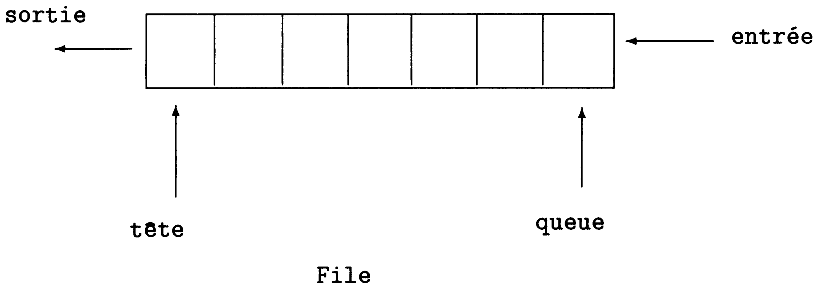
```
(define (calculette exp-postfixe)
  (let ((pile (make-pile!:vide)))
    (letrec ((evaluer
              (lambda (exp)
                (if (null? exp)
                  (pile!:sommet pile)
                  (let ((s (car exp)))
                    (if (number? s)
                      (pile!:empiler s pile)
                      (begin
                        (case s
                          ((sqrt)(let ((v (pile!:depiler pile)))
                                   (pile!:empiler (sqrt v) pile)))
                          ((+)  (let ((v2 (pile!:depiler pile))
                                      (v1 (pile!:depiler pile)))
                                  (pile!:empiler (+ v1 v2) pile)))
                          ((*)  (let ((v2 (pile!:depiler pile))
                                      (v1 (pile!:depiler pile)))
                                  (pile!:empiler (* v1 v2) pile)))
                          ((-)  (let ((v2 (pile!:depiler pile))
                                      (v1 (pile!:depiler pile)))
                                  (pile!:empiler (- v1 v2) pile)))
                          ((/)  (let ((v2 (pile!:depiler pile))
                                      (v1 (pile!:depiler pile)))
                                  (pile!:empiler (/ v1 v2) pile))))
                        (evaluer (cdr exp)) )))))))
      (evaluer exp-postfixe))))

? (calculette '(7 3 + 16 sqrt *)) -> 40
```

Cette méthode d'évaluation d'une expression à l'aide d'une pile est à l'origine de la notion de machine à pile qui sera présentée au chap12 §6. L'expression à calculer s'interprète comme une liste d'instructions à effectuer.

Exercice 4 *Donner une autre version de la calculette en utilisant la structure de pile fonctionnelle.*

7.5 Structures de file

La file correspond à l'idée intuitive de la file d'attente, les clients sont servis dans l'ordre d'arrivée : premier entré premier sorti. Une file est donc un ensemble fini ordonné d'éléments d'un type donné `E`. Une file non vide possède un élément de *tête*, le premier candidat à sortir de la file et un élément en *queue*, le dernier arrivé.



Files fonctionnelles

Type abstrait FILE

On désigne par FILE l'ensemble des files et FILE* celles qui sont non vides. La structure de file admet les opérations suivantes :

`file:Vide`		$\rightarrow$	FILE
`file:Vide?`	FILE	$\rightarrow$	BOOL
`file:ajouter`	E $\times$ FILE	$\rightarrow$	FILE*
`file:enlever`	FILE*	$\rightarrow$	FILE
`file:tete`	FILE*	$\rightarrow$	E

Avec les relations :

```
(file:Vide?    (fileVide))          = #t
(file:Vide     (file:ajouter  s f)) = #f
(file:tete     (file:ajouter  s f)) = (if (file:Vide?  f) s  (file:tete f))
(file:enlever  (file:ajouter  s f)) = (if (file:Vide? f)
                                          file:Vide
                                          (file:ajouter s (file:enlever f)))
```

Implantation des files

On a une implémentation simple avec les listes : on ajoute en fin de liste et on enlève en tête de la liste.

```
(define   (file:Vide)  '())

(define  file:Vide?  null?)
```

```
(define (file:ajouter s f)
  (append f (list s)))

(define (file:enlever  f)
  (if (file:Vide? f)
    (display-alln "erreur  file vide ")
    (cdr  f)))

(define  (file:tete f)
  (if (file:Vide? f)
    (display-alln "erreur  file vide ")
    (car  f)))
```

Remarque 2 Cette implantation est correcte mais présente un défaut quand la taille de la file est importante. En effet, l'utilisation de la fonction **append** dans `file:ajouter` est coûteuse en doublets. On donnera plus loin une autre méthode.

Une application des files

Comme application de la structure de file, considérons le problème suivant : on se donne une liste L d'entiers strictement croissants et un entier N. On veut compter le nombre de couples $(x, x+N)$ avec x et $x+N$ présents dans la liste.

Par exemple, si `L = (0 2 3 4 6 7 9 10 12 13 14 16 20)` et `N = 8` alors il y a quatre couples solutions, ce sont : `(2 ,10), (4 ,12), (6 ,14), (12 , 20)`.

Une méthode évidente consiste, pour chaque entier x de la liste L, à regarder au plus les N suivants dans L pour voir si l'on trouve l'entier $x+N$. Mais cette méthode nécessite de parcourir plusieurs fois la liste. Voici une méthode n'utilisant qu'*un seul* parcours.
L'idée consiste à utiliser une file F de longueur $< N$ dans laquelle on examine au fur et à mesure chaque élément x de L :
- si la file F est vide, on y ajoute simplement x,
- si l'élément x est supérieur à tete $+N$, nous sommes sûrs que nous ne rencontrons plus d'élément y tel que $y = tete + N$ aussi peut-on l'enlever de F,
- si l'élément x est égal à $tete + N$, on enlève x de L et on l'ajoute dans F, et l'on comptabilise le couple $(x, tete)$
- si l'élément x est inférieur à $tete + N$, on enlève simplement x de L et on l'ajoute dans F.
D'où une fonction qui compte le nombre de couples $(x, x+N)$ dans la liste strictement croissante L :

```
(define (compter-les-couples  L N)
  (letrec ((compter
             (lambda (L resultat F)
               (cond ((null? L) resultat)
                     ((Ffile:vide? F)
                      (compter (cdr L) resultat (file:ajouter (car L) F)))
                     (else (let ((difference (- (car L) (file:tete File) )))
```

```
                                  (cond
                                    ((< N difference)
                                     (compter L resultat  (file:enlever F)))
                                    ((= N difference)
                                     (compter (cdr L)
                                              (+ resultat 1)
                                              (file:enlever (file:ajouter (car L)
                                                                          F))))
                                    (else (compter (cdr L)
                                                   resultat
                                                   (file:ajouter (car L) F)))
                                    ))))))
    (compter L 0 (file:Vide)))

? (compter-les-couples '(0 2 3 4 6 7 9 10 12 13 14 16 20) 8) -> 4
```

Files mutables

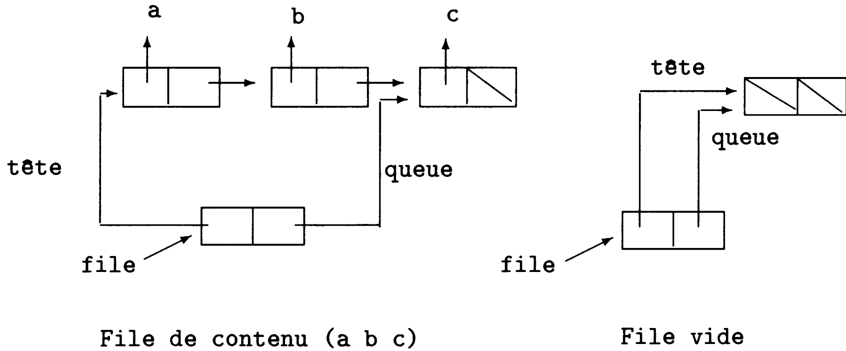
On note FILE! l'ensemble des files mutables et FILE*! celles qui sont non vides. Le type abstrait des files mutables est défini par les procédures :

`file!:Vide`		$\rightarrow$	FILE!
`file!:Vide?`	FILE!	$\rightarrow$	BOOL
`file!:ajouter`	E × FILE!	$\rightarrow$	FILE!*
`file!:enlever`	FILE!*	$\rightarrow$	FILE!
`file!:tete`	FILE!*	$\rightarrow$	E

On a les relations suivantes :

```
(file!:Vide?  (file!:vide))               = #t
(file!:Vide?  (file!:ajouter!  s f ))     = #f

(file!:tete (file!:ajouter!  s f))        = (if (file!:Vide?  f)
                                                s
                                                (file!:tete  f))

(if (file!:Vide? f)
   (file!:enlever! (file!:ajouter! s f)) =  s̄

(if (not (file!:Vide? f))
  ;; l'état après exécution de
   (file!:enlever! (file!:ajouter! s  f))  =
 ;; est le même qu'après exécution de
   (file!:ajouter! s (file!:enlever!  f)))
```

On peut adapter aux files les implantations décrites pour les piles. On a indiqué que l'utilisation de append pouvait être un défaut dans l'implantation fonctionnelle. On élimine cet inconvénient en considérant un pointeur sur la queue de la file. Cela permet d'ajouter un élément en un temps indépendant de la longueur de la file.



File de contenu (a b c)   File vide

Une file mutable est donc un doublet, son car pointe sur la liste constituée des éléments de la file, son cdr pointe sur le dernier doublet de cette liste.

```
(define (file!:vide)
  (cons '() '()))

(define (file!:vide? file)
  (null? (car file)))

(define (file!:tete  file)
  (if (file!:vide? file)
    (display-alln "erreur file vide ")
    (caar file)))

(define (file!:ajouter! s file)
  (let ((doublet (cons s '())))
    (if (null? (car file))
      (set-car! file doublet)
      (set-cdr! (cdr file) doublet))
    (set-cdr! file doublet)))

(define (file!:enlever! file)
  (if (file!:vide? file)
    (display-alln "erreur file vide ")
    (begin (set-car! file (cdar file))
           file)))
```

Exercice 5 *1. Réécrire la fonction* `compter-les-translates` *avec une file mutable.*

2. Reprendre les files fonctionnelles en introduisant un pointeur sur l'élément de queue.

3. Adapter aux files mutables l'implantion par prototypes.

Les files sont surtout utilisées en simulation pour représenter des files d'attente, voir un exemple au chapitre 8 §10.

7.6 Structures de table

Une table est un ensemble fini de correspondances entre des «clés» et des «valeurs» ; à toute clé on associe une valeur. Par exemple, dans un magasin l'association entre un article et son prix sera consignée dans un tarif, en informatique un environnement sera représenté aussi par une table. On peut voir une table comme une généralisation d'un vecteur ayant des indices non entiers ; cette idée sera développée plus loin dans le cadre de l'adressage dispersé. Si l'on dispose d'une fonction des clés vers les valeurs, celle-ci peut naturellement être vue comme une table. Par exemple la table des codes ASCII des caractères se représente par la fonction `char->integer`. Mais il est rare de pouvoir disposer d'une fonction explicite jouant le rôle d'une table. En général, il faudra construire une donnée Scheme de type table.

Quelles sont les opérations sur les tables? Nous avons essentiellement besoin de consulter une table en y recherchant la valeur `v` associée à une clé `c`. L'exemple typique est un dictionnaire. On peut aussi souhaiter mettre à jour la table en y ajoutant ou supprimant un couple `(c , v)`, ou encore en modifiant la valeur associée à une clé comme dans l'exemple de l'actualisation d'un tarif.
Pour définir le type abstrait de table, il faut se donner le domaine C des clés, le domaine V des valeurs associées, une notion d'égalité sur chacun de ces domaines pour comparer les objets.
L'ensemble des tables sur ces domaines est noté TABLE, c'est un sous-ensemble des fonctions de C dans V définies sur une partie finie de C.

Tables fonctionnelles

On a les opérations :

```
table-vide                          --> TABLE
```

```
table:consulter      C × TABLE  -->  V  ou #f
```

Cette opération rend la valeur associée à une clé ou une valeur d'erreur comme `#f` si elle n'appartient pas à V.

```
table:ajouter      C × V × TABLE --> TABLE
```

Cette opération rend la table comportant en plus la liaison `c` $\rightarrow$ `v` et supprime la liaison précédente de `c` si elle existait.

```
table:enlever      C × TABLE      --> TABLE
```

Cette opération rend la table sans liaison pour la clé `c`.

Exercice 6 *Donner des relations liant ces fonctions.*

L'implantation fonctionnelle est immédiate car on sait que Scheme dispose du type prédéfini des a-listes pour représenter les associations entre clé et valeur. Selon le prédicat de comparaison choisi parmi `equal?`, `eqv?`, `eq?`, on utilise la fonction de consultation correspondante `assoc`, `assv`, `assq`. Pour fixer les idées on prendra le prédicat `equal?` (pour permettre de considérer des clés de type string).

```
(define (table:vide) '())

(define (table:consulter cle table)
  (let ((doublet (assoc cle table)))
    (if doublet
      (cdr doublet)
      #f)))

(define (table:ajouter cle val table)
  (cond ((null? table)(cons (cons cle val) '()))
        ((equal? cle (caar table))(cons (cons cle val)(cdr table)))
        (else  (cons (car table)(table:ajouter cle val (cdr table))))))

(define (table:enlever cle  table)
  (cond ((null? table) '())
        ((equal? cle (caar table))(cdr table))
        (else  (cons (car table)(table:enlever cle (cdr table))))))
```

Exercice 7 *Ecrire une fonction qui construit la table des fréquences des mots d'un fichier de texte. Utiliser la fonction du chapitre 4 §5 pour chercher les mots. Les mots du fichier seront les clés et le nombre d'apparitions d'un mot sera la valeur associée. On donnera au §10 une méthode plus efficace tirant parti de l'ordre alphabétique sur les mots.*

Tables mutables

On désigne par TABLE! l'ensemble des tables mutables avec les opérations :

```
table!:vide                        --> TABLE!
```

```
table!:consulter       C x TABLE!  -->  V  ou #f
```

Cette opération rend la valeur associée à une clé ou une valeur d'erreur comme `#f` si elle n'appartient pas à V.

```
table!:ajouter       C x V x TABLE! --> TABLE!
```

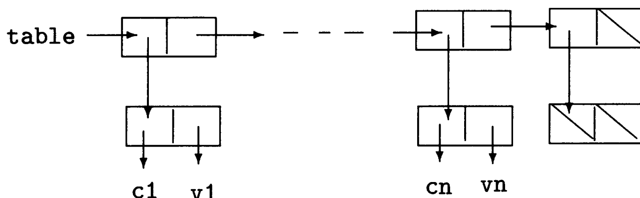
Cette opération rend la table modifiée par l'ajout de la liaison `c` $\rightarrow$ `v` et suppression de la liaison précédente de `c` s'il elle existait.

```
table!:enlever       C x TABLE!      --> TABLE!
```

Cette opération rend la table où l'on a supprimé la liaison pour la clé `c` si elle existait.

On donne une implantation par modification physique de a-listes. Comme toujours, l'impossibilité de modifier la liste vide conduit à représenter une table vide

par une a-liste non vide. On choisit pour cela une clé que l'on sait ne pas appartenir à l'ensemble des clés considérées (par exemple la liste vide). Les modifications physiques pour ajouter ou supprimer un élément en tête d'une liste non vide ont été expliquées au chapitre 4 §3.




```
(define (table!:vide) (list (cons '() '())))
```
On suppose que () n'est pas une clé. C'est le cas quand les clés sont des symboles, aussi on utilisera souvent cette manière d'initialiser une a-liste que l'on voudra pouvoir modifier physiquement.

```
(define  table!:consulter table:consulter)

(define (table!:ajouter cle valeur table)
  (cond ((null? (cdr table))(ajouterEnTete!  (cons cle valeur) table))
        ((equal? cle (caar table))(set-cdr! (car table) valeur))
        (else  (table!:ajouter cle valeur (cdr table)))))

(define (table!:enlever cle  table)
  (cond ((null? (cdr table)) table)
        ((equal? cle (caar table))(SupprimerTete! table))
        (else  (table!:enlever cle (cdr table)))))
```

Avec les fonctions auxiliaires suivantes, recopiées du chapitre 4 §3;

```
(define (AjouterEnTete! s L)
  (set-cdr! L (cons (car L)(cdr L)))
  (set-car! L s)
  L)

(define (SupprimerTete!  L)
  (set-car! L (cadr L))
  (set-cdr! L (cddr L))
  L)
```

Exercice 8 *Donner une implantation des tables mutables basée sur les prototypes. On s'inspirera de la méthode utilisée pour les piles mutables.*

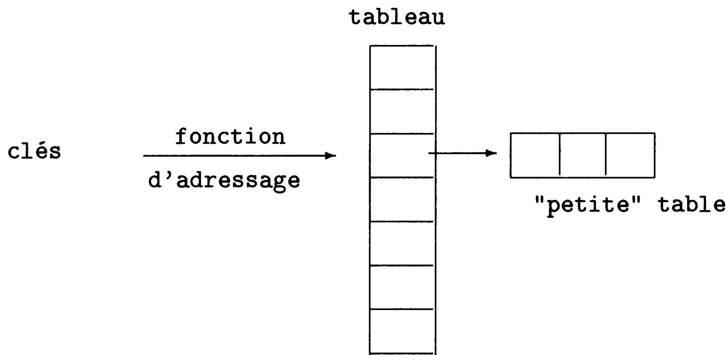
7.7 Adressage dispersé

Principe

Pour ajouter ou enlever une clé, les méthodes précédentes utilisaient un parcours de la table jusqu'à trouver la clé concernée. On dit qu'il s'agissait de tables linéaires

car le temps de consultation est, dans le pire des cas, proportionnel à la longueur de la table.
Connaissant la clé, on aimerait accéder à la valeur plus directement, comme on accède à une composante d'un vecteur en connaissant son indice. Cependant, les clés ne sont pas en général des entiers d'où l'idée d'utiliser une fonction qui transforme toute clé en un entier ; c'est le principe de l'adressage dispersé (*hash-coding* en anglais).

On se donne un vecteur de taille N. Pour associer un indice à chaque clé, on utilise une fonction dite «*d'adressage dispersé*» dont le rôle est d'associer à toute clé un indice entier dans $[0\ N[$. L'idéal est que cette fonction soit injective, mais en général ce n'est pas possible car il y a beaucoup plus de clés possibles que d'indices distincts, aussi espère-t-on que les valeurs associées aux clés se dispersent bien dans l'intervalle $[0\ N[$.
La dispersion dépend de l'ensemble des clés possibles et de la fonction d'adressage choisie. Il reste ensuite à résoudre le problème des *collisions* : comment stocker les valeurs dont les clés s'envoient sur le même indice? Une technique, appelée adressage avec chaînage, consiste à placer dans chaque case du tableau une table au sens précédent et on stocke dans ces tables partielles les ensembles (clé , valeur associée) à cette case du tableau. Bien entendu, plus N est petit plus les tables composantes risquent d'être longues, aussi le choix de N résulte d'un compromis entre occupation mémoire et vitesse de consultation.



Adressage dispersé

Voici une implantation de l'adressage dispersé avec un tableau de tables mutables. On utilise un prototype qui prend en paramètre la taille du tableau et la fonction d'adressage à utiliser. Notons que l'on doit initialiser *chaque* composante du tableau avec une *nouvelle* table mutable vide, sinon les modifications physiques seraient partagées par toutes les composantes ! On ajoute une méthode pour afficher toutes les a-listes composantes.

```
(define (make-tableAdressage:vide  taille  fctAdressage)
  (let ((tableau  (make-vector taille))
```

```
    (do  ((i 0 (+ 1 i)))
         ((= i taille))
      (vector-set!  tableau i (table!:vide)))
    (lambda (message . Largt)
      (case message
        ((consulter) (let* ((cle (car Largt))
                            (index (fctAdressage cle)))
                       (table!:consulter cle
                                         (vector-ref tableau index))))
        ((ajouter)  (let* ((cle (car Largt))
                           (index (fctAdressage cle))
                           (valeur (cadr Largt)))
                      (table!:ajouter cle valeur
                                      (vector-ref tableau index))))
        ((enlever)  (let* ((cle (car Largt))
                           (index (fctAdressage cle)))
                      (table!:enlever cle (vector-ref tableau index))))
        ((affiche) (do  ((i 0 (+ 1 i)))
                        ((= i taille))
                     (display-alln i  " " (vector-ref tableau i))))
        ))))
```

Fonctions d'adressage

Supposons que l'on souhaite construire une fonction d'adressage dispersé dans le cas où les clés sont les chaînes de caractères. Une fonction très simple consiste à associer à une chaîne la somme des codes ASCII de ses caractères modulo un entier. Si l'on veut que la valeur de la fonction d'adressage reste dans un intervalle $[0\ N[$, on prend le reste modulo N.
Considérons comme exemple le cas où les clés sont des noms d'identificateurs prédéfinis en Scheme, et prenons $N = 11$. On désire avoir un entier dans l'intervalle $[0\ 11[$.

```
(define (adressage s)
  (remainder (apply + (map char->integer (string->list s))) 11))

? (map adressage  '("let"  "if" "lambda" "set!" "define" "cond" "+"
           "begin"  "-"  "null?"  "pair?"  "number?" "odd" "sqrt" ))
(6 9 4 2 3 2 10 0 1 0 7 8 3 7)
```

On constate qu'il y quatre collisions pour ces quinze chaînes et que l'index 5 n'est pas utilisé, ce qui dénote une dispersion assez médiocre! Heureusement, il existe des fonctions d'adressage plus performantes que l'on trouvera dans les références. Créons la table d'adressage dispersé correspondant à cette liste de noms en prenant comme valeur associée le symbole correspondant puis affichons le contenu de cette table.

```
(define table-scheme (tableAdressage:vide 11  adressage))

(for-each  (lambda (str)
```

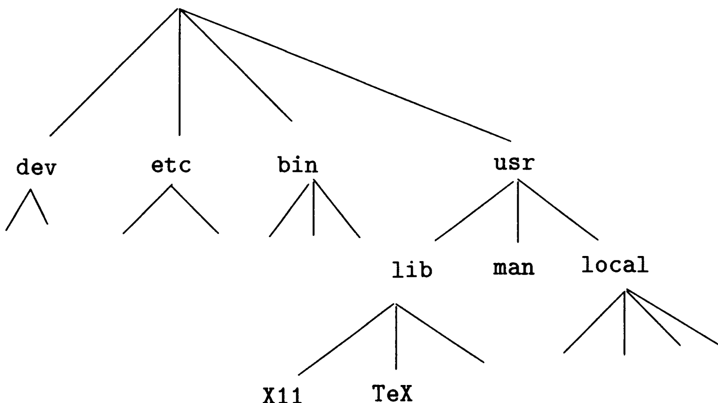
```
  (tableAdressage:ajouter! str  (string->symbol str) table-scheme))
          '("let"  "if" "lambda" "set!" "define" "cond" "+" "begin"
            "-"  "null?"  "pair?"  "number?" "odd" "sqrt"))

? (tableAdressage:affiche table-scheme)
0 (("begin" . begin) ("null?" . null?) (()))
1 (("-" . -) (()))
2 (("set!" . set!) ("cond" . cond) (()))
3 (("define" . define) ("odd" . odd) (()))
4 (("lambda" . lambda) (()))
5 ((()))
6 (("let" . let) (()))
7 (("pair?" . pair?) ("sqrt" . sqrt) (()))
8 (("number?" . number?) (()))
9 (("if" . if) (()))
10 (("+" . +) (()))
```

7.8 Structure d'arbre

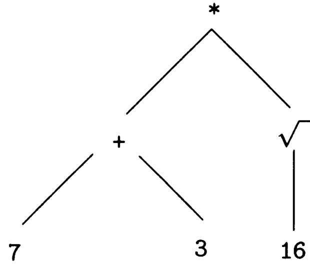
La structure d'arbre est omniprésente en informatique car on manipule couramment des objets hiérarchisés : l'arborescence des fichiers dans un système d'exploitation, la structure des expressions en mathématiques, les expressions Scheme, la structure grammaticale d'une phrase, l'arbre de recherche des coups suivants dans un jeu ...

Exemples d'arbres

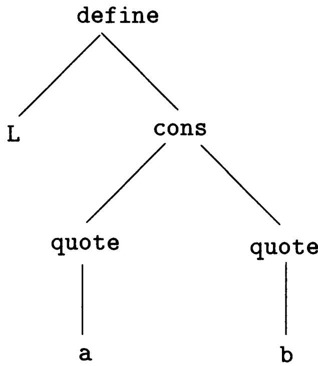
Il n'y a pas de structure unique d'arbre, mais une grande variété de structures pour modéliser la diversité des applications. Voici quelques visualisations d'arbres dans différents domaines.



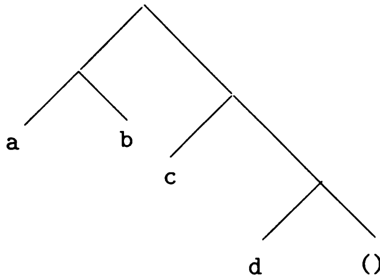
Extrait d'une hiérarchie de fichiers



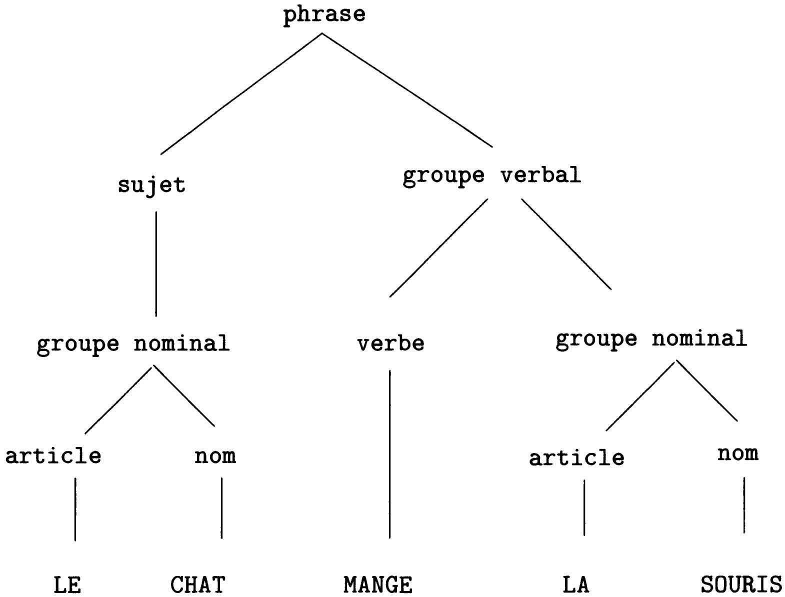
Expression $(7+3) * \sqrt{16}$



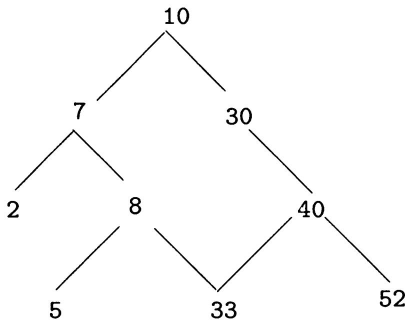
Expression Scheme (define L (cons 'a 'b))



S-expression ((a . b) c d)



Arbre d'analyse grammaticale

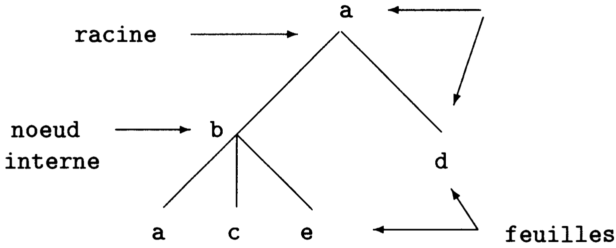


Ceci n'est pas un arbre mais un graphe à cause de 33

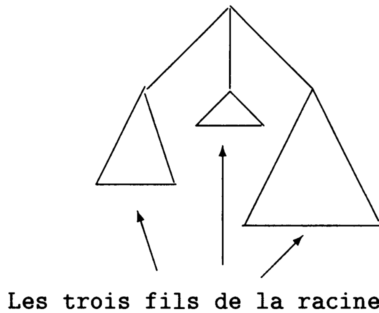
Terminologie

Il est de tradition en informatique de dessiner les arbres avec la *racine* en haut et les extrémités des *branches* (appelées *feuilles*) en bas. L'information qui est éventuellement associée à un *nœud* s'appelle une *étiquette*. Les branches qui partent d'un nœud pointent sur des sous-arbres appelés les *fils* du nœud. Les feuilles sont donc les nœuds qui n'ont pas de fils. Les nœuds qui ne sont pas des feuilles s'appellent des nœuds *internes*.

On appelle *sous-arbre* d'un arbre soit cet arbre ou un sous-arbre d'un de ses fils.



Terminologie des arbres



Les trois fils de la racine

7.9 Arbres binaires

On commence par la structure d'arbre la plus simple : les arbres binaires étiquetés.

Un arbre binaire c'est soit l'arbre vide (qui n'a pas de nœud) soit un arbre dont chaque nœud possède exactement deux fils qui sont des arbres binaires (éventuellement vides). On dit qu'il est étiqueté par des étiquettes dans un ensemble E si à chaque nœud est associé un élément de E.

Type d'abstrait arbre binaire

Notons ARBRE2 l'ensemble des arbres binaires étiquetés, et ARBRE2* ceux qui sont non vides. Un arbre binaire est donc de la forme

arbre2 =
- soit la constante `Arbre2:vide`,
- soit un arbre binaire constitué d'une étiquette `e` et deux arbres binaires appelés fils gauche et fils droit.

On a donc deux fonctions de construction, une pour construire un arbre vide, l'autre pour construire un arbre avec deux fils :

```
arbre2:vide                              --> ARBRE2
arbre2:cons    E x ARBRE2 x ARBRE2       --> ARBRE2 *
```

Les accesseurs permettent d'accéder à la racine et aux deux fils d'un arbre non vide :

```
arbre2:racine       ARBRE2*    --> E
arbre2:filsg        ARBRE2*    --> ARBRE2
arbre2:filsd        ARBRE2*    --> ARBRE2
```

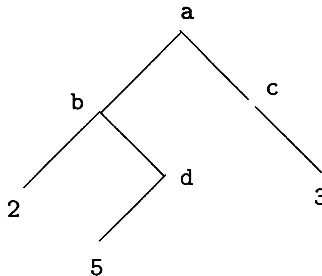
On a un prédicat pour tester si un arbre est vide :

```
arbre2:vide?        ARBRE2     --> BOOL
```

Avec les relations sémantiques suivantes :

```
(arbre2:filsg    (arbre2:cons  e a1  a2) )  = a1
(arbre2:filsd    (arbre2:cons  e a1  a2) )  = a2
(arbre2:racine   (arbre2:cons  e a1  a2) )  = e
(arbre2:vide?    (arbre2:cons  e a1  a2) )  = #f
(arbre2:vide?    (arbre2:vide) )            = #t
```

Un nœud dont les deux fils sont vides s'appelle une *feuille*. Voici un exemple d'arbre binaire, on ne visualise pas les arbres vides. Ici les feuilles sont étiquetées par des entiers et les nœuds internes par des lettres.

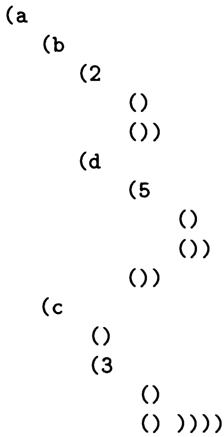


Représentations d'un arbre binaire

Il y a de multiples possibilités pour représenter en Scheme la structure d'arbre binaire. Une méthode naturelle consiste à représenter l'arbre vide par la liste vide et un arbre non vide par une liste constituée de l'étiquette de sa racine et de la représentation de ses deux fils.

```
(arbre2:vide)                -->   ()
(arbre2:cons    e  a1 a2)    -->   (e   A1   A2)
```
où `Ai` désigne la représentation de l'arbre `ai`.

Avec cette représentation, l'exemple précédent correspond à la liste suivante (on a placé les fils d'un même nœud en colonne).

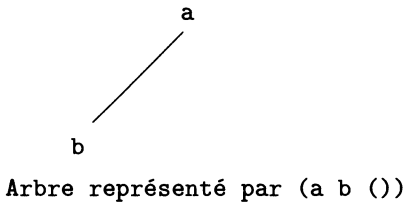


```
(a
   (b
      (2
          ()
          ())
      (d
          (5
              ()
              ())
          ())
   (c
       ()
       (3
           ()
           () ))))
```

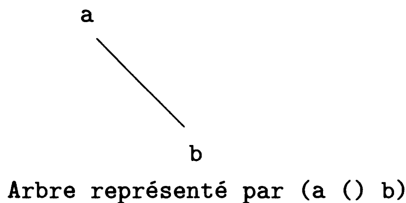
On voit que cette représentation est assez lourde à cause de la présence des arbres vides. Les arbres vides ne servent qu'à définir les feuilles d'un arbre binaire. Voici une deuxième représentation où les feuilles sont représentées par leurs étiquettes (on suppose que les étiquettes sont représentées par des atomes) et un arbre non réduit à une feuille est représenté comme précédemment :

```
arbre2:vide                  -->    ()

feuille d'étiquette  e  -->    e

(arbre2:cons   e  a1 a2)  -->    (e   A1   A2)
```

Cette représentation conserve bien la distinction entre fils droit et fils gauche :



Arbre représenté par (a b ())



Arbre représenté par (a () b)

La représentation de l'arbre précédent est plus compacte :

```
(a
   (b
      2
      (d
         5
         () ))
   (c
      ()
      3))
```

Voici une implantation des opérations sur les arbres binaires avec cette dernière représentation.

```
(define (arbre2:Vide)  '())

(define arbre2:Vide?  null?)

(define (arbre2:feuille? arbre)
  (not (pair?  arbre)))

(define (arbre2:cons  etiquette  arbre1 arbre2)
  (if (and (Arbre2:Vide? arbre1)(Arbre2:Vide? arbre2))
    etiquette
    (list etiquette  arbre1 arbre2)))

(define (arbre2:filsg  arbre)
  (cond ((Arbre2:Vide?  arbre)
          (display "erreur: pas de fils pour un arbre vide "))
         ((arbre2:feuille? arbre) (Arbre2:Vide))
         (else (cadr arbre))))

(define (arbre2:filsd  arbre)
  (cond ((Arbre2:Vide?  arbre)
          (display "erreur: pas de fils pour un arbre vide "))
         ((arbre2:feuille? arbre) (Arbre2:Vide))
         (else (caddr arbre))))

(define (arbre2:racine arbre)
  (cond ((Arbre2:Vide?  arbre)
         (display "erreur: pas de racine pour un arbre vide "))
         ((arbre2:feuille? arbre) arbre)
         (else (car arbre))))
```

Hauteur d'un arbre binaire

Il est important de remarquer que l'on peut oublier la représentation choisie pour programmer des algorithmes sur les arbres. Par exemple, écrivons une fonction `arbre2:hauteur` qui donne la hauteur d'un arbre binaire : c'est-à-dire la longueur de la plus longue branche :

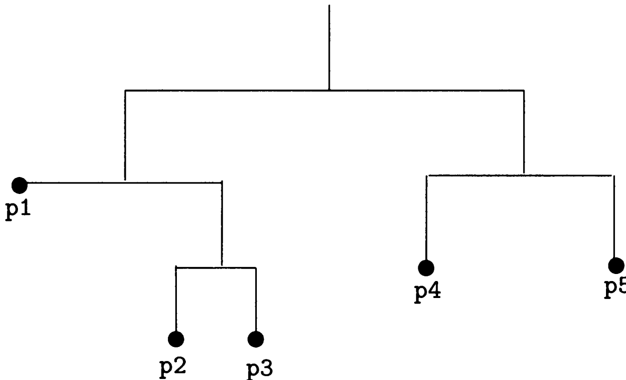
```
(define (arbre2:hauteur arbre)
```

```
  (cond ((Arbre2:vide?  arbre) 0)
        ((arbre2:feuille? arbre) 0)
        (else (+ 1 (max  (arbre2:hauteur (arbre2:filsg arbre))
                         (arbre2:hauteur (arbre2:filsd arbre)))))))

? (arbre2:hauteur  '(a (b 2 (d 5 ())) (c ()  3)))  --> 3
```

Exercice 9 *1. Ecrire une fonction qui donne le nombre de nœuds d'un arbre binaire.*

2. Ecrire un prédicat `meme-squelette?` *qui teste si deux arbres binaires sont semblables à la valeur près des étiquettes.*

3. Ecrire une fonction `poids` *qui calcule la somme des valeurs des étiquettes d'un arbre binaire à étiquettes entières. En déduire un prédicat* `Calder?` *qui teste si, en tous les nœuds, les deux fils ont le même poids.*



Arbre à la Calder

Explorations d'un arbre binaire

Un arbre sert souvent de structure de stockage aussi on peut être amené à explorer le contenu d'un arbre. On dispose de nombreuses stratégies pour visiter un arbre. Les arbres étant une structure définie récursivement, la programmation récursive des visites est immédiate. Pour chaque stratégie de visite, on rend la liste des étiquettes des nœuds visités.

Visite en pré-ordre

Principe : on visite la racine puis de la même façon le fils gauche et le fils droit :

```
(define  (arbre2:VisitePreordre arbre)
  (cond
    ((Arbre2:Vide?  arbre) '())
```

```
    ((arbre2:feuille? arbre) (list  (arbre2:racine arbre)))
    (else (cons (arbre2:racine arbre)
                (append (arbre2:VisitePreordre (arbre2:filsg arbre))
                        (arbre2:VisitePreordre (arbre2:filsd arbre)))))))
```

```
? (arbre2:VisitePreordre  '(a (b 2 (d 5 () )) (c ()  3)))
(a b 2 d 5 c 3)
```

On peut en donner une version itérative en éliminant les deux appels récursifs par l'utilisation d'une pile d'arbres. On empile les sous-arbres qui restent à visiter. Plus précisément, pour visiter un arbre on stocke dans une liste **Lnoeuds** sa racine (s'il est non vide) et on empile ses deux fils (d'abord le fils droit, pour que le gauche soit placé au sommet) dans la pile des arbres à visiter. Puis on visite de la même façon le premier arbre dépilé. On répète ce procédé tant que la pile d'arbres est non vide. A la fin la variable **Lnoeuds** contient la liste des nœuds visités du dernier au premier, aussi rend-on en résultat cette liste renversée.
C'est un exemple d'élimination d'une récursivité non terminale en utilisant une pile.

```
(define (VisitePreordre-It arbre)
  (letrec ((visite-aux
            (lambda (pile-arbres Lnoeuds)
              (if (pile:Vide? pile-arbres)
                (reverse Lnoeuds)
                (let ((arbre (pile:sommet pile-arbres)))
                  (if (arbre2:vide? arbre)
                    (visite-aux (pile:depiler pile-arbres) Lnoeuds)
                    (visite-aux (pile:empiler
                                   (arbre2:filsg arbre)
                                   (pile:empiler (arbre2:filsd arbre)
                                                 (pile:depiler pile-arbres)
                                (cons (arbre2:racine arbre) Lnoeuds))))))))
    (visite-aux (pile:empiler arbre (pile:vide)) '())))
```

```
? (VisitePreordre-It '(a (b 2 (d 5 ())) (c () 3)))
(a b 2 d 5 c 3)
```

Visite en post-ordre

Principe : on visite les deux fils puis la racine.

```
(define  (arbre2:VisitePostordre arbre)
  (cond
    ((Arbre2:Vide?  arbre) '())
    ((arbre2:feuille? arbre) (list  (arbre2:racine arbre)))
    (else (append (arbre2:VisitePostordre  (arbre2:filsg arbre))
                  (arbre2:VisitePostordre  (arbre2:filsd arbre))
                  (list (arbre2:racine arbre))))))
```

```
? (arbre2:VisitePostordre  '(a (b 2 (d 5 () )) (c ()  3)))
(2 5 d b 3 c a)
```

Visite en inordre

Principe : on visite le fils gauche, puis la racine, puis le fils droit.

```
(define  (arbre2:VisiteInordre arbre)
  (cond
    ((Arbre2:Vide?  arbre) '())
    ((arbre2:feuille? arbre) (list  (arbre2:racine arbre)))
    (else (append (arbre2:VisiteInordre  (arbre2:filsg arbre))
                  (list (arbre2:racine arbre))
                  (arbre2:VisiteInordre  (arbre2:filsd arbre))  ))))

? (arbre2:VisiteInordre  '(a (b 2 (d 5 () )) (c ()  3)))
(2 b 5 d a c 3)
```

Visite en largeur

Principe : on visite par niveau de hauteur. On commence par la racine, puis les racines de tous les fils, puis les racines des petits fils, ...
La considération des racines des fils conduit à passer par l'intermédiaire d'une fonction qui visite une liste d'arbres (on dit une *forêt*) : on visite les racines de chaque arbre puis on fait de même pour la forêt des fils.

```
(define  (arbre2:VisiteLargeur arbre)
  (letrec ((ListeFils (lambda (arbre)
                        (if (arbre2:Vide? arbre)
                          '()
                          (list (arbre2:filsg arbre)
                                (arbre2:filsd arbre)))))
           (VisiteForet
             (lambda (Larbres)
               (if (null? Larbres)
                 '()
                 (append
                   (append-map (lambda (arbre)
                                 (if (arbre2:Vide? arbre)
                                   '()
                                   (list (arbre2:racine arbre))))
                               Larbres)
                   (VisiteForet
                     (append-map ListeFils Larbres)))))))
    (VisiteForet (list arbre))))

? (arbre2:VisiteLargeur  '(a (b 2 (d 5 () )) (c ()  3)))
(a b c 2 d 3 5)
```

Exercice 10 *Eliminer la récursivité dans la visite en largeur en utilisant une file d'arbres.*

Affichage d'un arbre binaire

On peut profiter d'une visite en largeur pour effectuer l'affichage d'un arbre binaire. En l'absence de primitives graphiques, il ne s'agit pas de réaliser un dessin complet avec les arêtes (voir chapitre 14 §3), mais d'une visualisation des étiquettes par niveau. On affiche sur une même ligne les étiquettes d'un même niveau.

```
(define  (arbre2:affiche arbre)
  (letrec ((ListeFils (lambda (arbre)
                        (if (arbre2:Vide? arbre)
                            '()
                            (list (arbre2:filsg arbre)
                                  (arbre2:filsd arbre)))))
           (AfficheForet
             (lambda (Larbres)
               (if (null? Larbres)
                 (display "")
                 (begin
                   (for-each
                     (lambda (arbre)
                       (if (not (arbre2:Vide? arbre))
                         (display (arbre2:racine arbre))))
                       Larbres)
                   (newline)
                   (AfficheForet
                   (append-map ListeFils Larbres )))))))
    (AfficheForet (list arbre))))

? (arbre2:affiche  '(+ (- a (carre b ())) (* c  (/ e (+ d 1)))))
 +
 - *
 a carre c /
 b e +
 d 1
```

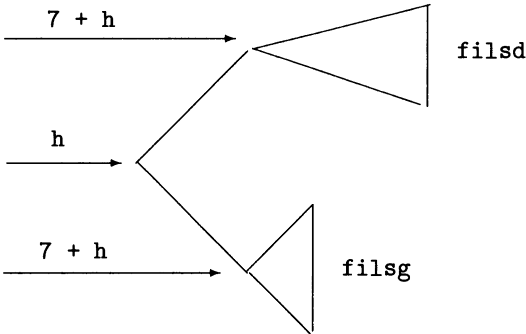
Mais cet affichage n'est pas satisfaisant. En effet, il ne permet pas de reconstruire l'arbre binaire car on ignore de quels nœuds dépendent les étiquettes de la ligne suivante. Pour pallier cet inconvénient, on adapte une solution élégante empruntée à [Wir76] qui affiche un arbre binaire après rotation de $-90°$.

Principe :

- on affiche le fils droit après rotation de -90 et avec une indentation de $h + 7$,

- on affiche la racine à la ligne suivante avec une indentation de h de la marge,

- on affiche le fils gauche après rotation de -90 et avec une indentation de $h + 7$.



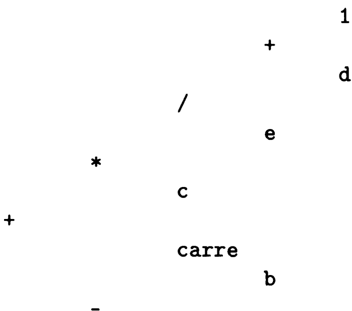
Affichage d'un arbre tourné de 90 degrés

```
(define (Arbre2:affiche90  arbre h)
  (if (Arbre2:Vide? arbre)
    (display "")
    (begin
      (Arbre2:affiche90 (arbre2:filsd arbre) (+ h 7))
      ;; dessin du fils droit après rotation de 90 et indentation
      (indente h)
      (display-alln (arbre2:racine arbre))
      ;; affichage de la racine avec une indentation
      (Arbre2:affiche90 (arbre2:filsg arbre) (+ h 7)))
      ;; dessin du fils gauche après rotation de 90 et indentation
    ))
```

Pour indenter de h, on affiche une chaîne de h espaces :

```
(define (indente h)
  (display (make-string   h)))
```

```
? (Arbre2:affiche90  '(+ (- a (carre b ()))
                      (* c
                        (/ e (+ d 1)))) 0)
```



Arbres binaires ordonnés

Quand les étiquettes sont dans un ensemble *totalement ordonné*, on peut définir la sous classe des arbres binaires *ordonnés*. On exige qu'en tout nœud l'étiquette soit supérieure (au sens large) à l'étiquette de son fils gauche (s'il est non vide) et inférieure (strictement) à l'étiquette de son fils droit (s'il est non vide).

Ce type d'arbre se prête bien au stockage d'objets qu'il faudra ensuite pouvoir retouver rapidement.
Pour savoir si une étiquette est présente dans un arbre ordonné, on compare cette étiquette avec la racine. Si elle est différente, on doit continuer la recherche dans le fils gauche ou le fils droit selon la relation d'inégalité satisfaite. Dans le cas où l'arbre n'est pas trop dégénéré (c'est-à-dire possède une répartition assez équilibrée des fils à gauche et à droite), la recherche est beaucoup plus rapide qu'une simple visite. En revanche, si l'arbre est complètement dégénéré (par exemple s'il ne comporte que des fils droits), on est ramené à une recherche linéaire.

```
(define (trouve?  etiquette arbre)
  (if (Arbre2:Vide? arbre)
    #f
    (let ((racine (arbre2:racine arbre)))
      (cond
        ((= etiquette  racine) #t)
        ((< etiquette  racine) (trouve? etiquette (arbre2:filsg arbre)))
        (else (trouve?  etiquette (arbre2:filsd arbre)))))))

? (trouve?  7  '(10 (5 2 7) 14)) -> #t
? (trouve?  12  '(10 (5 2 7) 14)) -> #f
```

Un arbre binaire ordonné vérifie en tout nœud n la relation

$$\text{(racine (filsg n))} \leq \text{(racine n)} < \text{(racine (filsd n))}.$$

On en déduit que toutes les étiquettes du fils gauche sont $\leq$ à la racine et que toutes les étiquettes du fils droit sont $>$ à la racine. Par conséquent, une visite en pré-ordre retourne la liste des étiquettes *triées* par ordre croissant. D'où une méthode pour le tri d'une liste L. On transforme la liste en un arbre binaire ordonné et on effectue une visite en inordre de cet arbre.
Pour construire l'arbre associé à la liste, on part d'un arbre vide puis pour chaque élément de la liste, on ajoute un nouveau nœud étiqueté par cet élément.

```
(define (ajouterNoeud etiquette arbre)
  (if (Arbre2:Vide? arbre)
    (arbre2:cons  etiquette  (Arbre2:Vide)  (Arbre2:Vide))
    (let ((racine (arbre2:racine arbre)))
      (if (<= etiquette  racine)
        (arbre2:cons racine
                    (ajouterNoeud etiquette (arbre2:filsg arbre))
                    (arbre2:filsd arbre))
        (arbre2:cons racine
                    (arbre2:filsg arbre)
                    (ajouterNoeud etiquette (arbre2:filsd arbre)))))))
```

En itérant ces ajouts, la fonction `liste->arbre2` transforme la liste en un arbre :

```
(define (liste->arbre2 L arbre)
  (if (null? L)
    Arbre
    (liste->arbre2  (cdr L)
                    (ajouterNœud (car L) arbre))))
```

D'où la fonction de tri par arbre binaire :

```
(define (trier-par-arbre2 L)
  (arbre2:VisiteInordre  (liste->arbre2 L (Arbre2:Vide))))

? (trier-par-arbre2 '(7 5 2 1 3 9 0 4)) -> (0 1 2 3 4 5 7 9)
```

Exercice 11 *Cette méthode pour ajouter un nœud est gourmande en place mémoire car on reconstruit l'arbre augmenté. On pourrait envisager de faire une mutation pour modifier physiquement l'arbre initial. Ecrire une fonction d'adjonction physique. Il est conseillé d'utiliser la première représentation des arbres binaires, car la représentation actuelle d'une feuille par un atome n'est pas mutable.*

Remarque 3 Cette technique de représentation par arbre binaire ordonné peut être aussi conseillée pour implanter la structure de table mutable quand les clés sont dans un ensemble ordonné. On stocke dans l'étiquette de chaque nœud la clé et la valeur associée.

7.10 Structure générale d'arbre

En général, le nombre de fils peut varier d'un nœud à l'autre, c'est la structure générale d'arbre.

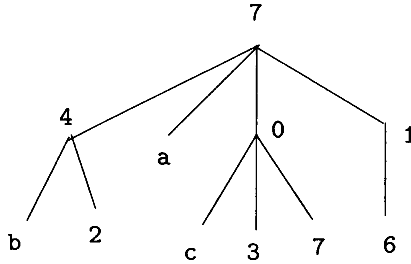
Définitions

La structure d'arbre avec des nœuds étiquetés par des étiquettes dans un ensemble E est définie récursivement par :

Arbre étiqueté = une étiquette e et une liste (éventuellement vide) $T_1 \dots T_n$ d'arbres étiquetés.

Remarque 4 Notons que l'on n'introduit pas la notion d'arbre vide, aussi ne peut-on pas considérer la structure d'arbre binaire comme un cas particulier de la structure d'arbre.

On note ARBRE l'ensemble des arbres. Pour un arbre T, les arbres $T_1 \dots T_n$ sont appelés les *fils* de T, l'étiquette e s'appelle la *racine* de T. Le nombre de fils en un nœud s'appelle le *degré* de ce nœud. Un nœud de degré 0 s'appelle une *feuille*.



Type abstrait arbre

Voici la signature du constructeur, des accesseurs et d'un prédicat :

```
arbre:cons       E  x  Liste d'ARBRES    -->   ARBRE
arbre:Racine                      ARBRE  -->  E
arbre:Fils               N   x    ARBRE  -->   ARBRES
```
(l'argument entier correspond au numéro (≥ 1) du fils)
```
arbre:Lfils     :              ARBRE    -->  Liste d'ARBRES
```
(la liste des fils d'un arbre)
```
arbre:Feuille?   :             ARBRE    -->  BOOL
```

Avec les relations sémantiques :

```
(arbre:Feuille? (arbre:cons e  ())        = #t
(arbre:Racine   (arbre:cons e  Larbres)) = e
(arbre:Lfils    (arbre:cons e  Larbres)) = Larbres
(arbre:Fils  k  (arbre:cons e  Larbres)) = Le k ième  élément
                                           de la liste Larbres
```

Implantation de la structure d'arbre

Comme pour les arbres binaires, il y a de nombreuses façons de représenter un arbre, en voici une très simple. Si l'arbre est réduit à une feuille, on le représente par l'étiquette de cette feuille (supposée ne pas être une liste) et sinon on le représente par la liste constituée de sa racine et de ses fils.

```
(define  (Arbre:cons   e   Larbres)
  (if (null? Larbres)
    e
    (cons  e  Larbres)))

(define (arbre:feuille?  arbre)
  (not (pair?  arbre)))

(define (arbre:Lfils  arbre)
  (if (arbre:feuille?  arbre)
    '()
     (cdr arbre)))
```

```
(define (arbre:racine arbre)
  (if (arbre:feuille?  arbre)
    arbre
    (car arbre)))

(define (arbre:fils k arbre)
  (if (arbre:feuille? arbre)
    (display "erreur: une feuille n'a pas de fils ")
    (list-ref  arbre  k)))
```

L'arbre de la figure précédente est représenté par la liste

```
 (7 (4 b 2)
    a
    (0 c 3 7)
    (1 6))
```

Hauteur d'un arbre

La hauteur d'un arbre s'obtient en ajoutant 1 au maximum des hauteurs de ses fils :

```
(define  (arbre:hauteur arbre) ;
  (if (arbre:feuille? arbre)
    0
    (+ 1 (apply max (map  arbre:hauteur (arbre:Lfils arbre))))))
```

Exploration d'un arbre

Les stratégies de visites définies pour les arbres binaires se généralisent immédiatement au cas des arbres. Voici la visite en pré-ordre, on laisse en exercice le soin d'étudier les autres.

```
(define  (arbre:VisitePreordre arbre)
  (let ((racine (arbre:racine arbre)))
    (if (arbre:feuille? arbre)
      (list arbre)
      (cons racine
           (append-map arbre:VisitePreordre (arbre:Lfils arbre))))))

? (arbre:VisitePreordre ' (7 (4 b 2) a (0 c 3 7) (1 6)) )
 (7  4 b 2 a 0 c 3 7 1 6)
```

7.11 Méthodes de recherche et de tris

On a vu au §3 que la consultation d'une table conduit à une exploration linéaire de la suite des valeurs des clés, d'où une complexité en temps de l'ordre du nombre des clés. C'est inacceptable pour de grandes tables : on ne consulte pas un dictionnaire en le parcourant du début à la fin ! Dans la plupart des applications

on dispose d'une notion d'ordre naturel sur les clés (ordre sur les nombres, ordre lexicographique sur les chaînes, ...), ce qui permet de trier les clés.

Recherche par dichotomie

Quand les clés sont triées, on peut pratiquer une recherche par dichotomie : on compare la clé cherchée avec celle du milieu de la table puis, selon la réponse, on cherche dans la moitié inférieure ou supérieure de la table. En itérant ce procédé, on obtient un encadrement de plus en plus fin de l'éventuelle solution.
On a déjà utilisé un principe analogue pour chercher une étiquette dans un arbre binaire ordonné. La recherche par dichotomie nécessite de pouvoir accéder directement au milieu de la table, aussi il est plus naturel de représenter la suite des clés non par une liste mais par un tableau.
On passe en paramètre le prédicat de comparaison, de façon à pouvoir particulariser cette recherche pour divers types de clés, et on retourne une fonction qui teste si une clé est dans un vecteur. La forme curryfiée est naturelle ici.

```
(define trouve?
  (lambda (p?)
    (lambda (cle vecteur)
      (letrec ((recherche-aux
                (lambda (inf sup)
                  (if (< sup inf)
                    #f
                    (let* ((milieu (quotient  (+ inf sup) 2))
                           (valeur (vector-ref vecteur milieu)))
                      (cond ((p? cle milieu)
                             (recherche-aux inf  (- milieu 1)))
                            ((p? milieu cle)
                             (recherche-aux (+ milieu 1) sup))
                            (else #t)))))))
        (recherche-aux 0 (1- (vector-length vecteur)))))))
```

On spécialise la recherche avec le prédicat $<$ sur les nombres :

```
(define recherche-nombre (trouve? <))

? (recherche-nombre  4 (vector 1 5 8 12 14 15 )) -> #t
? (recherche-nombre  6 (vector 1 5 8 12 14 15 )) -> #f
```

Exercice 12 *Donner une implantation de cette méthode dans le cas où l'on cherche dans une liste.*

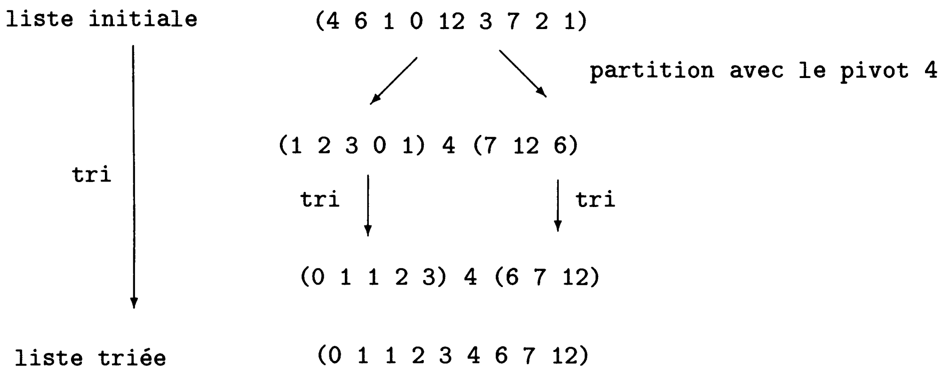
Pour pouvoir utiliser ce type de recherche, on est conduit à trier les clés. Il existe une très grande variété de méthodes de tri. Le choix d'une méthode dépend en grande partie des propriétés statistiques des suites à trier. Une méthode peut être satisfaisante pour des suites d'une vingtaine d'éléments et inadéquate pour des suites d'un millier d'éléments et vice et versa. Certaines méthodes — dites stables — conservent l'ordre des éléments déjà dans le bon ordre, d'autres pas. Certaines méthodes nécessitent des opérations non disponibles sur la suite

des éléments à trier (exemple tri de fichiers, ...). On a déjà vu une méthode de tri à l'occasion des arbres binaires ordonnés, on va en décrire deux autres basées sur le slogan «diviser pour accélérer» : le tri rapide et le tri fusion.

Tri rapide d'une liste

Principe : on choisit un élément de la liste appelé pivot (par exemple le premier élément) et l'on partitionne le reste de la liste en deux sous-listes :
- la liste des éléments inférieurs au pivot (notée `Linf`),
- la liste des éléments supérieurs ou égaux au pivot (notée `Lsup`).

Puis on concatène ces deux listes, après les avoir triées par cette méthode, en intercalant le pivot entre elles.
On peut schématiser le fonctionnement de cet algorithme sur un exemple :



La fonction de tri rapide est connue en anglais sous le nom de *quicksort.*

```
(define (quicksort L)
  (if (or (null? L)(null? (cdr L)))
    L
    (let* ((pivot (car L))
           (l1.L2 (partition (cdr L) pivot '() '() ))
           (L1 (car l1.L2))
           (L2 (cdr l1.L2)))
      (append (quicksort L1)(cons pivot (quicksort L2))))))

? (quickSort '(4 6 1 0 12 3 7 2 1)) -> (0 1 1 2 3 4 6 7 12)
```

La fonction de partition accumule dans des listes `Linf` et `Lsup` les éléments de `L` inférieurs (resp. supérieurs) au pivot et retourne à la fin le doublet constitué des listes `Linf` et`Lsup`.

```
(define (partition L pivot Linf Lsup)
  (cond
    ((null? L) (cons Linf Lsup))
```

```
    ((< (car L) pivot)
     (partition (cdr L) pivot (cons (car L) Linf) Lsup))
    (else (partition (cdr L) pivot Linf (cons (car L) Lsup) ))))

? (partition '(6 1 0 12 3 7 2 1)  4  '() '())
((1 2 3 0 1) 7 12 6)
```

On démontre que ce tri a une complexité en moyenne d'ordre $n \log n$, si n est le nombre d'objets à trier.

Exercice 13 *Démontrer la terminaison et la correction des procédures récursives* `partition` *et* `quicksort`*.*

Exercice 14 *Ecrire une fonction de tri par insertion (ou du joueur de cartes). Quand un joueur ramasse sa donne, il prend la première carte, puis insère la seconde à sa place, puis la troisième, ... On construit progressivement une nouvelle liste en insérant chaque élément de la liste initiale à sa place dans la nouvelle liste. On démontre que ce tri a une complexité en n^2 ce qui le limite aux petites listes.*

Tri rapide d'un vecteur

Un vecteur étant une structure mutable, on procède par modifications successives du vecteur pour le transformer en un vecteur trié. On note $V[p \dots q]$ le sous vecteur de V constitué des éléments d'indices de p à q. Pour trier le vecteur $V[p \dots q]$ on choisit un élément pivot (par exemple le premier élément) et on réarrange les éléments de ce vecteur pour que le sous-vecteur $V[p \dots j]$ contienne des éléments inférieurs ou égaux au pivot et $V[j+1 \dots q]$ des éléments supérieurs ou égaux au pivot et l'on applique le même procédé aux deux sous vecteurs ...
Le point délicat est la procédure de partition d'un vecteur $V[p \dots q]$. Elle procède par une suite de transpositions des composantes du vecteur. On prend comme pivot le 1er élément du tableau. On calcule l'indice i_1 du premier élément (en partant du bas du vecteur) supérieur ou égal au pivot. On calcule l'indice j_1 du premier élément (en partant du haut du vecteur) inférieur ou égal au pivot puis on échange ces deux éléments et l'on recommence en partant de $i_1 + 1$ et $j_1 - 1$. On arrête quand i_1 dépasse j_1 et l'on retourne l'indice j_1. A la fin, le sous-vecteur $V[p \dots j_1]$ est constitué d'éléments $\leq$ pivot et $V[j_1{+}1 \dots q]$ est constitué d'éléments $\geq$ pivot.

```
(define (vector-partition! V p q)
  (let ((pivot (vector-ref V p)))
    (letrec ((partitionner!
               (lambda (i j)
                 (let ((i1 (do ((m i (+ m 1)))
                               ((<= pivot (vector-ref V m)) m)))
                       (j1 (do ((m j (- m 1)))
                               ((<= (vector-ref V m) pivot) m) )))
                   (cond ((< i1 j1)
                          (echanger! V i1 j1)
                          (partitionner! (+ 1 i1) (- j1 1)))
```

```
                    (else j1))))))
      (partitionner! p q) )))
```

On teste avec le vecteur suivant :

```
? (define V (vector  2 1 5 2 4))
? (vector-partition!  V 0 4) -> 1
```

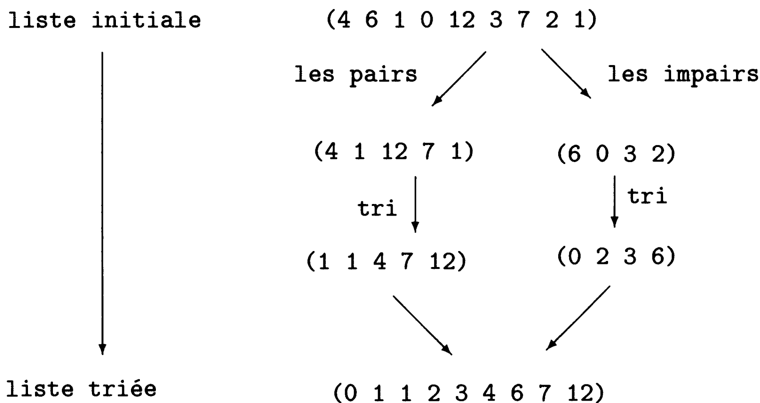
D'où la partitions de `V` en `#(2 1)` et `#(5 2 4)` pour le pivot 2.

La procédure de tri rapide pour un vecteur `V` utilise une fonction récursive auxiliaire qui effectue le tri rapide du sous-vecteur $V[g \dots d]$; ce vecteur est non vide si l'indice du bas g est inférieur à l'indice du haut d:

```
(define (vector-quickSort! V)
  (letrec ((subvector-quicksort!
            (lambda (g d)
              (if (< g d)
                (let ((j (vector-partition! V g d)))
                  (subvector-quicksort! g j)
                  (subvector-quicksort! (+ j 1) d)))
                V)))
   (subvector-quicksort! 0 (1- (vector-length V)))))
```

Tri fusion d'une liste

Le principe est proche du précédent. On partitionne la liste en deux sous-listes de longueur égales (à une unité près), on trie les deux sous-listes par le même procédé et on les fusionne.
Pour partager la liste en deux sousd- listes, on peut considérer les éléments d'indice pair et ceux d'indice impair :



Tri fusion d'une liste

```
(define (LesElementsPairs L)
  (if (or (null? L) (null? (cdr L)))
    L
    (cons (car L) (LesElementsPairs (cddr L)))))

(define (LesElementsImpairs L)
  (LesElementsPairs (cdr L)))
```

La fusion de deux listes croissantes en une liste croissante est donnée par la fonction :

```
(define (fusion L1 L2)
  (cond ((null? L1) L2)
        ((null? L2) L1)
        ((< (car L1)(car L2)) (cons (car L1) (fusion (cdr L1) L2)))
        (else (cons (car L2) (fusion L1 (cdr L2))))))
```

D'où la fonction de tri par fusion :

```
(define  (triFusion L)
  (if (or (null? L) (null? (cdr L)))
    L
    (let ((L1 (triFusion (LesElementsPairs   L)))
          (L2 (triFusion (LesElementsImpairs L))))
      (fusion L1 L2))))
```

Exercice 15 *Etendre aux vecteurs la méthode du tri fusion. Pour partager un vecteur en deux parties, il sera plus efficace de considérer les éléments d'indice inférieur à la demi-longueur et ceux d'indice supérieur.*

7.12 Notions sur les graphes

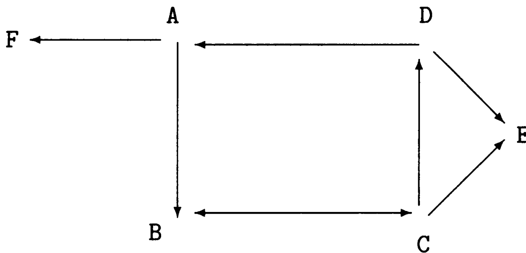
La grande diversité des situations que l'on peut modéliser par un graphe entraîne une énorme richesse en algorithmes. Mais ces algorithmes ne servent souvent que pour des domaines très précis et en fait il n,y a pas beaucoup d'algorithmes d'utilisation courante pour les besoins propres de l'informatique. On ne présentera donc que quelques algorithmes très simples. De plus, contrairement aux arbres, un graphe n'est pas une structure récursive, aussi la programmation y est moins élégante.

Définitions et terminologie

Les graphes servent à visualiser des relations d'interdépendance entre des objets. On distingue des graphes *ordonnés* et des graphes *non ordonnés* selon que la relation à visualiser est orientée ou non. Notons que la terminologie concernant les graphes n'est pas tout à fait normalisée, mais les différences entre les auteurs sont mineures.

Un graphe G est constitué par la donnée d'un ensemble V de points appelés *sommets* et d'une relation binaire R entre ces points. Quand la relation est symétrique on dit que le graphe est non orienté.

Sur la figure suivante, l'ensemble V des sommets est égal à `{A B C D E}` et la relation binaire est constituée par la donnée de l'ensemble des couples :
R = `{(A B) (A F) (B C ) (C B) (C D) (C E) (D A) (D E)}`.



Graphe 1

Un couple, comme `(A B)`, de R s'appelle une *arête* du graphe. La flèche est un moyen graphique pour distinguer l'arête `(A B)` de l'arête `(B A)`.

On appelle *successeur* d'un sommet `S`, un sommet `S'` tel que `(S S')` soit une arête du graphe. Ainsi les successeurs de `C` dans le graphe 1 sont `D` et `E`.

On appelle *chemin* dans un graphe, une liste de sommets $s_1 \ldots s_N$ tels que chaque sommet $s_j + 1$ est un successeur de s_j et on dit que c'est un *cycle* si de plus $s_1 = s_N$.

Un graphe orienté est dit *connexe* si pour tout couple de sommets distincts s_1 , s_2 il existe un chemin allant de s_1 à s_2 et un chemin de s_2 à s_1.

Les graphes peuvent aussi servir à modéliser les jeux à un joueur : les sommets représentent l'ensemble des positions possibles et les successeurs d'un sommet sont les positions atteignables en un coup de ce sommet. Le but du jeu se ramène à trouver un chemin joignant la position initiale à un sommet appartenant à un ensemble de positions dites gagnantes.

Exercice 16 *On considère les cases d'un échiquier comme les sommets d'un graphe et deux sommets sont reliés par un arc si l'on peut aller de l'un à l'autre en un seul coup du cavalier. Ce graphe est-il connexe ?*

Implantation et exploration d'un graphe

Les algorithmes sur les graphes font souvent appel à la liste des successeurs d'un sommet ce qui conduit à représenter un graphe orienté par une a-liste[2] :
`(...(sommet ses-successeurs) ...)` .
On choisit de faire figurer dans la a-liste même les nœuds qui n'ont pas de successeurs. Par exemple, on définit le graphe1 par :

[2]Un graphe *non orienté* sera représenté de la même façon : on donne, pour chaque sommet, la liste de ses successeurs (même si c'est un peu redondant.)

```
(define  Graphe1
  '((A B F) (B C) (C B D E) (D A E) (E) (F)))
```

Pour accéder aux successeurs d'un nœud et à la liste des nœuds, on définit les fonctions :

```
(define (ListeSuccesseurs sommet Graphe)
  (cdr (assoc sommet Graphe)))

(define (ListeSommets Graphe)
  (map car Graphe))

? (ListeSuccesseurs 'C  Graphe1) -> (b d e)
? (ListeSommets  Graphe1)        -> (a b c d e f)
```

Exploration d'un graphe

Comme pour les arbres, les algorithmes sur les graphes sont souvent dérivés des algorithmes de visite. On part d'un sommet s_0 et on essaye de visiter le graphe en allant toujours à un sommet adjacent. Selon que l'on gère la liste des successeurs comme une pile ou une file, on obtient une visite en profondeur ou en largeur. Mais, contrairement aux arbres, cette méthode peut boucler à cause de la présence de cycles. Pour ne pas revisiter un sommet, on utilise une liste des sommets déjà visités.

La fonction de visite d'un graphe à partir d'un nœud utilise une fonction auxiliaire `visiteAux`. On se donne une liste de sommets en attente d'être visités, et on commence la visite par le premier de cette liste, puis à partir de ses successeurs ... puis on part du sommet suivant de la liste non encore visité ... C'est l'analogue pour les graphes de l'exploration en pré-ordre des arbres.

```
(define (visite G s0)
  (letrec ((visiteAux
           (lambda (en-attente DejaVisite)
           ;; visite de G en évitant les noeuds déjà visités
           ;; et en partant des nœuds de la liste  en-attente
             (if (null? en-attente)
               DejaVisite
               (let* ((nouveauSommet (car  en-attente))
                      (resteAvisiter (cdr en-attente))
                      (SesSucc (ListeSuccesseurs nouveauSommet
                                                 G)))
                 (if (member nouveauSommet  DejaVisite)
                   (visiteAux  resteAvisiter  DejaVisite)
                   (let ((NouveauVisite
                          (visiteAux SesSucc
                                     (cons nouveauSommet
                                           DejaVisite))))
                     (visiteAux resteAvisiter
                                NouveauVisite))))))))
    (reverse (visiteAux  (list s0) '()))))
```

```
? (visite Graphe1 'a) -> (a b c d e f)
```

Exercice 17 *Ecrire une version itérative de cette fonction en stockant les prochains sommets à visiter dans une pile.*

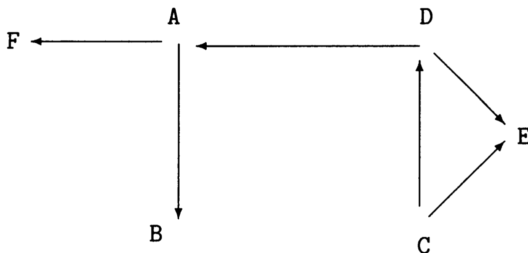
En partant d'un sommet s_0, on peut visiter tous les sommets que l'on peut atteindre à partir de s_0 ; cet ensemble de sommets s'appelle la *composante connexe* de s_0.

```
(define  Graphe2
   '((A B F) (B A C) (C B D E) (D A C E) (E D C) (F A)))

 ? (visite Graphe2 'A ) -> (a b c d e f)
```

Exercice 18 *On dispose de trois bidons: un bidon plein d'eau de 4L et deux bidons vides: un de 3L et un de 1L. Comment diviser les quatre litres en 2 fois 2 litres par transvasement de bidons. On représentera l'ensemble des configurations possibles par un graphe, chaque sommet étant un triplet des trois nombres donnant les volumes d'eau dans chaque bidon. Il s'agit de trouver un chemin entre deux configurations données.*

Graphes acycliques et tri topologique

Un cas particulier important de graphes orientés est constitué par ceux qui n'ont pas de cycle. On les appelle des graphes acycliques (en anglais DAG, pour Directed Acyclic Graph). Par exemple, `A B C D A` est un cycle du `Graphe1`. Le graphe suivant est un DAG :



Graphe 3

Dans un DAG, la relation binaire du graphe définit un ordre (partiel) sur les sommets. On définit la relation $A \succeq B$ si $A = B$ ou, s'il existe un chemin d'origine le sommet `A` se terminant en `B`. Cette relation est évidemment transitive et l'absence de cycle entraîne qu'elle est irréflexive :

$$A \succeq B \quad \text{et} \quad B \succeq A \quad \Rightarrow \quad A = B$$

C'est donc une relation d'ordre partiel.

L'intérêt d'un DAG est que l'on peut trouver un ordre *total* $<$ compatible avec cet ordre partiel. Autrement dit, on peut exhiber une liste $(S_1 \dots S_N)$ croissante pour $<$ de tous les sommets du DAG, de sorte que si l'on a $S_j \succeq S_i$, alors on a $j \geq i$.

Trouver un tel ordre total s'appelle faire un *tri topologique*. Par exemple, si un ensemble de tâches à effectuer doivent vérifier des relations de précédence (pour compiler tel et tel fichier il faut avoir déjà compilé tel et tel fichier ...), on désire trouver une manière pour effectuer toutes les tâches tout en satisfaisant les relations. Voici une méthode simple pour faire un tel tri :

- on prend pour S_1 un sommet minimal pour l'ordre $\succeq$, c'est-à-dire un sommet sans successeur (remarquer que l'existence d'un tel sommet découle facilement de l'absence de cycle),
- on supprime ce sommet du graphe et les arêtes qui y arrivent, puis on recommence en choisissant un sommet minimal dans le graphe restant ... jusqu'à épuisement des sommets ou échec si le graphe n'est pas un DAG.

Programmons cette méthode. La fonction `sommet-min` retourne un sommet sans successeur ou `#f` s'il y a un cycle dans le graphe `G` :

```
(define (sommet-min G)
  (if (null? G)
    #f
    (let ((sommet (caar G)))
      (if (pair? (ListeSuccesseurs sommet G))
        (sommet-min (cdr G))
        sommet))))
```

On teste avec le graphe de la figure ci-dessus.

```
(define Graphe3
  '((a b f) (b) (c d e) (d a e) (e) (f)))

? (sommet-min Graphe3 ) -> b
```

Quand on a supprimé un sommet d'un graphe, on construit un graphe où l'on a également supprimé les arêtes adjacentes à ce sommet. Pour supprimer un élément d'une liste, on utilise la fonction `remove` introduite au chapitre 2 §3.

```
(define (supprimerSommet s G)
  (if (null? G)
    '()
    (let ((sommet (caar G))
          (ses-successeurs (cdar G)))
      (cond ((member s ses-successeurs)
             (cons (cons sommet (remove  s ses-successeurs))
                   (supprimerSommet s (cdr G))))
            ((equal? s sommet)(supprimerSommet s (cdr G)))
            (else (cons (car G)(supprimerSommet s (cdr G)))) ))))

? (supprimerSommet 'e Graphe3) -> ((a b f) (c d) (d a) (f))
```

Enfin, la fonction qui réalise le tri topologique d'un graphe retourne une liste triée des nœuds ou un message d'échec s'il y a un cycle :

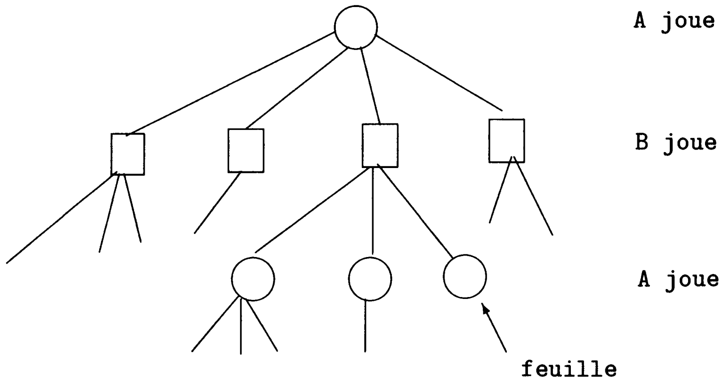
```
(define (triTopologique G)
  (if (null? G)
    '()
    (let* ((s0 (sommet-min G))
           (G-s0 (supprimerSommet s0 G)))
      (if  s0
        (let ((Lsommets (triTopologique G-s0)))
          (if Lsommets
            (cons s0 Lsommets)
            (display-alln "il y a un cycle ")))
        (display-alln "il y a un cycle ")))))

? (triTopologique Graphe3) -> (e d c f a)
? (triTopologique Graphe2) -> il y a un cycle
```

7.13 Compléments : jeux à deux joueurs et coupures alpha-bêta

Arbres de jeux

On considère un jeu où *deux* joueurs jouent à tour de rôle. On représente l'ensemble de toutes les parties possibles à l'aide d'un arbre, appelé `arbre de jeu`.

La racine représente l'état initial du jeu. Appelons `A` le joueur qui joue en premier. On définit les fils de la racine comme étant toutes les positions possibles après que `A` ait joué. On définit les fils de chacune de ces positions comme étant l'ensemble des positions possibles après que `B` ait joué ... On construit ainsi un arbre, ses feuilles sont les positions qui permettent de décider du résultat du jeu. On suppose que le jeu ne permet pas la nullité d'une partie, aussi une feuille est une position gagnante ou une position perdante pour le joueur qui se trouve dans cette position. On suppose que toute partie se termine, alors une partie correspond à une branche allant de la racine à une feuille.



Arbre de jeu

Fonction d'évaluation

Chaque joueur essaye de jouer de façon à définir une branche qui conduise à une feuille gagnante pour lui. On affecte à chaque position dans l'arbre une valeur 1 ou -1 définie par :

- +1 si le joueur qui joue à partir de cette position peut gagner quelle que soit la manière dont se défendra l'adversaire,
- -1 si le joueur qui joue à partir de cette position ne peut pas obliger l'adversaire à perdre.

On ne connaît pas a priori la valeur à associer à une position, sauf pour les feuilles. En effet, la règle du jeu nous indique pour chaque feuille si c'est une position gagnante ou non pour un joueur. Mais, si on connaît les valeurs $v_1 \ldots v_k$ des *fils* d'une position du joueur A, alors on en déduit sa valeur v. En effet, si tous les v_j valent 1, le joueur B est certain de gagner, elle vaut donc -1 pour A. S'il existe au moins un v_j égal à -1, alors, en jouant cette position, A est certain de gagner. Par conséquent, la valeur de v est $v = \max(-v_1, \ldots, -v_k)$. D'où une fonction F qui calcule la valeur d'une position :

```
(define (F position)
  (if (finale? position)
    (valeur position)
    (apply max (map (lambda (position-suivante)
                      (- (F position-suivante)))
                    (Lpositions-suivantes position)))))
```

Cette fonction utilise trois fonctions qui dépendent du jeu considéré :

- le prédicat `finale?` qui indique que dans cette position l'un des joueurs a perdu et l'autre a gagné,
- la fonction `valeur` qui donne la valeur +1 ou -1 d'une position finale en fonction des règles du jeu,

- la fonction `Lpositions-suivantes` qui retourne la liste des positions possibles à partir d'une position donnée, c'est en quelque sorte la définition du jeu.

Le calcul de la valeur `(F position)` nécessite un parcourdst complet de l'arbre du jeu. La connaissance de `F` enlève tout intérêt au jeu car on sait dès le départ si `A` est certain de gagner ou non. Pour les jeux intéressants (échec, dames, go, ...) l'arbre est trop vaste pour qu'une exploration *complète* soit effectivement faisable (avec les machines d'aujourd'hui et même celles de demain).

Pour obtenir une «approximation» `F-estimation` de `F`, une méthode consiste à limiter la profondeur de l'exploration à une certaine valeur appelée `profondeur-max`. Pour les positions qui sont à la profondeur maximum et qui ne sont pas des feuilles, on fournira une estimation de leur valeur au moyen d'une fonction appelée *heuristique*. Une heuristique fournira pour chaque position une valeur comprise entre des constantes `-MAX` et `MAX`. Plus grande est la valeur, plus favorable est supposée être la position pour le joueur concerné. Il est légitime de penser que la fiabilité d'une heuristique s'améliore quand on se rapproche des feuilles. On va donc utiliser l'heuristique pour les positions à la profondeur maximum et estimer la valeur des autres positions par la méthode de calcul précédente. Pour obtenir la fonction `F-estimation`, il suffit d'ajouter une nouvelle condition d'arrêt quand la profondeur maximum est atteinte.

```
(define (F-estimation position prof-max  heuristique)
  (letrec ((F-aux
             (lambda (position profondeur)
               (if (or (zero? profondeur)(finale? position))
                 (heuristique position)
                 (apply max (map (lambda (position-suivante)
                                   (- (F-aux position-suivante
                                              (- profondeur 1))))
                                 (Lpositions-suivantes position)))))))
    (F-aux position prof-max)))
```

On se propose d'optimiser ce calcul sans changer la valeur de la fonction `F-estimation`. Pour contrôler l'ordre des calculs du maximum, on réécrit cette fonction en remplaçant l'itérateur `map` par une fonction `LeMax` qui accumule dans la variable `valeur` le maximum cherché. La fonction `LeMax` parcourt la liste des positions à la recherche de la valeur maximum ; cette valeur est représentée par la variable de même nom et est initialisée à `-MAX`.

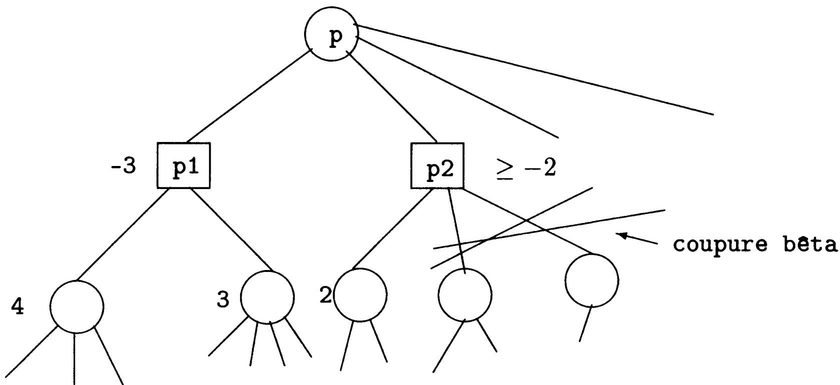
```
(define (F-estimation  position  prof-max  heuristique)
  (letrec ((F-aux (lambda (position profondeur)
                    (if (or (zero? profondeur)(finale? position))
                      (heuristique position)
                      (LeMax (Positions-suivantes position)
                             profondeur  -MAX))))

           (LeMax (lambda (Lpositions  profondeur valeur)
                    (if (null? Lpositions)
                      valeur
                      (let ((NlleValeur (- (F-aux (car Lpositions)
```

```
                                        (- profondeur 1)))))
                    (if (< valeur NlleValeur)
                      (LeMax (cdr Lpositions) profondeur NlleValeur)
                      (LeMax (cdr Lpositions) profondeur valeur)))))))
   (F-aux position  prof-max)))
```

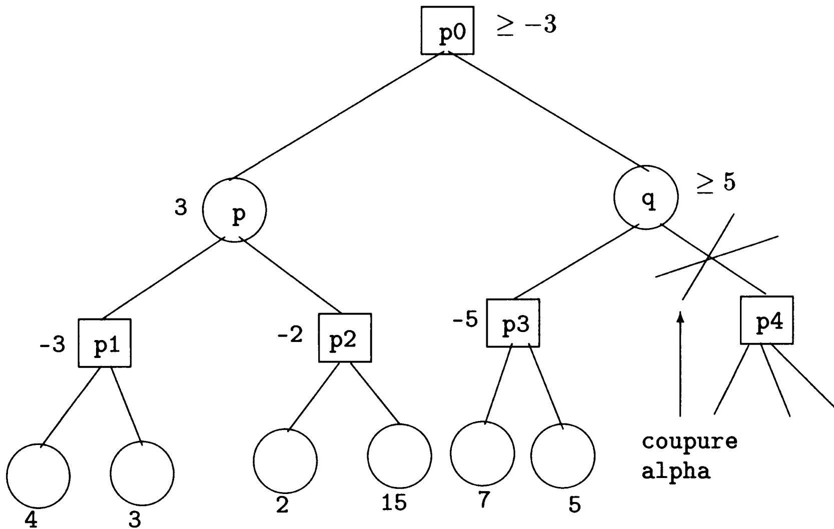
Coupures alpha et bêta

La fonction `F-aux` parcourt la liste des positions suivantes, de la gauche vers la droite, ce qui correspond à une exploration en pré-ordre de l'arbre de jeu. La connaissance de la valeur courante de `F-aux` peut amener dans certains cas à abandonner la considération d'autres positions car on peut dire a priori qu'elles ne changeront pas la valeur déjà trouvée.

Considérons le cas de figure suivant. On a calculé la valeur de la position **p1** et l'on est en train de calculer celle de **p2**. La valeur de **p1** étant -3, celle de **p** sera supérieure à 3, et comme on sait déjà que la valeur de **p2** sera supérieure à -2 il est inutile d'achever son calcul. C'est ce que l'on appelle traditionnellement la *coupure bêta*.



La coupure alpha correspond au même principe mais au coup précédent.



Ayant calculé la valeur 3 pour la position p et la valeur -5 pour p3, il est inutile d'examiner la position p4 car on sait déjà que la position q aura une valeur ≥5 et donc ne contribuera pas à p0.

Pour permettre ces optimisations, on ajoute en paramètre aux fonctions F1 et F-aux une estimation, appelée beta, du maximum et on renomme en alpha la valeur courante. Comme la valeur cherchée sera entre alpha et beta, on peut se permettre d'interrompre dès que alpha ≥ beta. Le rôle des valeurs alpha et beta est inversé à chaque changement de joueur à cause du changement de signe dans le calcul des valeurs.

```
(define (F-estimation  position prof-max  heuristique)
  (letrec ((F-aux (lambda (position profondeur alpha beta)
                   (if (or (zero? profondeur)(finale? position))
                     (heuristique position)
                     (LeMax (Lpositions-suivantes position)
                            profondeur alpha beta))))

          (LeMax (lambda (Lpositions  profondeur alpha beta)
                   (if (or (null? Lposition)(>= alpha beta))
                     alpha
                     (let ((NlleValeur
                            (- (F-aux (car Lpositions)
                                      (- profondeur 1)
                                      (- beta) (- alpha)))))
                       (if (< alpha NlleValeur)
                         (LeMax (cdr Lpositions) profondeur
                                NlleValeur beta)
                         (LeMax (cdr Lpositions) profondeur
                                alpha beta)))))))
    (F-aux position  prof-max  -MAX MAX)))
```

Exercice 19 *Tester ces fonctions en prenant comme arbre de jeu l'arbre ci-dessus.*

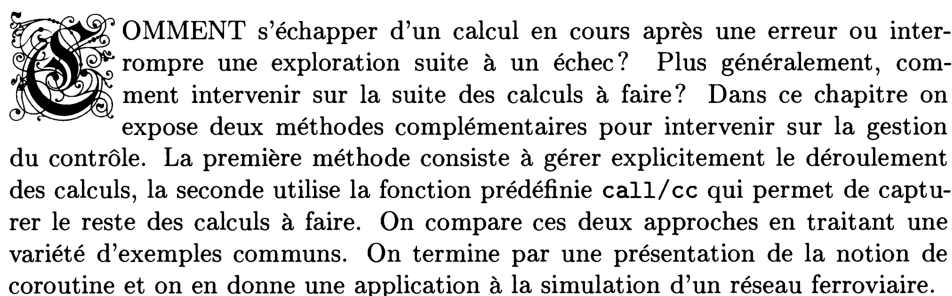
Il existe déjà des programmes qui jouent aux échecs à un haut niveau. Mais la mise en œuvre de ces méthodes sur un jeu non trivial demande un travail de longue haleine.

7.14 De la lecture

La littérature sur l'algorithmique et les structures de données est très riche, citons quelques ouvrages par ordre chronologique [Wir76, HS78, FGS90, CLR95].
Pour les jeux à deux joueurs, on peut consulter [HS78, Nil92, Nor92].

Chapitre 8

Programmation par continuation



COMMENT s'échapper d'un calcul en cours après une erreur ou interrompre une exploration suite à un échec? Plus généralement, comment intervenir sur la suite des calculs à faire? Dans ce chapitre on expose deux méthodes complémentaires pour intervenir sur la gestion du contrôle. La première méthode consiste à gérer explicitement le déroulement des calculs, la seconde utilise la fonction prédéfinie `call/cc` qui permet de capturer le reste des calculs à faire. On compare ces deux approches en traitant une variété d'exemples communs. On termine par une présentation de la notion de coroutine et on en donne une application à la simulation d'un réseau ferroviaire.

8.1 Continuation d'un calcul

Scheme étant un langage fonctionnel, l'action principale consiste à enchaîner des calculs, c'est-à-dire à prendre le résultat d'une fonction puis à le fournir à une ou plusieurs autres fonctions jusqu'au résultat final. La fonction qui transforme le résultat d'un calcul intermédiaire en résultat final s'appelle la *continuation* de ce calcul.

Par exemple, pour calculer $\frac{1}{n!}$, on calcule $n!$ puis on inverse cette valeur. Dans ce cas, la continuation du calcul de $n!$ pour obtenir le résultat final est la fonction `(lambda (x)(/ 1 x)`. Donnons-nous une fonction `k` d'une variable qui représente la continuation du calcul et désignons par `cont-fac` la fonction composée de factorielle par la continuation `k`:

```
(cont-fac n k) = (k n!)
```

Partant de la définition usuelle de factorielle, essayons d'en déduire une définition *directe* de `cont-fac`.

```
(define (fac n)
  (if (zero? n)
    1
    (* n (fac (- n 1)))))
```

Pour $n = 0$, on a :
`(cont-fac 0 k) = (k (fac 0)) = (k 1).`

Pour $n > 0$, on a :
`(cont-fac n k) = (k (fac n)) = (k (* n (fac (- n 1))))`
Pour passer de la valeur de `v = (fac (- n 1))` à la valeur finale `(k (* n (fac (- n 1))))`, il faut appliquer à v la nouvelle continuation `(lambda (v)(k (* n v)))`, d'où l'égalité :

```
(cont-fac n k) = (cont-fac (- n 1)  (lambda (v)(k (* n v))) ).
```

En résumé, on obtient pour `cont-fac` la définition

```
(define (cont-fac n k)
  (if (zero? n)
    (k 1)
    (cont-fac (- n 1) (lambda (x) (k (* n x))))))
```

Si l'on désire calculer la valeur de $\frac{1}{n!}$, on utilise la continuation
`k = (lambda (v)(/ 1 v))` :

```
? (cont-fac 5 (lambda (v)(/ 1 v)))  -> 1/120
```

Si l'on ne veut que la valeur de $n!$, il suffira de prendre pour continuation la fonction identité :

```
(fac n) = (cont-fac n (lambda (v) v))
```

Le lecteur attentif aura remarqué que la version avec continuation a aussi l'avantage d'être sous forme récursive *terminale*. Traçons le calcul de 4! :

```
? (cont-fac 4 (lambda (v) v))
Entry (cont-fac 4 '#[procedure #xCAEB0])
|Entry (cont-fac 3 '#[procedure #xCAEAA])
| Entry (cont-fac 2 '#[procedure #xCAEA4])
|  Entry (cont-fac 1 '#[procedure #xCAE9E])
|   |Entry (cont-fac 0 '#[procedure #xCAE98])
|   |==> 24
|   ==> 24
| ==> 24
|==> 24
==> 24
24
```

Contrairement au calcul défini par `fac`, on voit qu'il n'y a pas empilement des calculs qu'il reste à faire, puisque ces calculs sont stockés dans les continuations successives. On a bien une forme itérative, mais il ne faut pas trop s'illusionner sur un éventuel gain en place mémoire. En effet, il faut ranger en mémoire des fonctions de plus en plus complexes.
Ce style de programmation, qui consiste à passer de la forme `fac` à la forme `cont-fac`, s'appelle la *programmation par passage de continuation* (*continuation passing style*, en abrégé CPS). On en donnera des applications aux paragraphes 3, 4 et 5. Voyons quelques exemples de mise sous forme `cont-f` d'une fonction `f`, où en général `cont-f` sera définie par :

```
(cont-f  arg1 ... argN  k)  =  (k  (f  arg1 ... argN))
```

8.2 Programmation par passage de continuation

La fonction somme au premier niveau en style CPS

Commençons par un exemple voisin de factorielle : la fonction qui additionne les éléments d'une liste plate de nombres. On part de la définition usuelle de somme :

```
(define (somme Lnb)
  (if (null? Lnb)
    0
    (+ (car Lnb)
       (somme (cdr Lnb))))).
```

La version par continuation doit vérifier les égalités :

```
(cont-somme '() k) = (k 0)

(cont-somme Lnb  k) = (k (+ (car Lnb) (somme (cdr Lnb)))
```

Pour passer de la valeur de `(somme (cdr Lnb))` à la valeur finale, il faut appliquer la continuation `(lambda (v)(k (+ (car Lnb) v)))`, soit :

```
(cont-somme Lnb k) = (cont-somme (cdr Lnb)(lambda (v)(k (+ (car Lnb) v))))
```

D'où la définition de `cont-somme` :

```
(define (cont-somme Lnb k)
  (if (null? Lnb)
    (k 0)
    (cont-somme (cdr Lnb) (lambda (v)(k (+ (car Lnb) v))))))
```

La fonction somme à tous les niveaux en style CPS

Compliquons un peu en considérant la somme des éléments à tous les niveaux d'une liste quelconque de nombres, c'est la fonction `somme*` :

```
(define (somme* Lnb)
  (cond ((null? Lnb) 0)
        ((pair? (car Lnb))(+ (somme* (car Lnb))(somme* (cdr Lnb))))
        (else (+ (car Lnb) (somme* (cdr Lnb))))))
```

Pour définir `(cont-somme* Lnb k)`, considérons chaque branche du `cond` :

- la première branche doit évidemment retourner `(k 0)`,
- la dernière branche se traite exactement comme le cas de `somme`, on retourne :

```
(cont-somme* (cdr Lnb) (lambda (v)(k (+ (car Lnb) v))))))
```

- la deuxième branche est plus complexe à cause des deux appels récursifs. Il s'agit d'expliciter :

```
 (k (+ (somme* (car Lnb))(somme* (cdr Lnb)))))
```

Si l'on considère que l'on calcule en premier `(somme* (car Lnb))` alors le reste des calculs est donné par

```
 (lambda (v)(k (+  v  (somme* (cdr Lnb)))))
```

D'où la valeur de la deuxième branche :

```
(somme*  (car Lnb) (lambda (v)(k (+  v  (somme* (cdr Lnb))))))
```

Dans la lambda, la valeur de la sous-expression `(k (+ v (somme* (cdr Lnb))))` s'obtient en appliquant la continuation `(lambda (w)(k (+ v w)))` à la valeur de `(somme* (cdr Lnb))`. Elle s'écrit donc encore :
`(cont-somme* (cdr Lnb) (lambda (w)(k (+ v w))))`
D'où finalement une définition de `cont-somme*`

```
(define (cont-somme* Lnb k)
  (cond ((null? Lnb) (k 0))
        ((pair? (car Lnb))
         (cont-somme* (car Lnb)
                      (lambda (somme-car)
                        (cont-somme* (cdr Lnb)
                                     (lambda (somme-cdr)
                                       (k (+ somme-car somme-cdr)))))))
        (else (cont-somme* (cdr Lnb)
                           (lambda (somme-cdr)
                              (k (+ (car Lnb) somme-cdr)))))))
```

Remarque 1 Pour mieux faire comprendre le rôle de chaque continuation, on a baptisé les paramètres formels des lambda expressions avec les noms des valeurs qu'ils sont censés représenter : `somme-car` et `somme-cdr`.

Cette formulation de `cont-somme*` a été faite en supposant que, lors du calcul de la somme :

```
 (+ (somme* (car Lnb))(somme* (cdr Lnb)))
```

c'était (somme* (car Lnb)) qui était évalué en premier. On aurait pu tout aussi bien supposer que ce soit (somme* (cdr Lnb)), alors on serait arrivé à la forme équivalente suivante :

```
(define (cont-somme* Lnb k)
  (cond ((null? Lnb) (k 0))
        ((pair? (car Lnb))
         (cont-somme* (cdr Lnb)
                      (lambda (somme-cdr)
                        (cont-somme* (car Lnb)
                                     (lambda (somme-car)
                                       (k (+ somme-car somme-cdr )))))))
        (else (cont-somme* (cdr Lnb)
                           (lambda (somme-cdr)
                              (k (+ (car Lnb) somme-cdr)))))))
```

On voit que la programmation par continuation oblige à expliciter l'ordre d'évaluation des arguments d'un appel de fonction alors que Scheme ne le précise pas.

Exercice 1 *1. En partant de la définition usuelle de* **renverse** *donnée par :*

```
(define (renverse L)
  (if (null? L)
    L
    (append (renverse (cdr L)) (list (car L)))))
```

Ecrire la définition de la fonction **cont-renverse**.

2. Faire de même pour la fonction **renverse*** *qui inverse à tous les niveaux :*

```
(define (renverse* L)
  (cond ((null? L)  L)
        ((pair? (car L))(append (renverse* (cdr L))
                                (list (renverse* (car L)))))
        (else (append (renverse* (cdr L)) (list (car L))))))
```

8.3 Continuation et valeurs multiples

Ce style de programmation par continuation permet une programmation élégante des fonctions qui doivent rendre plusieurs valeurs.

Maximum d'une liste et indice d'un élément maximisant

Etant donné une liste (non vide) de nombres, on désire calculer, en une seule passe, le maximum de cette liste *et* l'index d'un élément maximum. Une première méthode (non élégante) consiste à définir une fonction max&index qui retourne ces deux valeurs dans une paire :

```
(define (max&index Lnb)
  (if  (null? (cdr Lnb))
    (cons (car Lnb) 0)
    (let ((max&ind (max&index (cdr Lnb)))
          (x (car Lnb)))
      (if (<= x (car max&ind))
        (cons (car max&ind) (1+ (cdr max&ind)))
        (cons x 0)))))

? (max&index '(1 5 3  14 1 2 14 5)) -> (14 . 6)
```

Considérons maintenant une version de cette fonction avec une continuation `k` qui prend *deux* arguments : le maximum trouvé et un index :

```
(define (cont-max&index L k)
  (if (null? (cdr L))
    (k (car L) 0)
    (cont-max&index (cdr L)
      (lambda (max-cdr index-cdr)
        (if (<= (car L) max-cdr)
          (k max-cdr (1+ index-cdr))
          (k (car L) 0))))))
```

On retrouve le cas particulier de la fonction `max&index` en utilisant la continuation `(lambda (max index) (cons max index))`. Si on n'a besoin que du maximum, on appelle `cont-max&index` avec la continuation `(lambda (max index) max)`.

Quotient et reste de la division euclidienne

La division euclidienne est aussi un bon exemple de fonction à valeurs multiples. Supposons que Scheme ne fournisse pas les fonctions `quotient` et `remainder`, on peut les regrouper en une seule fonction qui permettra de calculer toute fonction de ces deux valeurs.
Commençons par la version naïve qui retourne la paire `(quotient . reste)` :

```
(define (quotient&reste a b)
  (if (< a b)
    (cons 0 a)
    (let ((q.r (quotient&reste (- a b) b)))
      (cons (+ 1 (car q.r)) (cdr q.r)))))
```

Pour rendre le quotient et le reste, on utilise une continuation `k` qui est une fonction de ces deux arguments. En remplaçant `(cons 0 a)` par `(k 0 a)`
et
`(cons (+ 1 (car q.r)) (cdr q.r))` par `(k (+ 1 q) r)))))`
on obtient la définition de `cont-quotient&reste` :

```
(define (cont-quotient&reste a b k)
  (if (< a b)
    (k 0 a)
    (cont-quotient&reste (- a b) b
      (lambda (q1 r1) (k (+ q1 1) r1)))))
```

Egalité de Bezout

On a donné au chapitre 6 §1 une fonction pour calculer les coefficients u, v de l'égalité de Bezout :

Pour tous entiers a et b, il existe des entiers relatifs u et v tels que :

$$\text{(Bezout)} \quad au + bv = d \quad \text{où } d \text{ est le pgcd de } a \text{ et } b$$

La nécessité de calculer trois valeurs : `u, v, d` avait conduit à rendre la liste `(u v d)` :

```
(define (bezout a b)
  (cond ((= 0 a)(list 1 1 b))
        ((= 0 b)(list 1 1 a))
        ((< a b)(let ((u&v&d (bezout a (- b a))))
                  (list (- (car u&v&d)(cadr u&v&d))
                        (cadr u&v&d)
                        (caddr u&v&d))))
        (else (let ((u&v&d (bezout (- a b) b)))
                (list (car u&v&d)
                      (- (cadr u&v&d)(car u&v&d))
                      (caddr u&v&d)))))))
```

Donnons-en maintenant une version avec une continuation `k` à trois paramètres `u, v, d`. On remplace `list` par la continuation `k` et les éléments qui composaient la liste `u&v&d` deviennent les paramètres de la continuation :

```
(define (cont-bezout a b k)
  (cond ((= 0 a)(k 1 1 b))
        ((= 0 b)(k 1 1 a))
        ((< a b)(cont-bezout a (- b a)
                             (lambda (u v d)
                               (k (- u v) v d))))
        (else (cont-bezout (- a b) b
                           (lambda (u v d)
                             (k u (- v u) d))))))


? (cont-bezout 99 78 list) -> (15 -19 3)
```

La version avec continuation est plus élégante et on ne passe pas son temps à construire des listes que l'on doit aussitôt après décomposer en éléments.

Fibonacci par continuation

On a vu au chapitre 3 §9 une méthode pour avoir une version itérative de `fib`. Elle consiste à utiliser deux accumulateurs : un pour représenter u_n et l'autre pour u_{n-1}. Remplaçons ces deux accumulateurs par une continuation à deux paramètres représentant u_n et u_{n-1}. On veut avoir :

$$\texttt{(cont-fib n k)} = k(u_{n-1}, u_n).$$

Pour $n = 1$, on en déduit que :

(cont-fib 1 k) $= k(u_0, u_1) = k(0, 1)$
Pour $n > 1$, on aura successivement :
(cont-fib n k) $= k(u_{n-1}, u_n)$
$= k(u_{n-1}, u_{n-1} + u_{n-2})$
= ((lambda (u v) (k v (+ v u))) u_{n-2} u_{n-1})
=(cont-fib (- n 1)(lambda (u v) (k v (+ v u))))

D'où la définition de cont-fib :

```
(define (cont-fib n k)
  (if (= 1 n)
    (k 0 1)
    (cont-fib (- n 1) (lambda (u v) (k v (+ v u))))))
```

Comparer le temps de calcul de Fibonacci de 20 avec cette définition :

```
? (cont-fib 20  (lambda (u v) v) ) -> 6765
```

et avec celui utilisant la définition usuelle :

```
(define (fib n)
  (if (< n 2)
    n
    (+ (fib (- n 1)) (fib (- n 2)))))
```

Exercice 2 *Utiliser une continuation pour compter le nombre d'appels à* fib *quand on évalue* (fib 20) *selon le principe de la définition usuelle. La continuation à utiliser retournera deux valeurs : la valeur de* fib *et le nombre d'appels utilisés.*

8.4 Continuation et échappement

Puisque la continuation est une représentation de la suite des calculs, son utilisation s'impose si l'on souhaite modifier cette suite de calculs. On va illustrer cet aspect avec l'exemple classique du produit d'une liste de nombres pouvant comporter un 0.

Produit avec échappement

Partons de la définition naturelle du produit des éléments d'une liste :

```
(define (produit L)
  (if (null? L)
    1
    (* (car L) (produit (cdr L)))))
```

Si la liste comporte un 0, on fait beaucoup de calculs inutiles alors que ce produit est nul. Pour le voir, exécutons cette fonction en traçant les appels à la fonction de multiplication * et ceux de la fonction produit.

```
? (produit '(51 6 0 7 8))
Entry (produit '(51 6 0 7 8))
|Entry (produit '(6 0 7 8))
| Entry (produit '(0 7 8))
|  Entry (produit '(7 8))
|   |Entry (produit '(8))
|   | Entry (produit '())
|   | ==> 1
|   | Entry (* 8 1)
|   | ==> 8
|   |==> 8
|   |Entry (* 7 8)
|   |==> 56
|  ==> 56
|  Entry (* 0 56)
|  ==> 0
| ==> 0
| Entry (* 6 0)
| ==> 0
|==> 0
|Entry (* 51 0)
|==> 0
==> 0
0
```

Voici une deuxième version qui teste si l'on rencontre 0 au cours du calcul :

```
(define (produit1 L)
  (cond ((null? L) 1)
        ((zero? (car L)) 0)
        (else (* (car L) (produit1 (cdr L))))))

? (produit1 '(51 6 0 7 8))
Entry (produit1 '(51 6 0 7 8))
|Entry (produit1 '(6 0 7 8))
| Entry (produit1 '(0 7 8))
| ==> 0
| Entry (* 6 0)
| ==> 0
|==> 0
|Entry (* 51 0)
|==> 0
==> 0
0
```

On constate qu'on ne fait plus que les multiplications par 0 avec les nombres qui précèdent le premier zéro, ce qui est beaucoup plus satisfaisant. Donnons une version par continuation de cette fonction :

```
(define (cont-produit1 L k)
```

```
  (cond ((null? L) (k 1))
        ((zero? (car L))(k  0))
        (else (cont-produit1 (cdr L)
                             (lambda (p-cdr) (k (* p-cdr (car L))))))))
```

```
? (cont-produit1 '(51 6 0 7 8)  (lambda (x) x))
Entry (cont-produit1 '(51 6 0 7 8) '#[procedure #x142B34])
|Entry (cont-produit1 '(6 0 7 8) '#[procedure #x142B2E])
| Entry (cont-produit1 '(0 7 8) '#[procedure #x142B28])
|  Entry (* 0 6)
|  ==> 0
|  Entry (* 0 51)
|  ==> 0
| ==> 0
|==> 0
==> 0
0
```

On retrouve exactement les mêmes multiplications par 0, mais on a en plus l'avantage de la récursivité terminale.
On peut encore faire mieux : comme la continuation nous donne le contrôle de la suite des calculs, on peut décider que si l'on rencontre 0, on renvoie directement 0 en abandonnant tous les autres calculs :

```
(define (produit2 L k)
  (cond ((null? L) (k 1))
        ((zero? (car L))  0)
        (else (produit2 (cdr L)
                        (lambda (p-cdr) (k (* p-cdr (car L))))))))
```

L'appel de cette fonction sur la même liste retourne 0 sans effectuer la moindre multiplication :

```
? (produit2 '(51 6 0 7 8)  (lambda (x) x))
Entry (produit2 '(51 6 0 7 8) '#[procedure #x142B12])
|Entry (produit2 '(6 0 7 8) '#[procedure #x142B0C])
| Entry (produit2 '(0 7 8) '#[procedure #x142B06])
| ==> 0
|==> 0
==> 0
0
```

Attention, cette fonction vérifie encore l'égalité :

```
 (produit2 L k) = (k (produit L))
```

que pour les fonctions `k` telles que `(k 0) = 0`. Ce qui est bien le cas de la fonction identité.

Echappement en cas d'erreur

La détection d'une erreur est un autre exemple de situation où l'on désire abandonner complètement un calcul. Si au cours du calcul du produit des éléments de la liste on rencontre un élément qui n'est pas un nombre, Scheme déclenche une erreur car le test zero? ne s'applique qu'aux valeurs numériques :

```
 ? (produit2 '(42  27 a 51  6 0 7 8)  (lambda (x) x) )
*** ERROR -- NUMBER expected  (zero? 'a)
```

On peut souhaiter traiter soi-même cette erreur, ne serait-ce que pour donner un message en français. Il suffit d'appliquer exactement la même méthode que pour l'échappement quand on rencontre 0 : on teste si l'on a affaire à un nombre et sinon on s'échappe en n'appelant pas la continuation :

```
(define (produit3 L k)
  (cond ((null? L) (k 1))
        ((not (number? (car L)))
         (display-alln "*** ERREUR -- on attend un nombre: " (car L)))
        ((zero? (car L))  0)
        (else (produit3 (cdr L)
                        (lambda (p-cdr) (k (* p-cdr (car L))))))))

? (produit3 '(42  27  a  6 0 7 b  8)  (lambda (x) x) )
*** ERREUR -- on attent un nombre: a
```

Exercice 3 *Ecrire une fonction qui calcule le produit des inverses d'une liste de nombres.*

Valeurs multiples et récupération d'échec

On cherche à résoudre un problème dont la solution est à valeurs multiples et pouvant aussi échouer. Cela conduit à utiliser une continuation pour expliciter l'utilisation des résultats en cas de succès. Il est aussi utile de considérer une continuation pour indiquer ce qu'il reste à faire en cas d'échec.
Considérons la recherche du premier élément d'une liste plate L et satisfaisant à une condition p?. Si un tel élément existe, on désire le retourner ainsi que l'indice de sa position. D'où l'utilisation d'une continuation en cas de succès pour renvoyer ces deux valeurs. On emploie aussi une autre continuation pour indiquer ce qu'il faut faire après un échec.

```
(define (cherche L p? cont-succes cont-echec)
  (letrec ((cherche-aux
             (lambda (L i)
               (cond ((null? L)(cont-echec))
                     ((p? (car L))(cont-succes (car L)  i))
                     (else (cherche-aux (cdr L) (+ i 1)))))))
    (cherche-aux L 0)))
```

On teste, avec comme continuation en cas d'échec le retour du symbole echec.

```
? (cherche '(1 4 3 2 7 9) (lambda (x)(<= 5 x)) cons (lambda () 'echec))
(7 . 4)
? (cherche '(1 4 3 2 0 1) (lambda (x)(<= 5 x)) cons (lambda () 'echec))
echec
```

Exercice 4 *Généraliser au cas de la recherche dans une S-expression d'un atome vérifiant* **p?**. *On retournera l'atome trouvé et sa profondeur au sein de la S-expression.*

Pour rendre l'utilisation de la continuation `cont-echec` plus convaincante, on considère une recherche avec deux critères **p?** et **ps?**. On cherche en priorité un élément satisfaisant le critère **p?** mais en cas d'échec on relance la recherche avec le critère secondaire **ps?**. Pour permettre de changer de critère de recherche et de continuation d'échec, on les passe en paramètres de la fonction locale `cherche-aux`.

```
(define (cherche L p? ps? cont-succes cont-echec)
  (letrec ((cherche-aux
             (lambda (L i p? cont-echec)
               (cond ((null? L)(cont-echec))
                     ((p? (car L))(cont-succes (car L)  i))
                     (else (cherche-aux (cdr L) (+ i 1) p? cont-echec)))))
           (cont-echec1 (lambda ()(cherche-aux L 0 ps? cont-echec))))
    (cherche-aux L 0 p?  cont-echec1)))

? (cherche '(1 4 3 2 0 1) (lambda (x)(<= 5 x)) (lambda (x)(<= 3 x)) cons
                          (lambda () 'echec))
(4 . 1)

? (cherche '(1 -2 1 2 0 1) (lambda (x)(<= 5 x)) (lambda (x)(<= 3 x)) cons
                           (lambda () 'echec))
echec
```

Au lieu de s'arrêter à la première solution, on va s'intéresser à l'énumération des solutions.

8.5 Générateur de solutions (I)

Dans la recherche des solutions d'un problème, on désire parfois obtenir les diverses solutions à la demande. On fournit une première solution et, si l'utilisateur le souhaite, on en fournit une autre, ... Pour réaliser un tel générateur de solutions, il faut capturer la suite des recherches à faire lorsque l'on a trouvé une solution afin d'être en mesure de relancer la recherche à partir de cet état.
Considérons le problème de la recherche des atomes d'une S-expression qui satisfont un prédicat donné `p?`.

Recherche interactive

Voici une première version où l'on interroge l'utilisateur pour savoir s'il désire avoir une autre solution. S'il répond «oui», on rappelle la continuation `k` de la fonction de recherche.

Pour une paire on commence à chercher dans le car et on place dans la continuation l'exploration du cdr. Si l'on trouve un atome satisfaisant le prédicat, on appelle la continuation de succès qui a pour rôle d'interroger l'utilisateur pour savoir s'il désire oui ou non relancer la recherche. Si ol'n a rien trouvé, on renvoie le message «echec».

```
(define (generateur-interactif  s p? cont-echec)
  (letrec ((chercher
            (lambda (s k)
              (cond ((null? s) (k))
                    ((pair? s) (chercher (car s)
                                         (lambda ()
                                           (chercher (cdr s) k))))
                    ((p? s) (cont-succes s k))
                    (else (k)))))

           (cont-succes (lambda (s k)
                          (display-alln "solution : " s "
                                        autre solution : o/n : ")
                          (if (eq? (read) 'o)
                            (k)
                            (display-alln "au revoir")))))
   (chercher s cont-echec)))
```

Cherchons les nombres présents dans une S-expression :

```
? (generateur-interactif '(a (2 . c) ((0)) b 7)  number?
                                                 (lambda () 'echec))
solution : 2  autre solution : o/n : o
solution : 0  autre solution : o/n : n
au revoir
```

Création de générateur

Voici une autre version où l'on se dispense du dialogue avec l'utilisateur. Un générateur de solutions retourne après chaque appel une nouvelle solution ou bien appelle la continuation d'échec s'il n'y a plus de solution. Pour cela, après chaque succès on remplace la continuation de succès par la suite des recherches qui restent à faire. Au départ cette continuation lance la recherche.

```
(define (make-generateur s p? cont-echec)
  (letrec ((chercher
            (lambda (s k)
              (cond
                ((null? s) (k))
                ((pair? s) (chercher (car s)
                                     (lambda ()(chercher (cdr s) k))))
                ((p? s)(set! cont-succes k) s)
                (else (k)))))
           (cont-succes (lambda ()(chercher s cont-echec))))
    (lambda ()(cont-succes))))
```

```
? (define genere-nombre
     (make-generateur  '(a (2 . c) ((0)) b 7) number?
                                              (lambda () 'echec)))

? (genere-nombre)
2

? (genere-nombre)
0

? (genere-nombre)
7

?(genere-nombre)
echec
```

Exercice 5 *On dit que deux S-expressions ont mêmes feuilles si les expressions aplaties sont égales. Si ces expressions sont très grandes et qu'elles diffèrent dès le début, il ne semble pas économique de construire les deux listes de feuilles puis de les comparer; ce problème classique s'appelle «*same fringe*» en anglais.*
L'idée naturelle est de parcourir ces expressions et de s'arrêter avec #f dès que deux feuilles correspondantes ne sont pas égales. Il faut donc un moyen de parcourir à la demande les expressions, d'où l'utilisation du générateur. Pour cela, écrire une boucle de parcours de deux S-expressions en utilisant un générateur de feuilles de chaque arbre. On trouvera au chapitre 10 §7 une autre solution avec les flots.

8.6 La fonction `call/cc`

Continuation d'une sous expression

Comment passer de la valeur d'une sous-expression `e` à la valeur de l'expression complète `E = (...e ...)`?

Si l'on remplace l'occurrence de `e` par une variable notée `[]`, alors la valeur de `E` s'obtient en appliquant la lambda expression `(lambda ([]) (...[] ...))` à la valeur de `e`. Autrement dit, cette lambda expression est la continuation `k` qu'il faut appliquer à la valeur de `e`, notée `val-e`, pour obtenir la valeur de `E`:

`valeur de (...e ...) = (k e)`

Exemples

- Si `E = (* 4 (+ x 2))` et `e = x` alors
  `k = (lambda ([]) (* 4 (+ [] 2)))`
- Si `E = (if (= i 1) (+ i x) (- i 1)))` et `e = x` alors
  `k = (lambda ([])(if (= i 1) (* i []) (- i 1)))`
  Ici cette expression peut se simplifier en `(lambda ([])(+ 1 []))`

- Si E =
```
(let ((u 1)
      (v 4))
  (if (number? u)
    (+ u v)
    (cons u v)))
```
  et si e désigne l'occurrence de v dans (+ u v) alors :

```
k = (lambda ([])
     (let ((u 1)
           (v 4))
       (if (number? u)
         (+ u [])
         (cons u v))))
```

  qui peut se réduire à

```
(lambda ([])(+ 1 [])
```

Définition de call/cc

Le langage Scheme rend visible à l'utilisateur cette continuation par l'intermédiaire de la primitive call-with-current-continuation, en abrégé call/cc. La syntaxe est (call/cc *fct1*) où *fct1* désigne une fonction d'un seul argument.
Comment Scheme évalue l'expression suivante ?

```
E = (. . . (call/cc  (lambda (k)  corps) . . .)
```

L'appel de call/cc a pour effet de lier le paramètre k avec la continuation de l'occurrence de la sous expression (call/cc (lambda (k) corps) dans E.
La valeur de la sous expression e = (call/cc (lambda (k) corps)
dans E = (...e ...) est :
- celle de l'expression corps si son évaluation n'appelle pas k , d'où ensuite celle de E
- si on a un appel (k s), tous les calculs en cours sont abandonnés et la valeur de e est celle de s, et celle de E est donnée par (k s).

Exemples

```
? (call/cc (lambda (k)(+ 2 (* 7 (k 3)))))
3
```

Ici call/cc est appelée au sommet de l'expression, aussi la continuation k est l'identité : k = (lambda ([]) []). Comme k est appelée dans le corps de ce call/cc, la valeur rendue est (k 3) = 3.

```
? (* 4 (+ (call/cc (lambda (k)(* 5 8)))
          2))
168
```

ici la continuation k n'est pas appelée, donc la valeur de la sous-expression (call/cc (lambda (k)(* 5 8)) est celle de son corps (* 5 8).

```
? (* 4 (+ (call/cc (lambda (k)(* (k 5) 8 ))) 2))
28
```

Cette fois, la continuation k est appelée, elle a pour valeur : (lambda ([])(* 4 (+ [] 2))), donc l'expression a pour valeur (k 5) soit 28.

On peut emboîter des continuations, considérons l'expression :

```
? (+ 2 (call/cc (lambda (k1)
                 (- 9 (call/cc (lambda (k2)
                                (* 7  (k1 (+ 21 (k2 5))))))))))
```

Les continuations sont données par :

```
k1 = (lambda ([])(+ 2 []))
```
et
```
k2 = (lambda ([])(+ 2 (call/ cc (lambda (k1)(- 9 [])))))
   = (lambda ([])(+ 2 (- 9 []))).
```

L'évaluation de (* 7 (k1 (+ 21 (k2 5)))) fait un appel à k1, on rend donc la valeur de (k1 (+ 21 (k2 5))) = (+ 2 (+ 21 (k2 5)). L'évaluation de la sous--expression (+ 21 (k2 5)) fait un appel à k2, on rend donc (k2 5) d'où la valeur 6.

Exercice 6 *Donner les valeurs des expressions suivantes :*
(call/cc (lambda (k1)(+ 10 (call/cc (lambda (k2)(5 (k2 3)))))))*
(call/cc (lambda (k1)(+ 10 (call/cc (lambda (k2)((k1 5) 3))))))*
(call/cc (lambda (k1)(+ 10 (call/cc (lambda (k2)((k1 5) (k2 3)))))))*
(call/cc (lambda (k1)(+ 10 (call/cc (lambda (k2)((k2 3) (k1 5)))))))*

Capture de continuation

La continuation de l'évaluation d'une sous-expression est un objet connu du langage servant à implanter notre langage Scheme. Le fait de rendre accessible dans le langage de programmation un objet du langage d'implantation s'appelle pompeusement une *réification*. La réification de la continuation en une fonction Scheme d'une variable permet d'en disposer comme toute autre fonction. Par exemple, on peut l'affecter à une autre variable ou la passer en paramètre à une fonction. C'est une valeur de première classe.

Sauvons dans une variable k2 la continuation capturée en évaluant :

```
? (cons (car (call/cc (lambda (k)(set! k2 k)(list (k '(a b))))))
        '(c d))
(a c d)
```

On vérifie bien que k2 est la fonction (lambda (L)(cons (car L) '(c d))) :

```
? (k2 '(u v w)) -> (u c d)
```

La fonction `call/cc` fournit un moyen très (trop?) puissant pour agir sur le contrôle des calculs. En particulier, on peut refaire avec `call/cc` les diverses modifications du contrôle que l'on vient de décrire à l'aide de la programmation par continuation.

8.7 Call/cc et échappements

On reprend la question des échappements du §4 avec l'utilisation de `call/cc`.

Produit avec échappement

On commence par capturer la continuation au moment de l'appel du calcul du produit, puis, si on rencontre 0, on s'échappe en appelant cette continuation.

```
(define (produit L)
  (call/cc (lambda (exit)
             (letrec ((produit-aux
                        (lambda (L)
                          (cond ((null? L) 1)
                                ((zero? (car L)) (exit 0))
                                (else (* (car L)
                                         (produit-aux (cdr L))))))))
               (produit-aux L)))))
```

Pour vérifier que l'on n'effectue aucune multiplication quand la liste comporte un 0, on demande la trace de l'opération de multiplication :

```
? (trace *)

? (produit '(42  27  51  6 0 7 8))
0

? (produit  '(4  7 8))
Entry (* 8 1)
==> 8
Entry (* 7 8)
==> 56
Entry (* 4 56)
==> 224
224
```

Echappement en cas de réussite

La même technique de capture de continuation permet de s'échapper d'une recherche dès que l'on a trouvé une solution. Revenons à la recherche d'un atome d'une S-expression `s` qui vérifie un prédicat donné `p?`.

```
(define (recherche  s p?)
  (call/cc (lambda (succes)
             (letrec ((recherche-aux
                       (lambda (s)
                         (cond ((null? s) #f)
                               ((pair? s)(or (recherche-aux (car s))
                                             (recherche-aux (cdr s))))
                               ((p? s)(succes  s))
                               (else #f)))))
              (recherche-aux s)))))

? (recherche '(2 4 (6  (7))  3) odd? ) -> 7
? (recherche '(2 4 (6  (8))  3) odd? ) ->#f
```

Exercice 7 *Définir des fonctions (*`interrompre` `message`*) et (*`reprendre`*) telles que :*

- *l'appel de (*`interrompre message`*) interrompt l'évaluation de l'expression, il y a affichage du message et retour au top level ;*
- *l'appel de (*`reprendre`*) au top level permet de reprendre l'évaluation de la dernière expression interrompue (voir une solution dans la Slib [Jaf]).*

Tester avec la fonction :

```
(define (f x)
  (if (= x 0)
    1
    (begin  (interrompre "x= " x)
            (* x (f (- x 1))))))
```

Menus et échappements

Pour réaliser une interface utilisateur, on utilise souvent un système de boucles dirigées par des menus. Il est facile de quitter une boucle pour revenir à la boucle englobante mais il est plus délicat de quitter tout depuis une boucle plus profonde. On va donner le principe d'un tel échappement en utilisant la capture de continuation.

On commence par considérer une simple boucle, appelée `menu`, avec deux options : `lire` ou `quitter`. On utilise la boucle `do` avec un test toujours `#f` pour réaliser une boucle infinie. Pour quitter, on capture la continuation d'appel :

```
(define (menu)
  (call/cc (lambda (quitter)
             (do ()
                 (#f)
               (newline)
               (display "(L)ire  (Q)uitter-menu : ")
               (case (read)
                 ((L) (lire))
                 ((Q) (quitter 'fin-depuis-menu))
                 (else  #t))))))
```

Les choses se compliquent quand l'action de lire est aussi une boucle avec un menu. Le lecteur peut choisir de lire une revue ou une thèse ou de revenir au menu principal. Mais on veut aussi pouvoir quitter tout depuis cette boucle. Essayons la méthode suivante, elle appelle la continuation quitter pour tout quitter.

```
(define (lire)
  (call/cc (lambda ( quitter-lire)
             (do ()
                 (#f)
               (newline)
               (display "(R)evue (T)hèse (Q)uitter Quitter-(L)ire : ")
               (case (read)
                 ((R) (display "lecture de revue")(newline))
                 ((T) (display "lecture de thèse")(newline))
                 ((L) (quitter-lire 'fin-lecture))
                 ((Q) (quitter  'fin-depuis-lire))
                 (else  #t))))))
```

On essaye :

```
? (menu)

(L)ire  (Q)uitter-menu : L

(R)evue (T)hèse (Q)uitter Quitter-(L)ire : R
lecture de revue

(R)evue (T)hèse (Q)uitter Quitter-(L)ire : T
lecture de thèse

(R)evue (T)hèse (Q)uitter Quitter-(L)ire : Q
*** ERROR   -- Unbound variable: quitter
```

C'était prévisible car la portée de la variable quitter est limitée au corps de la lambda qui l'introduit. Il faudrait avoir une notion de portée dynamique pour que quitter soit encore visible pendant l'exécution de la fonction lire. Pour résoudre ce problème, on rend la continuation quitter visible dans lire en l'ajoutant en paramètre de cette fonction :

```
(define (menu)
  (call/cc (lambda (quitter)
             (do ()
                 (#f)
               (newline)
               (display "(L)ire  (Q)uitter-menu : ")
               (case (read)
                 ((L) (lire quitter))
                 ((Q) (quitter 'fin-depuis-menu))
                 (else  #t))))))
```

```
(define (lire c)
  (call/cc (lambda ( quitter-lire)
             (do ()
                 (#f)
               (newline)
               (display "(R)evue (T)hèse (Q)uitter Quitter-(L)ire : ")
               (case (read)
                 ((R) (display "lecture de revue")(newline))
                 ((T) (display "lecture de thèse")(newline))
                 ((L) (quitter-lire 'fin-lecture))
                 ((Q) (c 'fin-depuis-lire))
                 (else  #t))))))
```

Le lecteur connaissant Common Lisp comparera avec une solution utilisant les formes `catch` et `throw`.

8.8 Générateur de solutions (II)

L'utilisation de `call/cc` nous permet de donner une autre définition de la fonction `make-generateur` du §5. On n'a plus besoin de passer la continuation en paramètre de la fonction locale `chercher`. Quand on trouve une solution, on fait appel à `call/cc` pour capturer la suite de la recherche et on affecte cette continuation à la variable locale `cont-succes`. Au début, cette variable est initialisée avec la recherche à faire. On utilise une deuxième continuation `succes` pour interrompre la recherche dès que l'on a découvert une solution, cette continuation est obtenue en capturant la continuation d'appel. L'enchaînement des recherches dans le cas d'une paire est déterminé par l'évaluation en séquence de:
`(chercher (car s))` et `(chercher (cdr s))`

```
(define (make-generateur s  p?)
  (let ((succes  '?))
    (letrec ((cont-succes (lambda (any)(chercher s)))
             (chercher
               (lambda (s)
                 (cond ((null? s) 'fini)
                       ((pair? s) (chercher (car s))
                                  (chercher (cdr s) ))
                       ((p? s)(call/cc (lambda (k)(set! cont-succes k)
                                                  (succes  s))))
                       (else 'fini)))))
      (lambda ()
        (call/cc (lambda (k)(set! succes k)
                           (cont-succes 'fini)))))))
```

On utilise le même test qu'au §5 :

```
? (define genere-nombre  (make-generateur  '(a (2 . c) ((0)) b 7) number?)

? (genere-nombre)
2
```

```
? (genere-nombre)
0

? (genere-nombre)
7

? (genere-nombre)
fini
```

8.9 Une fonction *erreur*

Il peut être utile d'intercepter les erreurs quand on désire appliquer un traitement spécial ou bien si l'on ne veut pas être ramené sous le top level de Scheme par le déclenchement d'une erreur Scheme.

Echappement en cas d'erreur

Si l'on n'est pas assuré à l'avance que la liste L ne comporte que des nombres, on peut tester la nature des éléments afin d'éviter de déclencher une erreur Scheme quand le test zero? est appelé sur une valeur non numérique. Il suffit d'ajouter à la fonction produit-aux un cas supplémentaire d'échappement :

```
(define (produit L)
  (call/cc (lambda (exit)
             (letrec ((produit-aux
                        (lambda (L)
                          (cond ((null? L) 1)
                                ((not (number? (car L)))
                                 (exit 'erreur))
                                ((zero? (car L)) (exit 0))
                                (else (* (car L)
                                         (produit-aux (cdr L))))))))
               (produit-aux L)))))

? (produit  '(42  27 a 51  6 0 7 8))
erreur
```

Mais ce n'est pas idéal car :

```
? (+ 3 (produit '(12 37 a 41)))
*** ERROR -- NUMBER expected
```

En effet, notre message d'erreur est ensuite ajouté à 3 d'où déclenchement d'une erreur Scheme. Il fallait abandonner tous les calculs en attente, c'est à dire intervenir sur la continuation et la remplacer par un retour au top level. Pour cela, on baptise *abort* la continuation du top level. On la capture en évaluant ;

```
? (call/cc (lambda (k)(set! *abort* k)))
```

En effet, la continuation du top level est liée avec le paramètre formel `k` ; ce qui permet de la sauver dans la variable globale `*abort*`. En utilisant la continuation `*abort*`, on pourra abandonner tout calcul et revenir au top level. D'où une nouvelle version du produit :

```
(define (produit L)
  (call/cc (lambda (exit)
             (letrec ((produit-aux
                        (lambda (L)
                          (cond ((null? L) 1)
                                ((not (number? (car L)))
                                 (*abort* 'erreur))
                                ((zero? (car L)) (exit 0))
                                (else (* (car L)
                                         (produit-aux (cdr L))))))))
               (produit-aux L)))))

? (+ 3 (produit '(12 37 a 41)))
erreur
```

C'est mieux, mais on aimerait un message d'erreur plus précis. Il suffit de remplacer `*abort*` dans `produit` par une fonction `*erreur*` qui appellera `*abort*` après avoir effectué l'affichage désiré :

```
(define (*erreur*  . messages)
  (display "*** ERREUR ")
  (apply display-alln  messages)
  (*abort* #f))
```

On demande au lecteur d'ajouter cette fonction à ses utilitaires Scheme car on l'utilisera librement dans la suite.

Voici notre version finale pour le produit des éléments d'une liste :

```
(define (produit L)
  (call/cc (lambda (exit)
             (letrec ((produit-aux
                        (lambda (L)
                         (cond
                          ((null? L) 1)
                          ((not (number? (car L)))
                           (*erreur* "on attend un nombre : '' (car L)))
                          ((zero? (car L)) (exit 0))
                          (else (* (car L)
                                   (produit-aux (cdr L))))))))
              (produit-aux L)))))

? (+ 3 (produit '(12 37 a 41)))
*** ERREUR on attend un nombre : a
#f
```

Protection d'une fonction par une pré-condition

Il est parfois important de se prémunir contre l'utilisation de valeurs inadéquates pour les arguments d'une fonction. Une méthode consiste à insérer en tête du corps de la fonction une *pré-condition*. C'est une expression qui est chargée d'interrompre l'exécution de l'appel si une certaine expression est fausse. Il y a également affichage d'un message d'erreur.

```
(define (assert  pre-condition  message)
  (if (and *mode-test* (not pre-condition))
    (*erreur* message)))
```

On utilise une variable globale `*mode-test*` pour indiquer si l'on désire ou non activer les pré-conditions. Par exemple, on ne peut les rendre actives que pendant la période mise au point.

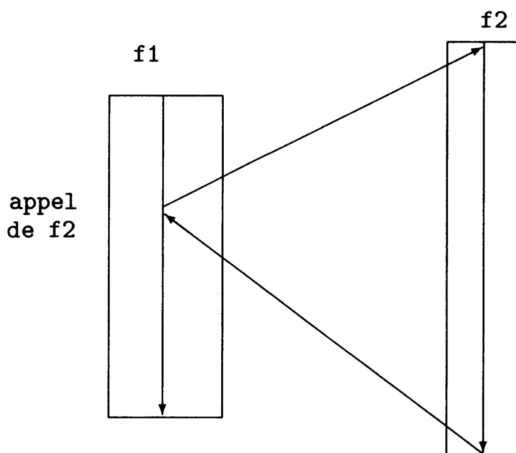
Voici un exemple. Si un Scheme dispose des nombres complexes, la fonction `sqrt` peut accepter les nombres négatifs. Pour avoir une fonction `rac2` qui ne prend que des arguments positifs, on pose :

```
(define (rac2 x)
  (assert (<= 0 x) "l'argument doit être positif")
  (sqrt x))
```

8.10 Coroutines

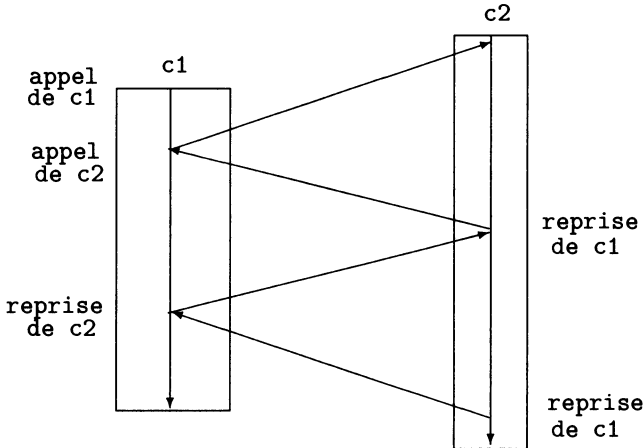
Introduction aux coroutines

Quand une fonction `f1` appelle une fonction `f2`, l'exécution de `f1` est interrompue et ne reprendra qu'après l'exécution *complète* de l'appel de `f2`. Pour cela, `f1` doit mémoriser la suite des calculs restant à faire. On schématise ce protocole par le diagramme suivant :



Appel de la fonction f2 dans f1

Il y a une dissymétrie entre les rôles de `f1` et `f2`. Le protocole de coroutinage permet de conserver une symétrie entre les rôles des fonctions considérées, on les appelle des *coroutines*. Dans le cas de deux coroutines `c1` et `c2`, on peut avoir le diagramme suivant :



Appels et reprises de coroutines

On commence par un appel de `c1`, puis au cours de l'exécution de `c1`, on appelle une première fois `c2` alors `c1` mémorise le reste des calculs à faire. L'exécution de `c2` commence jusqu'à l'ordre de reprise de `c1`. Alors `c1` reprend son cours et `c2` mémorise à son tour le reste de ses calculs à faire. L'exécution de `c1` déclenche un deuxième appel à `c2` qui reprend son exécution depuis son interruption. Puis de nouveau on reprend l'exécution de `c1` ...
L'exécution de plusieurs processus sur machine monoprocesseur est un exemple typique d'une telle situation. Le système d'exploitation alloue, selon des règles de priorité, un quantum de temps pour l'exécution (partielle) de chaque processus, ce qui permet de les modéliser par un ensemble de coroutines.
Pour simplifier, on se restreint aux coroutines sans paramètre. De même que l'on dispose de la définition d'une fonction `f` par `(define f ...)` et de l'appel par `(f ...)`, on a besoin d'une méthode de définition des coroutines et d'un moyen pour reprendre (ou démarrer) l'exécution d'une coroutine.
On va définir une fonction `make-coroutine` de création de coroutines. Elle permet la définition d'une coroutine de nom ***une-coroutine*** sans paramètre par :

```
(define une-coroutine (make-coroutine (lambda (reprendre) corps)))
```

Le paramètre formel ***reprendre*** désigne, dans le corps, le nom de la fonction servant à déclencher la reprise de l'exécution d'une autre coroutine : ainsi, l'appel de `(`***reprendre*** *c*`)` dans le corps d'une coroutine, provoque la reprise de l'exécution de la coroutine *c*.

Voici deux coroutines `c1` et `c2` qui fonctionnent selon le schéma ci-dessus :

```
(define c1 (make-coroutine
  (lambda (reprendre)
    (display-alln "debut de c1")
    (reprendre   c2)
    (display-alln "suite de c1")
    (reprendre   c2)
    (display-alln "fin de c1"))))

(define c2 (make-coroutine
  (lambda (reprendre)
    (display-alln "debut de c2")
    (reprendre   c1)
    (display-alln "suite de c2")
    (reprendre   c1)
    (display-alln "fin de c2"))))
```

Le premier appel de c1 se fait par (c1) :

```
? (c1)
debut de c1
debut de c2
suite de c1
suite de c2
fin de c1
```

De même si l'on commence par appeler c2 on obtient :

```
? (c2)
debut de c2
debut de c1
suite de c2
suite de c1
fin de c2
```

Construction des coroutines

Comment construire une coroutine? Une coroutine possède une variable locale appelée sa-continuation qui sert à sauvegarder sa continuation en cas d'interruption. Il faut donc que l'appel de la fonction reprendre ait pour effet de :

- sauvegarder la continuation de la fonction en cours dans sa-continuation (grâce à un call/cc),
- puis appeler la coroutine passée en argument à la fonction reprendre, ou plus précisément lancer l'exécution de sa continuation.

Au début la variable sa-continuation est initialisée avec l'appel du corps de la coroutine. D'où la définition de la fonction make-coroutine :

```
(define (make-coroutine proc) ; proc = (lambda (reprendre) corps)
  (letrec ((sa-continuation (lambda () (proc reprendre)))
         ;; variable pour stocker  la continuation de la coroutine
```

```
          (reprendre
          ;; pour reprendre  l'exécution d'une  coroutine  f
            (lambda (f)
              (call/cc (lambda (k)
                         (set! sa-continuation (lambda ()(k 'any)))
                         (f)))) )
          ;; déclenche la sauvegarde de la continuation de la coroutine
          ;; puis on appelle  f
     ;; au depart, la continuation est l'appel de la procedure
     (lambda ()(sa-continuation))))
```

Exercice 8 *Etendre* ***make-coroutine*** *au cas des coroutines ayant des paramètres. Pour cela, la fonction* ***reprendre*** *sera de la forme* ***(lambda (f . argts) ...).***

Producteur/consommateur

Un exemple type d'utilisation des coroutines est le problème du *producteur/consommateur.* C'est le cas où un processus produit (ou modifie) une donnée qu'un autre processus utilise au fur et à mesure. La difficulté est de synchroniser les deux processus pour que toute donnée produite puisse être éventuellement consommée. En voici une illustration très simple : le producteur est une boucle qui incrémente de 0 à 5 une variable globale n, le processus consommateur doit afficher au fur et à mesure les valeurs paires de n.

```
(define n -1)

(define producteur (make-coroutine
                     (lambda(reprendre)
                       (do ()
                           ((= n 5))
                         (display-alln "production ")
                         (set! n (+ n 1) )
                         (reprendre consomateur)))))

(define consomateur (make-coroutine
                      (lambda(reprendre)
                        (do ()
                            ((= n 5))
                          (if (even? n)(display-alln "consommation de n : "
                          (reprendre producteur)))))
? (producteur)
production
consommation de n : 0
production
production
consommation de n : 2
production
production
consommation de n : 4
production
```

8.11 Compléments : simulation ferroviaire

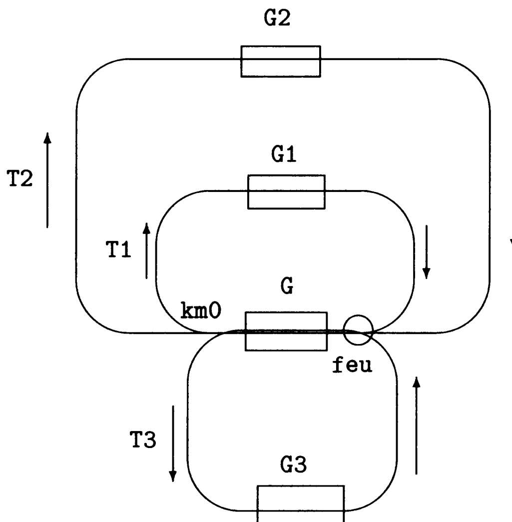
Modélisation d'un petit réseau ferroviaire

On va utiliser des coroutines pour simuler l'évolution de trois trains sur trois trajets ayant un tronçon commun. L'accès à ce tronçon est protégé par un feu de signalisation.

- Le train T1 effectue un trajet1 qui est une boucle possédant une gare G1 et le tronçon commun.
- Le train T2 effectue un trajet2 qui est une boucle possédant une gare G2 et le tronçon commun.
- Le train T3 effectue un trajet3 qui est une boucle possédant une gare 3 et le tronçon commun.

Le tronçon commun possède à son entrée un feu et comporte une gare G, l'autre extrémité de ce tronçon est par définition le km 0 de tous les trajets.

A l'instant 0 les trois trains partent du km 0 sur leurs voies respectives et le feu est au vert. Après avoir traversé leurs gares respectives, les trois trains se dirigent vers le feu. Le premier train qui se présente au feu le franchit et le fait passer au rouge, ensuite chaque train qui arrive au feu est arrêté et rangé dans une file d'attente. Quand un train quitte la gare G, le feu repasse au vert, ce qui libère le premier train en attente au feu, ce train s'engage à son tour sur le tronçon commun ce qui remet le feu au rouge.
On se propose d'indiquer l'enchaînement des mouvements des trains en fonction du temps.
L'avancement de chaque train est représenté par une coroutine que l'on interrompt quand le train est bloqué par le feu ou en arrêt dans une gare et que l'on relance quand le train quitte la gare ou passe le feu.



Définition des trains et des trajets

Un train sera représenté par un prototype qui répond à des messages concernant son nom, son heure de départ d'une gare, s'il est en arrêt au feu, la coroutine qui le régit, ...

```
(define (make-train  trajet nom)
  (letrec ((heure-depart 0)
           (heure-arrivee 0)
           (bloque-au-feu? #f)
           (sa-coroutine (make-coroutine
                          (lambda (reprendre)
                            (trajet train reprendre))))
           (train (lambda message
                    (case (car message)
                      ((nom) nom)
                      ((sa-coroutine) sa-coroutine)
                      ((heure-depart) heure-depart)
                      ((set-heure-depart)
                       (set!  heure-arrivee (cadr message)))
                      ((heure-arrivee) heure-arrivee)
                      ((set-heure-arrivee)
                       (set!  heure-arrivee (cadr message)))
                      ((bloque-au-feu?) bloque-au-feu?)
                      ((set-bloque-au-feu?)
                       (set!  bloque-au-feu? (cadr message)))))))
    train))
```

On représente l'écoulement du temps par une suite d'entiers que l'on génère en faisant appel à une fonction système (`runtime`) qui simule l'écoulement du temps. La fonction `heure` retourne à chaque appel l'heure exprimée en nombre d'unités de temps (disons des minutes) écoulées depuis le début de la simulation.

```
(define (heure)
  (inexact->exact(round (- (runtime) time0))))
```

On représente chaque trajet par une fonction qui est une boucle sans fin simulant les actions du train en fonction de sa position en km sur son trajet. A certaines positions kilométriques des informations sont affichées ou des instructions sont données au train : arrêt du train quand il arrive dans une gare et durée de l'arrêt, comportement du train à l'approche du feu, action sur le feu quand un train libère le tronçon commun.

```
(define (trajet1 train reprendre)
  (let ((position -1)
        (nom-train  (train 'nom)))
    (do ()
        (#f)
      (set! position (+ position 1))
      (case position
        ((0)  (display-alln (heure) "    "  nom-train  " passe en 0 ")
```

```
          (temps-trajet 2 train) (activer reprendre))
     ((10) (display-alln (heure) "  " nom-train  "
                        arrive en gare G1 3min d'arret ")
          (arret 3 train)
          (activer reprendre)
          (display-alln (heure) "  " nom-train  " quitte la gare G1 ")
          (temps-trajet 3 train) (activer reprendre))
     ((30) (display-all (heure) "   " nom-train   " arrive au feu  ")
          (bloque *feu* train)(activer reprendre)
          (temps-trajet 1 train)(activer reprendre))
     ((35) (display-alln (heure) "   " nom-train  "
                        arrive en gare G 5min d'arret ")
          (arret 5 train)(activer reprendre)
          (display-alln (heure) "   " nom-train  " quitte la  gare G ")
          (debloque *feu* reprendre)
          (set! position 0))    ))))

(define (trajet2 train reprendre)
  (let ((position -1)
        (nom-train  (train 'nom)))
    (do ()
        (#f)
      (set! position (+ position 1))
      (case position
        ((0)  (display-alln (heure) "   "  nom-train  " passe en 0 ")
              (temps-trajet 3 train)(activer reprendre) )
        ((20) (display-alln (heure) "  " nom-train  "
                           arrive en gare G2 5min d'arret ")
              (arret 5 train)(activer reprendre)
              (display-alln (heure) "   " nom-train  " quitte la gare G2 ")
              (temps-trajet 4 train) (activer reprendre))
        ((40) (display-all (heure) "   "  nom-train  " arrive au feu  ")
              (bloque *feu* train)(activer reprendre)
              (temps-trajet 1 train) (activer reprendre))
        ((45) (display-alln (heure) "   " nom-train  "
                            arrive en gare G 8min d'arret ")
              (arret 8 train)(activer reprendre)
              (display-alln (heure) "   " nom-train  " quitte la gare G ")
              (debloque *feu* reprendre)
              (set! position 0)) ))))

(define (trajet3 train reprendre)
  (let ((position -1)
        (nom-train  (train 'nom)))
    (do ()
        (#f)
      (set! position (+ position 1))
      (case position
        ((0) (display-alln (heure) "   "  nom-train  " passe en 0 ")
             (temps-trajet 1 train)(activer reprendre) )
        ((8) (display-alln (heure) "   " nom-train  "
                          arrive en gare G3 5min d'arret ")
```

```
            (arret 5 train)(activer reprendre)
            (display-alln (heure) "   " nom-train  " quitte la  gare G3 ")
            (temps-trajet 1 train) (activer reprendre))
       ((20) (display-all (heure) "   "  nom-train  " arrive au feu  ")
             (bloque *feu* train)(activer reprendre)
             (temps-trajet 1 train) (activer reprendre))
       ((25) (display-alln (heure) "  " nom-train  "
                          arrive en gare G 10min d'arret ")
             (arret 10 train)(activer reprendre)
             (display-alln (heure) "   " nom-train  " quitte la  gare G ")
             (debloque *feu* reprendre)
             (set! position 0)) ))))
```

L'ordre d'arrêt du train pendant une certaine durée a pour effet de fixer l'heure de son redémarrage :

```
(define (arret duree train)
  (train 'set-heure-depart  (+ (heure)  duree)))
```

La durée d'un trajet fixe l'heure d'arrivée à la prochaine étape :

```
(define (temps-trajet duree train)
  (train 'set-heure-arrivee (+ (heure) duree)))
```

Gestion du feu de signalisation

Les interactions entre le feu et le train qui se présente au feu sont :
- si le feu est rouge, on arrête le train et on le met en file d'attente,
- si le feu est vert, le train passe et le feu devient rouge,

```
(define (bloque feu train)
  (display-alln "     " (feu 'couleur))
  (if (eq? 'rouge (feu 'couleur))
    (begin
      (display-alln "     arret au feu rouge de  " (train 'nom) )
      ((feu 'file) 'ajouter train)
      (train 'set-bloque-au-feu? #t))
    (feu 'set-rouge)))
```

Les interactions entre le feu et la file d'attente des trains sont :
- la fonction debloque libère le premier train en attente dans la file,
- le feu passe au vert si la file devient vide sinon il reste au rouge.

```
(define (debloque feu reprendre)
  (let ((file-feu (feu 'file)))
    (cond ((file-feu 'vide?)
           (feu 'set-vert)(display-alln "     feu repasse au vert"))
          (else (let ((train (file-feu 'supprimer-tete )))
                  (train 'set-bloque-au-feu? #f)
                  (display-alln "    le train " (train 'nom) "
```

```
                                                   quitte le feu ")
              (reprendre (train 'sa-coroutine))
              (when (file-feu 'vide?)
                   (feu 'set-vert)
                   (display-alln "     feu repasse au vert")))))))
```

Pour activer les trains, on les place dans une liste circulaire représentée par la variable globale `*list-circ-train*`.
L'ordre `activer` permet de relancer le premier train de cette liste qui ne soit pas en arrêt dans une gare ou bloqué au feu ou pas encore arrivé à la prochaine étape :

```
(define (activer reprendre)
  (letrec ((loop
             (lambda (liste)
               (let ((train (car liste)))
                 (if (or (train 'bloque-au-feu?)
                         (< (heure) (train 'heure-depart))
                         (< (heure) (train 'heure-arrivee)))
                   (loop (cdr liste))
                   train)))))
    (let ((train (loop *list-circ-train*)))
      (reprendre (train 'sa-coroutine)))))
```

Le feu est représenté par un prototype qui possède une couleur, une file d'attente de trains et qui peut changer de couleur :

```
(define (make-feu)
  (let ((couleur 'vert)
        (file (make-file)))
    (lambda (message)
      (case message
        ((couleur) couleur)
        ((file) file)
        ((set-rouge) (set! couleur 'rouge))
        ((set-vert)  (set! couleur 'vert ))))))
```

Pour la file d'attente on utilise la représentation par prototype donnée au chapitre 7§3

```
(define (make-file)
  (let  ((contenu '()))
    (lambda message
      (case (car message)
        ((vide?) (null? contenu))
        ((tete) (car contenu))
        ((suivant) (cdr contenu))
        ((ajouter)
         (set! contenu (append contenu (list (cadr message)))))
        ((supprimer-tete)
         (begin1 (car contenu) (set! contenu (cdr contenu))))))))
```

Exécution de la simulation

Pour lancer la simulation, on doit créer le feu, les trois trains, les placer dans une liste circulaire, initialiser l'heure et enfin faire démarrer un train (disons le train No 1). Le jeu des coroutines déclenchera ensuite l'appel des autres trains.

```
(define *feu* 'any)
(define time0 'any)
(define train1 'any)
(define train2 'any)
(define train3 'any)
(define *list-circ-train* 'any)


(define (init)
  (set! *feu* (make-feu))
  (set! time0 (runtime))
  (set! train1 (make-train  trajet1 'T1))
  (set! train2 (make-train  trajet2 'T2))
  (set! train3 (make-train  trajet3 'T3))
  (set! *list-circ-train* (list train1 train2 train3))
  (set! *list-circ-train* (append! *list-circ-train* *list-circ-train*))
  ((train1 'sa-coroutine)))
```

Lançons la simulation, il y a affichage dans la colonne de gauche de l'écoulement du temps dans une unité arbitraire.

```
? (init)
0   t1 passe en 0
0   t2 passe en 0
0   t3 passe en 0
2   t3 arrive en gare G3 5min d'arret
2   t1 arrive en gare G1 3min d'arret
3   t2 arrive en gare G2 5min d'arret
5   t1 quitte la  gare G1
7   t3 quitte la  gare G3
8   t3 arrive au feu vert
8   t1 arrive au feu rouge
     arret au feu rouge de  t1
8   t2 quitte la  gare G2
9   t3 arrive en gare G 10min d'arret
12   t2 arrive au feu rouge
     arret au feu rouge de  t2
19   t3 quitte la  gare G
     le train t1 quitte le feu
19   t3 arrive en gare G3 5min d'arret
20   t1 arrive en gare G 5min d'arret
24   t3 quitte la  gare G3
25   t1 quitte la  gare G
     le train t2 quitte le feu
     feu repasse au vert
25   t1 arrive en gare G1 3min d'arret
```

```
25   t3 arrive au feu  vert
26   t3 arrive en gare G 10min d'arret
26   t2 arrive en gare G 8min d'arret
*** INTERRUPT
```

Architecture du programme de simulation

Les variables globales.

`*feu*`
Variable globale pour désigner le feu de signalisation.

`train1`
variable globale pour désigner le train no 1.

`train2`
variable globale pour désigner le train no 2.

`train3`
variable globale pour désigner le train no 3.

`*list-circ-train*`
variable globale pour désigner la liste circulaire composée des trois trains.

Les fonctions.

`(init)`
La fonction principale initialise les variables globales et lance la simulation.

> `(make-feu)`
> Pour créer un prototype simulant le comportement du feu de signalisation.
>
> > `(make-file)`
> > Pour créer un prototype modélisant une
> > File d'attente des trains au feu.
>
> `(make-train trajet nom)`
> Création d'un prototype qui modélise l'évolution d'un train sur son trajet,
> Un des champs est une coroutine qui représente l'avancement du train.
>
> > `(make-coroutine proc)`
> > Création d'une coroutine.
>
> `(trajet1 train reprendre)`
> Fonction qui associe un trajet au train No 1, elle simule une
> succession de parcours de la voie No 1.
>
> `(trajet2 train reprendre)`
> Fonction qui associe un trajet au train No 2, elle simule une
> Succession de parcours de la voie No 2.

`(trajet3 train reprendre)`
Fonction qui associe un trajet au train No 3, elle simule une succession de parcours de la voie No 3.

`(debloque feu reprendre)`
Débloque le premier train de sa file d'attente en relançant sa coroutine.

`(activer reprendre)`
Active le premier train (non arrêté en gare ou bloqué par le feu) de la liste circulaire *list-circ-train* en relançant sa coroutine.

`(bloque feu train)`
On bloque un train au feu.

`(temps-trajet duree train)`
Fixe le champ `heure-arrivee` d'un train après un voyage de durée donnée.

`(heure)`
Simule l'écoulement du temps par un entier lié à la fonction système `runtime`.

`(runtime)`
Fonction système qui donne le temps écoulé depuis le début de la session Scheme.

`(arret duree train)`
La connaissance de la durée d'un arrêt en gare permet de fixer l'heure de départ d'un train.

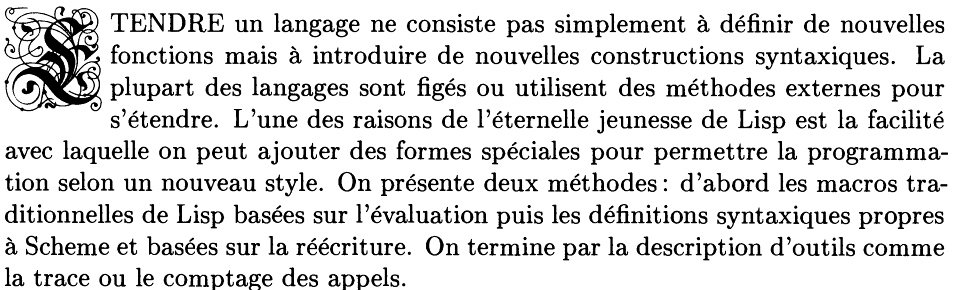
Cette modélisation d'un réseau de trains peut servir de modèle pour la simulation d'un système de gestion de processus par un seul processeur.
Quand certaines parties des codes des processus partagent une variable, il faut veiller à ce qu'ils ne la modifient pas en même temps. On arrive à la notion de section critique de code. En analogie avec le tronçon de voie commune, on protège l'exécution d'une section critique par une extension logicielle de la notion de sémaphore.

8.12 De la lecture

La programmation par continuation est développée dans [FWH92, App92, SJ93, Que94].
On trouvera d'autres présentations et des applications de `call/cc` dans [Dyb87, SF90, SJ93].

Chapitre 9

Macros et extensions syntaxiques



ÉTENDRE un langage ne consiste pas simplement à définir de nouvelles fonctions mais à introduire de nouvelles constructions syntaxiques. La plupart des langages sont figés ou utilisent des méthodes externes pour s'étendre. L'une des raisons de l'éternelle jeunesse de Lisp est la facilité avec laquelle on peut ajouter des formes spéciales pour permettre la programmation selon un nouveau style. On présente deux méthodes : d'abord les macros traditionnelles de Lisp basées sur l'évaluation puis les définitions syntaxiques propres à Scheme et basées sur la réécriture. On termine par la description d'outils comme la trace ou le comptage des appels.

9.1 Introduction à la notion de macro

Pour introduire l'idée de macro, on commence par un exemple.

Une nouvelle forme `ifn`

Quand on utilise la forme `if`, on a parfois envie de mettre en premier la branche correspondant à la partie sinon. Pour cela, il suffit d'utiliser un `if` où l'on permute les places des expressions `e1` et `e2` :

```
 (ifn test  e1  e2)   =   (if  test  e2  e1)
```

La première idée est de définir `ifn` par une définition habituelle :

```
(define  (ifn test e1 e2)
  (if test
    e2
    e1))
```

Essayons cette fonction `ifn`:

```
? (let ((x 5))
    (ifn (= 0 x) (/ 1 x) x))
1/5
```

Parfait ! Encore un essai :

```
? (let ((x 0))
    (ifn (= 0 x) (/ 1 x) x))
*** ERROR -- Division by zero
```

Catastrophe ! L'expression `(/ 1 x)` a été évaluée. C'etait prévisible car, comme toute fonction, `ifn` évalue tous ses arguments.

Exercice 1 *Que pensez-vous de la valeur de (fac 2) quand on écrit la fonction factorielle avec notre ifn :*

```
(define (fac n)
  (ifn (zero? n)
    (* n (fac (- n 1)))
    1))
```

Définition et appel d'une macro

Comme la forme `if`, la forme `ifn` ne doit jamais évaluer les deux branches de l'alternative. Il n'est donc pas possible de définir `ifn` par le mécanisme des définitions de fonctions. Il faut disposer d'un moyen qui transforme l'appel
`(ifn test e1 e2)` en l'appel `(if test e2 e1)`.
Un moyen, introduit dès les premières versions de Lisp, est basé sur le mécanisme des *macros*. En gros, une macro sert à engendrer un code qui sera ensuite évalué. La plupart des implantations de Scheme fournissent une forme pour définir des macros. La syntaxe est analogue à une définition de fonction :

```
(define-macro (nom-macro x1 ... xk)
  corps)
```

Ce qui est nouveau c'est le mécanisme utilisé à *l'appel* d'une macro. Le calcul de la valeur de `(nom-macro e1 ...ek)` se fait en deux temps :

1er temps (appelé macro-expansion) On lie les paramètres formels `x1 ...xk` avec les arguments `e1 ...ek` NON EVALUES, puis on évalue le corps de la macro dans cet environnement. On obtient une expression appelée *expansion*.

2ème temps On évalue l'expansion dans l'environnement courant de l'appel[1].

[1]On décrit ici un fonctionnement des macros qui est souvent disponible, mais ce n'est pas le seul et il ne fait pas partie de la norme Scheme. Le lecteur se reportera à la documentation accompagnant sa version de Scheme.

La phase de macro-expansion a donc pour objet de construire le code à évaluer, on bénéficie ici du fait qu'une *expression Scheme est aussi une valeur.*

Appliquons ce mécanisme à notre exemple initial. On définit une macro `ifn` par :

```
(define-macro  (ifn test e1 e2)
  (list  'if  test  e2  e1))
```

L'évaluation de :

```
(let ((x 0))
  (ifn (= 0 x) (/ 1 x) x))
```

revient à évaluer l'appel `(ifn (= 0 x) (/ 1 x) x))` dans un environnement où `x = 0`. Détaillons les deux phases de ce calcul :

- la phase de macro-expansion consiste à évaluer le corps de la macro `(list 'if test e2 e1))` dans un environnement où :
  - le paramètre `test` a pour valeur la S-expression `(= 0 x)`,
  - le paramètre `e1` a pour valeur la S-expression `(/ 1 x)` ,
  - le paramètre `e2` a pour valeur la S-expression `x`,

  d'où l'expansion `(if (= 0 x) x (/ 1 x))` ;
- le deuxième temps consiste à évaluer cette expression, sachant que `x = 0`. On obtient bien la valeur `0`.

Utilisation des macrocaractères ` , ,@

La difficulté est de trouver l'expression du corps de la macro pour que la phase de macro-expansion construise l'expression désirée. Pour cela, on dispose de diverses aides.
On a vu au chapitre 2 §11 l'utilisation des macrocaractères ` , ,@ pour faciliter la construction d'expressions Scheme. Ils sont surtout utilisés pour écrire le corps des macros car ils permettent d'en donner une forme se rapprochant de l'expansion à construire.
Par exemple, l'expression `(list 'if test e2 e1))` est équivalente à `` `(if ,test ,e2 ,e1) `` mais cette dernière suggère mieux le résultat à obtenir :

```
(define-macro  (ifn test e2 e1)
  `(if  ,test  ,e2  ,e1))
```

Pour s'assurer qu'une expression s'expanse comme on l'espère, les implantations fournissent en général une fonction appelée **macroexpand** qui réalise la phase d'expansion d'un appel :

```
? (macroexpand  '(ifn (= 0 x) (/ 1 x) x)) )
(if (= 0 x) x (/ 1 x))
```

Exercice 2 *Ecrire des macros* `tete` *et* `reste` *qui se comportent comme les fonctions* `car` *et* `cdr`*.*

9.2 Utilisations des macros

On a souvent dit que la programmation des macros était un véritable sport ; comme pour tous les sports on commence par une petite mise en forme. Elle consiste en exemples qui illustrent les utilisations les plus courantes des macros.

Définition d'une fonction par une macro

On peut accéder jusqu'au quatrième élément d'une liste par une combinaison `c...r`, écrivons un moyen pour accéder au cinquième. On peut tout simplement utiliser une fonction :

```
(define (fct-5eme  L)
  (car (cddddr  L)))
```

Mais on peut aussi définir une macro :

```
(define-macro (macro-5eme  L)
  `(car (cddddr ,L)))

? (macro-5eme '(a b c d e f)) -> e
```

Il est intéressant de peser les avantages et les inconvénients de la définition par une macro. Le principal (et ici le seul!) avantage repose sur le comportement des macros utilisées dans une définition de fonction. Par exemple, définissons une fonction `sixieme` à partir de la macro

```
(define (sixieme L)
  (macro-5eme (cdr  L)))
```

On l'essaye ;

```
? (sixieme  '(a b c d e f)) -> f
```

Parfait, modifions maintenant la définition de la macro `macro-5eme` :

```
(define-macro (macro-5eme  L)
  ''toto)
```

On essaye à nouveau :

```
? (sixieme  '(a b c d e f)) -> f
```

Contrairement à ce qui se passe quand on modifie une définition de fonction, la nouvelle définition de la macro n'est pas prise en compte. En effet, la macro a été expansée dans le corps de la fonction `sixieme` et n'est donc plus visible, cette fonction se comporte comme si on avait posé :

```
(define (sixieme L)
  (car (cddddr (cdr L))))
```

On économise donc un appel de fonction par rapport à la définition :

```
(define (sixieme L)
  (fct-5eme (cdr L)))
```

Ce n'est qu'après réévaluation de la première définition de `sixieme` que l'on aura bien :

```
? (sixieme  '(a b c d e f))  -> toto
```

Moralité :
- il faut définir les macros avant les fonctions qui les utilisent[2],
- il faut réévaluer les définitions des fonctions qui utilisent des macros qui ont été modifiées.

Un inconvénient des macros est que ce ne sont pas des fonctions ! Cette lapalissade pour préciser que les fonctionnelles comme `map, for-each, apply, ...` ne peuvent pas prendre une macro en argument :

```
? (map macro-5eme '(a b c))
*** ERROR -- Unbound variable: macro-5eme
```

Comme pour les formes spéciales, on peut souvent contourner la difficulté en enveloppant la macro dans une lambda :

```
? (map (lambda (s)(macro-5eme s)) '( a b c))
(toto toto toto)
```

En général, il n'est pas avantageux d'utiliser les macros pour faire ce qui peut être fait par une fonction. La principale utilisation des macros est ailleurs.

Macros et modifications physiques : `incr!`, `push!`, `pop!`

On avait constaté au chapitre 5 §5 que le mode d'évaluation des fonctions ne permettait pas de définir une fonction pour incrémenter une variable.
On se donne une variable globale numérique `n` :

```
? (define n  5)
```

Pour l'incrémenter, il faut générer le code `(set!  n (+ 1 n))` d'où la macro :

```
(define-macro (incr! nb)
  `(set! ,nb (+ ,nb 1)))

? (incr! n)

? n -> 6
```

Exercice 3 *Ecrire une macro* `decr!` *pour décrémenter.*

[2]On préconise de regrouper les définitions des macros en début de fichier.

Pour définir des macros à un nombre arbitraire de paramètres, la lambda liste d'une macro accepte aussi la forme avec un doublet.
Par exemple, généralisons la macro `incr!` en permettant un argument supplémentaire facultatif qui précise la valeur de l'incrément.

```
(define-macro (incr! symb . Lincrement)
  (if (null? Lincrement)
    `(set! ,symb (+ ,symb 1 ))
    `(set! ,symb (+ ,symb ,(car Lincrement)))))
```

Détaillons le calcul de `(incr!  n 10)`. La phase de macro-expansion consiste à évaluer :

```
(if (null? Lincrement)
  `(set! ,symb (+ ,symb 1 ))
  `(set! ,symb (+ ,symb ,(car Lincrement)))
```

où `Lincrement` a pour valeur la liste `(10)` et `symb` a pour valeur n. L'expansion est donc `(set!  n (+ n 10))`. Son évaluation réalise bien l'incrémentation de la variable n.

De façon analogue, on peut écrire une macro pour modifier une variable à valeur liste en lui ajoutant en tête la valeur d'une expression. Cette macro s'appelle traditionnellement `push!` :

```
(define-macro (push! s L)
  `(set! ,L (cons ,s ,L)))
```

et sa cousine `pop!` pour enlever l'élément de tête d'une liste

```
(define-macro (pop!  L)
  `(set! ,L (cdr ,L)))
```

Ce type de macro nous permet de proposer une implantation très naturelle de la structure de données `piles mutables`. On représente la pile vide par la liste vide :

```
(define   pile!:Vide   '())

(define  pile!:Vide?  null?)
```

L'empilement est réalisé sur le modèle de `push!` mais, afin de rendre le nouvel état de la pile, on l'enveloppe dans un `begin` :

```
(define-macro (pile!:empiler s p)
  `(begin
     (push! ,s ,p)
     ,p))
```

Le dépilement est basé sur le modèle de `pop!` mais est un petit peu plus délicat. En effet, on doit s'assurer que la pile est non vide et, dans l'affirmative, il faut sauvegarder le sommet pour le rendre en résultat.

```
(define-macro (pile!:depiler  p)
  '(if (pile!:vide? ,p)
     (*erreur* " pile vide")
     (let ((sommet (car ,p)))
       (begin (pop! ,p)
              sommet))))

? (define pile  pile!:Vide)

? (pile!:empiler 'a pile)
(a)

? (pile!:depiler  pile)
a

? (pile!:depiler  pile)
ERREUR --  pile vide
```

Exercice 4 *Définir sur le même principe une implantation des files mutables.*

Après ces échauffements, on arrive à l'utilisation principale des macros.

9.3 Définitions de nouvelles formes spéciales

Le mécanisme des macros permet de définir de nouvelles formes spéciales. On a déjà vu la forme `ifn`, voyons d'autres exemples simples.

Macro `when`

On a souvent besoin d'un `if` ayant une seule alternative mais pouvant être composée d'une suite de S-expressions. On l'appelle la forme `when`, sa syntaxe est :

```
(when test s1 ... sN)
```

Si le test est vrai, on évalue en séquence les expressions `s1 ...sN` et sinon on retourne `#f`. Voici son équivalent à l'aide d'un `if` :

```
(if test (begin s1 ... sN) #f)
```

D'où la macro :

```
(define-macro (when test  s1 . Larg)
  '(if ,test
     (begin ,s1 ,@Larg)
     #f))
```

Noter l'usage de la liste `Larg` pour les arguments facultatifs ; son contenu est inséré dans le `begin` en utilisant la technique des macrocaractères `'` `,` `@`.
Sa sœur s'appelle `unless`, elle correspond à un `when not` et s'écrit donc :

```
(define-macro (unless test . s1 Larg)
  '(if (not ,test)
     (begin ,s1 ,@Larg)
     #f))
```

Macros dotimes et dolist

Pour répéter un nombre connu de fois une action, il est commode de disposer d'une forme spéciale `dotimes` de syntaxe `(dotimes var exp corps)`. On exécute le `corps` un nombre de fois donné par la valeur initiale de l'expression à valeur entière `exp`. La variable `var` est initialiée à `0` et est incrémentée de `1` à chaque tour.
On peut réaliser cette boucle avec la forme `do` :

```
(do  ((var 0 (+ var 1)))
     ((= var exp))
  corps)
```

D'où la macro qui génère cette expression :

```
(define-macro (dotimes  var exp . Lcorps)
  '(do  ((,var  0 (+ ,var 1)))
        ((= ,var ,exp))
      ,@Lcorps))

? (dotimes (i (+ 2 3))
    (write i))
01234
```

Mais ce n'est pas tout à fait correct : dans le cas où l'évaluation du corps modifie la valeur de la borne `exp`, le résultat sera en général erroné. Pour y remédier, on calcule la valeur de la borne avant d'entrer dans la boucle `do` :

```
(define-macro (dotimes i exp  . Lcorps)
  '(let ((n ,exp))
     (do ((,i 0 (+ ,i 1)))
         ((= ,i n))
       ,@Lcorps)))
```

On a parfois besoin de répéter une action pour toutes les valeurs d'une variable qui parcourt une liste. Dans ce but on introduit la forme `dolist`[3], sa syntaxe est `(dolist x L corps)`. La variable `x` parcourt la liste `L` et pour chaque valeur on évalue le `corps` ; comme pour `dotimes` il n'y pas de valeur rendue, seul l'effet de bord importe.
On réalise cette boucle avec une fonction locale `loop` :

```
(define-macro (dolist i L  . Lcorps)
  '(letrec ((loop
             (lambda (N)
```

[3] Les formes **dotimes** et **dolist** sont prédéfinies en Common Lisp mais avec une syntaxe légèrement différente.

```
              (if (not (null? N))
                (let ((,i (car N)))
                  ,@Lcorps
                  (loop (cdr N)))))))
      (loop ,L)))
```

Voici une application de cette macro. On se donne deux listes L1, L2 d'entiers, et on cherche tous les couples de variables x1 dans L1 et x2 dans L2 tels que le produit x1 * x2 soit un entier donné. Pour cela, il suffit d'essayer tous les couples de telles variables et d'afficher les couples qui satisfont à la condition :

```
(define (les-solutions L1 L2 n)
  (dolist x1 L1
    (dolist x2 L2
            (if (= n (* x1 x2))
              (display-alln x1 " * " x2  " = " n)))))

? (les-solutions '(3 7 5 1 9 2) '(20 5 4 8 13 10)   20)
5 * 4 = 20
1 * 20 = 20
2 * 10 = 20
```

Il est facile de modifier cette méthode pour trouver une première solution et ne donner les autres que sur demande. Il suffit de capturer la continuation au moment de l'appel pour la rappeler quand l'utilisateur ne désire plus de solution :

```
(define (solution-sur-demande L1 L2 n)
  (call/cc
    (lambda (stop)
      (dolist x1 L1
        (dolist x2 L2
           (if (= n (* x1 x2))
             (begin
              (newline)
              (display-all x1 " * " x2  " = " n "  autre solution? o/n: ")
              (if (eq? 'n (read)) (stop #t)))))))))

? (solution-sur-demande '(3 7 5 1 9 2) '(20 5 4 8 13 10)   20)
5 * 4 = 20  autre solution? o/n: o

1 * 20 = 20  autre solution? o/n: n
#t
```

Remarque 1 On verra à la lumière du §4 que les macros `pile!:depiler`, `dotimes` et `dolist` ne sont pas irréprochables (cf exercice 6).

Exercice 5 *1. Ecrire une macro* `if*`, *de syntaxe d'appel*
`(if* test e1 e2 ...)`, *qui s'expanse en la forme*
`(if test e1 (begin e2 ...))`.

2. *Ecrire une macro* **selon-signe** *de syntaxe d'appel:*

   (selon-signe exp-nb si-positive si-nulle si-negative).

   Sa valeur dépend du signe de l'expression à valeur numérique ***exp-nb****. Selon que cette expression est positive, nulle ou négative, on rend respectivement la valeur de l'expression* ***si-positive, si-nulle*** *ou* ***si-negative.***

Définition avec documentation : defun

Quand on souhaite réaliser un nouveau langage, on écrit en général un interprète ou un compilateur dans un langage existant (voir chapitres 21 et 22). Mais Scheme permet une troisième approche : on réalise une sur-couche pour intégrer les spécificités du nouveau langage tout en continuant à bénéficier des fonctionnalités de Scheme. Evidemment, les macros sont le moyen idéal pour faire cette sur-couche. Afin d'illustrer cette approche, on ne va pas définir un nouveau langage mais se contenter d'une variation sur le processus de définition d'une fonction. On trouvera, dans le même esprit, une extension objet de Scheme au chapitre 11 §3 et l'introduction des listes infinies au chapitre 10 §5.
On se propose d'intégrer la méthode utilisée en Common Lisp pour documenter une fonction. Quand on documente une fonction avec des commentaires, cette information reste sur le papier car l'évaluateur ne la voit pas. En Common Lisp, si l'on place une chaîne en tête du corps d'une fonction, cette chaîne est considérée comme étant un texte de documentation associé à la fonction.
La définition d'une fonction en Common Lisp se fait par la forme `defun`.
Par exemple, on définit une fonction `last` par (on utilise la syntaxe de Common Lisp pour `defun`)

```
(defun  last (L)
" retourne le dernier cdr de la liste L ou () si elle est vide"
  (if (or (null? L)(null? (cdr L)))
    L
    (last (cdr L))))
```

Après évaluation de cette définition, on peut accéder à la documentation par l'appel de la fonction Common Lisp `documentation` (ici, on simplifie un peu la syntaxe de Common Lisp) :

```
 ? (documentation 'last)
"retourne le dernier cdr de la liste L ou () si elle est vide"
```

Notons que la chaîne de documentation est facultative. Pour ajouter cette possibilité dans Scheme, on définit une macro `defun` qui réalise ce comportement. On stockera les documentations des fonctions dans une a-liste globale `*documentations*` de doublets de la forme `(nom-fct . chaine-de-doc)`. La fonction `documentation` consulte cette a-liste, elle peut être définie par :

```
(define (documentation nom-fct)
  (let ((doublet (assq nom-fct *documentations*)))
    (if doublet
      (cdr doublet)
      (*erreur* "pas de documentation pour la fonction : " nom-fct))))
```

La macro `defun` analyse la première expression du corps de la fonction ; si c'est une chaîne on l'ajoute à la a-liste `*documentations*` et l'on génère la définition habituelle via `define`, et sinon on se contente de générer la définition habituelle :

```
(define-macro (defun nom Lparam . Lcorps)
  `(begin
     (if (string? ',(car Lcorps))
       (push! (cons ',nom ,(car Lcorps))  *documentations*))
     (define ,nom (lambda ,Lparam
       ,@Lcorps))))
```

Après évaluation de la définition précédente de `last` avec `defun`, on a :

```
? (documentation 'last)
" retourne le dernier cdr de la liste L ou () si elle est vide"

? (last '(a b c)) -> (c)
```

Mais si on ne fournit pas de documentation, on a le comportement habituel

```
(defun dernier-element (L)
;; retourne le dernier élément de la liste L ou  () si elle est vide
  (cond ((null? L) '())
        ((null? (cdr L))(car L))
        (else (dernier-element (cdr L)))))

? (documentation 'dernier-element)
pas de documentation pour la fonction : dernier-element
```

Si l'on souhaite ne pas avoir à quoter le nom de la fonction dont on demande la documentation, il suffit d'écrire une macro qui génère l'appel de documentation avec le nom quoté. On appelle cette macro `documentationq`, le q pour signaler que la quotation est déjà faite. C'est un bon exemple de l'utilisation d'une macro pour simplifier l'interface utilisateur.

```
(define-macro (documentationq nom-fct)
  `(documentation  ',nom-fct))

? (documentationq last)
" retourne le dernier cdr de la liste L ou () si elle est vide"
```

9.4 Problèmes de capture avec les macros

Les macros semblent un outil merveilleux permettant de modeler à sa guise un langage. C'est vrai, mais il y a des précautions à prendre. Des erreurs sournoises peuvent être provoquées par le phénomène dit de la *capture d'une variable.* Commençons par une macro en apparence très simple. Ecrivons une macro pour permuter les valeurs de deux variables globales.

Permutation de variables

Pour permuter les valeurs de deux variables globales `x` et `y`, on écrit naturellement :

```
(let ((tmp x))
  (set! x y)
  (set! y tmp))
```

où la variable `tmp` sert de sauvegarde temporaire pour `x`. D'où la définition d'une macro pour permuter deux variables quelconques :

```
(define-macro (permuter! x y)
  `(let ((tmp ,x))
     (set! ,x ,y)
     (set! ,y tmp)))
```

Pour la tester, on se donne deux variables globales `a` et `b` par

```
(define a "a")
(define b "b")
```

L'appel de `(permuter!  a b)` s'expanse en :

```
(let ((tmp a))
  (set! a b)
  (set! b tmp))
```

Après évaluation de

```
? (permuter! a b)
```

on a bien la permutation désirée

```
? a -> "b"
? b -> "a"
```

Supposons maintenant que l'on permute la variable `a` avec une variable globale qui a le mauvais goût de s'appeler `tmp` :

```
(define tmp "tmp")
```

Alors, après

```
? (permuter! a tmp)
```

on a

```
? a -> "b"
? tmp  -> "tmp"
```

Il n'y a pas eu de permutation ! Explication : la macro-expansion de l'appel `(permuter!  a tmp)` a pour valeur :

```
(let ((tmp a))
  (set! a   tmp)
  (set! tmp tmp))
```

L'évaluation de ce `let` commence par donner à la variable *locale* `tmp` la valeur de `a` (c.-à-d. actuellement `"b"`), puis on affecte cette valeur à `a` ce qui laisse `a` inchangée. Ensuite, on affecte à la variable *locale* `tmp` la valeur `"b"` de `tmp`. Cela n'a aucune incidence sur la variable *globale* `tmp` qui est masquée par la variable locale de même nom. On dit que le nom de la variable globale `tmp` a été *capturé* par celui de la variable locale. Bien entendu, il est facile de remédier à ce problème : comme le nom de la variable locale est indifférent, il suffit donc d'en changer pour faire disparaître ce phénomène. En appelant `tmp1` la variable locale, tout rentre dans l'ordre :

```
(let ((tmp1 a))
  (set! a   tmp)
  (set! tmp tmp1))
```

Renommage

Il reste cependant un problème, on ne sait pas à l'avance le nom des variables globales que l'on aura à permuter, aussi ne dispose-t-on pas d'une méthode générale pour empêcher cette capture de se reproduire. On pourrait utiliser pour la variable locale `tmp` un nom bizarre de façon à limiter les chances d'être aussi celui d'une autre variable. Cela peut suffire en pratique, mais cela ne donne pas de garantie absolue.

Pour être tranquille, il faudrait que la variable locale ait un nom dont le système nous garantisse qu'il n'a pas été utilisé au cours de la session. C'est pour cela que certaines implantations de Scheme fournissent une fonction, souvent appelée `gensym`, qui génère un symbole *distinct de tout* symbole pouvant être introduit par l'utilisateur. On *supposera* dans toute la suite disposer d'une telle fonction.

Revenons à la définition de `permuter!`, on commence par générer avec `gensym` le nom de la variable locale à utiliser

```
(define-macro (permuter! x y)
  (let ((tmp (gensym)))
    `(let ((,tmp ,x))
       (set! ,x ,y)
       (set! ,y ,tmp))))
```

Si la valeur de `(gensym)` est, par exemple, le symbole `g5` alors l'appel `(permuter!  a b)` s'expanse en :

```
(let ((tmp  g5))
  (let ((g5 a))
    (set! a b)
    (set! b g5)))
```

Cette fois, on est assuré qu'à chaque appel de `permuter!`, le nom généré pour la variable locale sera différent des noms des arguments.

On peut penser que ce phénomène de capture ne peut se rencontrer que pour des définitions de macros très spéciales. Malheureusement non ! Car il est potentiellement présent dans la plupart des définitions de macros. Aussi, dès que l'on doit introduire un nom auxiliaire dans une définition de macro, on fera appel à **gensym**.

Définition de la boucle `while`

Voici un autre exemple, très naturel, de macro pouvant donner lieu à une capture. Il s'agit d'ajouter à Scheme la forme spéciale **while** chère aux programmeurs Pascal. La syntaxe de son appel est :

```
(while   test   s1 ...)
```

Elle consiste à ne rien faire si le test est faux et à évaluer les expressions `s1 ...` s'il est vrai puis à recommencer. La valeur rendue est indéfinie. On réalise facilement une telle boucle avec un `letrec` :

```
(letrec ((loop (lambda ()
                 (if test
                   (begin s1 ... (loop))))))
  (loop))))
```

D'où une macro qui s'expanse en ce code :

```
(define-macro (while test . Lcorps)
  '(letrec ((loop (lambda ()
                    (if ,test
                      (begin ,@Lcorps (loop))))))
     (loop)))
```

On teste :

```
? (let ((x 4))
    (while (< 0 x)
           (display x)(set! x (- x 1))))
4321
```

Parfait ! Et si la variable **x** s'appelait `loop` ! :

```
? (let ((loop 4))
    (while (< 0 loop)
           (display loop)(set! loop (- loop 1))))
*** ERROR -- REAL expected
```

La fonction locale `loop` a caché la variable numérique `loop`. On connaît la parade, il faut générer le nom de la boucle locale à l'aide de **gensym** :

```
(define-macro (while test . Lcorps)
  (let ((loop (gensym)))
    '(letrec ((,loop (lambda ()
                       (if ,test
                         (begin ,@Lcorps (,loop))))))
       (,loop))))
```

On essaye à nouveau :

```
? (let ((loop 4))  ; capture
    (while (< 0 loop)
           (display loop)(set! loop (- loop 1))))
4321
```

plus de capture.

Exercice 6 *1. Les définitions des macros* `dotimes` *et* `dolist` *utilisent une variable locale* `n` *qui peut être la source d'une capture. Aussi, on demande de modifier ces macros en commençant par générer le nom de cette variable locale avec* `gensym`*.*

2. Définir une macro `repeat` *dont la syntaxe d'appel est*

`(repeat test s1 ...)` et dont la sémantique est analogue à `while` si ce n'est que l'on évalue d'abord les expressions `s1 ...` et si le test est faux on recommence.

3. Pour compléter notre panoplie de boucles Pascal, écrire une macro `for` de syntaxe d'appel `(for var bas haut s1 ...)` où `var` est un symbole, `bas` et `haut` sont des expressions à valeurs entières. Si la valeur de `bas` est inférieure à celle de `haut`, on fait varier la variable entière var de `bas` à `haut` et à chaque fois on évalue le corps `s1 ...`

9.5 Définitions récursives de macros

Peut-on définir une macro qui se rappelle dans son corps? Oui, à condition de s'assurer de l'absence de bouclage dans la phase de macro-expansion et dans la phase d'évaluation de l'expansion.
Redéfinissons la forme spéciale `and` à partir du `if` en se basant sur les égalités

```
(and)              = #t
(and s1)           = s1
(and  s1 s2 ... ) = (if  s1  (and s2 ...sN)  #f)
```

qui conduisent immédiatement à la définition d'une macro avec appel récursif :

```
(define-macro (and . Lexp)
  (if (null? Lexp)
    #t
    (if (null? (cdr Lexp))
      (car Lexp)
      `(if ,(car Lexp)
         (and ,@(cdr Lexp))
         #f))))
```

L'absence de bouclage est garantie par l'appel sur une liste de longueur plus petite.
La forme `or` se traite de manière analogue, on peut la définir récursivement à partir du `if` :

```
(or)              = #f
(or s1)           = s1
(or s1 s2  ... sN) = (let ((tmp s1))
                       (if tmp
                         tmp
                         (or  s2 ...)
```

Notons qu'il est incorrect d'écrire :

```
(or s1 s2  ... sN) =  (if s1
                        s1
                        (or  s2 ...)
```

car `s1` pourrait être évalué deux fois. Pour l'éviter, on a sauvegardé sa valeur dans la variable `tmp`. L'introduction de la variable locale `tmp` oblige à lui donner un nom par `gensym`, d'où finalement la définition de la macro `or` :

```
(define-macro (or . Lexp)
  (if (null? Lexp)
    #f
    (let ((tmp (gensym))
      `(let ((,tmp ,(car Lexp)))
         (if ,tmp
           ,tmp
           (or ,@(cdr Lexp)))))))
```

Exercice 7 *Donner des règles pour définir la forme* **cond** *à partir du* ***if*** *puis écrire une macro* **cond***.*

On voit que certaines formes spéciales de Scheme peuvent être réalisées par des macros. On va développer ce point de vue au paragraphe suivant.

9.6 Formes spéciales primitives et dérivées

On peut essayer de ramener à un minimum le nombre des formes spéciales de Scheme et de réaliser les autres à l'aide de macros. En fait, il est possible de se contenter des formes : `if, lambda, define, set!` et `quote` pour être en mesure de définir, en tant que macros, les autres formes : `begin, let, let*, letrec, cond, case, do` et `delay`[4]. D'ailleurs la terminologie Scheme n'emploie pas le terme de forme spéciale mais utilise à la place le terme *syntaxe*. A titre d'exemple, faisons-le pour les premières. Si on le souhaite, on peut préfixer leurs noms par mon : `mon-begin , mon-let ,...` pour ne pas écraser les formes prédéfinies.

Macro `begin`

Pour représenter la forme `(begin s1 s2 ...)` on peut utiliser une `lambda` sans paramètre. On utilise les égalités

[4]Cette forme sera étudiée au chapitre 10 §4.

```
(begin s1)        = ((lambda () s1))

(begin s1 s2 ...) = ((lambda (x) (begin s2 ...)) s1)
```

pour écrire la macro `mon-begin` :

```
(define-macro (mon-begin . Lexp)
  (if (null? (cdr  Lexp))
    (car Lexp)
    (let ((x (gensym)))
      `((lambda (,x) (mon-begin ,@(cdr lexp))) ,(car Lexp)))))
```

Exercice 8 *Définir une macro* ***begin1*** *qui évalue en séquence les expressions* *s1 s2 ...* *et retourne la valeur de la première. Sa syntaxe d'appel est* *(begin1 s1 s2 ...),*

Macro let

On a vu au chapitre 3 §1 que la forme `let` se ramène à un appel de lambda expression :

```
(let ((x1 e1)...(xk ek)) s1 s2 ...)
=
((lambda (x1...xk) s1 s2 ...)  e1...ek)
```

Notons que, s'il n'y a pas de liaison, on est ramené à la forme `begin`.
Pour construire cette lambda expression, il faut extraire les listes `(x1 ...xk)` et `(e1 ...ek)` de la liste des liaisons `((x1 e1) ...(xk ek))`. On a évidemment :

```
(x1 ...xk) = (map car liaisons)
```
et
```
(e1 ...ek) = (map cadr liaisons)
```

D'où la macro `let` :

```
(define-macro (let liaisons . corps)
  `((lambda ,(map car liaisons) ,@corps) ,@(map cadr liaisons)))
```

Macro let*

La macro `let*` est un bel exemple de macro récursive, on peut la spécifier par les deux égalités :

```
(let* () s1 s2...) = (let () s1 s2 ...)

(let* ((x1 e1))(x2 e2)...) s1 s2 ...) = (let ((x1 e1)) (let* ((x2 e2)...) s1 s2...)
```

D'où la macro `let*` :

```
(define-macro (let* liaisons . Lcorps)
  (if (null?  liaisons)
    `(let () ,@Lcorps)
    `(let (,(car liaisons)) (let* ,(cdr liaisons) ,@Lcorps))))
```

On l'essaye :

```
? (let* ((x 2)(y (* x 3)) ) (write x) (write y)(+ x y)) -> 268
```

Macro letrec

On a expliqué au chapitre 5 §4 comment la forme letrec

```
 (letrec  ((f1 e1)
                ...
            (fN eN))
   s1 ...)
```

peut s'exprimer à partir de let et set! en utilisant l'expression

```
(let ((f1 'any)
       ...
      (fN 'any))
   (let ((f1-aux  e1)
          ...
         (fN-aux  eN))
      (set! f1 f1-aux)
       ...
      (set! fN fN-aux)
      s1 ...))
```

Le deuxième let sert simplement à imposer une évaluation en parallèle des expressions ei. Mais, en pratique, les ei sont des lambda expressions et de plus on ne doit pas faire appel à la valeur d'une fj lors de l'évaluation des ei. Dans ces conditions, on peut se dispenser du let intermédiaire[5] et écrire plus simplement :

```
(let ((f1 '?)
      ...
      (fN '?))
  (set! f1 e1)
  ...
  (set! fN eN)
  s1 ...)
```

Pour générer le premier let, on utilise l'expression :

```
`(let ,(map (lambda (var)(list var ''?))  Lvar)
```
où Lvar = (f1 ... fN).

Pour générer la liste d'affectations, on utilise l'expression :

```
(map (lambda (var exp)(list 'set!  var exp)) Lvar Lexp)
```
où Lexp = (e1 ... eN).

D'où finalement, la macro letrec :

[5] Il aurait aussi fallu générer de nouveaux symboles pour les fi-aux.

```
(define-macro (letrec liaisons  . Lcorps)
  (let ((Lvar (map car liaisons))
        (Lexp (map cadr liaisons)))
    `(let ,(map (lambda (var)(list  var ''?)) Lvar)
       ,@(map (lambda (var exp)(list 'set! var exp)) Lvar Lexp)
       ,@Lcorps)))
```

Un essai :

```
? (letrec ((pair?   (lambda (n)
                      (if (zero? n) #t (impair? (- n 1)))))
           (impair? (lambda (m)
                      (if (zero? m) #f (pair?   (- m 1)))))
           (g (lambda (x) (+ x 1))))
    (impair?  (g 18)))
 #t
```

Exercice 9 *Donner une autre définition de* `letrec` *avec une macro récursive.*

Exercice 10 *On a indiqué au chapitre 4 §2 comment exprimer une boucle* **do** *à l'aide d'un* `letrec`*, en déduire l'écriture d'une macro* **do**.

Exercice 11 *On propose de définir une macro* `fluid-let`*. C'est une espèce de* `let`*, sans création de variable locale mais avec modification temporaire de variables visibles dans l'environnement de son appel. La syntaxe est celle du* `let` *:*

```
(fluid-let ((x1 e1)
            ...
            (xN eN))
  s1 s2 ...)
```

La sémantique consiste à sauver les valeurs des variables **xi**, *de leur affecter les valeurs des* **ei**, *puis de calculer le corps dans cet environnement et enfin de restaurer les valeurs des xi et de retourner la valeur du corps.*

```
(let ((tmp1 x1)
 ...
      (tmpN xN))
 (set! x1 e1)
  ...
 (set! xN eN)
 (let ((resultat (begin s1 s2 ...)))
   (set! x1 tmp1)
    ...
   (set! xN tmpN)
   resultat))
```

C'est souvent utilisé pour modifier temporairement une variable dans une fermeture, par exemple :

```
? (let* ((a 10)
         (f (lambda (u)(+ u a))))
    (fluid-let ((a 0))
      (list a (f 0))))
(0 0)
```

9.7 Liaisons destructurantes

On considère une application un peu plus complexe des macros ; elle servira de transition avec l'autre méthode de définition de macro.
Quand on veut représenter une fonction qui retourne plusieurs valeurs, on utilise parfois une liste de ces valeurs puis on accède ensuite à chaque élément de cette liste par un `let`. Par exemple, si une fonction `f` retourne une liste de deux valeurs, on est obligé d'écrire :

```
(let  ((L (f...))
  (let ((x1 (car L))
        (x2 (cadr L)))
    s1 s2 ...)))
```

ce qui est assez lourd. On aimerait pouvoir lier directement les variables locales `x1` et `x2` avec les deux éléments de cette liste. Si l'on se rappelle que le `let` est une forme dérivée d'un appel de lambda expression, on trouve immédiatement la solution. Il suffit d'écrire :

```
(apply (lambda (x1 x2) s1 s2 ...)  (f ...) )
```

pour obtenir le même résultat, ce qui beaucoup plus simple.

La forme `bind`

Ceci nous conduit à introduire une nouvelle forme que l'on baptise `bind`, l'appel est de la forme :

```
 (bind  (x1 ...)  exp  s1 s2 ...)
```

Il doit s'expanser en :

```
(apply  (lambda (x1 ...) s1 s2 ...)  exp).
```

On dit que la liste `(x1 ...xN)` *filtre* la valeur de `exp` si cette valeur est de la forme `(v1 ...vN)`. Alors, on lie `x1` avec `v1`,..., `xN` avec `vN` et l'on évalue `s1 s2 ...` dans cet environnement. Par exemple, la liste `(4 (a b) (+ 1 3))` est filtrée par la liste `(x1 x2 x3)` en donnant aux `xi` les valeurs `x1 = 4, x2 = (a b) et x3 = (+ 1 3)`.

On veut pouvoir aussi utiliser cette forme quand la valeur de `exp` n'est pas une liste ; on ajoute une autre règle pour traiter ce cas :

```
(bind  x  exp  s1 s2 ...) = ((lambda (x) s1 s2 ...)  exp)
```

On devra avoir les résultats suivants :

```
? (bind (x y z) '(1 3 5) (+ x y z)) -> 9
? (bind x (+ 4 6) (list x  1))      -> (10 1)
? (bind (x y) '(1) (+ x  y))        -> ** ERROR -- Wrong number of arguments
```

Voici la définition de cette macro, on doit vérifier si l'on filtre par une liste ou non ; pour ne pas préjuger on dira que l'on filtre par un motif.

```
(define-macro (bind motif  exp . Lcorps)
  (if (list? motif)
    `(apply (lambda ,motif ,@Lcorps) ,exp)
    `((lambda (,motif) ,@Lcorps) ,exp)))
```

Exercice 12 *On s'est contenté d'un filtrage de la liste au premier niveau, on pourrait envisager un filtrage à tous les niveaux mais nous ne le ferons pas. Cela conduirait à compliquer considérablement cette macro pour traiter des situations beaucoup moins fréquentes.*

La forme `bind-let`

Pour filtrer simultanément plusieurs expressions, on généralise la forme `let` en permettant des liaisons par `bind`. Appelons `bind-let` cette généralisation, sa syntaxe reprend celle du `let` :

```
(bind-let ((motif1 e1)
           ...
           (motifk ek))
  s1 s2 ...)
```

On effectue en parallèle les liaisons destructurantes des motifs avec les valeurs des expressions `ei`, puis on évalue le corps `s1 s2 ...` dans cet environnement. Voici l'expansion qui définit cette forme :

```
(let ((tmp1 e1)
      ...
      (tmpk ek))
  (bind motif1 tmp1
    (bind motif2 tmp2
             ...
             (bind motifk tmpk  s1 s2 ...))))
```

Le `let` sert à évaluer en parallèle les expressions `ei`, on peut ensuite procéder séquentiellement aux liaisons destructurantes. On remplace le `let` par un appel de lambda expression :

```
 ((lambda (tmp1 ... tmpk) binding ) e1 ... ek)
```

où l'on a posé :

```
binding = (bind m1 tmp1
           (bind m2 tmp2
                 ...
                 (bind mk tmpk  s1 s2 ...)))
```

La syntaxe de l'appel de `bind-let` est `(bind-let liaisons s1 s2 ...)`. Il s'agit de générer un appel de lambda. Comme pour le `let`, on calcule la liste des motifs par `(map car liaisons)` et la liste des expressions par `(map cadr liaisons)`. Pour éviter les captures, on génère une liste de k nouveaux symboles par `(map (lambda (x) (gensym)) liaisons)`, d'où la macro

```
(define-macro (bind-let liaisons  .  Lcorps)
  (let ((Lmotif (map car liaisons))
        (Lexp   (map cadr liaisons))
        (Ltmp   (map (lambda (x)(gensym)) liaisons)))
    (let ((binding (generer-binding  Lmotif Ltmp Lcorps)))
      `((lambda ,Ltmp ,binding) ,@Lexp))))
```

Il reste à préciser la génération de la suite d'appels `binding`. S'il n'y a pas de liaison c'est une forme `begin`, sinon on construit le premier `bind` et l'on fait un appel récursif pour générer les autres :

```
(define (generer-binding Lmotif Lvar Lcorps)
  (if (null?  Lmotif)
    `(begin ,@Lcorps)
    `(bind ,(car Lmotif) ,(car Lvar)
       ,(generer-binding  (cdr Lmotif) (cdr Lvar) Lcorps))))
```

La forme `bind-let` permet de mélanger des liaisons ordinaires et des liaisons destructurantes :

```
? (bind-let (((x y z) '(a b c))
             (u (+ 1 2))
             ((v  w) '(4 5)))
    (list x y z u v w))
(a b c 3 4 5)
```

La forme bind-let*

Comme pour le `let` en voici une version séquentielle :

```
(bind-let* ((motif1 e1)
            ...
            (motifk ek))
  s1 s2 ...)
```

définie par l'expansion :

```
(bind motif1 e1
  (bind motif2 e2
     ...
     (bind motifk ek  s1 s2 ...)))
```

C'est une macro récursive beaucoup plus facile à écrire que `bind-let` : on génère le premier `bind` et on la rappelle récursivement.

```
(define-macro (bind-let* liaisons  . Lcorps)
  (if (null? liaisons)
    '(begin ,@Lcorps)
    '(bind ,(caar liaisons) ,(cadar liaisons)
       (bind-let* ,(cdr liaisons) ,@Lcorps))))

? (bind-let* ( ((x y z) '(1 2 3))
               (u (+ 1 2 x))
               ((v  w) (list (+ z 2) 6)))
       (list x y z u v w))
(1 2 3 4 5 6)
```

La technique de filtrage introduite ici va être étendue à tous les niveaux pour fournir une autre méthode de définition des macros.

9.8 Extensions syntaxiques définies par filtrage

Le langage Scheme a introduit un moyen plus simple pour écrire des extensions syntaxiques. Au lieu de construire l'expansion par *évaluation* d'une expression, on utilise une méthode de *réécriture* basée sur la forme `define-syntax`.

Expansion par réécriture

Ecrivons la forme `ifn` avec ce nouveau procédé de définition de macro :

```
(define-syntax ifn
  (syntax-rules ()
    ((ifn test e1 e2)  (if test e2 e1)) ))
```

Où `define-syntax` permet de définir une transformation syntaxique de nom `ifn` et qui consiste en la règle de réécriture :

```
(ifn test e1 e2)  -->  (if test e2 e1)
```

Une règle de réécriture est constituée de deux parties : la partie à gauche de la flèche s'appelle le *motif* (*pattern* en anglais) et la partie à droite s'appelle le *modèle* (*template* en anglais)
On dit que le motif *filtre* l'expression `(ifn (= 0 x) (/ 1 x) x))` car cette expression est un cas particulier du motif en prenant :

`test = (= 0 x)`, `e1 = (/ 1 x)`, `e2 = x`.

La réécriture consiste à lui associer le modèle dans lequel on a remplacé les identificateurs `test`, `e1` et `e2` par les expressions précédentes. On trouve l'expansion désirée : `(if (= 0 x) x (/ 1 x))`
La sémantique de l'appel d'une nouvelle forme syntaxique est analogue à celle des macros, à ceci près que la macro-expansion est réalisée par ce mécanisme de réécriture. Par exemple, l'appel `(ifn (= 0 x) (/ 1 x) x))` est réécrit en `(if (= 0 x) x (/ 1 x))` puis cette expression est évaluée.

La forme `Define-syntax`

Il peut y avoir plusieurs règles de réécriture, d'où la forme générale de la définition d'une transformation syntaxique :

```
(define-syntax nom-macro
  (syntax-rules ()
    ((motif1  modele1)
      ...
     (motifk  modelek))))
```

Chaque règle de réécriture est constituée par une liste (*motifj modelej*) formée d'un motif et d'un modèle. On verra plus tard la raison de la liste vide qui suit l'identificateur `syntax-rules`.
Pour transformer une expression, on considère les motifs dans l'ordre *motif1, motif2, ...* et on cherche le premier motif qui la filtre et on réécrit l'expression avec le modèle associé.

9.9 Exemples de définitions par `define-syntax`

Pour comprendre ce mécanisme, le mieux est de refaire dans ce style certaines macros définies aux paragraphes précédents. Auparavant, précisons un peu la structure d'un motif.
Un motif est une liste qui doit commencer par le nom de la forme à définir ; par ailleurs un motif ne doit pas contenir deux fois le même identificateur[6].

Les formes `push!` et `incr!`

Il n'y a qu'une règle, elle indique que l'on doit réécrire l'appel
`(push! s L)` en `(set! L (cons s L))` :

```
(define-syntax push!
  (syntax-rules ()
    ((push! s L)  (set! L (cons s L)))))
```

Plus généralement, voici la forme pour empiler un élément dans une pile mutable :

```
(define-syntax pile!:empiler
  (syntax-rules ()
    ((pile!:empiler s p) (begin
                            (set! p (cons s p))
                            p))))
```

On constate que c'est beaucoup plus facile à écrire que par la méthode précédente. Il n'y a pas d'évaluation à l'expansion mais une simple réécriture, d'où la disparition de backquote, quasiquote et autres arobasques.

Exercice 13 *Ecrire de cette façon la macro* ***pile!:dépiler****.*

[6] On dit qu'il est linéaire. Pour plus de détails voir le chapitre 17 §4.

La forme incr! est un cas où l'on a besoin de deux règles de réécriture. Selon que l'on se donne ou non l'incrément, on a deux formes possibles pour le motif de l'appel, d'où deux règles.

```
(define-syntax incr!
  (syntax-rules ()
    ((incr! nb)     (set!  nb (+ nb 1)))
    ((incr! nb val) (set!  nb (+ nb val)))))
```

Les formes when et unless et le motif ...

La forme when pose un problème nouveau car le motif d'appel (when test s1 s2 ...) prend un nombre arbitraire ≥ 1 d'expressions si. La solution est très simple : les points de suspension ... font partie de la syntaxe pour écrire un motif. Un motif suivi de ... signifie la possibilité de 0 ou plus répétition de ce motif et deux motifs suivis de ... signifie la présence d'au moins un motif. D'où la définition de la forme when :

```
(define-syntax when
  (syntax-rules ()
    ((when test s1 s2 ...) (if test (begin s1 s2 ...) #f))))
```

Le corps du when devra donc comporter au moins une expression et toutes les expressions qui suivront s2 seront représentées par ... et réécrites telles quelles dans le corps du begin. L'appel :

```
(when (< 0 x) (write x) (set!  x (- x 1)) (write (* 2 x)))
```

sera réécrit en :

```
(if (< 0 x) (begin (write x) (set!  x (- x 1)) (write (* 2 x))) #f)
```

ici les points de suspension représentent l'expression (write (* 2 x)).

Exercice 14 *Redéfinir les formes* ***unless*** *et* ***if**** *décrites au §2.*

La forme defun

C'est un bon exemple de la simplification apportée par la méthode de réécriture. Il s'agit de transformer une définition du type :

```
(defun  nom-fct  Liste-parametres  s1 s2 ...)
```

en

```
(define  nom-fct (lambda Lparm s1 s2 ...))
```

Rappelons que si la première expression *s1* du corps est une chaîne, on l'ajoute dans la a-liste *Documentations*. Il suffit de transcrire ceci mot à mot pour obtenir la définition de la forme defun :

```
(define-syntax defun
  (syntax-rules ()
    ((defun nom-fct  Lparam   e1 e2 ...))
       (begin
         (if (string? 'e1)
           (begin (push!  (cons 'nom-fct e1)  *Documentations*)))
         (define nom (lambda Lparm e1  e2 ...)))))
```

Il n'y a rien à changer à la fonction documentation qui sert à consulter la a-liste. Pour éviter de quoter le nom de la fonction, on définit immédiatement la forme `documentationq` par :

```
(define-syntax documentationq
  (syntax-rules ()
    ((documentationq  nom) (documentation 'nom))))
```

La fonction `permuter!`

On se rappelle que c'est elle qui a attiré notre attention sur le fâcheux phénomène de capture. Qu'en est-il avec la nouvelle méthode? On commence par écrire la transformation de la façon la plus naturelle sans se préoccuper des captures :

```
(define-syntax permuter!
  (syntax-rules ()
    ((permuter! x y)  (let ((tmp x))
                        (set! x y)
                        (set! y tmp)))))
```

On la teste en prenant exprès une variable de nom `tmp` :

```
(define a    "a")
(define tmp "tmp")

? (permuter! a tmp)
```

Ensuite on a :

```
? a -> "tmp"
? tmp "a"
```

Miracle, il n'y a pas eu de capture ! En réalité, ce n'est pas un miracle mais un des gros avantages de cette méthode de définition des macros. Il y a renommage *automagique*[7] des variables pour éviter tout phénomène de capture. On dit, dans la description de Scheme, que ce type de macro-expansion est «hygiénique» par opposition aux «sales» macros de Lisp !

---

[7]signifie, dans le jargon des hackers, un procédé automatique dont le fonctionnement est un peu mystérieux.

La forme while

La forme while était aussi un cas où l'on a dû procéder à un renommage, sa définition est maintenant aussi simple que naturelle :

```
(define-syntax while
  (syntax-rules ()
    ((while test e1 ...) (letrec ((loop (lambda ()
                                         (if test
                                           (begin e1 ... (loop))))))
                            (loop)))))
```

et l'on vérifie bien que

```
? (let ((loop 4))
    (while (< 0 loop)
           (display loop)(set! loop (- loop 1))))
4321
```

Les formes and et or

Ce sont de bons exemples de l'utilisation de plusieurs règles. On avait décrit au §4 le and avec trois égalités :

```
(and)              = #t
(and s1)           = s1
(and  s1 s2 ... )  = (if  s1  (and s2 ...sN)  #f)
```

La définition par define-syntax consiste à recopier ces égalités selon les trois règles de réécriture :

```
(define-syntax and
  (syntax-rules ()
    ((and)       #t)
    ((and s1)    s1)
    ((and s1 s2 ...)  (if s1
                        (and s2 ...)
                        #f))))
```

Notons que le deuxième motif est un cas particulier du troisième, aussi il est important de les écrire dans cet ordre. Si l'on permute les deux derniers, on change la définition du and. D'une manière générale, il faut écrire en premier les motifs les plus spécifiques pour éviter qu'ils soient cachés par des motifs plus généraux.
On remarque par ailleurs que le nom de la forme à définir (ici and) peut aussi apparaître dans le modèle. C'est ce qui permet la récursivité au moment de l'évaluation de l'expansion.

Exercice 15 *Ecrire la définition du or.*

Une morale semble se dégager : ce procédé est beaucoup plus clair et sûr que la méthode des macros par évaluation. Mais deux questions restent en suspend :
- est-il aussi général ?
- est-il aussi efficace ?

9.10 Redéfinition de certaines formes spéciales

On continue notre étude comparative des deux méthodes de définition de macros avec certaines formes prédéfinies. On commence par la plus simple qui illustre bien la puissance de ... (*ellipsis* en anglais).

La forme `let`

On désire transformer l'appel `(let ((x1 e1) ...) s1 ...)` en appel de la lambda expression `((lambda (x1 ...) s1 ...) e1 ...)`. La définition de la macro `let` est identique à sa spécification :

```
(define-syntax let
  (syntax-rules ()
    ((let ((x1 e1) ...) s1 ...) ((lambda (x1 ...) s1 ...) e1 ...))))
```

On voit bien ici le caractère déclaratif de cette méthode, il suffit de dire ce que l'on veut obtenir sans avoir à préciser comment on le calcule.

La forme `let*`

Elle se définit par deux règles, selon qu'il y a ou non des liaisons :

```
(define-syntax let*
  (syntax-rules ()
    ((let* () s1 s2 ...)  ((let () s1 s2 ...)))
    ((let* ((x1 e1) (x2 e2) ...) s1 s2 ...)
     ((let ((x1 e1))
        (let* ((x2 e2) ...) s1 s2 ...))))))
```

Là encore, c'est la transcription immédiate de la spécification donnée au §5 :

La forme `letrec`

On rappelle la spécification utilisée au §5

```
(letrec ((f1 e1)
          ...
         )
  s1 s2 ...)
```

doit s'expanser en un `let` :

```
(let ((f1 '?)
       ...
      )
  (set! f1 e1)
  ...
  s1 s2 ...)
```

La technique des ... permet d'en donner une définition très claire :

```
(define-syntax letrec
  (syntax-rules ()
    ((letrec ((x1 e1) ...) s1 s2 ...) (let ((x1 '?) ...)
                                        (set! x1 e1) ...
                                        s1 s2 ...
                                       ))))
```

Exercice 16 *Ecrire une définition de la macro* `do`*.*

Exercice 17 *On utilise parfois une variante du* `letrec` *appelée* `let` *nommé. Sa syntaxe est voisine de celle du* `let` *et sa sémantique est basée sur le* `letrec`*. Elle s'écrit :*

```
(let  ident  ((x1 e1) ...) s1 s2 ...)
```

La meilleure méthode pour définir sa sémantique est d'en donner la forme `letrec` *équivalente :*

```
(letrec   ((ident  (lambda (x1 ...) s1 s2 ...)))
  (ident  e1 ...))
```

Ecrire sa définition avec `define-syntax`*.*

La forme cond

C'était trop beau pour cela dure ! L'écriture de la macro cond va nous poser un premier petit problème.
On décrit la forme cond par un ensemble de règles, d'abord des règles dans le cas d'une seule clause, puis celles avec plusieurs clauses :

```
(cond (test))                       = test
(cond (else s1 s2 ...))             = (begin s1 s2 ...)
(cond (test s1 s2 ...))             = (if test (begin s1 s2 ...))

(cond (test) c2 c3 ...)             = (or test (cond c2 c3 ...))
(cond (test s1 s2 ...) c2 c3 ...)   = (if test
                                        (begin s1 s2 ...)
                                        (cond c2 c3  ...))
```

Les règles No 2 et 3 ont le même motif si l'identificateur `else` se comportait comme l'identificateur `test`. Une expression comme `(cond (= x 0) #t)` est filtrée par ces deux règles. En réalité, on désire que la 2ème règle ne puisse pas la filtrer. Pour cela, il faut pouvoir indiquer que l'identificateur `else` ne peut représenter que lui-même et doit donc se retrouver tel quel dans l'expression à filtrer. Par exemple, on accepte que l'expression `(cond else #t)` soit filtrée par la deuxième règle mais pas l'expression `(cond (= x 0) #t)`.
Maintenant, on peut indiquer le rôle de la mystérieuse liste qui suit l'identificateur `syntax-rules` : on y place les identificateurs qui ne peuvent représenter qu'eux-mêmes au moment du filtrage. Voici une définition de la forme cond :

```
(define-syntax cond
  (syntax-rules (else)
    ((cond (test)) test)
    ((cond (else s1 s2 ...))  (begin s1 s2 ... ))
    ((cond (test s1 s2 ...))  (if test (begin s1 s2 ... )))
    ((cond (test) c2 c3 ...)  (or test (cond c2 c3 ...)))
    ((cond (test s1 s2 ...) c2 c3 ...)  (if test
                                             (begin s1 s2 ...)
                                             (cond  c2 c3 ...)))))
```

Exercice 18 *Donner une définition de la macro* `case`*.*

Mais on peut rencontrer des problèmes plus sérieux. Dans certaines situations, le caractère hygiénique est un handicap car on souhaite utiliser le phénomène de capture. On verra, au chapitre 11 §3, un exemple lors de la définition de la macro `defmethod`. La solution consiste à contrôler plus finement le filtrage par des fonctions de plus bas niveau. On ne s'étendra pas sur ces aspects car la norme Scheme n'est pas encore stabilisée dans ce domaine.

Liaisons destructurantes

Terminons par la redéfinition avec `define-syntax` des liaisons destructurantes introduites au §6.
Les deux règles qui définissent la forme `bind` conduisent à poser :

```
(define-syntax bind
  (syntax-rules ()
    ((bind (x1 ...) exp s1 s2 ...)
     (apply (lambda (x1 ...) s1 s2 ...) exp))
    ((bind x exp s1 s2 ...)
     ((lambda (x) s1 s2 ...) exp)))).
```

Et la définition récursive de `bind-let*` se transpose immédiatement en :

```
(define-syntax bind-let*
  (syntax-rules ()
    ((bind-let* () s1 s2 ...)
     (begin s1 s2 ...))
    ((bind-let* ((motif1 e1)(motif2 e2) ...) s1 s2 ...)
     (bind motif1 e1 (bind-let* ((motif2 e2) ...) s1 s2 ...)))))
```

En conclusion, la définition des macros par transformation syntaxique est très séduisante dans la plupart des cas, mais il reste des situations où cette méthode n'est pas assez puissante et oblige à des manipulations de plus bas niveaux qui peuvent être assez délicates. En général, les implantations de Scheme fournissent les deux méthodes ce qui permet d'utiliser la plus adéquate.

9.11 Compléments : trace et comptage des appels d'une fonction

La trace est un outil précieux, que faire si notre implantation de Scheme n'en fournit pas ? Tout simplement définir notre propre outil de traçage en Scheme. Commençons par un cas particulier.

Trace manuelle de factorielle

Supposons que nous voulions tracer notre sempiternelle fonction factorielle

```
(define fac
  (lambda (n)
    (if (zero? n)
      1
      (* n (fac (- n 1))))))
```

Il faut provoquer un affichage à l'appel et un affichage au retour de la valeur. On doit donc incorporer dans le code de factorielle des instructions d'affichage.
Avant de modifier son code, on sauve la version originale de factorielle :

```
(define fac-originale fac)
```

Puis, on encadre le calcul de factorielle par un message d'entrée et un message de sortie :

```
(define fac
  (lambda (n)
    (display-alln "appel de (fac " n")")
    (let ((valeur (fac-originale n)))
      (display-alln "=> (fac " n ") = " valeur)
      valeur)))
```

On essaye :

```
? (fac 4)
appel de (fac 4)
appel de (fac 3)
appel de (fac 2)
appel de (fac 1)
appel de (fac 0)
=> (fac 0) = 1
=> (fac 1) = 1
=> (fac 2) = 2
=> (fac 3) = 6
=> (fac 4) = 24
24
```

Il manque l'indentation mais c'est déjà assez explicite. Il est instructif de suivre le détail des appels :
- l'appel `(fac 4)` déclenche l'affichage de `appel de (fac 4)`,

- ensuite, le calcul de (fac-originale 4) appelle le code de la version originale de fac,
- il y a donc appel récursif de (fac 3) qui déclenche,
- ...
- après le calcul de (fac 0), il y a affichage du message de retour
=> (fac 0) = 1 puis terminaison des appels en cours.
Pour supprimer la trace, il suffit d'évaluer (define fac fac-originale)

Trace générale

Pour automatiser ce processus de modification d'une fonction, on définit une fonction, appelée avec-trace, qui associe à une fonction sa version avec trace. La fonction avec-trace prend en argument le nom de la fonction pour pouvoir l'afficher et la version originale pour calculer la valeur. Le code suit exactement le principe utilisé pour factorielle :

```
(define (avec-trace  nom-fct  fct-originale)
  (lambda Lpar
    (display-alln "appel de " (cons nom-fct   Lpar))
    (let ((valeur (apply fct-originale   Lpar)))
      (display-alln "=> " (cons nom-fct  Lpar) " = " valeur)
      valeur)))
```

Il reste à associer la version tracée avec le nom de la fonction originale, pour cela on utilise une macro pour générer le code de l'affectation :
(set! nom-fct (avec-trace 'nom-fct nom-fct)
C'est immédiat dans les deux styles de macro :

```
(define-macro (trace nom-fct)
  `(set!  ,nom-fct  (avec-trace ',nom-fct ,nom-fct)))
```

ou

```
(define-syntax  trace
  (syntax-rules ()
    ((trace nom-fct)  (set! nom-fct (avec-trace 'nom-fct nom-fct)))))
```

On teste :

```
? (trace fac)

? (fac 3)
appel : (f 3)
appel : (f 2)
appel : (f 1)
appel : (f 0)
valeur de (f 0) 1
valeur de (f 1) 1
valeur de (f 2) 2
valeur de (f 3) 6
6
```

Supprimer la trace avec untrace

Oui, mais comment restaurer la version originale? Ici c'est facile, car on dispose du code de factorielle. Il suffit de réévaluer la définition mais on préférerait disposer d'une méthode comme `untrace`.
Pour restaurer la version originale, il faut la sauvegarder quelque part ; on utilise une variable globale `*traced-fct*`[8] de type table. La trace doit d'abord vérifier si la fonction n'est pas déjà tracée : dans l'affirmative, on ajoute dans la table la fonction originale avec comme entrée le nom de la fonction. On utilise la structure de données table décrite au chapitre 7 §5.

```
(define *traced-fct (table-vide))

(define-syntax  trace
  (syntax-rules ()
    ((trace nom-fct)
     (let ((traced? (table:consulter 'nom-fct  *traced-fct*)))
       (if traced?
         (*erreur* "fct deja tracee :" 'nom-fct)
         (begin
           (set! *traced-fct* (table:ajouter 'nom-fct nom-fct *traced-fct*))
           (set! nom-fct (avec-trace 'nom-fct nom-fct)))))))
```

ou

```
(define-macro (trace nom-fct)
  `(let ((traced? (table:consulter ',nom-fct  *traced-fct*)))
     (if traced?
       (*erreur* "fct deja tracee :" ',nom-fct)
       (begin
         (set! *traced-fct* (table:ajouter ',nom-fct ,nom-fct *traced-fct*))
         (set! ,nom-fct (avec-trace ',nom-fct ,nom-fct))))))
```

On est maintenant en mesure de définir une macro `untrace` pour supprimer la trace. Il suffit d'aller chercher dans la table `*traced-fct*` la valeur originale de la fonction et de lui affecter ou d'afficher un message si elle n'est pas tracée.

```
(define-syntax untrace
  (syntax-rules ()
    ((untrace nom-fct)
     (let ((fct-originale (table:consulter 'nom-fct *traced*)))
       (if fct-originale
         (begin
           (set! nom-fct fct-originale)
           (set! *traced-fct* (table:enlever 'nom-fct  *traced-fct*)))
         (*erreur* "fonction non tracee : " 'nom-fct))))))
```

ou

[8]On pourrait éviter l'utilisation de cette variable globale en permettant à la version tracée de rendre la version originale quand on l'applique à un argument spécial, voir la Slib [Jaf] pour cette approche.

```
(define-macro (m-untrace nom-fct)
  `(let ((fct-originale (table:consulter ',nom-fct *traced-fct*)))
     (if fct-originale
       (begin
         (set! ,nom-fct fct-originale)
         (set! *traced-fct* (table:enlever ',nom-fct  *traced-fct*)))
       (*erreur* "fonction non tracee : " ',nom-fct))))
```

Trace avec indentation

Classiquement, les affichages des appels récursifs d'une fonction sont indentés pour mettre en évidence la profondeur de la récursivité. Pour décaler les affichages à chaque appel, il suffit d'ajouter une variable globale `*compteur*` qui compte les appels. A chaque appel, la variable `*compteur*` est incrémentée et elle est décrémentée à chaque retour. Chaque affichage est effectué avec une indentation de `*compteur*` espaces. Pour réaliser ces modifications, il suffit de remplacer la fonction `avec-trace` par :

```
(define (avec-trace  nom-fct   fct-originale)
  (lambda Lpar
    (incr! *compteur* )(indente *compteur*)
    (display-alln "appel de " (cons nom-fct   Lpar))
    (let ((valeur (apply fct-originale   Lpar)))
      (indente *compteur*)
      (display-alln "=> " (cons nom-fct  Lpar) " = " valeur)
      (decr! *compteur*)
      valeur)))
```

Elle utilise la fonction auxiliaire `indente` qui affiche n espaces :

```
(define (indente  nb-espace)
  (display (make-string nb-espac # )))
```

On dispose maintenant d'un système complet de trace. Pour le tester, il reste à initialiser la variable globale `*compteur*`.

```
(define *compteur* -1)
```

On trace la fonction de Fibonacci :

```
(define (fib n)
  (if (< n 2)
    n
    (+ (fib (- n 1))(fib (- n 2)))))

? (trace fib)

? (fib 4)
appel de (fib 4)
  appel de (fib 3)
   appel de (fib 2)
    appel de (fib 1)
    => (fib 1) = 1
    appel de (fib 0)
```

```
    => (fib 0) = 0
   => (fib 2) = 1
   appel de (fib 1)
   => (fib 1) = 1
  => (fib 3) = 2
  appel de (fib 2)
   appel de (fib 1)
   => (fib 1) = 1
   appel de (fib 0)
   => (fib 0) = 0
  => (fib 2) = 1
 => (fib 4) = 3
3

? (untrace fib)

? (fib 4) -> 3
```

Exercice 19 *Pour concentrer la trace sur certaines valeurs des arguments d'une fonction, on peut définir une trace conditionnelle. La condition est donnée par un prédicat que l'on applique aux arguments. Si le prédicat est vrai sur un argument, on procède aux affichages précédents, sinon on fait un calcul silencieux. Modifier la fonction* `avec-trace` *et la macro* `trace` *pour incorporer, de façon facultative, un tel prédicat. Par exemple, si on ne veut tracer* `fib` *que pour les valeurs* ≥ 3*, on écrira :*

```
?  (trace fib (lambda (n) (<= 3 n))).
```

Comptage des appels

On a indiqué au chapitre 5 §8 une méthode pour compter le nombre d'appels à une fonction. La méthode consistait à associer de façon manuelle un prototype à une fonction, ce prototype ayant un champ qui décompte les appels de la fonction. Voici une méthode plus simple et plus générale qui consiste à introduire temporairement un compteur dans le corps d'une fonction. On s'intéresse au nombre d'appels d'une fonction lors de l'évaluation d'une expression donnée. Durant l'évaluation de cette expression, on remplace la fonction par une fermeture contenant une variable `compteur`. Ensuite, on affiche la valeur de `compteur` et on retourne celle de l'expression. Voici une macro `compter-appels` qui réalise ce travail :

```
(define-syntax compter-appels
  (syntax-rules ()
    ((compter-appels exp fct)
     (let ((fct-originale fct)
           (compteur 0))
       (set! f (lambda Lpar
                 (incr! compteur)
                 (apply fct-originale Lpar)))
       (let ((valeur  exp))
         (set! fct fct-originale)
```

```
        (display-alln "nb d'appels de " 'fct " = " compteur)
        valeur))))
```

ou

```
(define-macro (compter-appels exp fct)
  `(let ((fct-originale ,fct)
         (compteur 0))
     (set! ,fct (lambda Lpar
                  (set! compteur (+ compteur 1))
                  (apply fct-originale Lpar)))
     (let ((valeur  ,exp))
       (set! ,fct fct-originale)
       (display-alln "nb d'appels de " ',fct " = " compteur)
       valeur)))
```

On essaye avec `fib`:

```
? (compter-appels (+ (fib 5)(fib 3)) fib)
nb appels de fib = 20
7
```

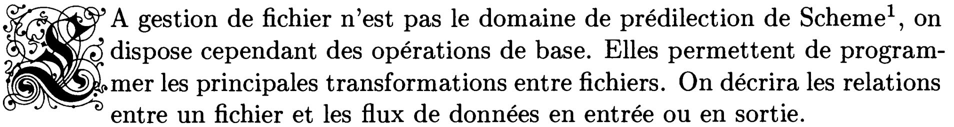
Exercice 20 *Donner une version plus compacte de la macro* ***compter-appels*** *en utilisant la forme* ***fluid-let*** *de l'exercice 11.*

9.12 De la lecture

Les macros pour Scheme sont aussi traitées dans [Dyb87, SF90] et pour Lisp dans [Gra94]. Une étude approfondie des macros est faite dans [Que94].

Chapitre 10

Fichiers, flux et flots



LA gestion de fichier n'est pas le domaine de prédilection de Scheme[1], on dispose cependant des opérations de base. Elles permettent de programmer les principales transformations entre fichiers. On décrira les relations entre un fichier et les flux de données en entrée ou en sortie.
La notion de flot est voisine de celle de flux, elle permet de manipuler des listes infinies. La programmation avec les flots favorise un style de programmation par composition de filtres. On présente en complément un petit système pour réaliser automatiquement l'indentation des S-expressions.

10.1 Fichiers et flux

Jusqu'à maintenant, les fonctions de lecture et d'écriture ne concernaient que les entrées au clavier et les affichages à l'écran. Mais on peut réaliser aussi ces fonctions avec des fichiers. Un fichier est un moyen pour stocker des suites de caractères en mémoire non volatile (disquette, disque dur, CD ROM, bande magnétique, ...). Pour lire ou écrire dans un fichier, il faut établir une connexion entre ce fichier et le système Scheme. Cette connexion permet d'établir un *flux* de caractères entre le fichier et les fonctions d'entrée/sortie de Scheme. Pour lire dans un fichier, on associe à ce fichier un *flux en entrée* qui alimente les fonctions de lecture. Pour écrire dans un fichier, on associe à ce fichier un *flux en sortie* qui est alimenté par les fonctions d'écriture.
Un nom de fichier est une chaîne dont la structure dépend des conventions utilisées par le système d'exploitation pour coder la hiérarchie des répertoires.
On dispose de plusieurs fonctions pour associer un flux à un fichier :

`(open-input-file` *`nom-fichier`*`)` → flux d'entrée associé à un fichier existant.
`(open-output-file` *`nom-fichier`*`)` → flux de sortie associé à un fichier à créer.

Après utilisation d'un flux, on doit le refermer par :

[1] Ce n'est pas un réel inconvénient car les systèmes d'exploitation actuels fournissent des outils très complets pour ce type de tâche.

`(close-input-port` *flux*`)` pour un flux d'entrée.
ou par
`(close-output-port` *flux*`)` pour un flux de sortie.

Pour relier un fichier avec un flux il est souvent plus préférable d'utiliser les fonctions suivantes :

```
(call-with-input-file fichier fct)
(call-with-output-file fichier fct)
```

car elles incluent la fermeture automatique du flux après usage. La variable *fct* doit être une fonction dont le seul paramètre sera lié au flux pendant l'exécution du corps de la fonction et le flux sera refermé après cette exécution. Dans le cas de `call-with-input-file`, le fichier doit déjà exister alors que pour la fonction `call-with-output-file` l'effet n'est pas spécifié si le fichier existe déjà.

Un flux est un objet Scheme dont on peut tester la nature par les prédicats :

`(input-port? s) -> #t` quand `s` est flux d'entrée,
`(output-port? s) -> #t` quand `s` est flux de sortie.

Le flux courant en entrée est donné par la fonction `(current-input-port)` et le flux courant en sortie est donné par la fonction `(current-output-port)`.
On dit que le clavier est le flux standard d'entrée et l'écran le flux standard de sortie.

10.2 Lecture de fichiers

Les fonctions de lecture

Toutes les fonctions de lecture de Scheme admettent un argument supplémentaire optionnel qui est un flux. La fonction

`(read [`*flux*`])`[2]

permet de lire la première S-expression présente en tête du fichier associé et place le début de la prochaine lecture après le dernier caractère de cette expression.
La fonction `read` est en fait un analyseur syntaxique pour les S-expressions. Elle transforme la représentation externe, donnée dans le fichier, en représentation interne qui sera manipulée par Scheme.
Quand on arrive en fin de fichier, un objet fin de fichier est rendu, on peut le reconnaître grâce au prédicat : `(eof-object? s)`.

Pour illustrer ces opérations, écrivons une fonction qui teste si une S-expression donnée est présente dans un fichier de S-expressions. C'est une simple boucle qui appelle `read` jusqu'à ce que l'on trouve la S-expression ou la fin du fichier.

```
(define (expression-presente? Sexp  nom-fichier)
  (letrec ((loop
             (lambda(flux)
```

[2]La notation [`flux`] signifie que l'argument `flux` est optionnel.

```
           (let ((expression-lue (read flux)))
             (cond ((eof-object? expression-lue)  #f)
                   ((equal? expression-lue Sexp)  #t)
                   (else (loop flux))))))
    (call-with-input-file  nom-fichier loop)))
```

On peut aussi lire un flux caractère par caractère avec la fonction :

(read-char [*flux*])

Pour lire le prochain caractère, mais sans avancer dans le flux, on utilise :

(peek-char [*flux*])

On ne peut pas lire par `read` les fichiers quelconques, il faut les parcourir caractère par caractère. Voici une fonction `voir-fichier` qui copie à l'écran le contenu d'un fichier. Elle fonctionne comme la fonction précédente : une boucle permet de lire les caractères jusqu'à la marque de fin de fichier, chaque caractère lu est affiché à l'écran.

```
(define (voir-fichier  nom-fichier)
  (letrec ((loop
            (lambda(flux)
              (let ((char (read-char flux)))
                (if (not (eof-object? char))
                  (begin (display char)
                         (loop flux)))))))
    (call-with-input-file  nom-fichier loop)))
```

Exercice 1 *1. Ecrire une fonction qui compte le nombre de caractères d'un fichier.*

2. Même question en ne comptant que les caractères qui ne sont ni un espace, ni une tabulation ou une marque de fin de ligne.

Fichiers de texte ou de blocs

Certains fichiers sont structurés en une succession de blocs. Par exemple, un fichier de texte est une succession de lignes, une fin de ligne est indiquée par le caractère `#\newline`.

Pour marquer la fin d'un bloc, on utilisera en général un ou plusieurs caractères distinctifs appelés *séparateurs*. La lecture de ces fichiers se fait souvent bloc par bloc. On écrit une fonction pour lire tout un bloc dans un flux et on retourne ce bloc sous forme de la chaîne de ses caractères. On passe en paramètre un prédicat `séparateur?` pour détecter les caractères pouvant servir de séparateurs. La fonction interne est une boucle qui accumule dans une liste les caractères du bloc et on les transforme à la fin en une chaîne. Avant d'appeler la lecture du bloc, on s'assure qu'il n'y plus de séparateur en les faisant lire par une fonction `lire-separateur` :

```
(define (lire-bloc  flux  separateur?)
  (letrec ((loop (lambda(Lcar)
                  (let ((c (peek-char flux)))
                    (if (separateur? c)
                      (list->string (reverse Lcar))
                      (loop (cons (read-char flux) Lcar)))))))
    (lire-separateurs  separateur? flux)
    (loop '())))
```

Exercice 2 *Ecrire la fonction* `lire-separateur`.

Si l'on désire lire ligne à ligne un fichier de texte, on utilisera le prédicat `separateur?` défini par `(lambda(char)(eq? #\newline char)`. Mais on peut aussi lire mot à mot, en convenant que les mots sont séparés par des caractères dits blancs : espace, tabulation, saut de ligne, saut de page et retour chariot. Ces caractères sont reconnus par le prédicat prédéfini `char-whitespace?`. Si l'on utilise des signes de ponctuation dans le texte, il suffit de les ajouter comme caractères séparateurs.

Exercice 3 *Ecrire une fonction, analogue au* `more` *Unix, pour voir à l'écran le contenu d'un fichier de texte par page de N lignes. Après l'affichage de chaque page, on demande à l'utilisateur s'il désire voir la page suivante ou quitter.*

Exercice 4 *Ecrire une fonction de comparaison de deux fichiers de texte. On retourne la première ligne où les fichiers diffèrent.*

10.3 Ecriture de fichiers

Les fonctions d'écriture

Les fonctions d'écriture admettent également un argument supplémentaire qui est un flux de sortie.

`(write s [flux])`, `(display s [flux])`, `(newline [flux])`, `(write-char carac [flux])`

Toutes ces fonctions provoquent une écriture et retournent une valeur indéfinie que nous n'afficherons pas. L'écriture par `display` est plus agréable pour l'œil mais le résultat de l'écriture par `write` est à nouveau lisible par Scheme.
Il suffit de comparer les affichages :

```
? (write   #\n)
#\n

? (display  #\n)
n

? (write   "voici une chaine")
"voici une chaine"

? (display   "voici une chaine")
voici une chaine
```

Copie de fichiers

L'opération de base entre fichiers est la copie. On procède caractère par caractère jusqu'à la marque de fin de fichier. Pour varier, on a écrit la boucle de parcours du fichier avec la forme do.

```
(define (copier-fichier source destination)
  (call-with-input-file source
    (lambda(input-flux)
      (call-with-output-file destination
        (lambda(output-flux)
          (do ((char (read-char input-flux) (read-char input-flux)))
              ((eof-object? char) )
            (write-char output-flux))))))
```

Si l'on a affaire à un fichier de code Scheme, on peut remplacer la lecture caractère par caractère par une lecture à base de S-expressions, mais alors les commentaires ne seront pas recopiés. Plus généralement, si l'on a un fichier structuré en blocs, on peut copier bloc par bloc à l'aide de fonctions de lecture et d'écriture de bloc. Voici une telle fonction pour copier bloc à bloc :

```
(define (copier-flux flux-in flux-out lire-bloc ecrire-bloc )
  (letrec ((loop
            (lambda()
              (let ((bloc (lire-bloc flux-in)))
                (if (eof-object? bloc)
                  'indefini
                  (ecrire-bloc bloc flux-out))))))
    (loop)))
```

Exercice 5 *Ecrire une fonction pour numéroter les lignes d'un fichier de texte.*

Conversions de fichiers

On peut modifier légèrement la fonction de copie de fichier pour en faire une fonction de transformation de fichier. Par exemple, on sait que la marque de fin de ligne varie selon les systèmes d'exploitation. Avec le système Unix, c'est le caractère line feed LF (code ASCII 10) qui est utilisé, pour les Macintosh c'est le caractère retour chariot CR (code ASCII 13) et pour les fichiers DOS ce sont les deux : CR puis LF. Pour transformer un fichier d'un système à l'autre, il faut procéder à une copie avec conversion des caractères de fin de ligne. Par exemple, voici une fonction pour transformer les fichiers Unix en fichiers Macintosh :

```
(define (Unix->Mac source destination)
  (call-with-input-file source
    (lambda(input-flux)
      (call-with-output-file destination
        (lambda(output-flux)
          (let ((LF (integer->char 10))
                (CR (integer->char 13)))
            (do ((char (read-char input-flux) (read-char input-flux)))
```

```
          ((eof-object? char) )
        (if (char=? char LF)
          (write-char CR   output-flux)
          (write-char char output-flux))))))))
```

Exercice 6 *1. Transformer les fichiers* DOS *en fichiers Unix.*

2. Dans un texte chaque mot qui suit un point a sa lettre initiale en majuscule, écrire une fonction qui corrige d'éventuels oublis de mise en majuscule dans un texte.

Fusion de fichiers

Le traitement de données très volumineuses ne peut se faire que par l'intermédiaire de fichiers. Par exemple la gestion des abonnés au téléphone, des clients d'une banque, des possesseurs du permis de conduire, ... Ceci nécessite des algorithmes adaptés à la structure de fichier pour trier les entrées, ajouter ou supprimer une entrée, ... C'est le domaine des systèmes de gestion de bases de données.
Une opération importante est la fusion de deux fichiers. On considère deux fichiers structurés en bloc et triés par ordre croissant pour une relation d'ordre donnée sur les blocs. Il s'agit de produire un fichier trié contenant l'ensemble des blocs de chaque fichier.
Le principe de la fusion est simple. Le cœur en est la fonction `fusion-flux`, c'est une boucle qui parcourt les deux fichiers à fusionner. Au départ, on lit le premier bloc de chaque fichier et on compare ces deux blocs. Le plus petit est écrit sur la sortie et comme l'autre bloc a été lu, on le garde en paramètre et on lit un nouveau bloc dans le fichier qui a fourni le bloc à écrire. On recommence la même opération avec ces deux blocs. Pour cela les deux blocs courants sont les paramètres de `fusion-flux`. Quand on atteint la fin d'un des fichiers, il reste à recopier la fin de l'autre sur le fichier de sortie, on le fait avec la fonction `copier-flux` définie plus haut.

```
(define (fusion-fichier f1 f2 f   lire ecrire inferieur?)
  (let ((flux1 (open-input-file f1))
        (flux2 (open-input-file f2))
        (flux  (open-output-file f)))
    (letrec ((fusion-flux
              (lambda(bloc1 bloc2)
                (cond ((and (eof-object? bloc1)(eof-object? bloc2))
                       'indefini)
                      ((eof-object? bloc1)
                       (ecrire bloc2 flux)
                       (copier-flux flux2 flux  lire ecrire))
                      ((eof-object? bloc2)
                       (ecrire bloc1 flux)
                       (copier-flux flux1 flux  lire ecrire))
                      ((inferieur? bloc1 bloc2)
                       (ecrire bloc1 flux)
                       (fusion-flux (lire flux1) bloc2 ))
                      (else (ecrire bloc2 flux)
```

```
                              (fusion-flux bloc1 (lire flux2)))
                        ))))
     (fusion-flux (lire  flux1) (lire  flux2))
     (close-input-port  flux1)
     (close-input-port  flux2)
     (close-output-port  flux))))
```

Exercice 7 *Il existe une méthode de tri de fichiers basée sur la constatation triviale suivante: la fusion de deux fichiers triés est un fichier trié de la longueur de la somme des longueurs des fichiers. L'idée consiste à faire une succession de fusions entre les sous-suites croissantes de blocs: la programmer.*

Exercice 8 *Ecrire une fonction pour insérer un nouveau bloc dans un fichier trié de blocs.*

10.4 Le couple force/delay

Pour exposer la notion de flot, on aura besoin d'une méthode pour retarder les évaluations. Voici une telle méthode basée sur la notion de promesse.

Promesse

On a parfois besoin de retarder l'évaluation d'une expression pour éviter d'avoir à le faire si cela ne s'avère pas indispensable. La forme spéciale `(delay` ***exp***`)` retourne une valeur, appelée *promesse*, qui permet de conserver l'expression ***exp*** *sans* l'évaluer.
Quand on a besoin de la valeur de l'expression contenue dans une promesse, on utilise la fonction `force`: `(force` ***promesse***`)`.

```
? (define  p  (delay (begin (display "valeur de l'expresion: ")
                             (+ 4 5))))

? (force p)
valeur de l'expression:  9
```

La première fois qu'une promesse est évaluée par la fonction `force`, sa valeur est *mémorisée.* Si l'on redemande une nouvelle fois de «forcer» la valeur d'une promesse, la valeur mémorisée est retournée sans qu'il y ait besoin d'une nouvelle évaluation de l'expression.

```
? (force p)
9            ;; la valeur est retournée sans affichage du message.
```

Pour toute expression `s`, on a la relation:

`(force (delay  s)) =` $\bar{\texttt{s}}$.

Une implantation de `delay` et `force`

Si une implantation de Scheme ne fournit pas les fonctions `delay`, `force`, on peut facilement les ajouter grâce au mécanisme des extensions syntaxiques. Commençons par la forme `delay`.
Pour éviter l'évaluation d'une expression, une méthode consiste à la placer dans le corps d'une lambda expression sans paramètre :

```
(delay  exp)    =  (lambda() exp)
```

Pour obtenir la valeur de la promesse, il suffit de procéder à l'appel de cette lambda :

```
((lambda() exp ))
```

D'où la fonction force :

```
(define (force promesse)
  (promesse))
```

Essayons avec l'expression précédente :

```
? (define p  (lambda() (begin (display "valeur de l'expresion : ")
                              (+ 4 5)))
p

? (force p)
valeur de l'expresion : 9
```

Mais si on redemande la valeur de la promesse :

```
? (force p)
valeur de l'expresion : 9
```

il y a de nouveau évaluation de l'expression. Aussi, il manque un moyen pour se rappeler la valeur de l'expression après son premier forçage de façon à éviter une nouvelle évaluation.
Une possibilité consiste à garder cette valeur dans la fermeture créée lors de l'évaluation de la lambda ; mais ce n'est pas suffisant. Il faut aussi pouvoir faire la distinction entre une promesse qui a été forcée et une qui ne l'a jamais été. Pour cela, on rajoute un booléen dans la fermeture de la lambda. L'appel `(delay exp)` doit s'expanser en :

```
(let ((promesse (lambda () exp))
      (non-evalue? #t)
      (valeur '?))
  (lambda()
    (if non-evalue?
      (begin (set! valeur  (promesse))
             (set! non-evalue? #f))
      valeur)))
```

On voit qu'après la première évaluation, la variable `valeur` représente la valeur et le booléen `non-evalue?` passe à `#f`.
Comme `delay` ne doit pas évaluer son argument, on utilise le mécanisme des extensions syntaxiques pour définir cette forme spéciale. Donnons la définition avec les deux techniques de macro :

```
(define-syntax delay
  (syntax-rules ()
    ((delay exp) (creer-promesse (lambda() exp)))))
```

ou

```
(define-macro (delay exp)
  `(creer-promesse (lambda() ,exp))).
```

La fonction de construction de la promesse prend en argument une lambda expression sans paramètre :

```
(define creer-promesse
  (lambda(suspension)
    (let ((non-evalue? #t)
          (valeur '?))
      (lambda()
        (if non-evalue?
          (begin (set! valeur  (suspension))
                 (set! non-evalue? #f)
                 valeur)
          valeur)))))
```

Si on refait les tests ci-dessus, on trouve bien le comportement espéré.

10.5 Flots et listes infinies

Une application importante de ce mécanisme d'évaluation retardée est la définition de listes infinies ou flots.

Les fonctions `f-cons`, `f-car`, `f-cdr` pour les flots

On construit une liste par application répétée de la fonction `cons`

```
(cons 'a (cons 'b ...(cons 'e '()))) -> (a b... e)
```

mais comment définir une liste infinie ? Si on utilise une suite infinie de `cons` :

```
(cons 'a (cons 'b ...(cons 'e ... ->?  (a b... e ... )
```

on ne sait pas comment évaluer le deuxième argument du premier `cons`. L'idée consiste à définir une nouvelle fonction `cons` qui n'*évalue pas* son deuxième argument mais en fait une *promesse*. Appelons `f-cons` cette nouvelle forme que l'on définit par une macro :

```
(define-macro (f-cons s1 s2)
  `(cons ,s1 (delay ,s2)))
```

ou

```
(define-syntax f-cons
  (syntax-rules ()
    ((f-cons s1 s2)  (cons  s1 (delay s2)))))
```

Pour récupérer la valeur du car, il suffit d'utiliser la fonction car habituelle :

```
(car (cons  s1  (delay  s2))) = s̄1
```

Mais pour accéder à la valeur de s2, il ne suffit pas de prendre le cdr, il faut aussi forcer le résultat :

```
(force (cdr  (cons  s1  (delay  s2))))) = (force (delay s2)) = s̄2
```

Appelons `f-cdr` ce cdr forcé, et par symétrie, appelons aussi `f-car` la fonction car habituelle

```
(define (f-cdr  e)
  (force (cdr e)))

(define  f-car  car)
```

Elles satisfont aux mêmes relations que car, cdr et cons :

```
(f-car (f-cons  s1 s2)) = s̄1    (f-cdr  (f-cons  s1 s2)) = s̄1
```

Revenons à notre liste infinie, on pose :

```
  e = (f-cons 'a (f-cons 'b ...(f-cons 'e ...) ...)

    = (cons 'a (delay  (f-cons 'b ...(f-cons 'e ...) ...)
```

Maintenant, le deuxième argument de `f-cons` n'est pas évalué, il n'est donc pas gênant qu'il désigne une suite potentiellement infinie de calculs. Calculons le `f-cdr` de `e` :

```
  (f-cdr e) = (f-cons 'a (f-cons 'b ...(f-cons 'e ...

            =  (cons 'a  (delay (f-cons 'b ...(f-cons 'e ...
```

On obtient une expression du même type, d'où la valeur du deuxième élément de cette liste en prenant le `f-car`. On constate que cette liste infinie permet néanmoins d'accéder à chacun de ses éléments par une suite de `f-car` et `f-cdr`. Bien entendu, on ne définit pas une liste infinie par des points de suspension mais avec une définition récursive.

Définition des flots

Ce type d'objet potentiellement infini s'appelle un *flot.* Ce nom ressemble à celui de flux et il lui est aussi sémantiquement voisin. Le flux associé à un fichier permet d'obtenir à la demande et dans l'ordre les éléments du fichier ; de même, un flot va permettre de générer à la demande et dans l'ordre les éléments de la suite infinie qu'il représente. Par exemple, la liste infinie ne comportant que des 1 `(1 1 1 ....)` est définie récursivement par :

```
(define  flot-de-1   (f-cons  1  flot-de-1 ))

? (f-car flot-de-1)
1

?(f-car (f-cdr flot-de-1))
1
...
```

Notons que ce mécanisme d'évaluation nous a permis de définir récursivement un objet sans avoir à donner une condition d'arrêt ! On reviendra plus en détail sur cette méthode d'évaluation au chapitre 21.

Les flots s'interprètent intuitivement comme des listes plates infinies, aussi va-t-on étendre aux flots certaines fonctions introduites pour les listes.
Pour accéder au n-ième élément d'un flot, on définit la fonction :

```
(define (f-nth flot n)
  (if (zero? n)
    (f-car flot)
    (F-nth (f-cdr flot) (- n 1))))

? (f-nth  flot-de-1   9)
1
```

En général, on préfixera par un `f` les fonctions sur les flots.
L'intervalle $[n \quad +\infty[$ des entiers supérieurs à l'entier n peut être représenté par un flot défini par :

```
(define  (f-intervalle n)
  (f-cons n (f-intervalle (+ n 1))))
```

En particulier, on appelle `f-entiers` le flot des entiers naturels :

```
(define f-entiers  (f-intervalle 0))
```

Pour visualiser le début d'un flot, on définit une fonction qui convertit les `n` premiers éléments d'un flot en une liste :

```
(define (flot->liste flot n)
  (if (zero? n)
    '()
    (cons (f-car flot)(flot->liste (f-cdr flot) (- n 1)))))

? (flot->liste f-entiers  8) -> (0 1 2 3 4 5 6 7)
```

Exercice 9 *1. Pouvez-vous définir l'égalité de deux flots ?*

2. Ecrire une fonction qui réalise la fusion de deux flots de nombres strictement croissants.

Opérations sur les flots

Il est facile d'étendre au cas des flots l'itérateur `map`:

```
(define (f-map  f . flots)
  (f-cons (apply f (map f-car flots))
          (apply f-map (cons f (map f-cdr flots)))))
```

Par exemple, le flot des entiers pairs peut s'obtenir en ajoutant `f-entiers` à lui-même :

```
? (flot->liste (f-map + f-entiers f-entiers) 8) -> (0 2 4 3 4 10 12 14)
```

On peut filtrer par un prédicat `p?` les éléments d'un flot pour ne garder que le flot de ceux qui satisfont au prédicat

```
(define (f-filtre p? flot)
  (if (p? (f-car flot))
    (f-cons (f-car flot)(f-filtre p? (f-cdr flot)))
    (f-filtre p? (f-cdr flot))))
```

Par exemple, les entiers impairs peuvent s'obtenir en filtrant les entiers non divisibles par 2 du flot des entiers :

```
(define (non-divisible-par? k)
  (lambda(n)(not (zero? (remainder n k)))))

? (define f-impairs (f-filtre (non-divisible-par? 2) f-entiers))

? (flot->liste  f-impairs  8) -> (1 3 5 7 9  11 13 15)
```

Exercice 10 *Définir une fonction qui construit le flot des entiers de la progression arithmétique de raison* **k** *et de premier terme* **a** *:* $\{ n = a + k\, r \mid k \geq 0\}$*. Etudier le flot des entiers communs à deux progressions arithmétiques de raisons différentes.*

Crible d'Eratosthène

Une application classique des flots est la méthode du crible d'Eratosthène pour énumérer les nombres premiers.
Comme 2 est le plus petit nombre premier, on part de la suite des entiers : `{2 3 4 5 6 7 8 9 10 11 12 13 ...}`. L'idée du crible consiste à supprimer les multiples de 2, il reste `{3 5 7 9 11 13 ...}` donc 3 est premier car il n'est pas divisible par un entier plus petit que lui. Puis, on supprime les multiples de 3, il reste `{5 7 11 13...}` donc 5 est premier ... Puis, on supprime les multiples de 5, il reste `{7 11 13 ...}` donc 7 est premier ...

A chaque étape, le plus petit entier restant est premier car il n'est multiple d'aucun nombre inférieur à lui. D'où la fonction `crible`, elle construit le flot constitué de chacun de ces entiers restant après avoir supprimé les multiples des entiers inférieurs :

```
(define (crible flot)
  (f-cons (f-car flot)
          (crible (f-filtre (non-divisible-par? (f-car flot))
                            (f-cdr flot)))))
```

On obtient le flot des nombres premiers en partant de l'intervalle n $\geq$ 2 :

```
? (define f-nb-premiers (crible (f-intervalle 2)))

? (flot->liste f-nb-premiers 25)
(2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97)
```

Si l'on souhaite visualiser les éléments d'un flot, on peut définir une fonction qui associe à un flot un générateur d'éléments. On utilise la technique des générateurs étudiée au chapitre 5 §5 :

```
(define (creer-generateur flot)
  (lambda()
    (let ((v (f-car flot)))
      (set! flot (f-cdr flot))
      v)))

(define gen-nb-premiers (creer-generateur f-nb-premiers))

? (gen-nb-premiers)
2

? (gen-nb-premiers)
3

? (gen-nb-premiers)
5
```

Listes et flots finis

La technique des flots est également intéressante pour traiter des listes ordinaires. Car l'évaluation à la demande des éléments d'une liste peut permettre d'éviter des calculs. On identifie une liste à un flot dont le début comporte les éléments de la liste et le reste est constitué de la répétition d'un même objet servant de marque. Pour éviter les confusions, il faudra utiliser une marque ne pouvant pas être égale à un élément de la liste concernée.

On appelle `eoflot` la variable qui a pour valeur cette marque de fin de flot fini.

```
(define eoflot 'eoflot)
```

Les flots dont les éléments sont égaux à cette marque à partir d'un certain rang sont appelés *flots finis*. On définit le *flot nul* comme étant le flot fini dont tous les éléments sont égaux à la marque :

```
(define f-null (f-cons  eoflot f-null))
```

Pour tester si un flot fini est nul, on pose :

```
(define (f-null? flot)
  (eq? (f-car flot) eoflot))
```

On est maintenant en mesure d'associer un flot fini à une liste :

```
(define (list->flot-fini L)
  (if (null? L)
    f-null
    (f-cons (car L)(list->flot-fini (cdr L)))))

? (flot->liste (list->flot '(a b c)) 7)
(a b c eoflot eoflot eoflot eoflot)
```

L'opération inverse consiste à recopier dans une liste les éléments d'un flot fini en s'arrêtant à la marque de fin de flot :

```
(define (flot-fini->list flot-fini)
  (if (f-null? flot-fini)
    '()
    (cons (f-car flot-fini)
          (flot-fini->list (f-cdr flot-fini)))))

? (flot-fini->list (list->flot-fini '(a b c))) -> (a b c)
```

Contrairement aux flots généraux, il est possible de comparer deux flots finis :

```
(define (f-equal? flot-fini1 flot-fini2)
  (cond ((f-null? flot-fini1)(f-null? flot-fini2))
        ((f-null? flot-fini2) #f)
        (else (and (equal? (f-car flot-fini1)(f-car flot-fini2))
                   (f-equal? (f-cdr flot-fini1)(f-cdr flot-fini2))))))
```

On peut étendre aux flots finis l'opération de concaténation :

```
(define (f-append-fini flot-fini1 flot-fini2)
  (if (f-null? flot-fini1)
    flot-fini2
    (f-cons (f-car flot-fini1)
            (f-append-fini (f-cdr flot-fini1) flot-fini2))))
```

10.6 Générateur de solutions (III)

On a déjà présenté deux méthodes (au chapitre 8) pour construire un générateur de solutions. La méthode des flots nous en donne une autre qui est peut-être plus naturelle à utiliser.
On considère le problème de la comparaison des ensembles de feuilles de deux arbres. On a suggéré au chapitre 8 §5 une méthode pour faire cette comparaison sans être obligé d'avoir à calculer les deux listes de feuilles. En voici une autre qui consiste à définir pour chaque arbre le flot fini de ses feuilles puis à comparer ces flots. Elle conserve donc l'idée naturelle de procéder en deux temps : d'abord un calcul des ensembles de feuilles puis la comparaison de ces ensembles. Mais l'utilisation des flots entraîne que ces calculs sont latents. En cas de différence, seul le calcul des feuilles permettant d'en décider sera effectué.
Le flot fini des feuilles d'un arbre binaire est construit par :

```
(define (f-feuilles arbre)
  (cond ((arbre2:vide? arbre) f-null)
        ((arbre2:feuille? arbre)(f-cons (arbre2:racine arbre) f-null))
        (else (f-append-fini (f-feuilles (arbre2:filsg arbre))
                             (f-feuilles (arbre2:filsd arbre))))))

? (flot-fini->list (f-feuilles '(a (b 2 (d 5 e )) (c ()  3)) ))
(2 5 e 3)
```

La comparaison des feuilles consiste simplement à comparer les flots associés :

```
(define (meme-feuilles? arbre1 arbre2)
  (let ((f-feuilles1 (f-feuilles arbre1))
        (f-feuilles2 (f-feuilles arbre2)))
    (f-equal? f-feuilles1 f-feuilles2)))

? (meme-feuilles? '(a (b 2 (d 5 () )) (c ()  3))
                  '(a (b 2 (d 5 () )) (c ()  3)))
#t

? (meme-feuilles? '(a (b 2 (d 5 () )) (c ()  3))
                  '(a (b 0 (d 5 () )) (c ()  3)))
#f
```

On voit que la technique des flots permet de concilier programmation intuitive et efficacité. Appliquons cette technique à l'exploration d'un arbre en vue de rechercher les nœuds qui satisfont un critère donné. On commence par associer à un arbre binaire le flot des étiquettes de ses nœuds. Pour cela on utilise une visite en pré-ordre (voir chapitre 7 §6)

```
(define (arbre->flot arbre)
  (if (arbre2:vide? arbre)
    f-null
    (let ((racine (arbre2:racine arbre) ))
      (f-cons racine
              (f-append-fini (arbre->flot  (arbre2:filsg arbre))
```

```
                                (arbre->flot  (arbre2:filsd arbre))))))

? (flot-fini->list (arbre->flot '(a (b 2 (d 5 () )) (c ()  3))))
 (a b 2 d 5 c 3)
```

Pour rechercher dans ce flot les étiquettes qui satisfont à un prédicat p?, il suffit de le filtrer :

```
(define (flot-solutions arbre p?)
  (f-filtre (lambda(x)(or (eq? eoflot x)(p? x)))
            (arbre->flot arbre)))
```

Si l'on souhaite générer les solutions à la demande, on utilise la fonction de création de générateur :

```
(define (gen-solutions arbre p?)
  (creer-generateur (flot-solutions arbre p?)))

(define etiquettes-numeriques
  (gen-solutions  '(a (b 2 (d 5 () )) (c ()  3)) number?))

? (etiquettes-numeriques)
2
5
3
eoflot
```

10.7 Flots et séries formelles

Ce paragraphe, à caractère mathématique, peut être sauté en première lecture.

Flot associé à une série formelle

Une série formelle $s = \sum_{i \geq 0} a_i X^i$ à coefficients numériques a_i généralise la notion de polynôme au cas d'une infinité de monômes. Il ne faut pas essayer de la considérer comme une fonction de X, d'ailleurs pour certaines valeurs de X il peut être impossible de lui associer une valeur. C'est un objet *formel* qui se manipule indépendamment de toute idée de fonction. La série formelle est parfaitement définie par la suite de ses coefficients, ou encore par le flot infini $(a_0, a_1, a_2, \ldots)$.
Il est facile de créer le flot des coefficients quand on connaît la fonction
$coeff\ :\ i \to a_i,$ il suffit d'appliquer cette fonction sur le flot des entiers.

```
(define (creer-serie fct-coeff)
  (f-map  fct-coeff  f-entiers))
```

Par exemple, la série associée à la fonction exponentielle $\sum_0^\infty \frac{X^n}{n!}$ peut être définie par :

```
(define serie-exp (creer-serie  (lambda(k)(/ 1 (fac  k)))))

? (flot->liste serie-exp  5) -> (1 1 1/2 1/6 1/24)
```

Opérations sur les séries formelles

On peut étendre aux séries les principales opérations faites sur les polynômes. La multiplication par un nombre consiste à multiplier chaque coefficient par ce nombre :

```
(define (mul-scalaire-serie   k  serie)
  (f-map  (lambda(n)(* k n)) serie))
```

Pour additionner deux séries, il suffit d'additionner les coefficients correspondants :

```
(define (add-serie  serie1 serie2)
  (f-map  + serie1  serie2))
```

Une primitive d'une série formelle s'obtient en prenant une primitive de chacun de ses monômes :

$$a_i X^i \to \frac{a_i X^{i+1}}{i+1}.$$

Pour réaliser cette opération sur le flot associé, on commence par remplacer chaque coefficient a_i par $a_i/i+1$:

```
(define (primitive serie)
  (f-map  (lambda(v n) (* v (/ 1 (+ n 1)))) serie f-entiers))
```

Mais ce n'est pas tout, il faut aussi décaler tous les coefficients, le i-ème devient le i+1-ème et choisir une valeur pour le nouveau coefficient d'indice 0. Par exemple, la primitive de la série exponentielle qui a pour coefficient 1 en 0 est encore la série exponentielle, cela peut être un autre moyen pour la définir. D'où une définition récursive de la série exponentielle :

```
(define serie-expo (f-cons 1 (primitive serie-expo)))
```

On vérifie que l'on retrouve les coefficients de la `serie-exp` :

```
? (flot->liste serie-expo 5) -> (1 1 1/2 1/6 1/24)
```

Exercice 11 *Définir par ce même procédé les séries des fonctions sin et cos. On notera que sin est la primitive nulle en 0 de cos et que −cos est la primitive de sin qui vaut 1 en 0.*

On peut multiplier les séries formelles. Le coefficient c_n de la série produit de $\sum a_i X^i$ par $\sum b_j X^j$ est donné par

$$c_n = \sum_{i=0}^{n} a_i \, b_{n-i}$$

La fonction qui calcule les coefficients c_n est générée par :

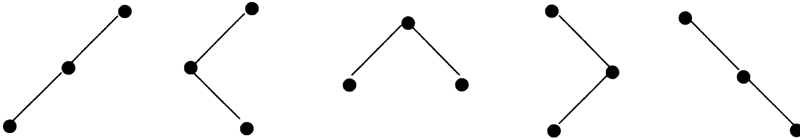
```
(define (coeff-produit-serie  serie1 serie2)
  (lambda(n)
    (letrec ((loop (lambda(coeff i)
                     (if (< n i)
                       coeff
                       (loop (+ coeff (* (f-nth serie1 i)
                                         (f-nth serie2 (- n i))))
                             (+ i 1))))))
      (loop 0 0))))
```

D'où la définition de la série produit :

```
(define (prod-serie s1 s2)
  (let ((coeff-produit (coeff-produit-serie  s1 s2)))
    (creer-serie coeff-produit)))

? (flot->liste (prod-serie serie-exp serie-exp) 10)
 (1 2 2 4/3 2/3 4/15 4/45 8/315 2/315 4/2835)
```

Applications au calcul du nombre d'arbres binaires

On désigne par b_n le nombre d'arbres binaires non étiquetés ayant n nœuds. On se propose de calculer les b_n (appelés nombres de Catalan). Par exemple, on a $b_0 = 1$ car l'arbre vide est le seul ayant 0 nœud et $b_3 = 5$ car il y cinq arbres binaires avec trois nœuds :



Pour calculer de façon systématique les coefficients b_n, on leur associe la série formelle $B = \sum_{n\geq 0} b_n X^n$; elle s'appelle la *série génératrice* des nombres b_n. La définition récursive de la structure d'arbre binaire :

```
Arbre2 =   Vide  ou   (cons-arbre2  racine Arbre2 Arbre2)
```

permet de démontrer (voir par exemple [Bur75]) que la série génératrice vérifie l'égalité :

$$B = 1 + X.B.B \qquad (*)$$

En traduisant cette égalité, on va obtenir une définition récursive du flot associé à B. Pour multiplier la série B par la variable X, il suffit de décaler ses coefficients de 1 et de donner la valeur 0 au coefficient d'indice 0. La série $X.B.B$ est donc associée au flot `(f-cons 0 (prod-serie B B))`.
Pour y ajouter la constante 1, il reste à incrémenter de 1 le coefficient d'indice 0. Pour cela on utilise la fonction qui ajoute une constante `c` à une série :

```
(define (add-cte c serie)
  (let ((a (f-car serie)))
    (f-cons (+ a c) (f-cdr serie))))
```

D'où la définition récursive de la série B :

```
(define B (add-cte 1 (f-cons 0 (prod-serie B B))))
```

Ce qui permet de calculer les valeurs des b_n:

```
? (flot->liste B 10) -> (1 1 2 5 14 42 132 429 1430 4862)
```

La méthode des flots est donc un bon moyen pour calculer certaines séries définies implicitement.

Remarque 1 Dans cet exemple, on peut aussi calculer la forme explicite des b_n. En résolvant l'équation du second degré (*), on trouve que la solution valant 1 pour $x = 0$ est donnée par :

$$B(X) = \frac{1 - \sqrt{(1 - 4X)}}{2X}$$

En identifiant la série B avec le développement en série entière du deuxième membre, il vient :

$$b_n = \frac{1}{n+1}\binom{2n}{n}$$

10.8 Compléments : indentation des S-expressions

Les éditeurs de texte pour Scheme fournissent en général un mode pour l'indentation automatique des S-expressions (*pretty print* en anglais). On se propose de décrire une telle méthode. Pour simplifier, on ne s'occupera pas des contraintes dues à une limitation de la longueur des lignes mais on privilégiera une disposition verticale.

Les modes d'indentation

Les méthodes d'indentation concernent seulement les formes spéciales car pour les autres fonctions on affiche la liste sans disposition particulière. On regroupe les formes spéciales qui utilisent le même schéma d'indentation. Bien entendu, une convention d'affichage est affaire de goût. On propose ici une méthode très simple[3] qui peut servir de base à beaucoup de raffinements. On utilise les regroupements suivants :

- Mode d'indentation pour `if, or, and, map, for-each`[4] :

Le nom de la forme spéciale et son premier argument sont affichés sur la même ligne et les autres arguments sont alignés verticalement sous le premier argument :

[3] Elle diffère un peu (notamment pour le `if`) de celle utilisée dans ce livre.

[4] `map` et `for-each` ne sont pas des formes spéciales mais on les inclut car on utilise aussi pour elles une présentation sur plusieurs lignes.

```
(if test
    si
    [sinon])
```

- Mode d'indentation pour `begin` et `cond`

Les arguments de la forme sont alignés verticalement sous le nom de la forme avec un décalage à droite de deux espaces :

```
(cond
  c1
  ...
  cN)
```

- Mode d'indentation pour `let` et `let*`

Les liaisons sont placées en colonne et commencent sur la même ligne que le nom de la forme spéciale. Les expressions du corps sont aussi alignées verticalement mais avec un décalage à droite de 2 espaces par rapport au nom de la forme. Comme les expressions `ei` sont en général simples, on ne propage pas l'indentation à ce niveau

```
(let ((x1 e1)
      ...
      (xK eK))
  s1
  ...
  sN)
```

- Mode d'indentation pour `letrec`

C'est le même que le précédent à une nuance près : ici les expressions `ej` peuvent être complexes car elles servent en général à définir des fonctions récursives, aussi on y propage l'indentation

```
(letrec ((x1 e1)
         ...
         (xK eK)
        )
  s1
  ...
  sN)
```

- Mode d'indentation pour `do`

Les définitions des variables locales et le corps sont disposés comme dans le cas précédent, la différence concerne le test de fin de boucle qui est aligné à la verticale de la première parenthèse des définitions de variables.

```
(do  ((var1  init1  incr1)
      ...
      (varK  initK  incrK))
     (test   [resultat])
  s1
  ...
  sN)
```

- Mode d'indentation pour `lambda, define, case`

Le nom de la forme spéciale et son premier argument sont affichés sur la même ligne et les autres arguments sont alignés verticalement sous le nom de la forme spéciale avec un décalage à droite de deux espaces :

```
(define (f x1 ...xN)
  s1
  ...
  sN)
```

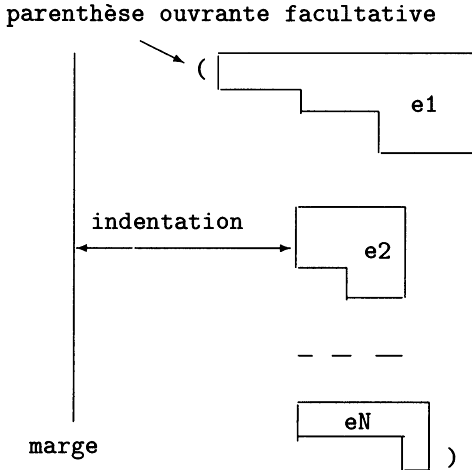
- Enfin les formes `set!` et `delay` sont écrites sans format spécial.

La fonction d'indentation

La fonction principale pour afficher une expression avec une indentation automatique s'appelle `pprint` (pour *pretty-print*). Elle prend en argument supplémentaire un entier qui indique de combien d'espaces on doit se déplacer à droite après un passage à la ligne. Cette fonction teste si elle a affaire à une forme spéciale, dans ce cas elle utilise la méthode prévue pour cette forme, sinon elle fait un affichage ordinaire.

```
(define (pprint exp indentation)
  (if (and (pair? exp)
           (symbol? (car exp)))
    (case (car exp)
      ((and or if map for-each)  (mode-and exp indentation))
      ((case define lambda)      (mode-case exp indentation))
      ((begin cond)              (mode-begin exp indentation))
      ((let let*)                (mode-let exp indentation))
      ((letrec)                  (mode-letrec exp indentation))
      ((do)                      (mode-do exp indentation))
      (else                      (write exp)))
    (write exp)))
```

Tous les modes d'affichage sont basés sur la fonction `modeV`. Elle réalise un affichage vertical d'une liste `(e1 e2 ...eN)`. Plus précisément, elle affiche, avec `pprint`, la première expression à partir de la position courante (si le booléen `Lpar?` vaut `#t`, on la fait précéder d'une parenthèse ouvrante). Puis on affiche en colonne verticale avec `pprint` les autres expressions. En représentant par des blocs l'affichage de chaque expression, l'appel de `(modeV L indentation Lpar?)` donne la disposition suivante :



Affichage de (e1 ...eN) en modeV

```
(define (modeV L indentation  Lpar? )
  (let ((indentation1 (if Lpar? (+ 1 indentation) indentation)))
    (if Lpar? (display "(" ))
    (pprint  (car L) indentation1)
    (for-each (lambda(e)
                (indentln indentation1)
                (pprint  e indentation1))
              (cdr L))
    (display ")" )))
```

Le décalage de 1, selon que l'on commence ou non par une parenthèse, oblige à utiliser la variable locale `indentation1`. La fonction `indentln` provoque un passage à la ligne et indente de l'entier passé en paramètre :

```
(define (indentln k )
  (newline)
  (display (make-string k #\space)))
```

Programmation de chaque mode

- Le mode and

On affiche le nom de la forme et son premier argument sur la même ligne puis les autres dessous en colonne avec le `modeV`. L'indentation est calculée pour qu'ils soient à la verticale du premier. Pour faire ce calcul, on ajoute à l'indentation courante le décalage provoqué par la parenthèse ouvrante, le nom de la forme et l'espace qui la suit.

```
(define (mode-and   exp indent)
  (let ((nom-forme (car exp)))
```

```
    (display-all "(" nom-forme " ")
    (modeV (cdr exp)
      (+ 2 indent (string-length (symbol->string nom-forme))) #f)))
```

- Le mode case

Il est pratiquement identique au précédent. L'alignement vertical ne se fait plus sous le premier argument mais en retrait de 2 par rapport au début de l'expression.

```
(define (mode-case  exp indent)
  (display-all "(" (car exp) " " (cadr exp))
  (indentln (+ indent 2))
  (modeV (cddr exp)  (+ indent 2)  #f))
```

- Le mode begin

C'est exactement le mode vertical :

```
(define (mode-begin exp indent)
  (modeV  exp  (+ indent 2)  #t))
```

- Le mode let

On affiche les liaisons en mode vertical comme le mode and et le corps en mode vertical comme le mode case :

```
(define (mode-let exp indent)
  (let ((nom-forme (car exp)))
    (display-all "(" nom-forme " ")
    (modeV (cadr exp)
           (+ 2 indent (string-length (symbol->string nom-forme))) #t)
    (indentln (+ indent 2))
    (modeV (cddr exp)  (+ indent 2)  #f)))
```

- Le mode do

On affiche les définitions des variables locales comme les liaisons du mode `let`, puis on affiche la partie test avec un décalage de un de moins et enfin le corps est traité comme le corps du `let` :

```
(define (mode-do exp indent)
  (display "(do " )
  (modeV (cadr exp)(+ 4 indent) #t )
  (indentln (+ indent 4))
  (write  (caddr exp))
  (indentln (+ indent 2))
  (modeV  (cdddr exp) (+ indent 2)  #f))
```

- Le mode letrec

On procède comme avec le mode `let`, mais on applique `pprint` à la partie expression de chaque définition de variable locale.

```
(define (mode-letrec exp indent)
  (display "(letrec (" )
  (let ((liaisons (cadr exp))
        (Lcorps (cddr exp)))
    (for-each (lambda(liaison)
                (display-all "(" (car liaison) " ")
                (pprint  (cadr liaison)
                         (+ 11 indent
                            (string-length (symbol->string (car liaison))))
                (display ")")
                (indentln (+ indent 9)))
              liaisons)
   (display ")")
   (indentln (+ indent 2))
   (modeV (cddr exp)  (+ indent 2)  #f)))
```

Testons notre fonction pprint sur l'expression (artificielle) suivante :

```
? (pprint '(define (f u v w) (if u (cond ((zero? u) a b)
                                         ((= v w) c) (else d))
                             (letrec ((g (lambda(x)(and a b c)))
                                      (h (lambda(y)
                                        (let ((i 1)(j 2))(+ i j y)))))
     (and (f a) (g b))))) 0)


(define (f u v w)
  (if u
      (cond
        ((zero? u) a b)
        ((= v w) c)
        (else d))
      (letrec ((g (lambda (x)
                    (and a
                         b
                         c)))
               (h (lambda (y)
                    (let ((i 1)
                          (j 2))
                      (+ i j y))))
              )
        (and (f a)
             (g b)))))
```

Enfin, pour afficher à l'écran l'indentation automatique d'un fichier de S-expressions, on utilise la fonction pprint-file qui applique pprint à chaque S-expression.

```
(define (pprint-file  source)
  (call-with-input-file source
    (lambda(input-flux)
      (do ((exp (read input-flux) (read input-flux)))
          ((eof-object? exp) )
```

```
(newline)(newline)
(pprint exp 0)))))
```

Exercice 12 *Modifier les fonctions précédentes pour écrire la version indentée d'un fichier dans un autre fichier.*

Architecture du programme d'indentation de S-expressions

`(pprint-file source)`
Pour afficher de façon indentée toutes les S-expressions d'un fichier.

> `(pprint exp indentation)`
> Affiche de façon indentée une S-expression, avec
> un retrait donné en cas de passage à la ligne.
>
> > `(mode-and exp indentation)`
> > Affiche une expression dans le style du `and`.
> >
> > `(mode-case exp indentation)`
> > Affiche une expression dans le style du `case`.
> >
> > `(mode-begin exp indentation)`
> > Affiche une expression dans le style du `begin`.
> >
> > `(mode-let exp indentation)`
> > Affiche une expression dans le style du `let`.
> >
> > `(mode-letrec exp indentation)`
> > affiche de façon indentée un `letrec`.
> >
> > A `(mode-do exp indentation)`
> > Affiche de façon indentée un `do`.
> >
> > > `(modeV Liste indentation Lpar? )`
> > > Affiche en colonne et de façon indentée chaque
> > > élément de la liste avec un retrait donné en cas
> > > de passage à la ligne. Si le booléen `Lpar?` vaut `#t`,
> > > on commence par afficher une parenthèse ouvrante.
> > >
> > > `(indentln k )`
> > > Passe à la ligne suivante et déplace le curseur
> > > de `k` espaces à droite.

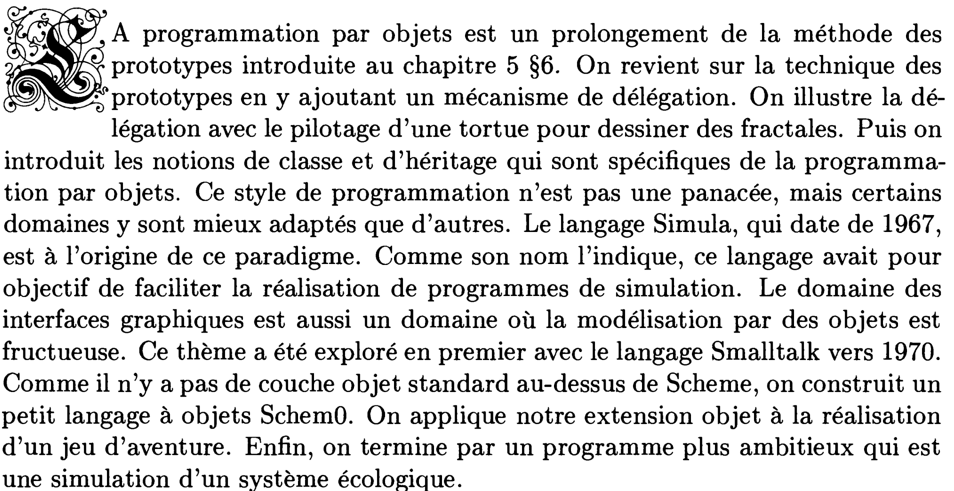
10.9 De la lecture

Les flots et leurs applications sont présentés dans [Bur75], ils sont aussi traités dans [ASS89, SF90].

On trouvera dans la Slib [Jaf] un système plus complet pour l'indentation des S-expressions. Cette question est aussi étudiée dans [SJ93].

Chapitre 11

Programmation par objets



LA programmation par objets est un prolongement de la méthode des prototypes introduite au chapitre 5 §6. On revient sur la technique des prototypes en y ajoutant un mécanisme de délégation. On illustre la délégation avec le pilotage d'une tortue pour dessiner des fractales. Puis on introduit les notions de classe et d'héritage qui sont spécifiques de la programmation par objets. Ce style de programmation n'est pas une panacée, mais certains domaines y sont mieux adaptés que d'autres. Le langage Simula, qui date de 1967, est à l'origine de ce paradigme. Comme son nom l'indique, ce langage avait pour objectif de faciliter la réalisation de programmes de simulation. Le domaine des interfaces graphiques est aussi un domaine où la modélisation par des objets est fructueuse. Ce thème a été exploré en premier avec le langage Smalltalk vers 1970. Comme il n'y a pas de couche objet standard au-dessus de Scheme, on construit un petit langage à objets Schem0. On applique notre extension objet à la réalisation d'un jeu d'aventure. Enfin, on termine par un programme plus ambitieux qui est une simulation d'un système écologique.

11.1 Principe de la programmation par objets

Si l'on doit définir des algorithmes complexes sur des objets simples, il est naturel d'organiser la programmation autour des aspects algorithmiques. En revanche, si l'on doit manipuler une grande variété d'objets complexes, il peut être préférable d'organiser la programmation autour des structures de données. Le contrôle est alors décentralisé au niveau des structures de données. Pour employer une métaphore, on passe d'une organisation où l'administration centrale décide de tout à une organisation où chaque niveau délègue le plus possible de responsabilité au niveau local.

Approche classique versus approche par objets

Pour illustrer ces deux types d'approche, on prend l'exemple classique des figures géométriques. On considère différents types de figures : `triangle, rectangle, carré, cercle,` ...
Si l'on adopte une approche classique, on écrira les fonctions : `perimetre, aire, afficher, ...` sur le modèle suivant :

```
(define (perimetre figure)
  (case (type-of figure)
    ((rectangle) (* 2 (+ (longueur figure)(largeur figure))))
    ((triangle)  (+ (coteAB figure)(coteBC figure)(coteCA figure)))
    ((cercle)    (* 2 PI (rayon figure)))
    ...
    (else (*erreur* " methode de calcul du perimetre  non connue "))))
```

Les informations concernant chaque type de figures sont disséminées dans diverses fonctions et si l'on veut ajouter un nouveau type de figure, il faudra *modifier* toutes ces fonctions.
Si l'on adopte une approche dirigée par les données, on regroupe pour chaque type de figure les fonctions qui la concernent, ce sont les concepts de *classe* et de *méthode* :

classe rectangle champs : `longueur, largeur`

méthodes : `périmètre, aire, affichage,` ...

classe triangle champs : `coteAB, coteBC, coteCA`

méthodes : `périmètre, aire, affichage,` ...

classe cercle champs : rayon

méthodes : `périmètre, aire, affichage ,...`

...

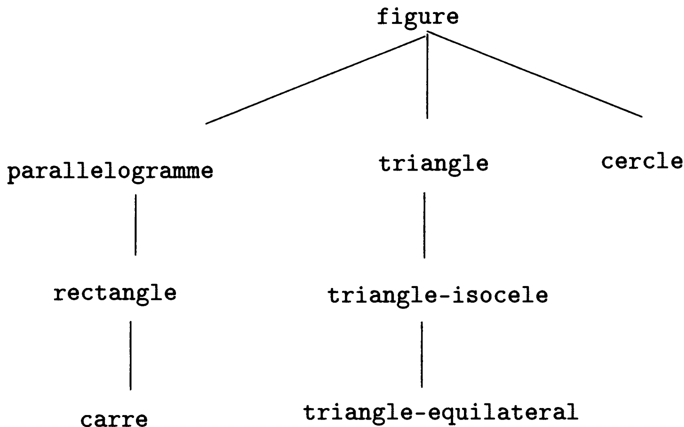
Les classes servent à définir les modèles généraux qui seront réalisés (on dit instanciés) en des objets spécifiques. Aussi, les méthodes et les noms des champs sont attachés aux classes mais les valeurs des champs sont normalement attachées aux objets.
On expliquera au §5 comment on crée un objet par instanciation de la classe qui lui sert de modèle.

Méthodes et surcharge

Si l'on demande le périmètre d'une figure, la méthode effectivement utilisée dépendra du type de la figure mais c'est le système objet, et non le programmeur, qui réalisera l'aiguillage. On dit que la fonction périmètre est *surchargée*, pour signifier que sous un même identificateur se cache des fonctions différentes. On dit aussi qu'elle présente un polymorphisme ad hoc. On peut voir la fonction `display` de Scheme comme un exemple de surcharge : la méthode pour imprimer un vecteur n'est pas la même que celle utilisée pour afficher un objet de type chaîne ou de type liste.

Classes et héritage

Si l'on considère la classe des triangles isocèles, on a envie de dire que certaines fonctionnalités se passent comme pour les triangles quelconques. On dit que la classe `triangle-isocele` *hérite* de la classe `triangle`. Par exemple, on peut hériter de la méthode du calcul du périmètre. De façon générale, on a une hiérarchie de classes ; une classe fille hérite de toutes les méthodes de sa mère, mais elle peut en masquer avec des méthodes plus spécifiques de même nom. Par exemple, si l'on introduit la classe des carrés comme classe fille des rectangles, on peut utiliser la fonction périmètre des rectangles ou bien la cacher par la méthode qui consiste à multiplier le côté par 4.
On définit de proche en proche toute une hiérarchie de classes. Dans le cas des figures, on peut représenter la relation d'héritage par un arbre :



Arbre d'héritage de figures géométriques

Décomposition en classes et évolutivité d'un logiciel

Un des avantages de la structuration en classes est de permettre d'ajouter un nouveau type de figure (c'est-à-dire une nouvelle classe) sans avoir à modifier une seule fonction existante. C'est une qualité importante, pour le génie logiciel, de pouvoir étendre un logiciel sans avoir à aller rechercher dans le code toutes les utilisations de certaines fonctions.
Les classes présentent aussi l'avantage de l'encapsulation : les objets ne sont manipulables que par les méthodes, on peut tout ignorer de l'implantation utilisée. Comme on l'a vu avec les prototypes, la présence de modificateurs fait que les objets sont des structures mutables, aussi la programmation par objets a souvent un fort parfum impératif mais avec les sécurités de l'encapsulation.

En résumé, on peut dire que ce qui caractérise la programmation par objets est la combinaison des notions de méthodes, de classes et d'héritage.

Avant de construire un petit langage à objets, on complète la notion de programmation par prototypes avec une dimension d'héritage, ou plus précisément de délégation.

11.2 Délégation et prototypes

On a vu que la technique des prototypes permettait de créer des structures composées et mutables.

Prototypes et fonction send

On ne peut les manipuler que par envoi de messages. Un schéma général possible est :

```
(define (make-prototype  init1  init2 . . .)
  (let ((champ1 init1)
        (champ2 init2)
        ....
        )
    (lambda (message . Largts)
      (case  message
        ((message1)  execution de la methode message1 )
        ((message2)  execution de la methode message2 )
        ...
        (else (*erreur* un message d'erreur si le message n'est pas prevu))
```

Les champs d'un prototype sont représentés par des variables locales du `let` mais on peut aussi utiliser des paramètres formels de la fonction de création.
Un prototype est une fonction, aussi pour exécuter une méthode avec un prototype, on évalue :

```
(un-prototype  'nom-methode  arg1 ...)
```

Cependant, il est fréquent d'utiliser une syntaxe plus conforme à la métaphore de l'envoi de message, pour cela on définit la fonction `send` :

```
(define (send  un-prototype  message . Largts)
  (apply un-prototype (cons message Largts)))
```

Délégation

Utilisons les prototypes pour représenter les couples de valeurs Scheme. On définit une fonction de création de couples, elle prend en arguments les valeurs initiales des composantes :

```
(define (make-couple x0 y0)
  (let ((x x0)
        (y y0))
    (lambda (message . Largts)
      (case   message
```

```
        ((type) 'couple)
        ((x) x)
        ((y) y)
        ((set-x!) (set! x (car Largts)))
        ((set-y!) (set! y (car Largts)))
        ((afficher) (display-all "x:" x "  y:" y))
        (else (*erreur*  "message inconnu : "   message)))))

? (define  c  (make-couple 3  '(b c)))
? (send c 'type) -> couple
? (send c 'set-x!  0)
? (send c 'afficher)
x:0  y:(b c)
```

Si les doublets mutables n'existaient pas en Scheme, les couples en seraient un succédané.

Modélisons maintenant les points du plan $x0y$, c'est-à-dire les couples de coordonnées (x, y) avec en plus des méthodes pour translater ou faire subir une rotation à un point. Bien sûr, on peut recopier l'implantation des couples et y ajouter d'autres méthodes. Mais il est plus satisfaisant de réutiliser ce qui est déjà connu pour les couples et de n'ajouter que les nouveautés ou différences. Pour cela, la fonction de création de `point` utilise un prototype `couple` en variable locale ; ce prototype permet de se dispenser de redonner la définition des méthodes communes avec les couples :

```
(define (make-point-2D x0 y0)
  (let ((couple (make-couple x0 y0)))
    (lambda (message . Largts)
      (case  message
        ((type) 'point2D)
        ((translater!) (couple 'set-x! (+ (couple 'x) (car Largts)))
                       (couple 'set-y! (+ (couple 'y) (cadr Largts))))
        ((tourner!) (let* ((angle (car Largts))
                           (cosinus (cos angle))
                           (sinus   (sin angle))
                           (x-old   (couple 'x))
                           (y-old   (couple 'y)))
                      (couple 'set-x! (+ (* x-old cosinus)(* y-old (- sinus)))
                      (couple 'set-y! (+ (* x-old sinus)(* y-old cosinus)))))
        ((afficher) (display " point ")(couple 'afficher))
        (else (deleguer message couple Largts))))))
```

Les méthodes pour translater et faire tourner un point font appel aux méthodes du prototype `couple` pour lire et modifier les composantes. La méthode `afficher` combine un affichage direct du type avec un appel à la méthode `afficher` de `couple`. Les messages non prévus dans les points sont transmis au prototype `couple` par la fonction de *délégation*. Elle fait suivre le message au prototype `couple` ; on «hérite» ainsi des fonctions définies pour les couples.

```
(define (deleguer message prototype  Largts)
  (apply prototype (cons  message Largts)))
```

```
? (define p2 (make-point-2D 4 6))
? (send p2 'translater! 2 2)
? (send p2 'afficher)
point x:6  y:8

? (send p2 'x) -> 6   ;; utilisation de la délégation
```

Exercice 1 *Définir une fonction pour calculer la distance de deux points du plan.*

De la même manière, la délégation va nous permettre de définir les points de l'espace avec un minimum de travail. Un point de l'espace peut être considéré comme un point du plan avec une composante supplémentaire : la hauteur `z`. On va donc associer à un point de l'espace, un point du plan (sa projection) et l'on déléguera à ce point du plan les opérations concernant les composantes `x` et `y`.

```
(define (make-point-3D x0 y0 z0)
  (let ((point2D (make-point-2D x0 y0))
        (z z0))
    (lambda (message . Largts)
      (case  message
        ((type) 'point3D)
        ((z) z)
        ((set-z!) (set! z (car Largts)))
        ((afficher) (point2D 'afficher)(display-all "  z:" z))
        (else (deleguer message point2D Largts))))))

(define p3 (make-point-3D 4 6 9))

? (send p3 'imprimer)
point x:4  y:6  z:9

? (send p3 'x) -> 4
```

Exercice 2 *Définir une fonction qui calcule la distance de deux points de l'espace.*

Exercice 3 *Définir les piles bornées à partir de la définition des piles données au chapitre 7 §2.*

Fonction `self`

On va modifier la méthode de création d'un prototype pour permettre aux prototypes de s'envoyer des messages à *eux-mêmes*. On appelle `self` la lambda expression construite. De plus, on modifie la fonction d'affichage : on mentionne le type au début de l'affichage.

```
(define (make-couple x0 y0)
  (let ((x x0)
        (y y0))
    (let ((self
```

```
            (lambda (message . Largts)
              (case  message
                ((type) 'couple)
                ((x) x)
                ((y) y)
                ((set-x!) (set! x (car Largts)))
                ((set-y!) (set! y (car Largts)))
                ((afficher) (display-all "type couple  x:" x "  y:" y))
                (else (*erreur*  "message inconnu : "  message)))))
      self))
```

Mais cette définition n'est pas toujours satisfaisante. Par exemple, si l'on veut faire calculer le type *au moment de l'affichage*, on a envie de demander à `self` son type. Pour que `self` soit visible dans son corps, il faut utiliser un `letrec` :

```
(define (make-couple x0 y0)
  (let ((x x0)
        (y y0))
    (letrec ((self
              (lambda (message . Largts)
                (case  message
                  ((x) x)
                  ((y) y)
                  ((type) 'couple)
                  ((set-x!) (set! x (car Largts)))
                  ((set-y!) (set! y (car Largts)))
                  ((afficher)
                   (display-all "type " (self 'type) "  x:" x "  y:" y))
                  (else (*erreur*  "message inconnu : " message)))))
      self)))

? (define  c (make-couple 1 2))

? (send  c 'afficher)
type couple  x:1  y:2
```

Exercice 4 *Reprendre la définition des prototypes points avec l'utilisation de la fonction* `self`.

11.3 Langage Logo

Pilotage d'une tortue

Une illustration classique de la programmation avec des prototypes est la réalisation de graphiques en pilotant un crayon par des ordres relatifs. Ce style de dessins est inspiré du langage Logo qui, à l'origine, servait à commander un petit animal cybernétique en forme de tortue. A chaque instant le crayon possède une position et une direction d'avancement. Les ordres sont donnés en valeurs *relatives* :

- l'ordre `avance` prend en argument une longueur et a pour effet de faire avancer la tortue de cette longueur dans la direction courante ;

- l'ordre `tourne` prend en argument un angle et a pour effet de faire tourner la tortue de cet angle ;

- on permet aussi de lever ou de baisser le crayon afin de permettre de déplacer la tortue sans dessiner (par exemple, pour faire des lignes pointillées).

On repère la position de la tortue, dans une fenêtre graphique, par ses coordonnées cartésiennes x, y dans un repère $x0y$ avec l'axe des y *dirigé vers le bas*. Les coordonnées doivent être des entiers aussi la fonction `reel->entier` sert à transformer en entiers les valeurs réelles.
On représente une tortue par un prototype à quatre champs :

- les coordonnées entières `x` et `y`,

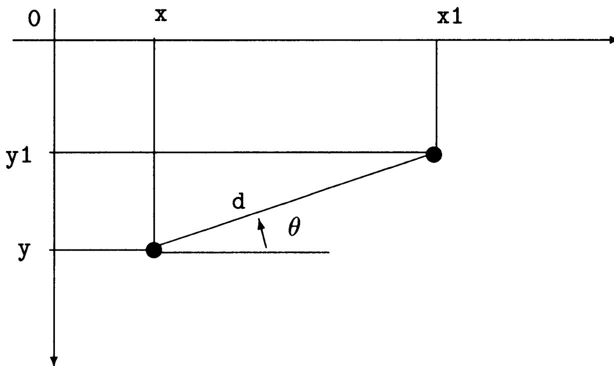
- l'angle `theta` (en radians) de la direction d'avancement avec l'axe Ox. Il est plus agréable de pouvoir donner l'ordre de tourner avec des angles exprimés en degrés. Comme les fonctions *sin* et *cos* utilisent la mesure des angles en radians, on définit une fonction `degre->radian` pour convertir en radians les angles exprimés en degrés,

- un booléen `baisse?` qui est vrai si le crayon écrit et faux sinon.

L'ordre d'avancer prend en argument un nombre `d` (négatif si on veut reculer) ; on calcule les nouvelles coordonnées exactes par les formules évidentes :

$$x_1 = x + d\ cos(\theta) \qquad y_1 = y - d\ sin(\theta)$$

D'où la nouvelle position `x1` , `y1` en prenant les parties entières.



L'ordre `tourne` consiste simplement à incrémenter l'angle `theta` de la valeur (en degré) passée en paramètre.
L'ordre `baisse` prend en argument un booléen qui sera affecté à la variable `baisser?`

L'état de la tortue est défini par les valeurs des quatre champs: `x, y, theta, baisse?`. On lit cet état par l'ordre `Letat` qui retourne la liste des valeurs de ces

champs. Enfin, pour modifier d'un coup cet état, on ajoute l'ordre `changer-etat` qui prend en arguments les nouvelles valeurs de ces champs.

Pour dessiner dans une fenêtre, on a besoin de deux primitives graphiques :

- `(move-to x y)` pour placer le crayon au point de coordonnées entières x, y,
- `(line-to x y)` pour tracer un segment de droite du point courant au point de coordonnées x, y.

Prototype de tortue

La fonction de création d'un prototype de tortue s'écrit naturellement :

```
(define  (make-tortue x y theta-degre baisse?)
  (let ((theta (degre->radian theta-degre))
    (lambda (message . Larg)
      (case message
        ((avance)
         (let((x1 (reel->entier (+ x (* (car Larg) (cos theta)))))
              (y1 (reel->entier (- y (* (car Larg) (sin theta))))))
           (if baisse?
             (line-to x1 y1)
             (move-to x1 y1))
           (set! x x1)(set!  y y1)))
        ((Tourne)  (set! theta  (+ theta (degre->radian  (car Larg)))))
        ((baisse)  (set! baisse? (car Larg)))
        ((Letat)   (list x y theta baisse?))
        ((changer-etat) (set! x (car Larg))
                        (set! y (cadr Larg))
                        (set! theta (caddr Larg))
                        (set! baisser? (cadddr Larg))
                        (move-to x y))
         ))))
```

avec les deux fonctions de conversion :

```
(define (reel->entier nb)
  (inexact->exact (round nb)))

(define (degre->radian degre)
  (/ (*  degre 3.14159) 180))
```

Dessiner avec une tortue

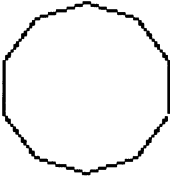
Pour utiliser une tortue, on commence par créer une fenêtre graphique avec le système de fenêtrage disponible dans notre environnement. Ensuite, on utilise `make-tortue` pour créer dans cette fenêtre un prototype tortue de nom `*tortue*` avec son crayon en position d'écriture et placé au centre de la fenêtre.

Commençons par des dessins de polygones réguliers. Un polygone à n côtés s'obtient en répétant n fois l'action d'avancer de la longueur du côté puis de tourner de$360/n$ degrés.

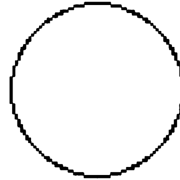
```
(define (polygone  cote nb-cotes tortue)
  (let ((alpha (/ 360 nb-cotes)))
    (do ((i 0 (+ 1 i)))
        ((= i nb-cotes))
      (tortue 'avance cote)
      (tortue 'tourne  alpha))))
```

Pour $n = 10$ on obtient un décagone et pour $n = 40$ on a déjà une bonne approximation d'un cercle :

```
? (polygone 30  10 *tortue*)
? (polygone 10  40 *tortue*)
```



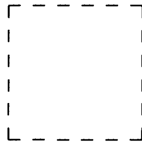
Polygone avec n=10



Polygone avec n=30

Pour tracer un carré en pointillé, il suffit de tracer chaque côté en alternant les positions levées et baissées du crayon :

```
(define (carre-pointille cote tortue)
  (do ((i 0 (+ 1 i)))
      ((= i 4))
    (do ((k 0 (+ 1 k)))
        ((= k 10))
      (tortue 'avance (quotient cote 10))
      (tortue 'baisse (odd? k)))
    (tortue 'tourne 90)))

? (carre-pointille 50 *tortue*)
```



Courbe de von Koch

Si l'on combine le dessin et la programmation récursive, on obtient des courbes, dites *fractales*. On commence par la courbe de von Koch. Elle est basée sur le motif suivant :

angle de 60 degrés

pas pas

pas pas

3 pas

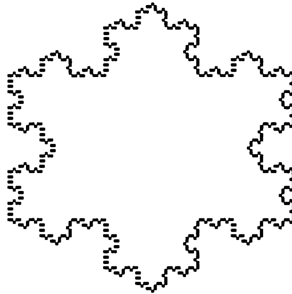
On trace ce motif de gauche à droite avec les ordres :
avancer de 1 pas ; tourner 60 degrés ; avancer de 1 pas ; tourner de -120 degrés ; avancer de 1 pas ; tourner de 60 degrés ; avancer de 1 pas.
Le principe du tracé récursif consiste à dessiner chaque côté en appliquant le même processus mais avec un `pas` d'un tiers du côté à parcourir car le dessin s'étend sur trois fois le pas. Le motif de base correspond à la courbe de von Koch d'ordre 1, le motif d'ordre `n` s'obtient en traçant chacun de ses côtés avec un motif d'ordre `n-1`.

```
(define (motif-vonKoch n pas tortue)
  (if (zero? n)
    (tortue 'avance pas)
    (begin
      (motif-vonKoch (- n 1) (/ pas 3) tortue)
      (tortue 'tourne 60)
      (motif-vonKoch (- n 1) (/ pas 3) tortue)
      (tortue 'tourne -120)
      (motif-vonKoch (- n 1) (/ pas 3) tortue)
      (tortue 'tourne 60)
      (motif-vonKoch (- n 1) (/ pas 3) tortue)
      )))
```

Pour faire un dessin rappelant les flocons de neige, on dispose trois motifs de von Koch comme les côtés d'un triangle équilatéral :

```
(define (flocon n cote tortue)
  (letrec ((motif-vonKoch
             (lambda (n pas)
               (if (zero? n)
                 (tortue 'avance pas)
                 (begin
                   (motif-vonKoch (- n 1) (/ pas 3))
                   (tortue 'tourne 60)
                   (motif-vonKoch (- n 1) (/ pas 3))
```

```
                (tortue 'tourne -120)
                (motif-vonKoch (- n 1) (/ pas 3))
                (tortue 'tourne 60)
                (motif-vonKoch (- n 1) (/ pas 3))
                )))))
    (do ((i 0 (+ i 1)))
        ((= i 3))
      (motif-vonKoch n cote)
      (tortue 'tourne -120))))

? (flocon  4 160 *tortue*)
```



Flocon

11.4 Application aux systèmes de Lindenmayer

Dans un organisme cellulaire le développement des cellules reproduit à chaque étape le même schéma de croissance. Le biologiste A. Lindenmayer a eu l'idée d'introduire un langage pour décrire la structure de chaque génération de cellules.

Système *L*

Décrivons le motif de von Koch avec son langage. Appelons `F` l'action d'avancer d'un pas, notons `+` l'action de tourner de 60 degrés et `-` celle de tourner de -60 degrés. Avec ces notations, le motif de von Koch est décrit par la suite des actions :

```
F + F - - F + F
```

Comme chaque pas est à son tour constitué d'un motif, l'idée de Lindenmayer est de décrire la récursion par la règle :

```
F --> F + F - - F + F        (*)
```

qui signifie que pour obtenir la génération suivante, il faut remplacer simultanément *tous* les `F` par le membre de droite.
Pour modéliser les embranchements que l'on rencontre fréquemment dans la nature (branches d'arbres, ...), il introduit une opération de branchement notée entre

crochets. De façon plus précise, une expression `[ ...]` provoque la sauvegarde de l'état de la tortue puis on exécute le contenu entre crochets et à la fin on restaure l'état sauvegardé.
Dans le cas général, le symbole + représente l'action de tourner d'un certain angle à gauche et le symbole - l'action de tourner d'un autre angle à droite. On introduit aussi le symbole `f` pour désigner l'action d'avancer sans dessiner. On arrive ainsi à la notion de systèmes de Lindenmayer, c'est une description de motifs par des règles du type (*) avec les symboles : `F , f, + , - , [ , ]`. Exemples :

- le système `F -> F [+ F] F [- F]`

  décrit un motif avec deux embranchements (un à gauche puis un à droite) séparés par une longueur de motif. Comme on le verra, son dessin rappelle la structure d'une herbe ;

- le système `F -> F [+ F] [- F]`

  décrit un motif avec deux embranchements en opposition, son dessin rappelle la structure d'un arbre.

Pour visualiser ces motifs, il faut généraliser notre notion de tortue. On la dote d'une pile pour se rappeler son état à l'entrée d'un branchement.

Tortue à pile et fractales

On utilise le principe de la délégation pour définir une nouvelle catégorie de tortue (dite à pile). Une tortue à pile comporte une pile locale et répond à deux nouveaux messages. Le message `empile` consiste à empiler l'état courant et le message `depile` consiste à remplacer l'état courant par celui qui est en sommet de pile et à dépiler. Les autres messages sont délégués à une tortue du type précédent.

```
(define  (make-tortue-pile x y theta-degre baisse?)
  (let ((pile '())
        (tortue (make-tortue x y theta-degre baisse?))
    (lambda (message . Larg)
      (case message
        ((empile)(set! pile (cons (tortue 'Letat) pile)))
        ((depile)
         (let ((etat  (car pile))
           (set! pile (cdr pile))
           (apply tortue  (cons 'changer-etat etat)) ))
        (else (deleguer message tortue Larg))))))
```

Appliquons ceci au tracé des deux motifs précédents.
La structure du programme suit strictement la règle du motif à dessiner. On appelle `deviationG` et `deviationD` les valeurs en degrés des angles des rotations + et -. Le passage d'une génération à la suivante utilise aussi un coefficient de réduction (il était de 1/3 pour le motif de von Koch).
On appelle `herbe` la fonction qui dessine le motif `F -> F [+ F] F [- F]` :

```
(define (herbe n pas tortue deviationG deviationD coeff-reduction)
  (letrec ((herbe-aux
             (lambda (n pas)
               (if (zero? n)
                 (tortue 'avance pas)
                 (begin
                   (herbe-aux (- n 1) (* coeff-reduction pas))
                   (tortue 'empile)
                   (tortue 'tourne deviationG)
                   (herbe-aux (- n 1) (* coeff-reduction pas))
                   (tortue 'depile)
                   (herbe-aux (- n 1) (* coeff-reduction pas))
                   (tortue 'empile)
                   (tortue 'tourne deviationD)
                   (herbe-aux (- n 1) (* coeff-reduction pas))
                   (tortue 'depile)
                   (herbe-aux (- n 1) (* coeff-reduction pas))
                   )))))
    (herbe-aux  n pas)))

? (herbe 4 300  *tortue*  15 -18 1/3)
```

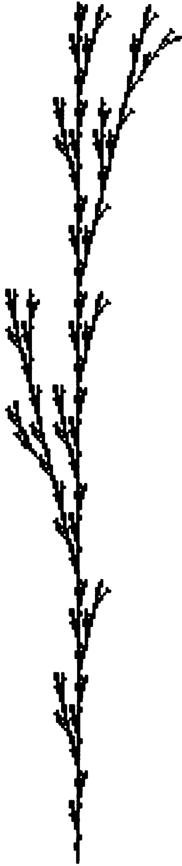
On appelle `arbre` la fonction qui dessine le motif `F -> F [+ F] [- F]` :

```
(define (arbre n pas tortue deviationG deviationD coeff-reduction)
  (letrec ((arbre-aux
             (lambda (n pas)
               (if (zero? n)
                 (tortue 'avance pas)
                 (begin
                   (arbre-aux (- n 1) (* coeff-reduction pas))
                   (tortue 'empile)
                   (tortue 'tourne deviationG)
                   (arbre-aux (- n 1) (* coeff-reduction pas))
                   (tortue 'depile)
                   (tortue 'empile)
                   (tortue 'tourne deviationD)
                   (arbre-aux (- n 1)(* coeff-reduction pas))
                   (tortue 'depile)
                   )))))
    (arbre-aux n pas)))
```

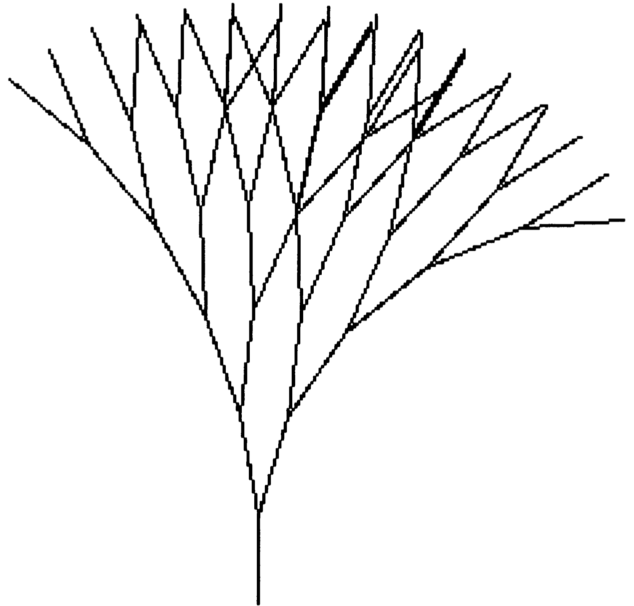
On teste avec une déviation droite un peu plus grande que la gauche, ce qui a pour effet de faire pencher l'arbre à droite.

```
? (arbre 5 500 *tortue* 10 -17 0.6)
```

Si l'on désire créer des paysages, on peut introduire de la diversité en tirant au hasard les valeurs des paramètres de ces fonctions.



Herbe fractale



Arbre fractal

11.5 Une extension objet de Scheme : SchemO

La méthode des prototypes n'utilise pas la notion de classe, car chaque prototype porte en lui-même la définition de ses méthodes. Une fois qu'un prototype est créé, il n'est plus possible de lui ajouter une nouvelle méthode. Au contraire, la structuration en classes va permettre d'avoir un comportement plus dynamique.
La notion de classe est une espèce de généralisation de la notion de type ; une classe ne représente pas un objet mais une catégorie d'objets ayant la même structure et partageant les mêmes méthodes. Il y a une grande diversité de systèmes à objets : des langages statiques, comme Eiffel, qui permettent un maximum de vérification à la compilation ou des langages dynamiques, comme CLOS, qui facilitent un développement incrémental des logiciels.
Conformément à l'esprit de Scheme, on va définir un langage objet qui se place dans la deuxième catégorie. Comme en informatique tout langage porte un nom, on baptise SchemO cette petite extension objet de Scheme. On commence par

expliquer son utilisation en reprenant l'exemple des points. On décrira ultérieurement l'implantation de SchemO.

Définition des classes et création d'objets en SchemO

Lors de la définition d'une classe, on ne précise que les noms des champs. Il y a création automatique des méthodes pour accéder ou modifier les champs. Les autres méthodes seront ajoutées après coup par l'utilisateur.

Par exemple, on définit la classe `couple` ayant les deux champs `x` et `y` par :

```
(make-class 'couple  #f  'x  'y)
```

L'argument `#f` signifie qu'elle n'hérite pas d'une autre classe. La syntaxe générale d'une définition de classe est :

```
(make-class  'nomClasse  'nomClasseMere  'champ1   'champ2 ...)
```

où *nomClasseMere* est `#f` ou le nom de la classe dont on hérite.

Pour créer un objet de la classe `couple`, on utilise la fonction `make-instance` :

```
(define  p0  (make-instance  'couple  'x  2   'y   (* 4  5) ))
```

La variable `p0` désigne un objet de la classe `couple` dont le champ `x` est initialisé avec 2 et `y` avec 20.
Le rappel du nom du champ devant sa valeur permet de les écrire dans n'importe quel ordre. On n'est pas obligé d'initialiser un champ, on ne mentionne dans `make-instance` que les champs que l'on initialise. La syntaxe générale est :

```
(make-instance  'nomClasse   'champj1  s1  ...   'champjk  sk)
```

Accesseurs et modificateurs

Les méthodes pour accéder ou modifier les champs sont *automatiquement* créées avec la classe. Pour accéder à la valeur du champ `champj`, on envoie le message `champj` et pour modifier sa valeur, on envoie le message `set-champj!`.

```
? (send  p0 'x) -> 2
? (send  p0 'set-x!   10)
? (send  p0 'x) -> 10
```

Définition des méthodes

Pour ajouter une méthode à une classe, on utilise la forme `defmethod` :

```
(defmethod  afficher  couple  ()
  (display-all "x:" (send self 'x)  "  y:" (send self 'y)))
```

On peut appliquer la méthode `afficher` au point `p0` que l'on avait créé :

```
? (send p0 'afficher)
x:10 y:20
```

La syntaxe pour définir une méthode est voisine de celle utilisée pour définir une fonction :

```
(defmethod nomMethode nomClasse (x1 ... xN)
  corps)
```

Dans le *corps*, la variable `self` désigne l'objet auquel sera appliqué la méthode. Après évaluation de cette définition, la méthode est ajoutée à l'ensemble des méthodes directement disponibles pour la classe indiquée (ou bien remplace une méthode existante de même nom).
L'appel d'une méthode se fait aussi par envoi d'un message à un objet :

```
(send objet 'nomMethode arg1 ...)
```

Si la méthode n'est pas trouvée dans la liste des méthodes de la classe de l'objet alors on cherche dans la classe mère et ainsi de suite ... Si l'on ne trouve rien, un message d'erreur est affiché.
Continuons l'illustration avec l'ajout de la classe `point-2D` comme classe fille de la classe `couple`. Elle hérite des champs de la classe couple :

```
? (make-class 'point-2D 'couple)
```

Créons un objet `point-2D` en initialisant seulement le champ `x` :

```
(define p2 (make-instance 'point-2D  'x 2))

? (send p2 'x) -> 2
```

Pour initialiser l'autre champ, on utilise le modificateur `set-y!` :

```
? (send p2  'set-y!  5)
? (send p2 'y ) -> 5
```

On ajoute des méthodes pour translater et tourner :

```
(defmethod  translater  point-2D (Dx Dy)
  (send self 'set-x! (+ Dx (send self 'x)))
  (send self 'set-y! (+ Dy (send self 'y))))

(defmethod  tourner  point-2D (angle)
  (let ((cosinus (cos angle))
        (sinus   (sin angle))
        (x-old   (send self 'x))
        (y-old   (send self 'y)))
    (send self 'set-x! (+ (* x-old cosinus)(* y-old (- sinus))))
    (send self 'set-y! (+ (* x-old sinus)(* y-old cosinus)))))

? (send p2 'translater  1 1)
? (send p2 'afficher)
x:3 y:6
```

L'utilisation de send-next

On a hérité de la méthode d'affichage de la classe couple. On souhaite maintenant particulariser cette méthode par l'affichage de la chaîne "point" avant l'affichage des champs. Pour ce faire, on définit une méthode afficher pour la classe point-2D qui appellera ensuite la méthode afficher de la classe mère. D'où l'utilité d'introduire une fonction send-next qui permet d'appeler la première méthode afficher trouvée en remontant dans les classes ancêtres de point-2D. Ici, il suffit de remonter à la classe couple pour trouver une telle méthode.

```
(defmethod  afficher  point-2D ()
  (display "point  ")
  (send-next  self 'point-2D 'afficher))

? (send p2 'afficher)
point  x:3 y:6
```

La syntaxe de send-next est voisine de celle de send :

```
(send-next  objet 'nomClasse 'nomMethode  arg1 . . .))
```

Elle effectue la recherche d'une méthode de nom *nomMethode* en remontant les classes ancêtres de la classe *nomClasse* ; si une telle méthode existe, elle est appliquée sinon il y a un message d'erreur.

Pour illustrer une autre utilisation de la fonction send-next, introduisons la classe des points du plan situés sur la bissectrice $x = y$.

```
(make-class 'point-bissec 'point-2D)
```

Quand on modifie la valeur de la composante x, on désire que la valeur de y soit également modifiée de façon à conserver l'égalité $x = y$ (de même si on change y). Ceci nous conduit à redéfinir les méthodes set-x!, set-y! pour cette classe :

```
(defmethod  set-x! point-bissec (v)
  (send-next self 'point-bissec 'set-x! v)
  (send-next self 'point-bissec 'set-y! v))

(defmethod  set-y! point-bissec (v)
  (send-next self 'point-bissec 'set-y! v)
  (send-next self 'point-bissec 'set-x! v))
```

L'utilisation de send-next est cruciale, car si l'on avait posé :

```
(defmethod  set-x! point-bissec (v)
  (send self 'set-x! v)
  (send self 'set-y! v))
```

on aurait créé un appel récursif de set-x! source d'un bouclage.

Résumé de la syntaxe de SchemO

Définition d'une classe.
`(make-class 'nomClasse 'nomClasseMere 'champ1 'champ2 ...)`

Création d'un objet.
`(make-instance 'nomClasse 'champj1 s1 ... 'champjk sk)`

Définition d'une méthode.
`(defmethod nomMethode nomClasse (x1 ... xN) corps)`

Lecture d'un champ.
`(send objet 'champ)`

Modification d'un champ.
`(send objet 'set-champ! arg)`

Appel d'une méthode.
`(send objet 'nomMethode arg1 ...)`

Appel d'une méthode à partir de la classe mère d'une classe donnée.
`(send-next objet 'nomClasse 'nomMethode arg1 ...)`

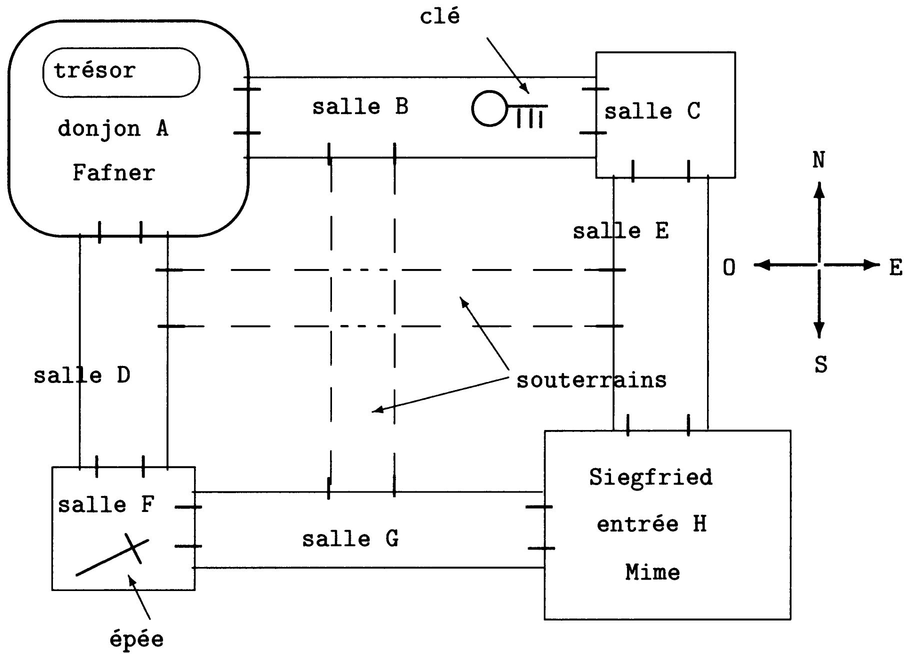
11.6 Jeu donjon et dragon

Les jeux sont propices à une programmation par objets. En effet, on utilise de nombreuses catégories d'éléments et l'état de chacun de ces éléments peut évoluer durant la partie.
On va illustrer cet aspect avec un petit jeu dans la lignée de donjon et dragon.

Règles du jeu : Siegfried et le dragon

L'aventure se passe dans un château. Le héros Siegfried aidé de son mentor Mime tente de récupérer un trésor rangé dans un coffre situé dans le donjon du château. Ce trésor est gardé par le dragon Fafner qui se déplace aléatoirement dans les salles du château et qui dévore les êtres qu'il rencontre dans une pièce. Seul Siegfried, s'il est en possession de son épée, peut tuer Fafner.

Au début de la partie Fafner est dans la salle (A) du trésor et Siegfried et Mime sont dans la pièce d'entrée (H) du château (voir plan). Le système place de façon aléatoire la clé du coffre dans une des huit salles et de même pour l'épée (ces salles ne sont pas connues de nos héros). Par ailleurs, Siegfried et Mime disposent chacun d'une énergie vitale initiale. On choisit aléatoirement chaque énergie de sorte qu'elle soit comprise entre 6 et 10 et de somme totale 16.



Plan du château et une position initiale des éléments

A chaque coup, le dragon tire au sort la direction dans laquelle il va se déplacer ou bien dormir. Dans le cas d'un déplacement, il s'effectue d'une case dans la direction donnée (il reste sur place s'il n'y a pas de porte dans la direction choisie). S'il entre dans une salle où se trouve un personnage, alors il le mange à moins que ce soit Siegfried muni de son épée, dans ce cas, c'est Fafner qui est tué.
Ensuite, le joueur choisit l'action à effectuer. Il doit indiquer le personnage (Siegfried ou Mime) concerné par l'action puis la nature de l'action. Cette action peut être de quatre sortes :

se déplacer : on doit préciser la direction : Nord, Sud, Est, Ouest. Le personnage est alors déplacé d'une case dans cette direction et son énergie personnelle est diminuée de 1. Si l'énergie du personnage devient nulle, il meurt d'épuisement.

voir : alors on nous informe des objets (clé, épée, trésor) présents dans la salle où l'on se trouve.

prendre : on doit préciser l'objet à prendre (on a une perte d'énergie de 1 si cette action n'est pas possible). Un personnage doit posséder la clé pour sortir le trésor du coffre. Il n'est pas exclu que Mime s'empare du trésor pour l'amener à Siegfried si ce dernier n'a plus beaucoup d'énergie.

abandonner : on peut abandonner à tout moment.

La partie est gagnée quand Siegfried est en possession du trésor et elle perdue si Siegfried meurt avant.

Définitions des classes

Voici la hiérarchie des classes utilisée pour modéliser notre jeu.

- Au sommet, on a la classe `element` avec un champ pour indiquer le nom de l'élément. On répartit les éléments en deux sous-classes : les éléments mobiles et les salles du château.

- La classe `salle` comporte quatre champs : `sud, nord, est, ouest`.

  La valeur d'un tel champ est la salle adjacente dans cette direction ou `#f` s'il n'y en a pas.

- La classe `element-mobile` comporte un champ `salle` pour désigner la salle contenant actuellement cet élément. La classe `element-mobile` comporte deux sous-classes : les choses et les créatures vivantes.

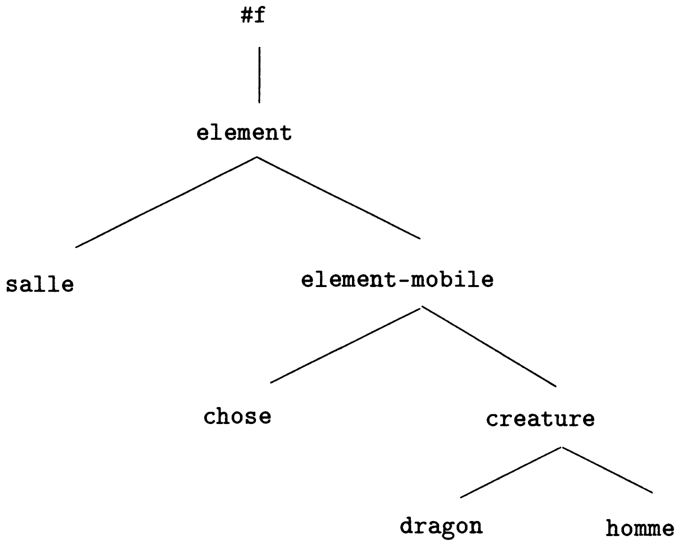
- La classe `creature` comporte le champ `vivant?` pour indiquer si la créature en question est encore vivante ou non.

- La classe `chose` comporte le champ `possede-par` pour indiquer la créature qui possède cette chose ou `#f` si elle n'est possession de personne.

- Enfin la classe `creature` comporte deux sous-classes : une classe pour les hommes et une classe pour les dragons.

- La classe `homme` comporte le champ `energie` qui contient la valeur de l'énergie vitale d'un homme.

D'où les définitions des classes qui traduisent cette hiérarchie :



Hiérarchie des classes

```
(make-class 'element #f 'nom)

(make-class 'salle 'element   'sud 'nord 'est 'ouest)

(make-class 'element-mobile   'element 'salle)

(make-class 'chose 'element-mobile 'possede-par)

(make-class 'creature  'element-mobile 'vivant?)

(make-class 'dragon 'creature)

(make-class 'homme  'creature 'energie)
```

Création des objets

On se donne des variables globales pour représenter les salles, les créatures et les objets.

```
(define sA '?)
(define sB '?)
(define sC '?)
(define sD '?)
(define sE '?)
(define sF '?)
(define sG '?)
(define sH '?)
```

```
(define Siegfried '?)
(define Mime      '?)
(define Fafner    '?)

(define tresor '?)
(define cle    '?)
(define epee   '?)
```

La fonction init sert à initialiser au début de chaque partie les valeurs de ces variables.

```
(define (init)
  (display-alln "creer les salles")      ;; les salles
  (set! sA (make-instance 'salle 'nom 'A))
  (set! sB (make-instance 'salle 'nom 'B))
  (set! sC (make-instance 'salle 'nom 'C))
  (set! sD (make-instance 'salle 'nom 'D))
  (set! sE (make-instance 'salle 'nom 'E))
  (set! sF (make-instance 'salle 'nom 'F))
  (set! sG (make-instance 'salle 'nom 'G))
  (set! sH (make-instance 'salle 'nom 'H))
  (set! *Lsalles* (list sA sB sC sD sE sF sG sH))

  (display-alln "creer les portes")      ;; les portes
  (for-each (lambda (salle Lportes)
              (for-each (lambda (message porte)
                          (send salle message porte))
                        '(set-sud! set-nord! set-est! set-ouest!)
                        Lportes))
            *Lsalles*
            (list (list sD #f sB #f)
                  (list sG #f sC sA)
                  (list sE #f #f sB)
                  (list sF sA sE #f)
                  (list sH sC #f sD)
                  (list #f sD sG #f)
                  (list #f sB sH sF)
                  (list #f sE #f sG)))

  (display-alln "creer les creatures")    ;; les créatures
  (set! Siegfried (make-instance 'homme 'nom 'Siegfried 'vivant? #t 'salle sH))
  (set! Mime      (make-instance 'homme 'nom 'Mime      'vivant? #t 'salle sH))
  (set! Fafner    (make-instance 'dragon 'nom 'Fafner   'vivant? #t 'salle sA))

  (display-alln "creer les objets")       ;; les objets
  (set! tresor (make-instance 'objet 'nom 'tresor 'possede-par #f 'salle sA))
  (set! cle    (make-instance 'objet 'nom 'cle     'possede-par #f))
  (set! epee   (make-instance 'objet 'nom 'Notung  'possede-par #f))

  ;; placer aléatoirement la clé et l'épée
  (let ((salle-cle  (list-ref *Lsalles* (random 8)))
```

```
        (salle-epee (list-ref *Lsalles* (random 8))))
    (send cle  'set-salle!   salle-cle)
    (send epee 'set-salle!   salle-epee))

  ;; initialiser les énergies de Siegfried et de Mine
  (let ((energie (random 5)))
    (send S 'set-energie! (+ energie 6))      ; énergie entre 6 et 10
    (send M 'set-energie! (- 10 energie))))  ; énergie entre 6 et 10
```

Définitions des méthodes

Pour modéliser les actions, on introduit les méthodes : `avance`, `mourir`, `prendre`, `combattre-dragon`, `affiche`.

La méthode `avance` concerne les créatures et prend en paramètre une direction. S'il y a une salle dans la direction donnée, on actualise la position de la créature et sinon on envoie un message d'erreur. Elle rend en résultat la nouvelle salle ou `#f` si l'on ne bouge pas.

```
(defmethod avance creature (direction)
  (let ((salle-suivante (send (send self 'salle) direction)))
    (if salle-suivante
      (send self 'set-salle! salle-suivante)
      (display-alln "pas de porte dans cette direction"))
    salle-suivante))
```

On complète cette méthode pour la classe `dragon`. Un dragon n'avance que s'il est vivant, dans ce cas on appelle la méthode de la classe mère pour actualiser sa salle. C'est un bon exemple d'utilisation de `send-next`. Enfin, si la nouvelle salle est déjà occupée par un homme, il y a combat.

```
(defmethod avance dragon (direction)
  (if (send self 'vivant?)
    (begin
      (send-next self 'dragon 'avance direction)
      (send siegfried 'combattre-dragon)
      (send mime 'combattre-dragon))
    (display-alln "le dragon est mort")))
```

On complète également cette méthode pour les hommes. Un homme n'avance que s'il est vivant, dans ce cas on appelle la méthode de la classe mère `creature` pour actualiser sa salle. Si l'on change effectivement de salle on doit également décrémenter l'énergie de l'homme et actualiser la salle de chaque objet possédé par un homme. Enfin, si l'homme rencontre le dragon, il y a combat.

```
(defmethod avance homme (direction)
  (if (send self 'vivant?)
    (let ((salle-homme (send-next self 'homme 'avance  direction)))
      (when salle-homme
            (send self 'decr-energie! 1)
```

```
          (for-each (lambda (chose)
                      (if (eq? self (send chose 'possede-par))
                        (send chose 'set-salle! salle-homme)))
                    (list epee cle tresor))
          (send self 'combattre-dragon)))
    (display-alln (send self 'nom) " est mort")))
```

Pour diminuer l'énergie d'un homme d'une quantité n, on a utilisé la méthode decr-energie!. Si l'énergie devient nulle, cela provoque la mort de l'homme par épuisement.

```
(defmethod decr-energie! homme (n)
  (let ((energie-actuelle (send self 'energie)))
    (if (< n energie-actuelle)
      (send self 'set-energie! (- energie-actuelle n))
      (send self 'mourir " par épuisement"))))
```

La mort d'une créature change la valeur du champ vivant? en #f. On donne la cause du décès en argument. Si Siegfried meurt le jeu est fini.

```
(defmethod mourir creature (message)
  (send self 'set-vivant?! #f)
  (display-alln "mort de " (send self 'nom) message)
  (if (equal? self siegfried)
    (*abort* "Vous avez perdu")))
```

Le combat d'un homme contre un dragon est représenté par une méthode. Ce combat n'a lieu que s'ils sont dans la même pièce et l'issue se discute selon l'identité et l'équipement de l'homme. Quand Siegfried est victorieux, son énergie est diminuée de 2.

```
(defmethod combattre-dragon homme ()
  (if (send fafner 'vivant?)
    (let ((salle-homme  (send self 'salle))
          (salle-dragon (send fafner 'salle)))
      (if (equal? salle-homme salle-dragon)
        (if (eq? self siegfried)
          (if (equal? (send epee 'possede-par) siegfried)
            (begin (send fafner 'mourir " tué par Siegfried")
                   (send siegfried 'decr-energie! 2))
            (send siegfried 'mourir " tué par le dragon"))
          (send mime 'mourir " tué par le dragon"))))))
```

Un homme ne peut prendre un objet que s'il est situé dans la même salle que lui. De plus, si c'est le trésor, il doit posséder la clé du coffre à moins que le trésor soit déjà en possession de Mime. Si c'est Siegfried qui s'empare du trésor, la partie est finie. Une tentative de prise de possession qui échoue coûte un point d'énergie à son auteur.

```
(defmethod prendre homme (objet)
  (let ((salle-homme  (send (send self   'salle) 'nom))
```

```
        (salle-objet  (send (send objet  'salle) 'nom)))
    (if (and (eq? salle-homme salle-objet)
             (or (not (eq? 'tresor (send objet 'nom)))
                 (equal? mime (send tresor 'possede-par))
                 (equal?  self  (send cle 'possede-par))))
      (begin (send objet 'set-possede-par! self)
             (display-alln (send objet 'nom)
                           " en possession de " (send self 'nom))
             (if (and (equal? objet tresor)(equal? self siegfried))
               (*abort* "Vous avez gagné")))
      (send self 'decr-energie! 1))))
```

Pour visualiser la situation à chaque phase du jeu, on demande aux créatures d'indiquer leurs états et possessions. Pour cela, on introduit une méthode générale d'affichage pour les créatures :

```
(defmethod affiche creature ()
  (if (send self 'vivant?)
    (display-all  (send self 'nom) " est dans la salle "
                  (send (send self 'salle) 'nom))))
```

On complète cette méthode pour les hommes en indiquant l'énergie et la liste des choses que possède un homme vivant.

```
(defmethod affiche homme ()
  (if (send self 'vivant?)
    (begin
      (send-next self 'homme 'affiche)
      (display-all  "  son energie = " (send self 'energie) "  ")
      (for-each (lambda (chose)
                  (if (eq? self (send chose 'possede-par))
                    (display-all (send chose 'nom) " en sa possession ")))
                (list epee cle tresor))
      (newline))
    (display-alln  (send self 'nom) " est mort")))
```

Déroulement d'une partie

Le jeu commence par une initialisation qui cré les salles, place les portes, cré les créatures avec une énergie initiale et les choses en plaçant aléatoirement la clé et l'épée dans des salles non connues du joueur.

La partie consiste ensuite en une boucle qui exécute alternativement le déplacement aléatoire du dragon puis l'action décidée par le joueur.
A chaque tour, on affiche l'état des protagonistes. La boucle est interrompue quand Siegfried meurt ou quand il s'empare du trésor ou si le joueur décide d'abandonner.

```
(define (jeu)
  (init)
  (display-alln "debut du jeu ")
  (do ((i 1 (+ i 1)))
```

```
    (#f)
   (display-alln "*************** coup no " i)
   (action-dragon)
   (send siegfried 'affiche)
   (send mime 'affiche)
   (action-joueur)
   (newline)))
```

Si le dragon est vivant, son action consiste à tirer au sort un entier dans [0 4] ; selon l'entier trouvé, il décide de dormir ou essaye de se déplacer dans la direction associée.

```
(define (action-dragon)
  (if (send fafner 'vivant?)
    (let ((n (random 5)))
      (if (= 4 n)
        (display  "le dragon dort ")
        (send fafner 'avance (list-ref '(ouest est nord sud) n)))
      (send fafner 'affiche)(newline))
    (display-alln "le dragon est mort ")))
```

Le joueur indique l'action à effectuer par un des personnages à l'aide d'une suite de menus.

```
(define (action-joueur)
  (display-alln "indiquer le personnage qui agit :
             (S)iegfried "  (if (send mime 'vivant?) "ou (M)ime ? " ""))
  (let ((personnage (if (eq? 'S (read)) siegfried mime)))
    (display-alln "choix de l'action : déplacer (N) (S) (E) (O)
                                       prendre  (C)lé (T)résor é(P)ée
                                       (V)oir (Q)uitter: ")
    (let ((action (read)))
      (case action
        ((N) (send personnage 'avance  'nord))
        ((S) (send personnage 'avance  'sud))
        ((E) (send personnage 'avance  'est))
        ((O) (send personnage 'avance  'ouest))
        ((C) (send personnage 'prendre 'cle))
        ((T) (send personnage 'prendre 'tresor))
        ((P) (send personnage 'prendre 'epee))
        ((V) (let ((salle-personnage (send personnage 'salle)))
               (for-each (lambda (chose)
                           (display-alln (send chose 'nom)
                                         (if (equal? (send chose 'salle)
                                                     salle-personnage)
                                           " est" " n'est pas")
                                         " dans cette salle"))
                          (list cle epee tresor))))
        ((Q) (*abort* "vous avez perdu par abandon"))
        (else (display-alln "action inconnue "))))))
```

Il resterait à faire une interface graphique/souris pour que ce jeu soit plus convivial. Ce serait aussi une excellente illustration des méthodes objets mais cela obligerait à utiliser des aspects non standard de Scheme.

11.7 Implantation de SchemO

Les dialectes Lisp ont été, dès l'origine, à l'avant-garde de l'expérimentation de nouveaux concepts pour la programmation par objets. Le système CLOS[1] en est un des points culminants avec son protocole méta-objet. De nombreuses extensions objets sont disponibles pour Scheme. Pour notre petit langage nous n'avons cherché ni l'originalité ni l'efficacité mais la simplicité.

Représentation des classes

Une classe sera représentée par un vecteur à quatre composantes :

```
classe = #(NomClasse NomMere LnomsChamps Lmethode)
```

où `Lmethodes` est une a-liste `( ...(nom-methode . fonction) ...)`
Pour accéder à une classe dont on connaît le nom on utilise une a-liste globale `*les-classes* = (... (nom . classe) ...)` initialisée à `((#f . #f))`.

```
(define *les-classes* (list (cons #f #f)))
```

La fonction `classe` retourne le vecteur représentant la classe dont on donne le nom :

```
(define (classe nom-classe)
  (let ((doublet (assq nom-classe *les-classes* )))
    (if doublet
      (cdr doublet)
      (*erreur* "pas de classe de nom : " nom-classe))))
```

On accède aux composantes d'une classe, dont on donne le nom, par les fonctions :

```
(define (nomMere nom-classe)
  (vector-ref (classe nom-classe) 1))

(define (Lchamps nom-classe)
  (vector-ref (classe nom-classe) 2))

(define (Lmethodes nom-classe)
  (vector-ref (classe nom-classe) 3))
```

[1] *Common Lisp Object System.*

Représentation des objets

Un objet sera représenté par un vecteur à deux composantes :

```
objet = #(NomClasse  Lchamp.valeur)
```

La deuxième champ est une a-liste constituée par les doublets (`nomChamp . sa-valeur` où `nomChamp` est un champ accessible directement ou par héritage et ayant été initialisé.
On définit les fonctions d'accès :

```
(define (NomClasse obj)
  (vector-ref obj 0))

(define (Lchamp.valeur  obj)
  (vector-ref obj 1))
```

Définition d'une classe

La définition d'une nouvelle classe par la fonction :

```
(make-class  'nomClasse  'nomClasseMere  'champ1  'champ2 ...)
```

consiste à créer un nouveau vecteur de classe.[2] Cette création se fait en quatre temps :

- on place, dans la composante `LnomsChamps`, les noms des champs indiqués dans la définition de la classe ;
- on définit les accesseurs et les modificateurs pour les champs cités dans la définition et on place ces méthodes dans la composante `Lmethodes` du vecteur ;
- les composantes `NomClasse` et `NomMere` sont initialisées
- enfin, on ajoute la nouvelle classe dans la a-liste globale `*les-classes*` et on retourne une valeur indéfinie.

```
(define  (make-class nom-classe nom-mere . Lchamps)
  (let ((NlleClasse
         (vector nom-classe  nom-mere  Lchamps (list (cons '() '())))))
    (if (not (null? Lchamps))
      (let ((Laccesseurs    (map make-accesseur    Lchamps))
            (Lmodificateurs (map make-modificateur Lchamps)))
        (vector-set! NlleClasse 3 (append (map  cons Lchamp Laccesseurs)
                                          (map  cons (map prefixe-set Lchamps)
                                          Lmodificateurs)))))
    (ajouter/modifier!  nom-classe NlleClasse *les-classes*)))))
```

[2]On pourrait éviter d'avoir à quoter les arguments en remplaçant la fonction `make-class` par une macro `defclass`, mais on a voulu réduire au maximum l'utilisation des macros.

Cette fonction utilise quatre fonctions auxiliaires :
`make-accesseur, make-modificateur, ajouter!` et `prefixe-set`.
La fonction `make-accesseur` retourne une lambda qui sert à accéder à la valeur du champ passé en paramètre. Il suffit de consulter la a-liste `Lchamp.valeur` de l'objet concerné :

```
(define (make-accesseur nom-champ)
  (lambda (objet)
    (let ((champ.valeur (assq nom-champ (Lchamp.valeur objet))))
      (if champ.valeur
        (cdr champ.valeur)
        (*erreur* "champ non-initialisé " nom-champ)))))
```

La fonction `make-modificateur` retourne une lambda qui sert à modifier la valeur du champ passé en paramètre. Cette lambda réalise une modification physique de la a-liste des `champ.valeurs` de l'objet concerné. Si le champ n'était pas initialisé, il est inséré physiquement dans cette a-liste.

```
(define (make-modificateur nom-champ)
  (lambda (objet exp)
    (let ((Lchamp&valeur (Lchamp.valeur objet)))
      (ajouter/modifier! nom-champ exp Lchamp&valeur))))
```

La fonction `ajouter/modifier!` modifie physiquement une a-liste en modifiant la valeur associée à une clé ou en ajoutant un nouveau doublet en queue si la clé n'existe pas (voir le chapitre 4 §3 pour `append2!`). La valeur retournée est indéfinie.

```
(define (ajouter/modifier! cle valeur  aliste)
  (let ((doublet (assq cle aliste)))
    (if doublet
      (set-cdr! doublet valeur)
      (append2! aliste (list (cons cle valeur)))))
   'indefinie)
```

La fonction `prefixer-set` transforme un nom de champ en le symbole `set-nom!` .

```
(define (prefixe-set nom)
  (string->symbol (string-append "set-" (symbol->string nom) "!")))
```

Création d'un objet

On crée un objet instance d'une classe par :

```
(make-instance nomClasse  champj1 valeurj1 ...)
```

Un objet est un vecteur dont la deuxième composante est la a-liste
`((`*`champj1 . valeurj1`*`)...)` ou `(())` si on ne fournit pas de valeurs initiales aux champs.

```
(define (make-instance nom-classe . champs&valeurs)
  (if (null? champs&valeurs)
    (vector nom-classe (list (cons '() '())))
    (let ((champs  (les-pairs   champs&valeurs))
          (valeurs (les-impairs champs&valeurs)))
      (vector nom-classe (map cons champs valeurs)))))
```

On extrait les éléments d'indice respectivement pair ou impair de la liste des `champs&valeurs` par les fonctions :

```
(define (les-pairs L)
  (if (null? L)
    '()
    (cons (car L)(les-pairs (cddr L)))))

(define (les-impairs L)
  (if (or (null? L)(null? (cdr L)))
    '()
    (cons (cadr L)(les-impairs (cddr L)))))
```

Définition d'une méthode

On utilise une macro `defmethod` pour définir une nouvelle méthode. Le principe consiste à créer une fonction `(lambda (self` *x1...xN*`)` *corps*`)` et à ajouter cette fonction dans la a-liste `Lmethodes` de la classe considérée.
Le paramètre formel `self` sert à désigner dans le *corps* l'objet receveur du message. Comme `define`, c'est une forme spéciale, aussi est-elle définie par une macro. L'appel :

```
(defmethod nom-methode nom-classe  (x1 ...)  s1 s2 ...)
```

doit s'expanser en l'expression :

```
(letrec ((nom-methode (lambda (self x1 ...) s1 s2 ...)))
  (ajouter/modifier! 'nom-methode  nom-methode
                     (Lmethodes 'nom-classe))))
```

Le `letrec` sert à permettre la définition de méthode récursive. La méthode est ajoutée (ou bien remplace une méthode de même nom) dans la liste des méthodes de cette classe. Voici la définition de `defmethod` avec le mécanisme des macros Lisp :

```
(define-macro (defmethod nom-methode nom-classe Lparametre . Lcorps)
  `(letrec ((,nom-methode (lambda ,(cons 'self Lparametre) ,@Lcorps)))
     (ajouter/modifier! ',nom-methode  ,nom-methode
                        (Lmethodes ',nom-classe))))
```

La définition par `define-syntax` présente un problème : la règle d'hygiène fait qu'un renommage empêchera l'identificateur `self` d'être capturé par un identificateur identique dans le corps. Or c'est précisément ce que l'on veut. Heureusement, il y a des méthodes de plus bas niveau pour y remédier mais nous ne les présenterons pas.

Envoi de message

L'appel de (send *objet 'nomMethode arg1...*) déclenche la recherche, dans les méthodes de la classe de l'objet, de la méthode ***nomMethode***. Si on ne la trouve pas, on cherche dans la classe mère ... En cas d'échec, on renvoie un message d'erreur. Cette recherche de méthode est réalisée par la fonction `find-method`:

```
(define (find-method  message NomClasse)
  (if NomClasse
    (let ((doublet (assq message (Lmethodes NomClasse))))
      (if doublet
        (cdr doublet)
        (find-method  message  (nomMere NomClasse))))
    (*erreur*  " methode inconnue: " message)))
```

Une fois la méthode trouvée, il suffit de l'appliquer aux arguments. On ajoute en tête de la liste des arguments l'objet receveur du message pour qu'il soit lié avec le paramètre formel `self`.

```
(define (send objet message . argts)
  (let ((methode (find-method message (NomClasse  objet))))
    (if methode
      (apply methode (cons objet argts)))))
```

La fonction send-next

La forme (send-next *objet 'nomClasse 'nomMethode arg1 ...*) se comporte comme send mais la recherche de méthode se fait à partir de la classe mère de la classe passée en paramètre.

```
(define (send-next  objet nomClasse message . argts)
  (let ((methode (find-method message  (nomMere  nomClasse)))
    (if methode
      (apply methode  (cons objet argts)))))
```

Cette couche objet ajoutée à Scheme est un bon exemple des possibilités d'extensions réalisables avec ce type de langage.

Pour revenir aux sources de la programmation par objets, nous allons étudier en complément une simulation. On trouvera d'autres utilisations de ce petit langage à objets dans le chapitre 14 §3 et §4.

11.8 Compléments: vie artificielle

On étudie les interactions qui se manifestent au court de l'évolution de plusieurs espèces partageant un même milieu appelé biotope.
On considère un biotope composé de lapins, de renards et où l'herbe pousse. Les lapins se nourrissent d'herbe et sont eux-mêmes la proie des renards. La prolifération des lapins est bénéfique pour les renards mais entraîne aussi une raréfaction

de l'herbe qui, conjuguée auxs prédations des renards, peut entraîner l'extinction des lapins et ensuite celle des renards.
Il est intéressant d'étudier les conditions pour que de telles espèces puissent continuer à survivre.
On peut dépasser le point de vue écologique pour étudier aussi le point de vue génétique. Chaque animal d'une espèce donnée hérite à sa naissance d'un patrimoine génétique qui va influencer ses aptitudes et son comportement. On pourra étudier le phénomène de la sélection naturelle, c.-à-d. si à terme il se dégage au sein de chaque espèce des animaux plus adaptés au milieu. On entre alors dans le domaine dit de la «vie artificielle» qui étudie l'apparition de comportements dits émergeants car non prévus à l'origine.
Pour illustrer la programmation par objet, nous allons modéliser, de façon très simpliste, ces types de phénomènes.

Le biotope est découpé par un quadrillage de zones, chaque zone peut comporter des lapins, des renards et de l'herbe. Les animaux de chaque espèce peuvent avoir différents comportements : se déplacer, manger, dormir, se reproduire, un lapin peut fuir et un renard peut tuer un lapin. Cette variété d'espèces et de comportements nous incite à utiliser une programmation par objets.

Les classes utilisées

Pour chaque animal on aura besoin de savoir : s'il est vivant, dans quelle zone il est présent, la liste de ses comportements et son degré de vitalité. On pourrait ajouter beaucoup d'autres caractéristiques : âge, sexe, couleur, ... mais on ne le fait pas pour ne pas trop alourdir cette simulation.
On est donc conduit à définir une classe `espece` dont les instances seront des animaux du biotope :

```
(make-class 'espece #f 'vivant?  'vitalite  'zone 'comportement)
```

Le champ `vivant?` est un booléen, le champ `vitalite` sera un nombre mesurant le degré d'énergie de l'animal, le champ `zone` donnera sa zone actuelle et le champ `comportement` représente la liste des comportements que l'animal peut adopter.
Comme on aura souvent à incrémenter d'une valeur `x` (positive ou négative) la vitalité on définit la méthode :

```
(defmethod incr-vitalite espece (x)
  (send self 'set-vitalite! (+ x (send self 'vitalite))))
```

Pour avoir des méthodes plus spécifiques aux lapins ou aux renards, on définit les deux sous-classes `lapin` et `renard` :

```
(make-class 'lapin   'espece)
(make-class 'renard  'espece)
```

Pour décrire le biotope, on a un quadrillage du plan $x0y$ en zones ; le biotope est une matrice carrée `*taille*` $\times$ `*taille*` de zones. Une zone est une classe qui comporte des champs pour la population d'animaux (liste de lapins et de renards), la quantité d'herbe, et la localisation `x` , `y` dans la matrice du quadrillage.

```
(make-class  'zone  #f   'population 'qte-herbe 'x 'y)
```

Pour tenir à jour le nombre total d'animaux de chaque espèce, on actualise en permanence des statistiques. Elles augmentent à chaque naissance et diminuent à chaque décès. Pour cela, on introduit une classe `statistique` avec un champ pour le nombre de lapins et un pour le nombre de renards :

```
(make-class 'statistique #f 'lapin 'renard)
```

On y joint une méthode pour incrémenter d'un entier `x` (positif ou négatif) le total des lapins ou des renards

```
(defmethod incr-statistique statistique (x nom-classe)
  (case nom-classe
    ((lapin)  (send self 'set-lapin! (+ x (send self 'lapin))))
    ((renard) (send self 'set-renard! (+ x (send self 'renard))))))
```

Pour afficher l'état des populations, on ajoute la méthode :

```
(defmethod afficher statistique ()
  (display-alln "**** bilan  nb-lapin: " (send self 'lapin) "
                             nb-renards: " (send self 'renard))
  (newline))
```

On désigne la statistique par une variable globale `*statistique*`, initialisée par la fonction :

```
(define (initialiser-statistique)
  (set! *statistique* (make-instance 'statistique 'lapin 0 'renard 0)))
```

Comportements communs aux espèces

La simulation consiste en une boucle représentant l'écoulement du temps, chaque passage dans le corps de la boucle provoque l'exécution du comportement de toutes les espèces qui se trouvent dans chacune des zones.
Les comportements possibles pour un lapin sont stockés dans une liste globale :

```
(define *comportement-lapin*  '(reproduire deplacer dormir fuir manger))
```

et ceux d'un renard :

```
(define *comportement-renard* '(reproduire attaquer dormir deplacer))
```

On remarque qu'un renard peut attaquer et qu'un lapin peut fuir.
A chaque appel du comportement d'un animal, on exécutera le premier comportement effectivement réalisable de sa liste des comportements. Mais il est important de savoir qu'à la naissance de chaque animal, cette liste est initialisée dans un ordre quelconque et cet ordre représente, en un sens, le caractère propre de chaque animal d'une espèce. Il est clair que celui qui commence toujours par essayer de manger a un caractère différent de celui qui essaye en priorité de se reproduire ...

```
(define (executer-comportement Lcomportement animal)
  (or (null? Lcomportement)
      (send animal (car Lcomportement))
      (executer-comportement (cdr Lcomportement) animal)))
```

Il faut que l'exécution d'un comportement renvoie #f si ce comportement n'est pas actuellement utilisable et #t si on a pu l'exécuter. Les comportements seront évidemment représentés par des méthodes. Les comportements communs seront des méthodes de la classe espece : se déplacer, mourir, dormir, se reproduire. Les croisements se feront entre deux animaux de même espèce et, pour simplifier, on fera abstraction du sexe.
Comme le biotope est représenté par un quadrillage, on évite d'avoir à traiter le cas particulier des positions en bordure en le considérant comme une surface torique. On utilise donc pour les déplacements une arithmétique modulo la *taille* du biotope :

```
(define (add-modulo-taille x y)
  (remainder (+ x y) *taille* ))

(define (sub-modulo-taille x y)
  (if (< (- x y) 0)(+ (- x y) *taille*)(- x y)))
```

Ainsi, la somme ou la différence des coordonnées d'une zone est toujours un entier de l'intervalle [0 *taille*[.
La méthode deplace-de réalise le passage d'un animal d'une zone de coordonnées x, y à la zone de coordonnées x+dx , y+dy. Ensuite, il faut actualiser les populations de ces deux zones :

```
(defmethod deplacer-de espece (dx dy)
  (let ((zone (send self 'zone))
    (let* ((x (send  zone 'x))
           (y (send  zone 'y))
           (x1 (add-modulo-taille x dx))
           (y1 (add-modulo-taille y dy)))
      (let ((nlle-zone (matrice-ref  *biotope* x1 y1)))
        (send self 'set-zone!  nlle-zone)
        (send zone 'supprimer-animal self)
        (send nlle-zone 'ajouter-animal self))))
```

Le déplacement proprement dit se fait en choisissant aléatoirement deux entiers dx et dy dans [0 3[. Puis, on appelle deplace-de avec ces paramètres, enfin on diminue la vitalité de la somme dx + dy.

```
(defmethod deplacer espece ()
  (let ((dx (randomN 3))
        (dy (randomN 3)))
    (send self 'incr-vitalite  (- (+ dx dy)))
    (send self 'deplacer-de dx dy)))
```

Rappelons que la fonction (randomN k) est définie au chapitre 5 §5, elle retourne un entier aléatoire dans l'intervalle [0 k[.

La méthode mourir met à #f le champ vivant? de l'animal et actualise le contenu de sa zone et la statistique. Il est prévu l'affichage d'un message en cas de décès si le booléen *avec-faire-part*? vaut #t.

```
(defmethod mourir espece ()
  (let ((sa-zone (send self 'zone))
        (sa-classe (NomClasse self)))
    (if *avec-faire-part?*
      (display-alln "mort d'un " sa-classe ))
    (send self 'set-vivant?! #f)
    (send sa-zone 'supprimer-animal self)
    (send sa-zone 'incr-herbe 4)
    (send *statistique* 'incr-statistique -1 sa-classe)))
```

La méthode dormir est très simple, elle consiste en une légère diminution de la vitalité :

```
(defmethod dormir espece ()
  (send self 'incr-vitalite -1))
```

La reproduction d'un animal est possible si sa vitalité est supérieure à sa vitalité initiale et s'il existe un animal de la même espèce dans sa zone ; si ces conditions sont remplies, on effectue le croisement :

```
(defmethod reproduire espece ()
  (let* ((sa-zone (send self 'zone))
         (vitalite (send self 'vitalite))
         (son-espece (NomClasse self))
         (population (send sa-zone 'population)))
    (if (< *vitalite-initiale* vitalite)
      (let ((partenaire (existe-animal-vivant son-espece
                                               (remove self population))))
         (if partenaire
           (send self 'croisement  partenaire)
           #f))
      #f)))
```

On a utilisé la fonction existe-animal-vivant pour chercher s'il existe, dans une liste d'animaux de la zone, un animal en vie d'une espèce donnée.

```
(define (existe-animal-vivant nom-classe  Lanimal)
  (some (lambda (animal)(if (and (eq? (NomClasse animal) nom-classe)
                                 (send animal 'vivant?))
                            animal
                            #f))
        Lanimal))
```

Le croisement de deux animaux de même espèce consiste à ajouter un nouvel animal dans la même zone que ses parents. Sa vitalité initiale sera la demi-somme

des vitalités des parents et il héritera du comportement d'un de ses parents[3]. On doit aussi actualiser la statistique et afficher un faire-part de naissance si le booléen `*avec-faire-part*?` vaut `#t`.

```
(defmethod croisement espece (partenaire)
  (let ((sa-zone (send self 'zone))
        (sa-vitalite (send self 'vitalite))
        (son-comportement (send self 'comportement))
        (vitalite-partenaire (send partenaire 'vitalite))
        (sa-classe (NomClasse self) ))
    (send self 'set-vitalite! (quotient sa-vitalite 2))
    (send partenaire 'set-vitalite! (quotient vitalite-partenaire 2))
    (if *avec-faire-part?*
      (display-alln " nouveau " sa-classe ))
    (let ((descendant
            (make-instance sa-classe
                           'vitalite
                           (quotient (+ sa-vitalite vitalite-partenaire) 2)
                           'zone sa-zone  'vivant? #t
                           'comportement son-comportement)))
      (send sa-zone 'ajouter-animal  descendant)
      (send *statistique* 'incr-statistique 1 sa-classe))
      #t))
```

La dernière méthode pour la classe `espece` réalise l'actualisation d'un animal. S'il est vivant et si sa vitalité est ≤ 0, alors il décède, sinon on lui demande d'exécuter son comportement :

```
(defmethod actualiser-animal  espece ()
  (if (send self 'vivant?)
    (if  (<= (send self 'vitalite) 0)
      (send self 'mourir)
      (let ((Lcomportement (send self 'comportement)))
        (executer-comportement Lcomportement self)))))
```

Comportements spécifiques aux lapins

Un lapin peut manger s'il y a de l'herbe dans sa zone. Il absorbe au maximum la valeur de la constante `*ration-herbe*` ; la quantité d'herbe de la zone est diminuée d'autant.

```
(defmethod manger lapin ()
  (let ((sa-zone (send self 'zone)))
    (let ((qte-herbe (send sa-zone 'qte-herbe)))
      (if (< 0 qte-herbe)
```

[3]On pourrait mélanger de façon plus réaliste la contribution des deux parents au comportement de la progéniture. Par exemple, si l'on code les comportements par des 0 et des 1 on peut les représenter par une espèce de chromosome ce qui permet d'attribuer au descendant un partage aléatoire des gènes. En développant ce point de vue, on aboutit aux algorithmes dits «génétiques» où ce type de reproduction est utilisé pour rechercher les maxima d'un critère d'adéquation.

```
        (let ((mange (min *ration-herbe* qte-herbe)))
          (send self 'incr-vitalite mange)
          (send sa-zone 'set-qte-herbe! (- qte-herbe mange))
        #f)))))
```

Les lapins sont des animaux prudents, aussi la méthode dormir doit être particularisée pour les lapins. Un lapin n'envisage de dormir que s'il n'y pas un renard dans sa zone :

```
(defmethod dormir lapin ()
  (let ((sa-zone (send self 'zone)))
    (if (existe-animal-vivant 'renard  (send sa-zone 'population))
      #f
      (send-next  self 'lapin 'dormir))))
```

Si un lapin aperçoit un renard dans une zone adjacente, il s'enfuit dans la direction opposée :

```
(defmethod fuir lapin ()
  (let* ((sa-zone (send self 'zone))
         (Lzones (send sa-zone 'Lzones-voisines))
         (renard (existe-animal-vivant 'renard
                   (apply append (map (lambda (zone)
                                        (send zone 'population))
                                      Lzones)))))
    (if renard
      (let ((x (send sa-zone 'x))
            (y (send sa-zone 'y))
            (zone-renard (send renard 'zone)))
        (let ((x-renard  (send zone-renard 'x))
              (y-renard  (send zone-renard 'y)))
          (send self 'deplacer-de
                     (remainder  (sub-modulo-taille x x-renard) 2)
                     (remainder  (sub-modulo-taille y y-renard) 2))))
      #f)))
```

Comportements spécifiques aux renards

L'attaque est une méthode propre aux renards, elle consiste à tuer un lapin qui se trouve dans sa zone. Cela augmente sa vitalité d'une quantité égale à celle du lapin tué. Un faire-part de décès est imprimé si `*avec-faire-part?*` est égal à `#t` :

```
(defmethod attaquer renard ()
  (let* ((sa-zone (send self 'zone))
         (lapin (existe-animal-vivant 'lapin (send sa-zone 'population))))
    (if lapin
      (let ((vitalite-lapin (send lapin 'vitalite)))
        (if *avec-faire-part?* (display "victime d'un renard "))
        (send lapin 'mourir)
        (send self 'incr-vitalite vitalite-lapin))
      #f)))
```

Les méthodes propres à une zone

On ajoute ou supprime un animal de sa population avec les méthodes suivantes :

```
(defmethod  ajouter-animal zone  (animal)
  (send self 'set-population!
             (cons animal (send self  'population))))

(defmethod  supprimer-animal zone  (animal)
  (send self 'set-population!
             (remove animal (send self  'population))))
```

Pour faire pousser l'herbe d'une quantité x donnée, on a :

```
(defmethod incr-herbe zone (x)
  (send self 'set-qte-herbe! (+ x (send self 'qte-herbe))))
```

Pour actualiser une zone, on doit actualiser chacune des espèces qui la composent :

```
(defmethod actualiser-zone zone ()
  (send self 'incr-herbe 1)
  (let ((population (send self 'population)))
    (for-each (lambda (animal)(send animal 'actualiser-animal))
              population)))
```

Pour fuir, le lapin a eu besoin de regarder la liste des zones adjacentes à une zone ; on les obtient en calculant la liste des coordonnées de ces zones :

```
(defmethod Lzones-voisines zone  ()
  (let ((x (send  self 'x))
        (y (send  self 'y)))
    (let ((Lcoord (Lcoord-voisines x y )))
      (map (lambda (coord)
             (matrice-ref *biotope* (car coord)(cdr coord)))
           Lcoord))))

(define (Lcoord-voisines i j)
  (map (lambda (direction)
         (cons (add-modulo-taille i (car direction))
               (add-modulo-taille j (cdr direction))))
       *Les8directions*))
```

La liste des coordonnées des zones voisines s'obtient en ajoutant, aux coordonnées de la zone, les huit directions possibles définies par la constante `*Les8directions*` :

```
(define *Les8directions*
  (let ((moins1 (sub-modulo-taille 0 1)))
    (list (cons 0 1)(cons 0 moins1)
          (cons 1 1) (cons 1 0)(cons 1 moins1)
          (cons moins1 1 ) (cons moins1 0)(cons moins1 moins1))))
```

-1,1	0,1	1,1
-1,0	0,0	0,1
-1,-1	0,-1	1,-1

Les huit directions de déplacement

Les initialisations

Avant de définir la boucle d'évolution de notre biotope, il faut introduire les fonctions d'initialisation.
On se donne au départ la probabilité pour qu'il y ait dans chaque zone un lapin, un renard, une quantité d'herbe. Chaque probabilité est définie en pourcentage par un nombre entier: %lapin, %renard, %herbe dans [0 100]. D'où une méthode pour initialiser le contenu d'une zone:

```
(defmethod initialiser zone (%lapin %renard %herbe)
  (if (< (randomN 100) %lapin)
    (send self 'ajouter-animal (initialiser-lapin self)))
  (if (< (randomN 100) %renard)
    (send self 'ajouter-animal (initialiser-renard self)))
  (if (< (randomN 100) %herbe)
    (send self 'incr-herbe 10)) )
```

Pour initialiser un lapin dans une zone, on lui affecte une vitalité égale à la constante *vitalite-initiale* et on lui attribue un comportement qui est une permutation aléatoire de la liste des comportements possibles *comportement-lapin*. C'est l'ordre des éléments de cette liste qui caractérise le «tempérament» d'un lapin.

```
(define (initialiser-lapin zone)
  (send *statistique* 'incr-statistique 1 'lapin)
  (make-instance 'lapin  'vivant? #t
                         'vitalite *vitalite-initiale*
                         'zone zone
                         'comportement (melanger-list *comportement-lapin*)))
```

La fonction utilisée pour mélanger une liste (ici très courte) consiste à se ramener à la fonction de mélange de vecteurs décrite au chapitre 5 § 7:

```
(define (melanger-list L)
  (vector->list (melanger-vecteur! (list->vector L))))
```

Même principe pour initialiser les renards d'une zone:

```
(define (initialiser-renard zone)
  (send *statistique* 'incr-statistique 1 'renard)
  (make-instance 'renard
                 'vivant? #t
                 'vitalite *vitalite-initiale*
                 'zone zone
                 'comportement (melanger-list *comportement-renard*)))
```

Exercice 5 *On aurait pu, au prix de légères complications, factoriser ces initialisations en une méthode de l'espèce ; le faire.*

Pour initialiser le biotope, qui est une matrice de zones, on commence par créer une matrice de zones vierges :

```
(define (make-biotope-vierge)
  (let ((matrice (init-matrice *taille*  *taille*)))
    (do ((y 0 (+ 1 y)))
        ((= y *taille*) matrice)
      (do ((x 0 (+ 1 x)))
          ((= x *taille*))
        (matrice-set! matrice x y
                      (make-instance 'zone 'population '()
                                     'qte-herbe 0  'x x 'y y)))))
```

Puis, on initialise le biotope en initialisant chacune de ses zones :

```
(define (initialiser-biotope %lapin  %renard %herbe)
  (set! *biotope* (make-biotope-vierge))
  (let ((action
         (lambda (zone)
           (send zone 'initialiser %lapin  %renard %herbe))))
    (for-each-element *biotope*  action)))
```

On a utilisé la fonction `for-each-element` pour effectuer une action donnée sur chaque élément d'une matrice ; elle se définit simplement par :

```
(define (for-each-element matrice action)
  (let ((nbLignes (nb-lignes matrice))
        (nbColonnes (nb-colonnes matrice)))
    (do ((y 0 (+ 1 y)))
        ((= y nbLignes) matrice)
      (do ((x 0 (+ 1 x)))
          ((= x nbColonnes))
        (action (matrice-ref matrice x y ))))))).
```

La simulation

La simulation consiste à :

- mettre les statistiques à zéro,
- initialiser la matrice du biotope,

- afficher le nombre initial de lapins et de renards,
- effectuer la boucle d'évolution, avec un nombre donné d'itérations.
  Chaque itération consiste à :
  - afficher le numéro de l'itération,
  - actualiser le biotope,
  - afficher les nouveaux effectifs de lapins et de renards.

On se donne en paramètres les pourcentages pour l'initialisation et la valeur du booléen `*avec-faire-part?*`.

```
(define (evolution  %lapin  %renard %herbe Nb-pas  avec-faire-part?)
  (set! *avec-faire-part?*  avec-faire-part?)
  (initialiser-statistique)
  (initialiser-biotope  %lapin  %renard %herbe)
  (send *statistique* 'afficher)
  (let ((action
          (lambda (zone)
            (send zone 'actualiser-zone))))
    (do ((i 0 (+ i 1)))
        ((= i Nb-pas) 'fini)
      (display-alln #\newline "**** jour no : " i)
      (for-each-element *biotope* action)
      (send *statistique* 'afficher))) )
```

Test

Pour pouvoir tester, on définit les constantes globales :

```
(define *taille* 4)
(define *vitalite-initiale*   30)
(define *ration-herbe*   4)
```

Lançons une exécution :

```
? (evolution 95 30 40 20  #t)

**** bilan  nb-lapin: 15    nb-renards: 9


**** jour no : 0
victime d'un renard mort d'un lapin
**** bilan  nb-lapin: 14    nb-renards: 9


**** jour no : 1
victime d'un renard mort d'un lapin
**** bilan  nb-lapin: 13    nb-renards: 9
```

```
**** jour no : 2
 nouveau lapin
**** bilan  nb-lapin: 14    nb-renards: 9


**** jour no : 3
 nouveau lapin
victime d'un renard mort d'un lapin
**** bilan  nb-lapin: 14    nb-renards: 9


**** jour no : 4
mort d'un renard
mort d'un renard
**** bilan  nb-lapin: 14    nb-renards: 7


**** jour no : 5
**** bilan  nb-lapin: 14    nb-renards: 7


**** jour no : 6
victime d'un renard mort d'un lapin
mort d'un renard
**** bilan  nb-lapin: 13    nb-renards: 6


**** jour no : 7
victime d'un renard mort d'un lapin
mort d'un renard
mort d'un lapin
mort d'un renard
**** bilan  nb-lapin: 11    nb-renards: 4


**** jour no : 8
victime d'un renard mort d'un lapin
mort d'un renard
**** bilan  nb-lapin: 10    nb-renards: 3


**** jour no : 9
**** bilan  nb-lapin: 10    nb-renards: 3


**** jour no : 10
victime d'un renard mort d'un lapin
**** bilan  nb-lapin: 9    nb-renards: 3


**** jour no : 11
```

```
victime d'un renard mort d'un lapin
**** bilan  nb-lapin: 8   nb-renards: 3


**** jour no : 12
victime d'un renard mort d'un lapin
**** bilan  nb-lapin: 7   nb-renards: 3


**** jour no : 13
victime d'un renard mort d'un lapin
**** bilan  nb-lapin: 6   nb-renards: 3


**** jour no : 14
victime d'un renard mort d'un lapin
**** bilan  nb-lapin: 5   nb-renards: 3

fini
```

Architecture du programme

Les variables globales

`*taille*`
Le biotope est un carré `*taille*` x `*taille*` de zones.

`*comportement-lapin*`
Liste des comportements possibles pour les lapins.

`*comportement-renard*`
Liste des comportements possibles pour les renards.

`*vitalite-initiale*`
Quantité d'énergie de tout animal à sa naissance.

`*avec-faire-part?*`
Le booléen indiquant si l'on souhaite signaler les naissances et les décès

Les classes

`(classe 'espece #f 'vivant?  'vitalite 'zone 'comportement)`
Un individu d'une espèce donnée est dans une zone, il possède
une certaine vitalité et a une liste de comportements possibles.

`(classe 'lapin 'espece)`
L'espèce des lapins.

`(classe 'renard 'espece)`
L'espèce des renards.

`(classe 'zone #f 'population 'qte-herbe 'x 'y)`
Une zone est un élément du quadrillage ; elle comporte une population de lapins, de renards et une quantité d'herbe.

`(classe 'statistique #f 'lapin 'renard)`
La statistique permet de tenir à jour le nombre de lapins et de renards

Les fonctions et méthodes

`(evolution %lapin %renard %herbe Nb-pas avec-faire-part?)` Elle réalise les initialisations en créant une quantité de lapins, de renards et d'herbe avec des probabilités données en pourcentage. Puis il y a exécution d'une boucle de `Nb-pas`. Chaque passage dans le corps de la boucle provoque l'exécution du comportement de chaque animal.

Les méthodes et fonctions auxiliaires

Cette présentation indentée est un peu arbitraire, car les méthodes peuvent être utilisées à tous les niveaux.

`(initialiser-statistique)`
Remise à zéro des effectifs.

`(initialiser-biotope %lapin %renard %herbe)`
On initialise chaque zone du biotope selon les pourcentages.

`(make-biotope-vierge)`
On crée une matrice qui représente le biotope.

`(initialiser zone %lapin %renard %herbe)`
On initialise aléatoirement la population de la zone avec des probabilités suivant les pourcentages donnés.

`(initialiser-lapin zone)`
On crée un lapin dans une zone, sa liste des comportements est obtenue par permutation aléatoire de la liste `*comportement-lapin*`.

`(initialiser-renard zone)`
On crée un renard dans une zone, sa liste des comportements est obtenue par permutation aléatoire de la liste `*comportement-renard*`.

`(melanger-list L)`
Rend une permutation aléatoire de la liste `L`.

`(actualiser-zone zone)`
Demande l'actualisation de l'état des espèces qui la composent, et une augmentation de l'herbe.

`(incr-herbe zone n)`
Pour augmenter de `n` la quantité d'herbe de la zone.

`(actualiser-animal espece)`
Déclenche le décès des animaux sans vitalité et demande l'exécution du comportement de ceux qui sont en vie.

`(executer-comportement Lcomportement animal)`
On exécute le premier comportement admissible de l'animal.

`(incr-vitalite espece x)`
Pour augmenter la vitalité de `x`.

`(deplacer espece)`
On choisit aléatoirement les composantes du déplacement et on diminue la vitalité d'autant.

`(deplacer-de espece dx dy)`
Réalise le changement de zone avec actualisation des populations des zones. Pour rester dans le quadrillage on le considère comme un tore.

`(add-modulo-taille x y)`
Addition de deux entiers modulo `*taille*`

`(randomN k)`
Rend un entier aléatoire dans l'intervalle [0 k[, (voir chapitre 5 §7).

`(reproduire espece)`
Si la vitalité de l'animal est suffisante et s'il y a un congénère dans sa zone, alors il y a croisement.

`(croisement espece partenaire)`
Création d'un descendant de vitalité égale à la demi somme de celle de ses parents.

`(ajouter-animal zone animal)`
On ajoute un animal dans la population d'une zone.

`(existe-animal-vivant nom-classe Lanimal)`
Retourne un congénère vivant dans la même zone ou `#f` s'il n'y en a pas.

`(dormir lapin)`
Ne dort que s'il n'y a pas de renard dans sa zone.

`(method fuir lapin)`
S'il y a un renard dans une zone adjacente, le lapin fuit dans la direction opposée.

`(sub-modulo-taille x y)`
Soustraction de deux entiers modulo `*taille*`.

`(Lzones-voisines zone)`
Liste des zones adjacentes à une zone.

`(Lcoord-voisines i j)`
Liste des coordonnées adjacentes à cette case.

`*Les8directions*`
Les huit directions admissibles de déplacement.

`(manger lapin)`
Le lapin mange sa ration d'herbe et la quantité d'herbe est diminuée d'autant.

`*ration-herbe*`
Maximum d'herbe mangeable en une fois par un lapin.

`(attaquer renard)`
Un renard peut attaquer un lapin dans sa zone.

`(dormir espece)`
Provoque une petite diminution de la vitalité de l'animal.

`(mourir espece)`
Actualisation des populations et des statistiques.

`(incr-statistique statistique x nom-classe)`
Incrémenter de `x` la statistique d'une espèce donnée.

`(supprimer-animal zone animal)`
Supprimer un animal de la population d'une zone.

`(for-each-element matrice action!)`
Exécuter une action sur chaque élément d'une matrice.

`(afficher statistique)`
Afficher le nombre de lapins et de renards.

Il y a de nombreuses améliorations à apporter à cette simulation. C'est un avantage de la programmation par objets que de pouvoir faciliter l'extension de cette maquette par l'ajout de nouvelles méthodes et de nouvelles classes d'animaux ayant de nouveaux comportements. Par exemple, on peut coder dans ses chromosomes le comportement de chaque animal et utiliser le mécanisme du croisement chromosomique pour obtenir le code génétique d'un descendant. On renvoie à la littérature sur les algorithmes génétiques pour ces aspects.

En expérimentant ce système, on verra qu'il est assez délicat de trouver des configurations qui ne provoquent pas trop rapidement la disparition d'une espèce. Heureusement que la nature dispose de mécanismes de régulation plus sophistiqués ; mais elle n'en demeure pas moins en équilibre très fragile.

11.9 De la lecture

On trouvera la description d'autres extensions objets pour Scheme par exemple dans [FWH92, Jaf, Que94].
On n'a pas abordé les mécanismes importants du méta-protocole, on renvoie à [MNC+89, KdRB92]. Pour le langage CLOS on renvoie à [Ste90].
On trouvera des informations sur les systèmes de Lindenmayer dans [PJS92].
La méthode des algorithmes génétiques est exposée dans [Gol94].

Partie II

Outils formels pour le programmeur

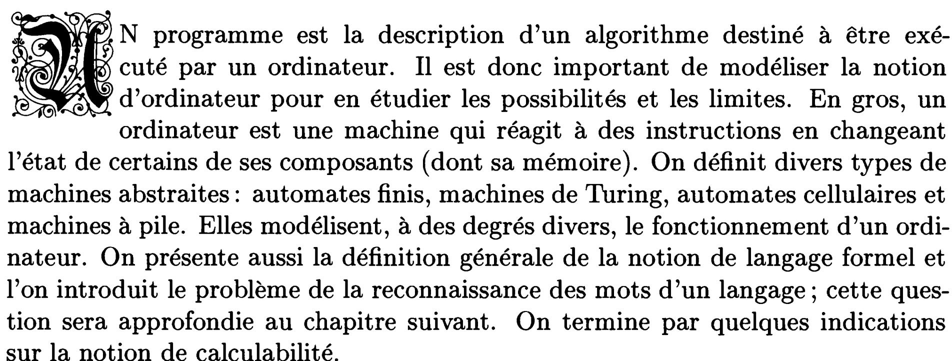
Quelques fonctions utiles

Dans la deuxième partie, il sera parfois commode de faire usage de certaines fonctions ou macros non standard mais définies dans la première partie. En voici la liste par ordre alphabétique. On conseille au lecteur de les regrouper dans un fichier qu'il évaluera si besoin.

Fonction/macro	page
`*abort*`	256
`*erreur*`	256
`acons`	50
`append2!`	110
`append-map`	80
`bind`	288
`creer-genVar`	143
`display-all`	84
`display-alln`	84
`every`	81
`filtre`	72
`gensym`	281
`intersection`	87
`remove`	36
`reunion`	86
`some`	81
`sublis`	49
`unless`	275
`when`	275

Chapitre 12

Automates, langages et ordinateurs

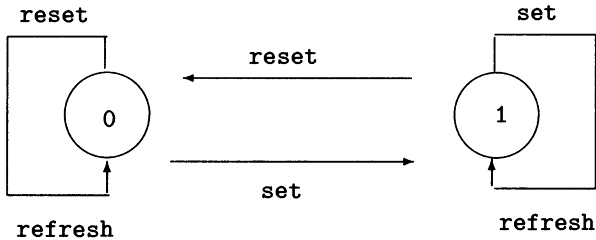


UN programme est la description d'un algorithme destiné à être exécuté par un ordinateur. Il est donc important de modéliser la notion d'ordinateur pour en étudier les possibilités et les limites. En gros, un ordinateur est une machine qui réagit à des instructions en changeant l'état de certains de ses composants (dont sa mémoire). On définit divers types de machines abstraites : automates finis, machines de Turing, automates cellulaires et machines à pile. Elles modélisent, à des degrés divers, le fonctionnement d'un ordinateur. On présente aussi la définition générale de la notion de langage formel et l'on introduit le problème de la reconnaissance des mots d'un langage ; cette question sera approfondie au chapitre suivant. On termine par quelques indications sur la notion de calculabilité.

12.1 Automates finis

Mémoire 1 bit

Introduisons la notion d'automate fini avec un exemple très simple. Modélisons l'élément mémoire le plus petit : la mémoire à 1 bit ! Une telle cellule mémoire peut avoir deux états que l'on appelle 0 et 1. Cette mémoire réagit à trois ordres : `set, reset` et `refresh`. L'ordre `set` la fait passer dans l'état 1, l'ordre `reset` la fait passer dans l'état 0 et `refresh` lui fait conserver son état. Visualisons son comportement à l'aide d'un diagramme : on représente les états par des cercles et les ordres de changement d'états servent à étiqueter les flèches de transition.



Mémoire à 1 bit

Le fonctionnement de cette mémoire peut se résumer dans une table T à deux entrées : l'état actuel et l'ordre reçu. Pour chacun de ces couples on indique l'état résultant :

	set	reset	refresh
1	1	0	1
0	1	0	0

Table des changements d'états

Automates finis

Si l'on assemble des mémoires à 1 bit, on peut définir des composants plus complexes : registres, mémoires N bits, ... Le nombre d'états peut devenir très grand mais reste fini. Cela conduit à la notion générale d'*automate fini déterministe.*

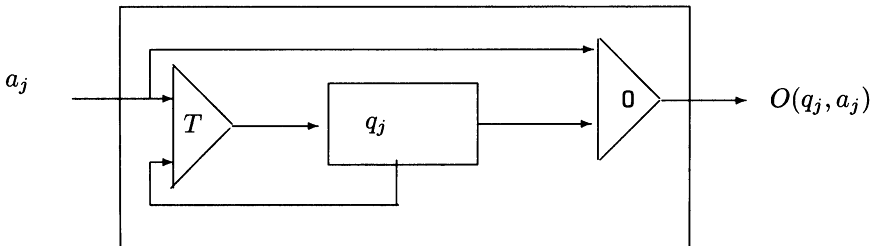
Définition 1 On se donne un ensemble fini Q d'états et un ensemble fini d'instructions I. Un automate fini déterministe est caractérisé par la donnée d'une fonction de transition $T : Q \times I \to Q$ qui donne la valeur du nouvel état $q_j = T(q_i, a_k)$ suite à l'exécution d'une instruction a_k depuis l'état q_i.

Le mot déterministe signifie que, pour chaque état et chaque entrée, il y un et un seul état suivant ; autrement dit que T est une fonction. On peut aussi considérer des *automates non déterministes* pour lesquels T est seulement une relation sur $Q \times I \times Q$. On peut donc avoir 0 ou plusieurs états suivants possibles. On verra, au §2 du chapitre suivant, que ce n'est pas une réelle extension de la notion d'automate déterministe.
Nous nous intéressons au comportement d'un automate fini déterministe lorsque celui-ci exécute une suite d'instructions (i.e. un programme). En supposant le temps discrétisé et scandé par une horloge, nous avons une suite d'états q_j vérifiant la relation de récurrence :

$$q_{j+1} = T(q_j, a_j)$$

Automates et machines

On peut voir un automate comme une machine synchrone (cadencée par l'horloge). Avec ce point de vue, il est souvent utile de considérer une fonction de sortie O (pour observer la machine), on dit que l'on a un transducteur. Si la fonction de sortie dépend de q_j et a_j, on dit que c'est une *machine de Mealy* et si elle ne dépend que de l'état q_j, on dit que c'est une *machine de Moore*.



Machine de Mealy

Dans le cas de notre mémoire à 1 bit, on prend comme fonction de sortie la fonction $O(q_j, a_j) = q_j$, ce qui en fait une machine de Moore.
Si l'on assemble de telles machines (en parallèle, en série, ...) on peut réaliser des fonctions électroniques plus complexes : additionneurs, multiplexeurs, ... C'est le domaine des circuits digitaux séquentiels.

Mais la notion d'automate dépasse largement le cadre de la modélisation de circuits électroniques. Par exemple, si l'on met bout à bout les lignes de la table de transition d'un automate, on obtient une bande qui est une espèce de «code génétiqueD» suffisant pour définir l'évolution de l'automate. L'analyse du comportement d'un automate a fait l'objet de nombreuses recherches, mais si l'on ne considère plus un seul automate mais une «soupe» de tels automates, alors le domaine est encore peu défriché. C'est l'un des thèmes de «l'algorithmique génétique» : on étudie des populations d'automates pouvant subir des mutations de leurs codes génétiques, effectuer des sélections et des reproductions, ... et l'on pourra ainsi obtenir une modélisation de certains phénomènes biologiques. La notion d'automate est aussi fortement liée à la notion de langage formel comme on va le voir dans la suite.

12.2 Notion de langage formel

Donnons quelques notions générales sur les langages.

Définition d'un langage

Définition 2 On se donne un ensemble de symboles distincts appelé *alphabet* A, on appelle *mot* sur cet alphabet toute suite finie de symboles ou la suite vide notée ϵ. La longueur du mot est la longueur de la suite.

L'ensemble des mots sur un alphabet A est désigné par A^*.
Par exemple, les entiers sont des mots (non vides) de l'alphabet
`{0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9}`. On peut aussi les représenter en numération binaire par des mots de l'alphabet `{0 , 1}`.
La numération romaine représente les entiers (pas trop grands) par certains mots sur l'alphabet `{I, V , X , C , L , M}`.

Définition 3 Un *langage* L sur un alphabet A est par définition une partie (finie ou non) de A^*.

Par exemple, si $A = \{0, 1\}$, l'ensemble des suites finies de 0 ou 1 et ayant un nombre impair de 1 est un langage sur cet alphabet. Si A est l'ensemble des lettres (accentuées ou non), alors l'ensemble des mots français est un langage. En informatique on s'intéresse aux langages de programmation ; ils sont construits sur un alphabet comportant un nombre fini de mots appelés mots-clés, des identificateurs admissibles, des nombres, ... et certains symboles spéciaux.

Opérations sur les langages

Comme un langage est une partie d'un ensemble A^*, on dispose sur les langages des opérations ensemblistes usuelles : union, intersection, complémentaire, etc.
Attention à ne pas confondre le langage vide $\emptyset$ avec le langage (non vide) réduit au mot vide ϵ.
Par ailleurs, on dispose sur les mots d'une opération binaire naturelle, la *concaténation* qui consiste à mettre «bout à bout» les symboles constituant deux mots pour en former un troisième.

$$a_1 \ldots a_h \quad , \quad b_1 \ldots b_k \quad \to \quad a_1 \ldots a_h b_1 \ldots b_k$$

Cette opération, notée . , est clairement associative, non commutative et admet le mot vide ϵ comme élément neutre.
Un mot v est un *sous-mot* d'un mot w s'il existe des mots x et y tels que $w = x.v.y$.
En particulier, v est un *préfixe* de w si $x = \epsilon$ et un *suffixe* si $y = \epsilon$.
Par extension, on définit le *produit* $L_1.L_2$ de deux langages comme étant l'ensemble des concaténations $m.n$ lorsque $m \in L_1$ et $n \in L_2$.
On peut alors définir l'opération $*$ (étoile) qui à un langage L associe le langage :

$$L^* = \{\epsilon\} \cup L \cup L.L \cup L.L.L \ \ldots$$

Autrement dit, c'est l'ensemble des mots que l'on peut construire par concaténation d'un nombre fini de mots de L. Cette notation est bien cohérente avec la notation A^* pour désigner l'ensemble de tous les mots sur l'alphabet A.

Il y a deux problèmes de base en théorie des langages : la *reconnaissance* et la *génération* :

- la reconnaissance consiste à décider si un mot appartient ou non à un langage donné ;
- la génération consiste à décrire par des procédés *finis* l'ensemble (généralement infini) des mots d'un langage.

Pour étudier ces problèmes, on distingue différentes classes de langages. On commence par une classe très particulière de langages : les langages réguliers et l'on verra des classes plus générales au chapitre suivant.

12.3 Reconnaissance d'un langage par un automate fini

Automate fini et langage régulier

A un automate fini déterministe, nous allons associer un langage. Pour cela, on distingue parmi ses états :
- un état initial q_0,
- un ensemble d'états finals Q_f.
Le *langage associé* aura pour alphabet l'ensemble des instructions de l'automate, il sera constitué des mots (ou suite d'instructions) qui permettent de passer de l'état initial à un état final,

$$L = \{a_1 \ldots a_k \mid q_0 \stackrel{a_1}{\rightarrow} \cdots \stackrel{a_k}{\rightarrow} q_j \; \in \; Q_f\}$$

Par exemple, le mot vide ϵ est dans L si et seulement si q_0 est aussi un élément final.
Un langage défini par ce procédé s'appelle un *langage régulier*. Par construction l'automate fournit une méthode de reconnaissance des mots de ce langage : on exécute la suite d'instructions constituée par un mot ; si l'on arrive dans un état final, alors le mot est dans le langage.

Exercice 1 *On reprend l'automate de la mémoire à 1 bit et l'on décide que son état initial est 0 et que 0 est aussi le seul état final. Caractériser les suites d'instructions* `set, reset, refresh` *qui forment les mots du langage régulier reconnu par cet automate.*

Implantation d'un automate fini

Nous allons implanter cette méthode de description d'un langage. On commence par modéliser la notion d'automate fini déterministe. C'est un objet qui change d'état quand il reçoit certaines instructions, il est donc naturel de le représenter avec la notion de prototype.
L'alphabet du langage à étudier sera représenté par des symboles et les mots par des listes de symboles.
La fonction de transition d'un automate est définie par la donnée d'une table, elle même modélisée par une a-liste de a-listes. Une a-liste de la forme :

```
( ... (etati  .   (...(inputj . etatk) ...)  ...)
```

indique qu'un automate étant dans l'état **etati** et lisant un symbole **inputj** passera dans l'état **etatk**.
Le prototype qui représente un automate répondra aux messages suivants :

etat : retourne l'état courant.

etat-suivant : réalise le changement d'état correspondant à la donnée lue en entrée.

etat-final? : teste si un état donné fait partie des états finals.

reset : remet l'automate dans son état initial.

Pour créer un automate, on se donne une table de transition, un état initial et un ensemble d'états finals. La table permet de calculer la fonction de transition.

```
(define (make-automate table-transition etat0  Letat-finals)
  (let ((etat etat0)
        (fct-transition
          (lambda (etat input)
            (cdr (assq input (cdr (assq etat table-transition)))))))
    (lambda (instruction . Largt)
      (case instruction
        ((etat) etat)
        ((changer-etat)
         (let ((etat-suivant (fct-transition etat (car Largt))))
           (set! etat  etat-suivant)))
        ((etat-final?) (if (memq etat Letat-finals) #t #f))
        ((affiche) (display-alln "etat:" etat))
        ((reset) (set! etat etat0))))))
```

Acceptation d'un mot

Pour savoir si un mot est accepté par un automate, on exécute la suite des transitions définie par la liste de symboles donnés en entrée puis l'on regarde si on a atteint un état final. La fonction `accepte-mot?` retourne un booléen indiquant si un mot a été accepté par un automate. Elle prend aussi en argument un booléen `bavard?` pour savoir si l'on désire une exécution avec affichage des états intermédiaires. Notons qu'avant de lancer l'automate, il faut penser à le remettre dans son état initial.

```
(define (accepte-mot? automate mot  bavard?)
  (letrec ((lire-mot
             (lambda (mot)
               (let ((etat (automate 'etat)))
                 (if bavard? (automate 'affiche))
                 (cond ((null? mot) etat)
                       (else (automate 'changer-etat (car mot))
                             (lire-mot (cdr mot))))))))
    (automate 'reset)
    (lire-mot mot)
    (automate 'etat-final?)))
```

Un exemple

Appliquons ceci à un exemple. On considère l'alphabet {0 , 1} et le langage constitué par les suites contenant un nombre de 1 qui soit multiple de 3.
Pour reconnaître ce langage, on peut utiliser un automate à trois états: q0, q1, q2, ayant le diagramme de transition suivant :

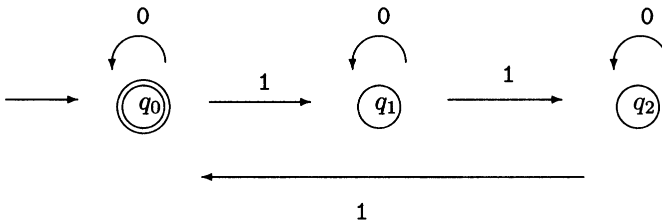


Diagramme des changements d'états

Les 0 ne font pas changer d'état, et les 1 font passer de l'état q_i à l'état q_{i+1}.
L'état initial (signalé par une flèche entrante) est q_0, c'est aussi le seul état final (schématisé par un double cercle).
On le code par une table :

```
(define table-mult3 '((q0 (0 . q0)(1 . q1))
                      (q1 (0 . q1)(1 . q2))
                      (q2 (0 . q2)(1 . q0))))
```

On définit l'automate associé par :

```
(define automate-mult3 (make-automate  table-mult3 'q0 '(q0)))
```

et l'on teste les mots suivants :

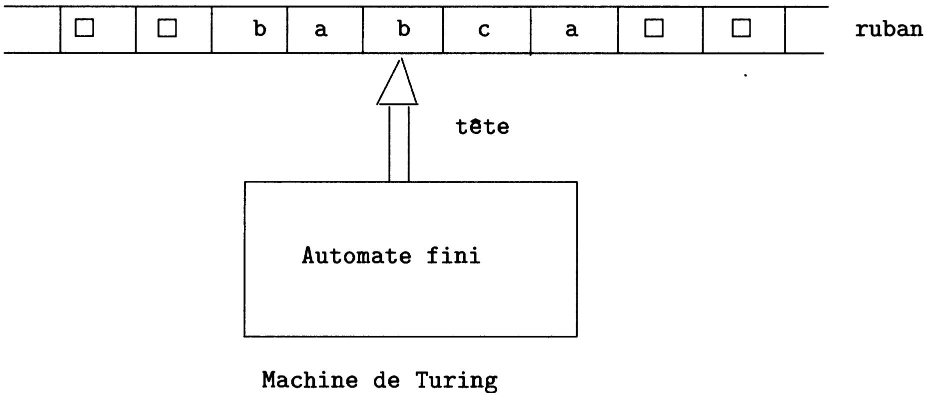
```
? (accepte-mot? automate-mult3 '(0 1 0 1 1 0) #t)
 -> q0 q0 q1 q1 q2 q0 q0 #t

? (accepte-mot? automate-mult3 '(1 1 0 1 1 0) #f)
-> #f
```

Supposons que nous voulions vérifier qu'une suite de parenthèses est une suite «bien» parenthésée au sens habituel. Par exemple ((()(est mal parenthésée, tandis que (( ))()() est bien parenthésée. On devine intuitivement qu'un automate fini ne pourra reconnaître que les suites ayant une profondeur de parenthésage bornée. En effet, considérons un automate à N états et une suite de $K > N$ parenthèses ouvrantes, alors après lecture des N premières parenthèses, on est certain que l'automate sera passé deux fois par le même état. Il ne pourra donc pas distinguer les deux niveaux de parenthèses qui ont conduit au même état. Pour mémoriser une suite non bornée d'états, il faut passer à des automates disposant d'une mémoire non bornée.

12.4 Machines de Turing

Définition d'une machine de Turing

On adjoint à un automate fini déterministe une mémoire non bornée constituée d'une bande infinie. Cette bande est reliée à l'automate par une tête de lecture/écriture. La tête permet de connaître le contenu de la case qui lui fait face ou d'écrire sur cette case. On obtient une machine inventée par Alan Turing pour donner un sens précis à la notion de fonction *calculable par une machine*:



Machine de Turing

La bande sert à stocker les instructions à exécuter et également à mémoriser des «symboles». Au départ, la bande contient un mot constitué d'un nombre fini de symboles, le reste de la bande ne comportant que des blancs que l'on matérialise par le symbole □.

Fonctionnement d'une machine de Turing

A chaque étape, la machine lit le symbole faisant face à sa tête et exécute, selon son état, les instructions données par sa table. Chaque instruction se compose de trois éléments : l'état suivant, le symbole à écrire sur la bande et la direction de déplacement de la tête. L'exécution d'une instruction consiste en :

- un changement d'état de l'automate fini,
- l'écriture sur la bande d'un symbole à la place de celui qui a été lu,
- un déplacement de la tête de lecture d'une case à gauche ou à droite.

Toutes ces actions ne dépendent que du symbole lu et de l'état courant.
A partir d'une configuration initiale {état initial, tête, bande}, la machine exécute une suite de pas pour s'arrêter quand on arrive dans un état final de l'automate. Le contenu de la bande représente alors (selon un certain codage) le résultat du calcul. Bien sûr, la machine peut «boucler» et l'exécution ne fournira aucun résultat.

Ce type de machine avec une mémoire non bornée est en fait une meilleure modélisation d'un ordinateur. En effet, un ordinateur a une mémoire finie ; c'est un automate fini qui a un nombre d'états possibles tellement grand que l'on ne peut espérer les énumérer tous pendant la vie de notre univers !

Implantation d'une machine de Turing

Pour expérimenter avec ce type de machine, nous allons en donner une implantation. Nous reprenons la méthode utilisée pour un automate fini, en y rajoutant une bande et une tête. La bande sera représentée par une liste de symboles et la tête par un indice indiquant sa position dans la bande. Comme pour les automates finis, la fonction de transition est définie par une table. La valeur associée à un état et un symbole lu est une liste constituée par le nouvel état, le symbole à écrire et une lettre (`G` ou `D`) pour indiquer la direction de déplacement de la tête.
La modification de la tête est réalisée par la fonction `deplacer-tete`, elle consiste à incrémenter ou décrémenter sa valeur selon que la direction est `D` ou `G`. Quand la tête est une des extrémités non encore parcourue de la bande, il faut ajouter un blanc à gauche ou à droite selon la direction et l'extrémité concernée.
La fonction `lire` réalise la lecture de la bande, c'est à dire retourne le symbole situé devant la tête.
La fonction `ecrire` réalise l'écriture sur la bande du symbole passé en argument.

```
(define (make-turing table etat0 Letat-finals tete0 bande0)
  (let ((etat etat0)
        (tete tete0)
        (bande bande0)
        (fct-transition
          (lambda (etat input)
            (cdr (assq input (cdr (assq etat table)))))))
    (let ((deplacer-tete
            (lambda (direction)
              (case direction
                ((G) (if (= 0 tete)
                         (set! bande (cons '□ bande))
                         (set! tete (- tete 1 ))))
                ((D) (if (= (- (length bande) 1) tete)
                         (begin (append! bande (list '□))
                                (set! tete (+ tete 1))))))))

          (lire (lambda ()(list-ref  bande tete)))

          (ecrire (lambda (symb)(set-car! (list-tail bande  tete) symb))))

      (lambda (instruction)
        (case instruction
          ((etat) etat)
          ((executer-pas) (bind (etat-suivant  output-symb direction)
                            (fct-transition etat (lire))
                            (set! etat etat-suivant)
                            (ecrire output-symb)
```

```
                               (deplacer-tete direction)))
        ((etat-final?) (if (memq etat Letat-finals) #t #f))
        ((affiche) (display-alln "etat:" etat " tete:" tete "  bande:" bande))
        ((reset) (set! etat etat0) (set! bande bande0) (set! tete tete0))))))
```

On a utilisé la liaison destructurante `bind` décrite au chap.9 §6.

Une machine de Turing exécute le programme situé sur sa bande et s'arrête quand on arrive dans un état final, le résultat du calcul est représenté par la configuration finale.

```
(define (executer-turing turing  bavard?)
  (letrec ((iterer-pas (lambda ()
                         (if bavard?  (turing 'affiche))
                         (unless (turing 'etat-final?)
                                 (turing 'executer-pas)
                                 (iterer-pas)))))
    (turing 'reset)
    (iterer-pas)
    (display-alln " configuration finale : ")
    (turing 'affiche)))
```

Application à la vérification du bon parenthésage

On peut construire une machine de Turing pour vérifier si une suite de parenthèses est bien construite. On utilise une machine à quatre états :
- q0 : on cherche une) en allant vers la droite,
- q1 : on cherche une (en allant vers la gauche,
- q2 : on cherche une) en allant vers la gauche,
- qf : seul état final.

On écrit `oui` sur la bande quand on est dans un état final d'acceptation de l'expression et on écrit `non` dans le cas contraire. On utilise un alphabet de 6 symboles[1] : `<` , `>` , `X` , □ , `oui` , `non`. Le symbole `X` sert à effacer les parenthèses qui s'accouplent. La table des transitions est donnée par la a-liste :

```
(define table
  '((q0 . ((> . (q1 X G))(< . (q0  <  D))(□ . (q2  □  G))(X . (q0 X D))))
    (q1 . ((> . (q1 > G))(< . (q0  X  D))(□ . (qf non G))(X . (q1 X G))))
    (q2 . ((> . (q2 > G))(< . (qf non G))(□ . (qf oui G))(X . (q2 X G))))) )
```

Le contenu initial de la bande est constitué de l'expression à vérifier encadrée par des caractères □ . On place la tête devant la première parenthèse (donc à la position 1) et on part de l'état q0.
Construisons une machine pour vérifier si < > est bien parenthésée :

```
(define turing-parenthesage
  (make-turing table  'q0 '(qf)  1 '(□ < > □ )))
```

On exécute :

[1] On a remplacé les parenthèses par des signes `<` , `>` à cause du rôle spécial des parenthèses en Scheme.

```
? (executer-turing turing-parenthesage    #t)
etat:q0 tete:1  bande:( □  < >  □ )
etat:q0 tete:2  bande:( □  < >  □ )
etat:q1 tete:1  bande:( □  < x  □ )
etat:q0 tete:2  bande:( □  x x  □ )
etat:q0 tete:3  bande:( □  x x  □ )
etat:q2 tete:2  bande:( □  x x  □ )
etat:q2 tete:1  bande:( □  x x  □ )
etat:q2 tete:0  bande:( □  x x  □ )
etat:qf tete:0  bande:( □  oui x x  □ )
 configuration finale :
etat:qf tete:0  bande:( □  oui x x  □ )
```

La réponse est oui.
On recommence avec l'expression (( ) :

```
(define turing-parenthesage
  (make-turing table 'q0 '(qf) 1 '( □  < < >  □ ) ))

? (executer-turing turing-parenthesage  #t)
etat:q0 tete:1  bande:( □  < < >  □ )
etat:q0 tete:2  bande:( □  < < >  □ )
etat:q0 tete:3  bande:( □  < < >  □ )
etat:q1 tete:2  bande:( □  < > x  □ )
etat:q0 tete:3  bande:( □  < x x  □ )
etat:q0 tete:4  bande:( □  < x x  □ )
etat:q2 tete:3  bande:( □  < x x  □ )
etat:q2 tete:2  bande:( □  < x x  □ )
etat:q2 tete:1  bande:( □  < x x  □ )
etat:qf tete:0  bande:( □  non x x  □ )
 configuration finale :
etat:qf tete:0  bande:( □  non x x  □ )
```

La réponse est non.

La notion de *machine* peut prendre des formes très diverses. A titre d'exemple, voici une autre famille d'automates qui possède la même puissance de calcul «universel» que les machines de Turing.

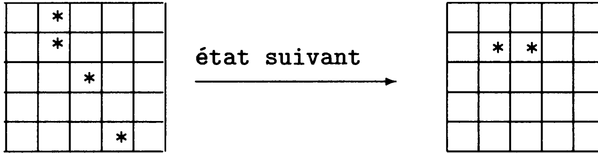
12.5 Automates cellulaires

Jeu de la vie en dimension 2

On veut décrire l'évolution d'un ensemble de cellules dont l'état binaire (vivante ou morte) n'est fonction que de l'état des cellules adjacentes. Un exemple classique est fourni par le «jeu de la vie» de Conway en *dimension 2.* On a un ensemble de cellules sur un quadrillage. La règle pour obtenir la population à l'instant suivant est :

- une cellule reste en vie si parmi les 8 cellules adjacentes il y en a 2 ou 3 en vie,

- une cellule naît si parmi les 8 cellules adjacentes, il y en a exactement 3 en vie.



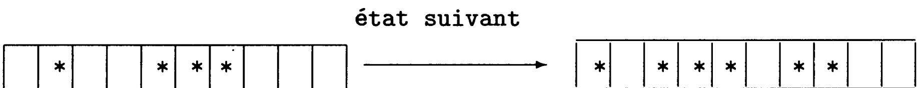
Jeu de la vie

Automate cellulaire en dimension 1

On va se contenter de modéliser les automates cellulaires en *dimension 1*. Dans ce cas, l'automate cellulaire est défini par la donnée d'une table qui indique l'état suivant de chacune des huit combinaisons possibles de état-cellule-gauche, état-cellule, état-cellule-droite. On représente un état par une liste de 0 ou 1 dans laquelle un 1 (resp. un 0) signifie vivante (resp. morte). Voici une table qui s'interprète en disant qu'une cellule naît ou reste en vie si, et seulement si, elle a une seule cellule adjacente en vie.

```
                       cellule droite
         état cellule
                                 état suivant
cellule gauche
                 0 0 1 -> 1
                 1 0 0 -> 1
                 0 1 1 -> 1
                 1 1 0 -> 1
                 1 0 1 -> 0
                 0 0 0 -> 0
                 0 1 0 -> 0
                 1 1 1 -> 0
```

Voici un exemple de transition selon cette table :



On représente une table par une a-liste dont les doublets sont de la forme :

```
((cellule-gauche  etat-cellule  cellule-droite) . etat-suivant )
```

Représentation d'un automate cellulaire par un prototype

Un automate cellulaire sera un prototype du même type que celui des automates précédents. Le point essentiel est le calcul de l'état suivant. Pour cela, on calcule en parallèle l'état suivant de chacune de ses cellules ; ce calcul est réalisé par la fonction locale `fct-transition`. Pour obtenir l'état suivant de la ième cellule de la liste, on doit connaître la liste des états des trois cellules de numéros : $i-1$, i et $i+1$; c'est l'objet de la fonction `Lvoisines`. Il y a un petit problème pour les cellules extrêmes. On considère la liste comme une liste circulaire, aussi la cellule gauche de la première cellule est identifiée avec la dernière et de même la cellule droite de la dernière est identifiée avec la première cellule. Pour afficher l'état de l'automate, on représente les cellules vivantes par une * et les cellules mortes par un blanc. Voici maintenant la fonction de création d'un automate cellulaire :

```
(define (make-autoCellulaire table etat0)
  (let ((etat etat0)
        (fct-transition
          (lambda (etat)
            (let ((IndiceMax (- (length etat0) 1)))
              (map (lambda (i)
                     (cdr (assoc (Lvoisines etat i IndiceMax) table)))
                   (iota IndiceMax))))))

    (lambda (message)
      (case message
        ((etat) etat)
        ((changer-etat) (set! etat (fct-transition etat)))
        ((affiche) (newline)
                   (map (lambda (x)
                          (if (zero? x)
                            (display " ")
                            (display "*")))
                        etat))
        ((reset)  (set! etat etat0))))))
```

Avec la fonction auxiliaire déjà mentionnée :

```
(define (Lvoisines L k IndiceMax)
  (list (list-ref L (if (zero? k) IndiceMax  (- k 1)))
        (list-ref L  k )
        (list-ref L (if (= IndiceMax k) 0 (+ k 1)))))
```

On rappelle que la fonction `iota` sert à construire la liste `(0 ...n)`

```
(define (iota n)
  (if (zero? n)
    '(0)
    (append  (iota  (- n 1)) (list n))))
```

Evolution d'un automate cellulaire

L'utilisation de cet automate consiste à effectuer un nombre donné de changements d'états.

```
(define (ItererAutomateCellulaire automate nb-iterations)
  (newline)
  (automate 'reset)
  (do ((i 0 (+ i 1)))
      ((= i nb-iterations))
    (automate 'affiche)
    (automate 'changer-etat)))
```

On teste avec la règle indiquée au début :

```
(define table '(((0 0 0) . 0)((0 0 1) . 1)((0 1 0) . 0)((1 0 0) . 1)
                ((0 1 1) . 1)((1 0 1) . 0)((1 1 0) . 1)((1 1 1) . 0)))
```

On prend comme état initial une cellule vivante au milieu de 20 cellules mortes.

```
(define automate (make-autoCellulaire table
              '(0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0)))
```

```
? (ItererAutomateCellulaire automate  18)

          *
         * *
        *   *
       * * * *
      *       *
     * *     * *
    *   *   *   *
   * * * * * * * *
  *               *
 * *             * *
*   *           *   *
** * *         * * **
 *    *       *    *
* *  * *     * *  * *
*  **   *   *   **  *
****** * * * * ******
     *         *
    * *       * *
```

On peut expérimenter visuellement et étudier diverses questions : périodicité, extinction, rapports avec le triangle de Pascal modulo 2, etc.
Mais revenons aux machines qui nous intéressent le plus : les ordinateurs ! Nous allons commencer par une machine très simple pour évaluer les expressions arithmétiques.

12.6 Machines à pile

La mémoire d'une machine de Turing ne permet d'écrire qu'à l'endroit que l'on vient de lire et l'on ne peut se déplacer que d'une case à la fois. Cela ne correspond pas aux possibilités des mémoires de nos ordinateurs ou RAM en anglais. En effet, RAM est l'acronyme de *Random Access Memory* et signifie que l'on peut accéder à tout moment en lecture ou en écriture à des endroits quelconques de la mémoire.

Mémoire et instructions d'une machine à pile

Chaque case est repérée par une *adresse*, aussi la structure de tableau est un bon modèle d'une RAM en Scheme.
La première chose que l'on attend d'une machine est qu'elle puisse évaluer des expressions arithmétiques comme `(10 + 97) * 72 - 13`. On a vu au chap.7 §3 que la structure de pile donnait un moyen systématique d'évaluation d'une expression à partir de sa mise sous forme postfixé. L'expression précédente s'écrit en postfixée `10 97 + 72 * 13`. On sait que cette forme postfixée se traduit immédiatement en la suite d'instructions pour l'évaluer avec une pile. Il vient dans ce cas la suite: `(empiler 10)`, `(empiler 97)`, `ADD`, `(empiler 72)`, `MUL`, `(empiler 13 )`, `SUB`, `STOP`.
L'instruction `ADD` signifie dépiler les deux arguments et empiler leur somme; les instructions `MUL` et `SUB` ont une signification analogue. L'instruction `STOP` sert à indiquer la fin du calcul. Pour exécuter ces instructions, on va construire une machine à pile.

Implantation d'une machine à pile

La machine à pile devra exécuter les instructions suivantes:

```
(ADD), (SUB), (MUL), (EMPILER  nombre),  (STOP)
```

Par souci d'uniformité, toutes les instructions sont représentées par une liste. Le nom d'une instruction s'appelle aussi le mnémonique:

```
(define Mnemo  car)
```

Quand il y a un argument, on y accède par la fonction:

```
(define argt-instr cadr)
```

La suite des instructions constitue le *code*, on le représente par un vecteur d'instruction. On accède à une instruction par son indice (appelé *adresse* de l'instruction) dans le vecteur. Cette adresse est contenue dans un registre de la machine appelé *compteur ordinal* et noté PC (pour *Program Counter*). Ce registre est modélisé par une variable locale `PC` initialisée à `0`. La lecture d'une instruction est faite par une fonction appelée `charger-instruction` (*fetch* en anglais). Elle récupère l'instruction située à l'adresse contenue dans `PC` puis incrémente la valeur de `PC`.

On utilise une des représentations des piles mutables du chap.7 §2 pour modéliser la pile de la machine. Après exécution des instructions, on affiche le résultat final du calcul qui se trouve au sommet de la pile.
Voici la fonction d'exécution du code par une machine à pile.

```
(define (executer-code  code)
  (let ((PILE (pile!:vide))
        (PC 0))
    (letrec (
            (charger-instr (lambda ()
                             (set! PC (+ PC 1))
                             (vector-ref code (- PC 1))))

            ;;**** la boucle d'exécution des instructions
            (exec-aux
              (lambda ()
                (let ((instr (charger-instr)))
                  (case (Mnemo instr)
                    ((EMPILER) (pile!:empiler (argt-instr instr) PILE)
                               (exec-aux))
                    ((ADD) (let* ((v2 (pile!:depiler PILE))
                                  (v1 (pile!:depiler PILE)))
                             (pile!:empiler (+ v1 v2) PILE) )
                           (exec-aux))
                    ((SUB) (let* ((v2 (pile!:depiler PILE))
                                  (v1 (pile!:depiler PILE)))
                             (pile!:empiler (- v1 v2) PILE))
                           (exec-aux))
                    ((MUL) (let* ((v2 (pile!:depiler PILE))
                                  (v1 (pile!:depiler PILE)))
                             (pile!:empiler (* v1 v2) PILE) )
                           (exec-aux))

                    ((STOP) (display-alln "==> " (pile!:sommet PILE)))
                    (else (*erreur* "instruction inconnue : " instr))))))
      (exec-aux))))
```

Compilation d'expressions arithmétiques

Pour calculer une expression avec notre machine à pile, on doit d'abord générer le code associé. La traduction d'une expression en un code machine s'appelle la *compilation* et sera étudiée plus systématiquement au chapitre 22 ; mais il est instructif de le faire dans ce cas très simple. On génère le code sous forme d'une liste puis on le recopie (on dit assembler) dans un vecteur. Pour simplifier, on suppose que l'expression à compiler est donnée sous forme préfixée.
La fonction `compiler` est constituée par une boucle, `compAux` , qui traduit l'expression en une liste d'instructions selon le schéma suivant :
- une constante `nb` est traduite en `(empiler nb)`,
- une expression générale `(op opd1 opd2)` est traduite en la séquence
`code-opd1    code-opd2    mnémonique.`

Le code généré est accumulé dans le paramètre `code-suivant` puis transformé en un vecteur.

```
(define (compiler exp)
  (let ((op    car)
        (opd1  cadr)
        (opd2 caddr))
    (letrec ((compAux
              (lambda (exp code-suivant)
                (if (number? exp)
                  (cons `(EMPILER ,exp) code-suivant)
                  (case (op exp)
                    ((+) (compAux (opd1 exp)
                                  (compAux (opd2 exp)
                                           (cons '(ADD) code-suivant))))
                    ((-) (compAux (opd1 exp)
                                  (compAux (opd2 exp)
                                           (cons '(SUB) code-suivant))))
                    ((*) (compAux (opd1 exp)
                                  (compAux (opd2 exp)
                                           (cons '(MUL) code-suivant))))
                  )))))
      (list->vector (compAux exp '((STOP)))))))
```

Compilons l'expression arithmétique du début :

```
(define code (compiler '(- (* (+ 10 97) 72) 13)))

? code
 #((empiler 10) (empiler 97) (add) (empiler 72) (mul) (empiler 13) (sub) (stop))
```

Calculons la valeur de l'expression en exécutant ce code avec notre machine à pile

```
? (exec-code  code)
resultat:7691
```

Exercice 2 *En pratique, la mémoire d'une machine n'est pas infinie, aussi reprenez ce paragraphe avec une machine à pile bornée. On surveillera les débordements de piles.*

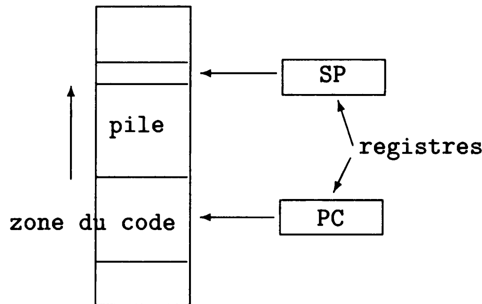
Exercice 3 *Etendre le type des opérations arithmétiques permises : ajouter par exemple la division et la racine carrée.*

12.7 Structure d'un ordinateur

La structure d'une machine à pile est assez voisine de l'architecture d'un ordinateur, mais il reste des différences.

Composants d'un ordinateur

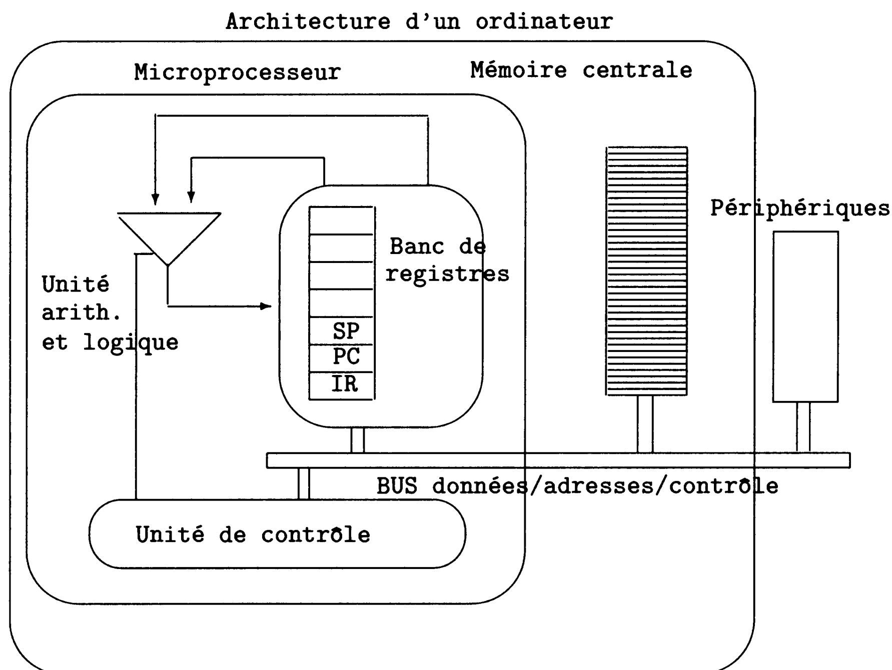
Dans un ordinateur, le code et la pile sont situés dans la mémoire centrale et la valeur PC est rangée dans un de ses registres.



Une organisation de la mémoire centrale

Un registre est analogue à une case mémoire mais est situé directement dans l'unité centrale ce qui permet d'y accéder beaucoup plus rapidemment. Un microprocesseur possède en général de nombreux registres (spécialisés ou non) pour stocker des résultats intermédiaires sans avoir à aller les ranger en mémoire. Par exemple, l'adresse du sommet de la pile est souvent rangée dans un registre appelé SP (*Stack Pointer*).

La communication entre l'unité centrale et la mémoire est faite par des nappes de fils appelées *bus*. Si la nappe comporte par exemple 32 fils, on dit que c'est un bus 32 bits car on peut y transmettre en parallèle 32 informations élémentaires. Sur ces fils peuvent circuler des données, des adresses mémoires, des ordres pour contrôler la synchronisation des opérations ; cela a conduit à distinguer entre bus des données, bus d'adresse, bus de contrôle. La communication entre l'unité centrale et la mémoire constitue le goulot d'étranglement de l'architecture de von Neumann des processeurs. Le rôle des registres est d'essayer de réduire l'utilisation des bus. On peut représenter l'architecture usuelle d'un ordinateur par le schéma suivant :



Exécution d'une instruction par un microprocesseur

L'exécution d'une instruction est classiquement décomposée en trois étapes : chargement, décodage et exécution :

chargement : - on lit l'instruction dans la mémoire à l'adresse contenue dans PC,

- on place cette instruction dans le registre d'instruction IR,
- on incrémente la valeur de PC.

décodage : l'unité de contrôle décode l'instruction obtenue et produit les signaux à envoyer pour lancer son exécution.

exécution : - si cette instruction utilise des opérandes situés en mémoire, on les charge dans des registres,

- on exécute l'instruction,
- on stocke l'éventuel résultat.

Cette architecture conduit à classer les instructions machines en trois catégories :
- les instructions qui réalisent des transferts de données entre les registres ou la mémoire,
- les instructions qui réalisent des calculs arithmétiques ou logiques élémentaires,
- les instructions qui modifient le séquencement des instructions en agissant sur la valeur de PC.

Certaines architectures (dites pipe-line) utilisées dans les processeurs RISC[2] permettent d'accélérer ce cycle, en commençant à charger l'instruction suivante pendant que l'on exécute l'instruction courante.

12.8 Calculabilité

Les machines de Turing ne présentent guère d'intérêt pratique, mais leur simplicité en fait un outil théorique important pour étudier la calculabilité et la complexité des fonctions. Par exemple, on peut utiliser une machine de Turing pour faire des calculs sur les entiers. On code un entier en numération unaire : l'entier n est représenté par une suite de n symboles 1.

Fonctions calculables

On dit qu'une machine de Turing calcule la fonction f de $\mathbf{N}$ dans $\mathbf{N}$ si, pour tout entier n, la machine partant d'une bande représentant n en unaire, termine avec une représentation de $f(n)$ sur sa bande (ou ne termine pas si la fonction n'est pas définie pour cette valeur n). On étend facilement cette notion aux fonctions à plusieurs variables entières en codant un tuple d'entiers en numération unaire par le codage de chacune des composantes séparées par un blanc.

Définition 4 Une fonction est dite *Turing-calculable* si l'on peut trouver une machine de Turing qui calcule cette fonction.

On peut définir de nombreuses notions de calculabilité plus ou moins naturelles par exemple, les fonctions calculables par un programme Scheme. On démontre que toutes les définitions «raisonnables» conduisent à la même classe de fonctions que l'on appelle «*récursives*». Ce qui a conduit le logicien Church à énoncer la thèse :
«Toute fonction calculable par un procédé effectif quelconque est récursive.»
Cela permet de dire que tous les langages de programmation ont la même puissance de calcul, même si les facilités de programmation peuvent être très différentes. Pourquoi utiliser Scheme plutôt qu'un langage machine? C'est pour laisser au compilateur le soin de traduire un programme lisible par un homme en une suite d'instructions exécutables par une machine. Mais la clarté d'expression d'un langage a souvent un coût en termes d'efficacité du code produit pour un type d'architecture. Le concepteur d'un langage doit trouver un compromis entre ces deux exigences. C'est en ce sens que l'on a pu dire que Lisp réalise un «optimum local» dans l'espace des langages de programmation, et c'est l'une des raisons de sa longévité.

Fonctions non calculables

Mais revenons aux fonctions calculables : y a-t-il des fonctions non calculables? La réponse est *oui* ! On le montre avec un simple argument de dénombrabilité :

[2] *Reduced Instruction Set Computer*, c.-à-d. : ordinateur à jeu d'instructions réduit. Plus que le jeu des instructions, c'est la structure plus uniforme des instructions et l'architecture pipe-line qui caractérise cette nouvelle génération de machines.

l'ensemble des fonctions de **N** dans **N** n'est pas dénombrable, alors que l'ensemble des fonctions Turing-calculable l'est. En effet, l'ensemble des machines de Turing est dénombrable, puisque l'on peut numéroter effectivement les tables de transition et les codes possibles. Mais cet argument n'est pas très satisfaisant, il laisse à penser que les fonctions non calculables sont des monstres que l'on ne peut jamais rencontrer. Au contraire, voici un exemple.

A une numérotation des machines de Turing correspond par définition une numérotation des fonctions Turing-calculables et donc des fonctions récursives:
$k \in \mathbf{N} \to f_k$.
Considérons la fonction H :

$$n \in \mathbf{N} \to H(n) = 1 \quad \textit{quand} \quad f_n(n) \quad \text{est définie et } 0 \textit{ sinon.}$$

On va montrer que H n'est *pas* récursive. Si elle l'était, la fonction :

$$n \in \mathbf{N} \to G(n) = f_n(n) + 1 \quad \textit{quand} \quad H(n) = 1 \textit{ et } 0 \textit{ sinon}$$

serait aussi calculable. Mais si g était calculable, il existerait un entier p_0 tel que $G = f_{p_0}$ ce qui est impossible car par construction $G(p_0) \neq f_{p_0}(p_0)$.
Cette contradiction prouve par l'absurde que la fonction H n'est pas récursive.

Machine de Turing universelle

De même qu'un interprète Scheme permet de calculer toute fonction calculable, on démontre que l'on peut construire une machine de Turing *universelle*, au sens où elle peut calculer toute fonction Turing-calculable. Précisons le sens de cet énoncé. Une machine de Turing est définie par sa table, or on peut tout coder avec deux symboles 0 et 1, de sorte que la donnée d'une machine de Turing T et d'un code C pour cette machine peut être considérée comme étant un code TC pour une autre machine.
On dit qu'une machine de Turing TU est universelle si, pour toute machine de Turing T et pour tout code C, la machine TU avec le code TC termine (ou non) avec la même configuration sur sa bande que la machine T avec le code C.
Grâce à l'existence d'une machine universelle, on peut prouver qu'il n'existe pas d'algorithme pour décider si, étant donnée une machine de Turing et une donné, l'exécution se termine ou non. C'est le problème de l'arrêt, en d'autres termes, on sait qu'il n'existe pas de procédé général permettant de savoir à l'avance si un programme boucle ou non.

Propriété décidable

Parallèlement à la notion de fonction calculable correspond la notion de propriété *décidable*. Une propriété P est une fonction de $\mathbf{N^k}$ à valeur vrai ou faux ou, ce qui revient au même, à valeur 1 ou 0 ; elle est dite décidable si cette fonction est récursive, autrement dit, si l'on peut trouver un procédé effectif pour décider en chaque point si sa valeur est vraie ou fausse. Par exemple, la propriété définie par $P(n) =$ vraie si n est un nombre premier, est décidable, en revanche on a vu plus haut que la propriété d'arrêt d'une machine de Turing est non décidable. On

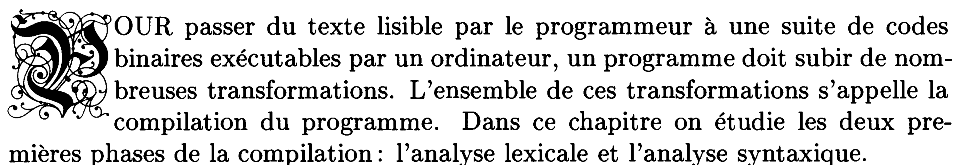
ignore si la propriété suivante est décidable : $P(n)$ = vraie si le développement décimal de π contient quelque part une suite d'exactement n chiffres 5 consécutifs. Il existe de nombreuses propriétés indécidables en logique et en mathématique, ce qui est heureux car cela signifie que la découverte dans ces domaines ne se réduira jamais à l'exécution de méthodes automatiques !

12.9 De la lecture

Pour les automates finis et les expressions régulières on renvoie à [ASU90].
Les liens entre automates et circuits digitaux sont développés dans [WH90] qui traite aussi de la structure des microprocesseurs. L'architecture des ordinateurs est détaillée dans [Tan77, HP94].
On trouvera des informations sur les automates cellulaires dans [PJS92].
Pour les machines de Turing et la calculabilité on renvoie par exemple à [Lal90].

Chapitre 13

Analyse lexicale et syntaxique



POUR passer du texte lisible par le programmeur à une suite de codes binaires exécutables par un ordinateur, un programme doit subir de nombreuses transformations. L'ensemble de ces transformations s'appelle la compilation du programme. Dans ce chapitre on étudie les deux premières phases de la compilation : l'analyse lexicale et l'analyse syntaxique.
Les briques de base d'un programme s'appellent des lexèmes : nombres, identificateurs, chaînes, ... Les caractères qui composent un programme sont regroupés en lexèmes durant la phase d'analyse lexicale. Les expressions régulières sont un formalisme commode pour décrire les diverses catégories de lexèmes. On fait le lien entre expressions régulières et langages reconnus par les automates finis. Ensuite, on regarde si les lexèmes peuvent s'ordonner selon un ensemble de règles dépendant du langage à analyser. C'est la phase de l'analyse syntaxique. Les expressions régulières ne sont pas assez puissantes pour décrire la syntaxe des langages de programmation, on utilise la notion de grammaire formelle. Enfin, on profite souvent de l'analyse syntaxique pour associer au programme analysé une forme représentant mieux sa structure profonde et appelée syntaxe abstraite. On illustrera ces différentes phases dans le cas du langage des expressions arithmétiques. Bien que l'analyse des programmes soit la motivation majeure pour un informaticien, les méthodes d'analyse présentées peuvent s'appliquer à beaucoup d'autres domaines.

13.1 Expressions régulières et langages associés

On a souvent besoin de désigner une famille de mots partageant un même modèle. On connaît l'utilisation de caractères de type joker pour désigner un ensemble de fichiers. La recherche de mots dans un fichier est un autre exemple où l'on souhaite pouvoir envisager une classe de mots. Si l'on ne cherche qu'un nombre fini de mots, il suffit de les énumérer. Mais si l'on désire chercher toutes les occurrences d'un entier quelconque dans un texte, il faut donner la forme générale de l'écriture d'un entier. On peut dire qu'elle commence éventuellement par un signe + ou - et qu'il

y a ensuite une suite d'au moins un chiffre. Pour faire cette description, on a utilisé l'alternative, la concaténation et la répétition. Ces opérations sont à la base du langage des expressions régulières.

Définition d'une expression régulière

Définition 1 Etant donné un alphabet, une *expression régulière* r sur cet alphabet est par définition soit :

- un symbole de l'alphabet,
- le symbole ϵ,
- le symbole $\emptyset$,
- une expression régulière entre parenthèse $(\; r \;)$,
- l'alternative de deux expressions régulières $r \mid s$,
- le produit de deux expressions régulières $r \; . \; s$,
- l'étoile d'une expression régulière r^*.

Les expressions régulières réduites à un symbole, ou à ϵ, ou à $\emptyset$ sont dites *atomiques* car les autres sont construites à partir d'elles.
Par exemple, sur l'alphabet $A = \{0, 1\}$ l'expression $(0.0)^*|(1.1)^*$ est une expression régulière.
Comme pour les expressions arithmétiques on peut convenir de règles de priorité pour diminuer l'usage des parenthèses ; on convient que par ordre de priorité décroissante on a l'étoile, puis la concaténation, puis l'union.

Langage associé à une expression régulière

Une expression régulière est un moyen fini pour décrire des langages pouvant être infinis. Le langage associé à une expression régulière r est noté $LANG(r)$, il est défini par les règles suivantes :

- $LANG(a) = \{a\}$; si a est un élément de l'alphabet considéré,
- $LANG(\epsilon) = \{\epsilon\}$; le langage réduit au mot vide,
- $LANG(\emptyset) = \emptyset$; le langage vide,
- $LANG((r)) = LANG(r)$; le même langage que r,
- $LANG(r \mid s) = LANG(r) \cup LANG(s)$; la réunion ensembliste des langages,
- $LANG(r \; s) = LANG(r).LANG(s)$; le produit[1] des langages associés,
- $LANG(r^*) = (LANG(r))^*$; l'étoile du langage associé à r.

[1] Au sens du chapitre 12 §2.

Pour faciliter l'écriture, on se permet aussi de nommer des expressions régulières par un identificateur, ce qui revient à ajouter la présence de variables dans les expressions régulières.
Par exemple, appelons `chiffre` l'expression régulière
`( 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 )`. Alors:

- les entiers naturels sont décrits par l'expression:

  `entier = chiffre .  (chiffre)*`

  car, d'après la définition de l'étoile, l'expression `(chiffre)*` représente une suite finie (ou vide) de chiffres.

- les entiers relatifs par:

  `entier-relatif = (- | + | ϵ ) .  entier`

- les nombres décimaux par:

  `entier-relatif | (entier-relatif .  "," .  entier)`

Les identificateurs admissibles de certains langages de programmation sont constitués d'une lettre suivie éventuellement de lettres ou de chiffres:
`ident = lettre .  (lettre | chiffre) *`
où `lettre` désigne l'expression régulière `(A | B |...| Z | a | b |...| z)`

Exercice 1 *Décrire les commentaires en Scheme par une expression régulière, on conviendra de noter la fin de ligne par le symbole* `eol`.

Exercice 2 *On dit que deux expressions régulières sont égales si elles désignent le même langage. Démontrer les égalités suivantes:*
$(r|r) = r$
$((r_1|r_2)|r_3) = (r_1|(r_2|r_3))$ *c'est l'associativité* [2] *de* $|$
$(r^*)^* = r^*$
$(r_1 \mid r_2)^* = (r_1^* \,.\, r_2^*)^*$ *où les* r_j *sont des expressions régulières.*
On inclut parfois dans les expressions régulières l'expression r^+, *c'est une abréviation de l'expression* $r.r^*$. *A-t-on* $(r^+)^* = (r^*)^+$ *?*

13.2 Recherche d'une expression régulière

Combinaison de fonctions de recherche

Le problème central est la reconnaissance des mots du langage associé à une expression régulière. Etant donné une expression régulière r et un mot m, il s'agit de décider si $m \in LANG(r)$ [3] Plus généralement, on se propose de rechercher un *préfixe* de m qui soit un mot de $LANG(r)$.

[2] On a déjà fait usage de cette associcativité pour écrire les expressions régulières comme `chiffre` ou `lettre`

[3] Les fonctions de la famille «grep» des systèmes UNIX servent à la recherche d'expressions régulières.

Pour résoudre ce problème, on va définir une fonction qui retourne deux valeurs : un préfixe conforme à l'expression régulière et le reste du mot.
Pour construire la fonction de recherche d'une expression régulière, on va combiner les fonctions de recherche des sous-expressions qui la constituent.
Pour cela, on associe à chaque méthode de construction d'expression régulière, une méthode pour combiner les fonctions de recherche des expressions régulières qui la composent. Aux trois constructeurs d'expressions régulières correspondent trois fonctions de combinaison :

- La recherche d'un préfixe accepté par l'expression $r_1|r_2$, consiste à utiliser la fonction de recherche associée à $r1$. En cas de succès c'est fini, sinon on appelle la fonction de recherche associée à r_2.[4] La considération de l'autre possibilité en cas d'échec est l'une des deux causes de retour en arrière dans la recherche d'un préfixe.

- La recherche d'un préfixe accepté par l'expression $r_1.r_2$, consiste à utiliser la fonction de recherche associée à r_1. En cas d'échec c'est fini, en cas de succès on continue avec la fonction de recherche associée à r_2 que l'on applique au reste du mot,

- La recherche d'un préfixe accepté par l'expression r^* , consiste à utiliser la fonction de recherche associée à r. En cas d'échec, on a accepté le mot vide, en cas de succès on refait de même avec le reste du mot. Ce cas d'échec est la deuxième cause de retour en arrière durant la recherche.

  Bien entendu, les deux possibilités de retour en arrière sont les principales sources d'inefficacité de cette méthode générale de recherche.

Fonction de recherche d'expressions régulières

Pour l'implantation, on représente un mot par une liste de symboles. Comme on retourne deux valeurs : le préfixe trouvé et le suffixe restant, il est commode d'utiliser un style de programmation par passage de continuation.
La fonction de recherche associée à une expression régulière sera une fonction de deux arguments : le mot où l'on cherche et une continuation `k` à deux arguments : le préfixe trouvé et le suffixe restant. Ce sera donc une lambda de la forme

```
(lambda (mot k) (k  prefixe reste))
```

Si l'on ne trouve pas de préfixe conforme à l'expression régulière on convient de dire que le mot trouvé est `#f`. On applique dans chaque cas les principes énoncés plus haut.
Appelons `rech1` et `rech2` les fonctions de recherche de deux expressions régulières r_1 et r_2, alors la fonction de recherche de l'expression $r_1|r_2$ est donnée par la fonction `rech-ou` définie par :

```
(define (rech-ou  rech1  rech2)
  (lambda (mot k)
```

[4] Cette stratégie favorise la reconnaissance par r_1 quand le mot est aussi acceptable par r_2.

```
    (rech1 mot (lambda (prefixe1 reste1)
                 (if prefixe1
                   (k prefixe1 reste1)
                   (rech2 mot k))))))
```

La fonction de recherche de l'expression $r_1.r_2$ est définie par :

```
(define (rech-et  rech1  rech2)
  (lambda (mot k)
    (rech1 mot (lambda (prefixe1 reste1)
                 (if prefixe1
                   (rech2  reste1 (lambda (prefixe2  reste2)
                                    (k (if prefixe2
                                         (append prefixe1 prefixe2)
                                         #f)
                                       reste2)))
                   (k #f mot))))))
```

La fonction de recherche de l'expression r^* est donnée par :

```
(define (rech-*  rech)
  (lambda (mot k)
    (rech mot (lambda (prefixe1 reste1)
                (if prefixe1
                  ((rech-* rech) reste1
                                 (lambda (prefixe2 reste2)
                                   (k (if prefixe2
                                        (append prefixe1 prefixe2)
                                        #f)
                                      reste2)))
                  (k '() mot))))))
```

On en déduit immédiatement une fonction de recherche pour l'expression $r^+ = r.r^*$

```
(define (rech-+  rech)
  (rech-et  rech  (rech-*  rech)))
```

Enfin, on traite directement les fonctions de recherche d'expressions régulières atomiques. Si l'expression régulière est réduite à un élément de l'alphabet du langage, que l'on représente par un symbole, on regarde si ce symbole est le premier symbole du mot.

```
(define (rech-symb symb)
  (lambda (mot k)
    (if (and (pair? mot) (eq? symb (car mot)))
      (k (list symb) (cdr mot))
      (k #f mot))))
```

On trouve toujours l'expression ϵ, avec comme préfixe le mot vide :

```
(define rech-epsilon
  (lambda (mot k)(k '() mot)))
```

Il est assez pénible d'écrire à la main la fonction de recherche associée à une expression régulière. Supposons, pour simplifier, les expressions régulières représentées en notation préfixée. Par exemple, avec des notations évidentes, la forme préfixe de l'expression $((a \mid b)^*) . c^+$ est représentée par `(et (* (ou a b)) (+ c))`.
Il est facile de générer la fonction de recherche associée à une expression régulière r donnée sous forme préfixe

```
(define (generer-fct-rech r)
  (cond ((eq? 'epsilon r) rech-epsilon)
        ((symbol? r)(rech-symb r))
        (else (case (car r)
                ((et) (rech-et (generer-fct-rech (cadr r))
                               (generer-fct-rech (caddr r))))
                ((ou) (rech-ou (generer-fct-rech (cadr r))
                               (generer-fct-rech (caddr r))))
                ((*) (rech-*   (generer-fct-rech (cadr r))))
                ((+) (rech-+   (generer-fct-rech (cadr r))))))))
```

Finalement, la fonction qui retourne le préfixe d'un mot conforme avec une expression régulière est donnée par :

```
(define (rech-exp-reguliere r  mot)
  ((generer-fct-rech r) mot (lambda (x y) x)))
```

Cherchons un préfixe d'une liste compatible avec l'expression $((a \mid b)^*) . c^+)$.

```
? (rech-exp-reguliere '(et (* (ou a b)) (+ c)) '(b a a c c d e)) -> (b a a c c)
? (rech-exp-reguliere '(et (* (ou a b)) (+ c)) '(b c a c c d e)) -> (b c)
? (rech-exp-reguliere '(et (* (ou a b)) (+ c)) '(c a a c c d e)) -> (c)
? (rech-exp-reguliere '(et (* (ou a b)) (+ c)) '(d a a c c d e)) -> #f
? (rech-exp-reguliere '(ou a epsilon)  '( a b a c c d e)) -> (a)
? (rech-exp-reguliere 'epsilon '(d a a c c d e)) -> ()
```

Ce programme fournit un bel exemple de programmation avec des fonctionnelles mais la méthode utilisée n'est pas totalement satisfaisante. En effet, le résultat peut dépendre de la façon dont on écrit l'expression régulière. Par exemple, si l'on cherche l'expression $x^*.x.y$ dans le mot `xxy` on aura un échec, car la sous-expression x^* lira tous les `x`. En revanche, il y aura succès si on utilise l'expression régulière équivalente x^+y. La stratégie utilisée est trop déterministe, on devrait remettre en cause chaque choix qui aboutit à un échec. On indique au paragraphe suivant une telle méthode, elle est basée sur la notion d'automate fini non déterministe.

Exercice 3 *Généraliser à la recherche de la première sous-liste compatible avec une expression régulière.*

Structure du programme de recherche d'expressions régulières

`(rech-exp-reguliere r mot)`
Retourne le préfixe du mot qui est conforme à l'expression régulière r.

`(generer-fct-rech r)`
Construit la fonction de recherche associée à une expression regulière donnée sous forme préfixée.

`(rech-ou rech1 rech2)`
Combine les fonctions de recherche de deux expressions régulières pour en faire une fonction de recherche du ou de ces expressions.

`(rech-et rech1 rech2)`
Combine les fonctions de recherche de deux expressions régulières pour en faire une fonction de recherche du et de ces expressions.

`(rech-* rech)`
Construit la fonction de recherche de l'étoile d'une expression régulière à partir de la fonction de recherche de cette expression.

`(rech-+ rech)`
Construit la fonction de recherche du + d'une expression régulière à partir de la fonction de recherche de cette expression.

`(rech-symb symb)`
Construit la fonction de recherche d'une expression régulière réduite à un symbole.

`rech-epsilon`
La fonction de recherche de l'expression ϵ.

Exemples avec les nombres

A titre d'exemple, voici quelques fonctions de recherche pour des catégories de nombres. On commence par définir une fonction de recherche pour les chiffres, on l'écrit directement sans passer par l'expression `(0 | 1 | ....)` :

```
(define  rech-chiffre
  (lambda (mot k)
    (if (and (pair? mot)(chiffre? (car mot)))
      (k (list (car mot)) (cdr mot))
      (k #f mot))))
```

On a utilisé le prédicat `chiffre?` défini par :

```
(define (chiffre? s)
  (and (integer? s)(< s 10)(<= 0 s)))
```

Comme un entier naturel est décrit par `chiffre+`, on obtient sa fonction de recherche :

```
(define  rech-entier-naturel
  (rech-+  rech-chiffre))
```

```
? (rech-entier-naturel '(1 2 3 a b) (lambda (prefixe reste) prefixe))
(1 2 3)
```

Pour un entier quelconque, il faut accepter un éventuel signe :

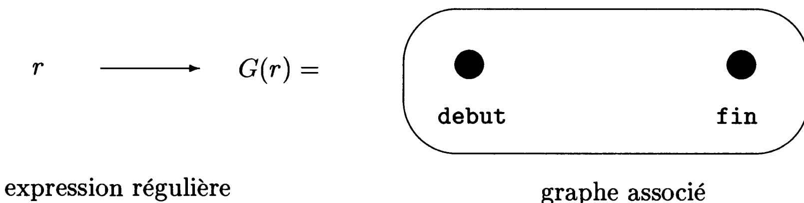
```
(define rech-entier
  (rech-et (rech-ou (rech-ou (rech-symb '+)(rech-symb '-)) rech-epsilon)
           rech-entier-naturel))

? (rech-entier  '(+ 1 5 a b) (lambda (x y) x)) -> (+ 1 5)
? (rech-entier  '(4 1 5 a b) (lambda (x y) x)) -> (4 1 5)
? (rech-entier  '(- a b) (lambda (x y) x)) -> #f
```

Exercice 4 *Ecrire la fonction de recherche d'un nombre décimal.*

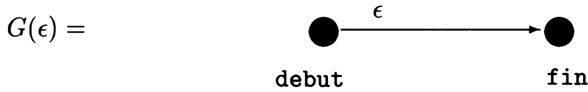
13.3 Expressions régulières et automates finis

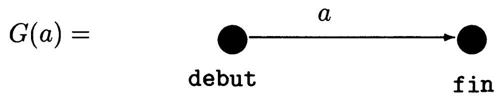
Graphe d'une expression régulière

On peut visualiser une expression régulière r par un graphe orienté $G(r)$ dont les arcs sont étiquetés par des expressions régulières atomiques. Ce graphe possède toujours un nœud appelé `debut` et un nœud appelé `fin`. On va le construire de sorte que la succession des étiquettes des chemins de `debut` à `fin` engendre les mots du langage de l'expression régulière. Cette construction et sa justification suivent exactement la définition récursive d'une expression régulière.

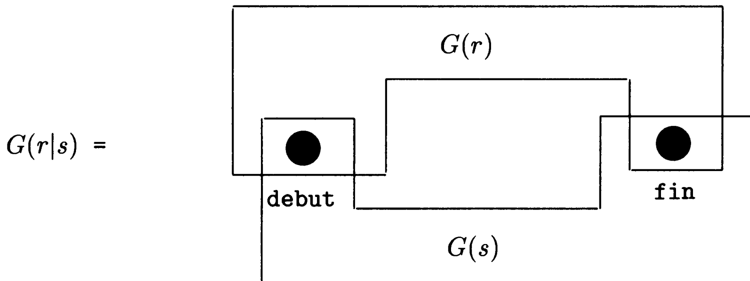


Pour les expressions régulières atomiques, on a les solutions évidentes :

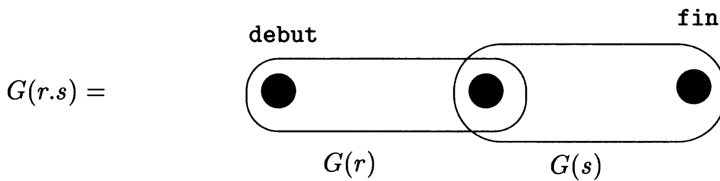




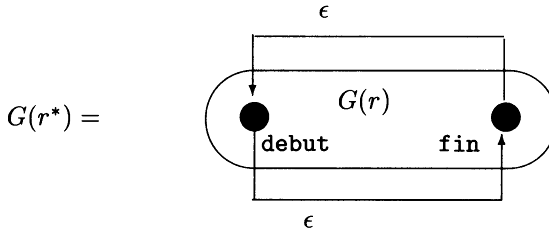
Considérons les expressions régulières composées :
Pour le ou, on peut emprunter au choix le graphe de r ou celui de s :



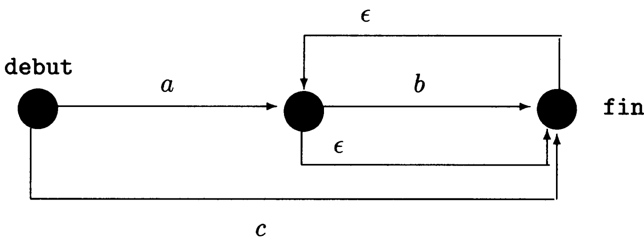
Pour le `et`, on doit emprunter successivement le graphe de r puis celui de s :



Pour l'opération étoile, on peut aller directement à la fin, ou emprunter un nombre fini de fois le graphe de r :



A titre d'exemple, voici le graphe associé à l'expression $(a \,.\, b^* \mid c)$ (après suppression d'arcs ϵ) superflus) :



Graphes et automates non déterministes

Ces graphes nous rappellent les graphes de transition associés aux automates finis du chapitre 13, où l'état initial serait le nœud `debut` et l'état final le nœud `fin`. Il y a effectivement des points communs, mais il y a aussi des différences :

- les graphes de transition des automates ont, depuis chaque nœud, une arête allant à chaque symbole du langage ce qui n'est pas obligatoire pour les graphes des expressions régulières,
- on doit étiqueter les arcs d'un graphe d'automate déterministe par les éléments de l'alphabet, ce qui exclut l'étiquette ϵ.

En fait, les graphes des expressions régulières correspondent à la notion plus générale d'*automates finis non déterministes.* c'est-à-dire les automates pour lesquels la «fonction» de transition est une relation. Par conséquent, ces automates reconnaissent les mêmes langages que ceux qui peuvent se décrire par les expressions régulières. On démontre que cette classe de langages est la même que la classe des langages, dits réguliers, reconnus par les automates finis déterministes. On renvoie à la bibliographie pour la preuve mais le principe en est assez simple. On associe à un automate non déterministe un automate déterministe qui reconnaît le même langage. Pour construire cet automate déterministe, on définit ses états comme étant l'ensemble de toutes les parties des états de l'automate non déterministe.
Le passage d'une expression régulière à un automate fini qui reconnaît le même langage est une espèce de compilation puisque la reconnaissance est alors faisable par une machine simple. En particulier, on a supprimé la présence des retours

en arrière en cas d'échec, ce qui présage d'une meilleure efficacité que la méthode basée sur les expressions régulières.

13.4 Analyse lexicale

Pour étudier la structure d'un programme (on dit faire son analyse syntaxique) on doit commencer par extraire les briques de base ou *lexèmes*. Les catégories de lexèmes varient selon les langages, mais on a généralement : les identificateurs, les nombres, les booléens, les mots-clés, les symboles spéciaux (par exemple, les parenthèses en Scheme), ... Toutes ces catégories peuvent être décrites par des expressions régulières.
La reconnaissance des lexèmes d'un programme doit être une opération rapide car le processus d'analyse syntaxique y fait appel un grand nombre de fois. Aussi, les lexèmes d'un langage de programmation sont définis par des expressions régulières présentant un non-déterminisme très limité. Plus précisément, on s'arrange pour qu'en général le caractère suivant lève l'ambiguïté en cas de doute sur la branche à suivre. Dans ces conditions, on peut écrire des analyseurs lexicaux sans passer par le processus de détermisation d'expressions régulières.

Catégories lexicales

Un analyseur lexical donnera pour chaque lexème sa *classe* et sa *valeur*. Par exemple, si le lexème est un entier, on retourne comme classe entier et comme valeur le nombre qu'il représente. L'analyse lexicale s'occupe aussi de reconnaître les commentaires pour pouvoir les sauter.
On se propose d'écrire un analyseur lexical assez générique pour être adaptable aux situations les plus courantes. On définit quatre classes de lexèmes que l'on trouve souvent dans les langages de programmation :

identificateurs Décrits par l'expression régulière : `lettre . (lettre | chiffre)*`

mots clés Décrits par l'expression régulière : `lettre . (lettre | chiffre)*`

nombres (pour simplifier, on se restreint aux entiers naturels) Ils sont décrits par l'expression régulière `chiffre . chiffre*`.

symboles spéciaux on considère des symboles spéciaux utilisés pour écrire les expressions arithmétiques : quelques opérateurs et les parenthèses, on se restreint à `( + | - | * | / | ( | ) | = | < | > )`.

La distinction entre identificateur et mot-clé se fait ensuite par consultation d'une table contenant tous les mots clés du langage. Pour simplifier, on a retenu que des symboles spéciaux mono-caractère.

Implantation d'un analyseur lexical

La méthode de reconnaissance est simple, on sélectionne sur le premier caractère et on lira le lexème jusqu'à un séparateur.

L'implantation est basée sur la fonction `make-lex` qui sert à créer un analyseur lexical, elle prend en paramètre :
- le flux d'entrée,
- la table des mots-clés qui permettra de faire la distinction entre un mot-clé et un identificateur,
- la table des symboles spéciaux, ce sera une a-liste qui à chaque caractère de symbole spécial associe son nom (par exemple `((#\= . egale)(#\< . pp) ...)`.

Décrivons les fonctions locales utilisées par `make-lex`.
- La fonction `car-suivant` sert à placer dans la variable locale `carCourant` le caractère suivant du flux.
- La fonction `Lire-commentaire` sert à sauter les commentaires, à titre d'exemple on a utilisé la syntaxe Scheme des commentaires.
- La fonction `Nom-symbole-special` associe à un caractère spécial son nom trouvé dans la a-liste `table-symboles-speciaux` puis appelle le carac. suivant.
- La fonction `Lire-symbole` retourne la chaîne maximale constituée de lettres ou de chiffres et commençant par une lettre donnée.
- La fonction `Classe-symbole` sert à distinguer la classe des mots-clés de celle des identificateurs en consultant la a-liste `table-mots-cles`.
- La fonction `Lire-entier` lit une suite maximale de chiffres et calcule le nombre associé.
- La fonction `lex-suivant` est la lambda expression retournée par `make-lex`, elle saute les espaces puis calcule le lexème suivant présent dans le flux. Elle retourne un doublet composé de la classe et de la valeur du lexème.

```
(define (make-lex flux  table-mots-cles  table-symboles-speciaux)
  (let ((carCourant       (read-char flux))
        (Lseparateurs     '(#\space #\newline ))
        (debutCommentaire #\;)
        (finCommentaire   #\newline)
        (LcarSpeciaux     '(#\>  #\<  #\= #\( #\)  #\+ #\-  #\*  #\\))
        (code-0           (char->integer #\0))
             )
    (letrec (
            (carSuivant (lambda ()(set! carCourant (read-char flux))))

            (Lire-commentaire
              (lambda ()
                (carSuivant)
                (cond ((eof-object? carCourant))
                      ((char=? carCourant finCommentaire)(carSuivant) )
                      (else (lire-commentaire)))))

            (Classe-caractere-special
              (lambda (char)
                (carSuivant)
                (let ((nom-symbole (cdr (assq char table-symboles-speciaux
                  (cons  nom-symbole (string char))))
```

```
          (Lire-symbole
            (lambda (str)
              (carSuivant)
              (if (or (char-alphabetic? carCourant)
                      (char-numeric? carCourant))
                (lire-symbole (string-append  str (string carCourant)))
                str)))

          (Classe-symbole
            (lambda (str)
              (let ((doublet (assoc str table-mots-cles)))
                (if doublet
                  (cons  'mot-cle (cdr doublet))
                  (cons  'identificateur str)))))

          (Lire-entier
            (lambda (n)
              (carSuivant)
              (if (char-numeric? carCourant)
                (lire-entier (+ (* 10 n) (- (char->integer carCourant)
                                            code-0)))
                (cons  'entier  n))))

          (lex-suivant
            (lambda ()
              (cond ((eof-object? carCourant) '(eof eof))
                    ((memq carCourant Lseparateurs)
                     (carSuivant) (lex-suivant))
                    ((char=? carCourant debutCommentaire)
                     (lire-commentaire) (lex-suivant))
                    ((memq carCourant LcarSpeciaux)
                     (Classe-caractere-special   carCourant))
                    ((char-alphabetic? carCourant)
                      (Classe-symbole (lire-symbole (string carCourant))))
                    ((char-numeric? carCourant)
                      (Lire-entier (- (char->integer carCourant) code-0))
                   ))))

      lex-suivant)))
```

Pour faire l'analyse lexicale d'un fichier source, on ajoute la fonction :

```
(define (analyse-lexicale source table-mots-cles table-symboles-speciaux)
  (let ((flux (open-input-file source)))
    (make-lex flux table-mots-cles table-symboles-speciaux )))
```

Tests

Testons sur un fichier `*source*` contenant la définition usuelle de factorielle :

```
(define (fac n)
  (if (= 0 n)
```

```
    1
    (* (fac (- n 1)))))
```

On se donne des extraits de la table des mots-clés de Scheme et des caractères spéciaux suffisants pour traiter cet exemple.

```
? (define *Table-mots-Cles*    '(("define" . define)("if" . if)))

? (define *Table-symboles-speciaux*
    '((#\- . moins) (#\* . mul)  (#\( . lpar)( #\) . rpar)(#\= . egale)))
```

On définit l'analyseur lexical du fichier `*source*` par :

```
? (define *lex*
    (analyse-lexicale source *table-mots-cles* *table-symboles-speciaux*))
```

Les appels successifs à la fonction `*lex*` donnent les lexèmes :

```
? (*lex*)
(symb-special . lpar)

? (*lex*)
(mot-cle . define)

? (*lex*)
(symb-special . lpar)

? (*lex*)
(identificateur . "fac")

...

? (*lex*)
(eof . eof)
```

Cet analyseur lexical est un exemple que l'on peut adapter facilement à d'autres langages.
Supposons que les commentaires commencent par `--` et que l'on accepte les entiers relatifs, alors la lecture du caractère `-` ne permet pas de distinguer entre : commentaire, entier négatif ou le signe `-`. Pour cela, il est commode d'introduire une nouvelle fonction locale `lire-commentaire-ou-entier-negatif-ou-signe-`. Cette fonction lèvera l'ambiguïté selon que le caractère suivant est le caractère `-`, un chiffre ou un séparateur.

Exercice 5 *Ajouter un traitement des erreurs avec localisation de l'erreur dans le texte source. Pour cela, on ajoute deux variables locales dans `make-lex` : `NoLigne` et `NoCaractere` qui indiqueront la position du caractère courant dans le texte source. Ces variables seront actualisées à chaque lecture de caractère et à chaque lecture du caractère `#\newline`.*

Il existe des générateurs d'analyseurs lexicaux. On y spécifie chaque classe de lexèmes par une expression régulière puis le système génère un automate déterministe optimisé pour la reconnaissance des lexèmes.

13.5 Grammaires formelles

Les expressions régulières nous ont permis de décrire la structure de diverses classes de lexèmes, mais elles ne sont pas assez générales pour décrire la structure des programmes admissibles pour un langage donné. Par exemple, on a vu que la classe des langages réguliers est une classe trop petite pour contenir le langage des expressions arithmétiques à cause de la profondeur arbitraire des parenthésages. On va décrire une méthode plus générale de description d'un langage mais en utilisant toujours un nombre fini de symboles, sinon il suffirait d'énumérer tous les mots du langage à décrire !

Définition d'une grammaire

Pour décrire un langage sur un alphabet A, on commence par présenter la notion de *grammaire formelle.*

Définition 2 Une grammaire formelle[5] G est la donnée d'un ensemble fini $NT = \{X_1, X_2, \ldots X_n\}$ de symboles (non dans l'alphabet A) appelés *non terminaux* ; par opposition, les symboles de A s'appellent les *terminaux.* Un des non-terminaux, souvent noté S, s'appelle *l'axiome* de la grammaire. Ensuite, on se donne un ensemble fini de règles (ou productions) de la forme : $X_i \to \alpha_i$ où α_i est un mot sur l'alphabet $\Sigma = A \cup NT$. Un même non terminal X_i peut être membre gauche de plusieurs règles ou d'aucune.

Par exemple, on considère l'alphabet $A = \{a\}$, on définit une grammaire G_1 ayant un seul non-terminal $NT = \{S\}$ et deux règles :
$S \to aS$
$S \to \epsilon$.

Langage associé à une grammaire

Pour associer un langage à une grammaire, on doit introduire quelques notions supplémentaires. On dit qu'un mot α de Σ^* se réécrit en un mot β de Σ^* si on obtient β par remplacement dans α d'un des non terminaux par le membre de droite d'une règle. On note $\alpha \longrightarrow \beta$ cette transformation.
Par exemple, avec la grammaire G_1 on a : $aa\underline{S} \longrightarrow aa\underline{aS}$ en remplaçant le S souligné par aS.
On écrit $\alpha \longrightarrow^* \beta$ si l'on peut passer de α à β par un nombre fini (voir 0) de telles transformations : $aSaS \longrightarrow^* aaaSaa$.

Le *langage associé à une grammaire* G est par définition l'ensemble des mots de A^* que l'on peut atteindre en partant de l'axiome de G :

$$Langage(G) = \{m \in A^* \mid \quad S \longrightarrow^* \quad m\}$$

Par exemple, il est facile de voir que $Langage(G_1) = \{\epsilon, a, aa, aaa, \ldots\} = a^*$.

[5]En fait, on définit une classe particulière de grammaires, dites hors contexte, mais elle sera largement suffisante pour nos applications.

Notons au passage que les expressions régulières comme l'union $(a|\ b)$ ou la concaténation $a.b$ peuvent être représentées respectivement par les règles :
$S \rightarrow a$
$S \rightarrow b$
ou
$S \rightarrow a.b$
Cela montre que l'on peut définir avec cette classe de grammaires, dites *hors contexte,* tous les langages réguliers. Mais on peut aussi décrire des langages non réguliers. Par exemple, le langage des expressions bien parenthésées sur l'alphabet réduit aux parenthèses (et) peut se définir par les règles :
$S \rightarrow ()$
$S \rightarrow SS$
$S \rightarrow (S)$

Notons qu'un même langage peut être défini par plusieurs grammaires qui seront dites *équivalentes.* Par exemple, la grammaire G_2 ayant les deux règles
$S \rightarrow Sa$
$S \rightarrow \epsilon$
définit clairement le même langage que G_1.
Signalons que l'on démontre qu'il n'existe pas d'algorithme pour déterminer si deux grammaires quelconques sont équivalentes ou non. C'est un exemple de propriété indécidable, l'étude des grammaires formelles en fournit de nombreux.
On va présenter certaines opérations sur les grammaires en s'appuyant sur l'exemple des expressions arithmétiques. C'est un exemple fondamental car ce type d'expressions se retrouve dans tous les langages de programmation.

Grammaires des expressions arithmétiques

Pour décrire la syntaxe d'un langage de programmation, c'est-à-dire la structure des expressions ou instructions admissibles (indépendamment de leurs significations éventuelles), on va utiliser les grammaires formelles avec des conventions de notations dites BNF (pour *Backus Naur Form*). On écrira les non-terminaux entre crochets, les terminaux entre apostrophes, les deuxièmes membres d'un même non-terminal seront séparés par une barre verticale et la flèche sera remplacée par : : =.

Expressions arithmétiques additives et arbre d'analyse

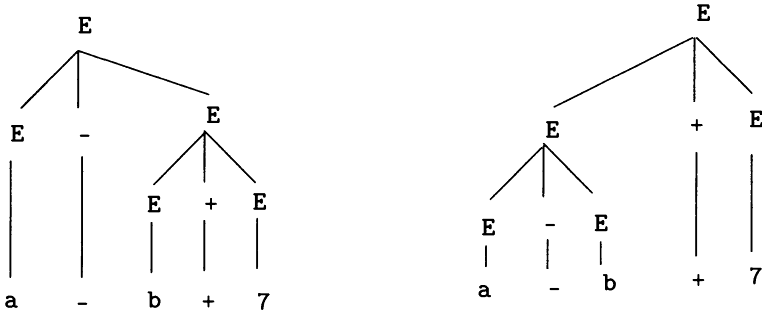
Voici une grammaire $G\pm$ décrivant les expressions additives comme $a - b + 7$:

```
<E>  ::=    <E> '+'  <E>  |   <E>  '-' <E>  |  '('  <E>  ')'
           |   'nombre' |   'identificateur'
```

où `'nombre'` et `'identificateur'` désignent des classes de terminaux ; ici l'axiome est noté <E>.

Un programme est syntaxiquement correct pour une grammaire G si c'est un mot de son langage. C'est le problème de la reconnaissance. Le passage de l'axiome à un mot du langage se fait par application d'une suite de règles à des non-terminaux précis.

On peut visualiser ceci par un arbre (dit d'analyse) où les nœuds correspondent à l'application d'une règle et les feuilles sont les terminaux qui composent de gauche à droite le mot.
Prenons l'exemple de l'expression `a - b + 7` avec la grammaire $G\pm$, on peut trouver *deux* arbres d'analyse :



On dit que la grammaire $G\pm$ est *ambiguë*, car il existe des mots ayant plusieurs arbres d'analyse. Dans le premier arbre, on a `b + 7` comme sous-arbre, ce qui correspond au regroupement `a - (b + 7)`, alors que le deuxième arbre représente le regroupement `(a - b) + 7`. Pour lever l'ambiguïté, c'est-à-dire faire un choix entre ces deux groupements, on remplace $G\pm$ par la grammaire suivante :

```
<E>  ::=   <E>  '+'  <T>  |  <E>  '-'  <T>  |  T
<T>  ::=  '('  <E>   ')'  |  nombre  |  identificateur
```

Avec cette grammaire, la seule possibilité est donnée par le deuxième arbre qui correspond au regroupement `(a - b) + 7`.

Expressions arithmétiques complétées

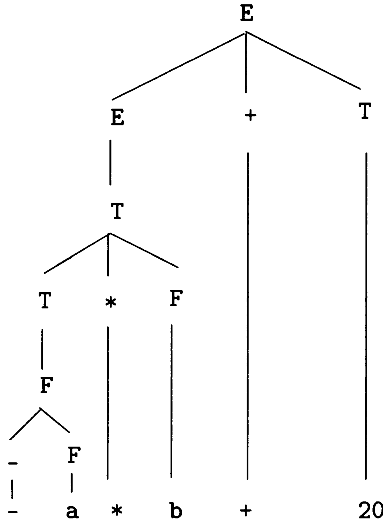
Si l'on ajoute les opérateurs multiplicatifs `*` et `/`, on doit ajouter un non-terminal <F> (F pour facteur) et on remplace la règle pour <T> par :

```
<T> ::= <T> '*' <F> | <T> '/' <F> | F
<F> ::= '(' <E> ')' | 'nombre' | 'identificateur'
```

Enfin, si l'on souhaite disposer du moins unaire, il faut rajouter une alternative au deuxième membre de <F> d'où finalement la grammaire dite ETF :

```
<E>  ::=   <E>  '+'  <T>  |  <E>  '-'  <T>  |  <T>
<T>  ::=   <T>  '*'  <F>  |  <T>  '/'  <F>  |  <F>
<F>  ::=  '(' <E>  ')'  |  '-'  <F>  | 'nombre' | 'identificateur'
```

Où <E> désigne les expressions arithmétiques, <T> les termes multiplicatifs et <F> les facteurs. L'introduction de ces non-terminaux se fait dans l'ordre inverse des conventions de priorité. L'opération la plus prioritaire est le moins unaire puis viennent à égalité `*` et `/` et enfin les deux opérations `+` et `-`.
Par exemple, l'expression `- a * b + 20`, s'analyse de façon unique selon l'arbre :



Ce qui montre que ses sous-expressions se regroupent de la façon suivante : `((- a) * b ) + 20.`

Exercice 6 *Ajouter à la grammaire ETF un non-terminal <P> et un terminal* `^` *de façon à accepter aussi les expressions comportant des puissances* `a^b` *(signifie* `a` *à la puissance* `b`*).*

Elimination de la récursivité à gauche

On dit que la grammaire ETF sous la forme ci-dessus est *récursive à gauche*, car le non-terminal <E> possède une règle où il apparaît en tête de son deuxième membre. La récursivité à gauche est une propriété à éviter pour pouvoir utiliser certaines techniques d'analyse syntaxique comme la descente récursive (voir plus loin).
Mais on peut trouver une grammaire équivalente n'ayant plus cette propriété. La transformation est basée sur la remarque simple suivante : le langage défini par la grammaire récursive gauche :

```
<S> ::= <S> 'a' | 'b'
```

est constitué des mots de la forme `b a*`. Or on a vu que `a*` peut être défini par la grammaire :

```
<X> ::= 'a' <X> | ε
```

Par conséquent, on peut remplacer les règles `<S> ::= <S> 'a' | 'b'` par les règles :

```
<S> ::= 'b' <X>
<X> ::= 'a' <X> | ε
```

qui est une grammaire non récursive gauche.
En appliquant ce procédé à la grammaire ETF, on obtient une grammaire équivalente et non récursive gauche :

```
<E> ::= <T> <E'>
<E'> ::= '+' <T> <E'> | ' -' <T> <E'> | ε
<T> ::= <F> <T'>
<T'> ::= '*' <F> <T'> | '/' <F> <T'> | ε
<F> ::= '(' <E> ')' | '-' <F> | 'nombre' | 'identificateur'
```

Ou encore, en remplaçant <T> <E'> par <E> et <F> <T'> par <T>,

```
<E> ::= <T> <E'>
<E'> ::= '+' <E> | ' -' <E> | ε
<T> ::= <F> <T'>
<T'> ::= '*' <T> | '/' <T> | ε
<F> ::= '(' <E> ')' | '-' <F> | 'nombre' | 'identificateur'
```

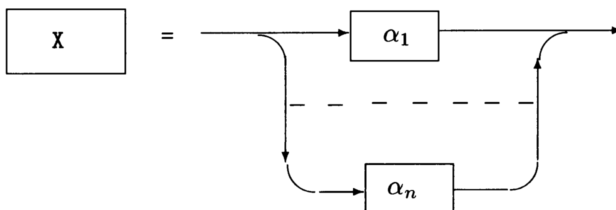
Formalisme BNF étendu

On peut en donner une présentation plus compacte en utilisant le formalisme BNF étendu (EBNF) qui autorise la fonctionnalité de l'étoile. Avec ce formalisme l'expression $\alpha*$ est désignée par $\{\alpha\}$. Par exemple, le langage engendré par le non-terminal <E'> est aussi décrit par {<±> <T>}
avec les règles supplémentaires <±> ::= '+' | '-' .
De même, le langage engendré par <Y> est aussi décrit par {<*/> <F>}
avec les règles supplémentaires <*/> ::= '*' | '/' .
D'où une version EBNF, plus compacte, pour le langage des expressions arithmétiques :

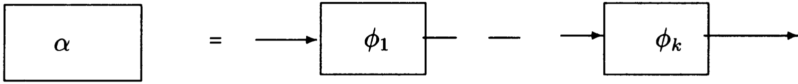
```
<E> ::= <T> {<± > <T>}
<±> ::= '+' | '-'
<T> ::= <F> {<*/> <F>}
<*/> ::= '*' | '/'
<F> ::= '(' <E> ')' | '-' <F> | 'nombre' | 'identificateur '
```

Diagrammes syntaxiques

Il est souvent plus commode de visualiser les règles d'une grammaire par des diagrammes.
A une grammaire on associe un diagramme syntaxique de la manière suivante :
à chaque règle X ::= α_1 | ... | α_n, on associe des diagrammes en parallèle :

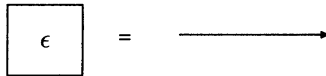


Si $\alpha = \phi_1 \ \ldots \ \phi_k$, alors le diagramme de α est la mise en série des diagrammes des ϕ_i :

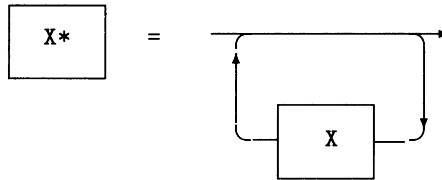


Si ϕ est un terminal `'a'`, alors :

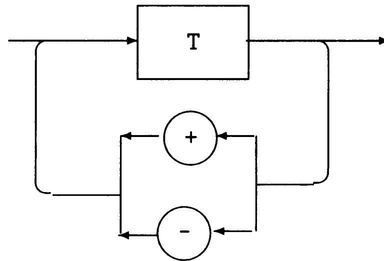
et si c'est ϵ, alors :



L'étoile est représentée par la boucle :



Par exemple, le diagramme syntaxique de la règle `<E>::= <T> {<± > <T>}` est :



A chaque parcours possible, en partant de l'axiome, on associe le mot obtenu en concaténant les terminaux qui étiquettent les flèches. Par construction, l'ensemble de ces mots est le langage défini par la grammaire.

Le formalisme des grammaires formelles est un procédé commode et puissant pour générer des langages. Il reste le problème de la reconnaissance des mots du langage c'est-à-dire la reconnaissance des mots syntaxiquement corrects pour une grammaire donnée.

13.6 Analyse syntaxique descendante

Méthode récursive descendante

On va présenter une méthode d'analyse syntaxique dite *récursive descendante.* Elle est simple à programmer et suffisamment générale pour nos besoins ultérieurs. Elle est qualifiée de descendante car on part de l'axiome et l'on essaye de trouver les règles qui permettent d'arriver au mot à analyser. Elle est récursive car elle consiste à associer à chaque non-terminal une fonction qui appelle récursivement les fonctions associées aux symboles du deuxième membre d'une règle de ce non terminal. Le problème sera le choix de la règle à utiliser quand un non-terminal possède plusieurs seconds membres. Pour cela, on fera une hypothèse qui permettra de décider de la règle à suivre à la vue du lexème suivant.

Pour expliquer cette méthode, commençons par un exemple un peu simpliste.
On se donne trois non-terminaux `<S>, <X>, <Y>`, deux terminaux `'a', 'b'` et la grammaire :

`S ::= X Y` ; production S_1
`X ::= b Y` ; production X_1
`      | a` ; production X_2
`Y ::= a X` ; production Y_1
`      | b` ; production Y_2

Si l'on veut décider que le mot **baab** est dans le langage, on doit le générer en partant de l'axiome `<S>`. Pour `<S>` on n'a pas le choix, on remplace `<S>` par `X Y`, on est ramené à analyser **baab** avec `X Y`. Le choix de la production à utiliser pour `X` est dicté par le premier terminal du mot **baab**. On doit donc choisir la production X_1, il reste à analyser **aab** par `Y Y`. On doit donc remplacer le premier `Y` par **a** `X`, il reste à analyser **ab** par `X Y`. On termine évidemment en appliquant les productions X_2 puis Y_2.

Exercice 7 *Les mots **aabb** et **bbab** sont-ils dans le langage défini par cette grammaire ?*

Programmation d'un analyseur

Ecrivons en Scheme un analyseur pour cette grammaire. On associe à chaque non-terminal une fonction d'analyse dont l'objet est de trouver dans le mot donné le préfixe maximal pouvant être généré à partir de ce non-terminal. L'acceptation d'un terminal consiste à avancer dans le mot ou à signaler une erreur en cas d'incohérence.
La fonction d'analyse d'un non terminal suit la structure des règles pour ce non--terminal. On représente un mot par une liste de symboles, cette liste est terminée par **eof** pour indiquer la fin du mot. Si en fin d'analyse tous les symboles de la liste on été traités alors la fonction `analyse-mot?` renvoie `#t` et `#f` sinon.

```
(define (analyse-mot? Lsymboles)
  (let ((symboleSuivant '?))
    (letrec (
```

```
          (Lire-SymboleSuivant
            (lambda ()
              (if (null? Lsymboles)
                #f
                (begin (set! SymboleSuivant (car Lsymboles))
                       (set! Lsymboles (cdr Lsymboles))))))

          (analyse-S
            (lambda ()
              (analyse-X)
              (analyse-Y)))

          (analyse-X
            (lambda ()
              (case SymboleSuivant
                ((b) (Lire-SymboleSuivant) (analyse-Y))
                ((a) (Lire-SymboleSuivant))
                (else
                 (*erreur* "on attend le terminal a ou b : "
                           symboleSuivant)))))

          (analyse-Y
            (lambda ()
              (case SymboleSuivant
                ((a) (Lire-SymboleSuivant) (analyse-X))
                ((b) (Lire-SymboleSuivant))
                (else
                 (*erreur* "on attend le terminal a ou b : "
                           symboleSuivant)))))
          )

      (Lire-SymboleSuivant)
      (analyse-S)
      (eq? 'eof SymboleSuivant) )))

? (analyse-mot?  '(b a a b eof))   ->   #t
? (analyse-mot?  '(a a b b eof))   ->   #t
? (analyse-mot?  '(b b a b eof))   -> ERREUR --: on attend le terminal a ou b
```

13.7 Grammaires LL(1)

On va définir une sous-classe de grammaire permettant de faire une analyse récursive descendante sans retour arrière. Il s'agit de donner une méthode pour faire le bon choix de la règle à utiliser quand un non-terminal possède plusieurs règles.

Ensembles Premier et Suivant

Dans l'exemple précédent, on a choisi la règle à utiliser en se basant sur le premier terminal du mot à analyser. On a utilisé la règle (ici unique) qui l'avait en tête. Cela conduit à introduire l'ensemble des terminaux générés en tête par une règle

de la forme `X ::=` ξ, où ξ est dans $(N \cup T)^*$. On définit l'ensemble *Premier* par

`Premier`(ξ) = {$a \in$ `T` | $\xi \longrightarrow^* a\,\xi'$ }

Quand un non-terminal est membre gauche de plusieurs règles :
`X ::=` ξ_1 | ... | ξ_n,
le choix de la règle à utiliser sera déterminé de façon unique si l'on a la condition :

(*) les ensembles `Premier`(ξ_j) $j = 1 \dots n$ sont deux à deux disjoints.

Il est facile de calculer ces ensembles dans notre exemple. Le non-terminal `X` possède deux règles, calculons l'ensemble **`Premier`** de chacune :
- **`Premier (b X) = {b}`** car tout mot généré par `b X` commence évidemment par le terminal `b`,
- **`Premier (a) = {a}`**.
Ces ensembles sont bien disjoints.
Le non-terminal `Y` possède deux règles, calculons l'ensemble **`Premier`** de chacune :
- **`Premier (a X) = {a}`**,
- **`Premier (b) = {b}`**, ils sont disjoints.
Le critère est bien satisfait par cette grammaire. Mais la situation considérée est trop simpliste car il n'y a pas de production conduisant à ϵ. Ajoutons une telle production :

`X ::=` ϵ ; production X_3

Analysons le mot `aa` ; on doit analyser `aa` par `X Y`. On peut utiliser la règle X_2, il reste à analyser `a` par `Y` ce qui est possible avec les règles Y_1 et X_3. Mais on pouvait aussi commencer avec la règle X_3 : il resterait à analyser `aa` avec `Y`, ce qui est possible avec les règles Y_1 et X_2. On constate que l'ajout de la règle X_3 a introduit une ambiguïté dans le choix des règles.
On doit raffiner notre critère quand il y a des «epsilon productions». On étend la définition de **`Premier`** :

`Premier`(ξ) $= \{\, a \in \mathrm{T} \mid \xi \longrightarrow^* a\,\xi' \} \cup \{\, \epsilon \,\}$ si $\xi \longrightarrow^* \epsilon$

et l'on note `Premier*`(ξ) l'ensemble `Premier`(ξ) privé de l'élément ϵ.

Quand on peut dériver ϵ à partir d'un non-terminal `X`, il faut considérer aussi les terminaux qui peuvent suivre `X` dans une dérivation de `S`, c'est l'ensemble :

`Suivant(X) = {` $a \in$ `T | S` $\longrightarrow^*$ α `X` a β `}`

D'où la condition supplémentaire, pour avoir un choix déterministe de la règle à utiliser

(**) si `X` $\longrightarrow^*$ ϵ on demande que les ensembles `Premier(X)` et `Suivant(X)` soient disjoints.

Ensemble directeur

Pour unifier ces conditions, on introduit l'ensemble directeur d'une règle X $\rightarrow \xi$ par :

Directeur(ξ) = Premier(ξ) si ϵ n'en fait pas partie et
Premier*(ξ) $\cup$ Suivant (X) sinon.

En remarquant que si X est membre gauche de plusieurs règles de la forme :
X ::= ξ_1 | ... | ξ_n , on a :

Premier(X) = Premier(ξ_1) $\cup$ ... $\cup$ Premier(ξ_n).

On en déduit que les conditions (*) et (**) sont équivalentes à la condition :

LL(1) les ensembles Directeur(ξ_j) $j = 1 \ldots n$ sont deux à deux disjoints.

Le 1 de LL(1) signifie que le choix de la production à utiliser est déterminé par la connaissance du symbole suivant. De façon plus précise, si un non-terminal possède plusieurs règles, on prend celle dont l'ensemble directeur contient le symbole suivant.

Remarque 1 Pour appliquer ce critère quand on utilise la forme EBNF, il faut convenir qu'une règle comme X::= {ξ} désigne en fait deux productions distinctes : X ::= ξ {ξ} | ϵ.

Pour avoir une grammaire LL(1), on peut être amené à la remplacer par une grammaire équivalente. Par exemple, considérons les productions S ::= a X | a Y : le terminal a sera dans les deux ensembles premiers. Pour l'éviter, il suffit de factoriser, c'est-à-dire introduire un nouveau non-terminal Z et poser :

S ::= a Z et Z ::= X | Y

Pour calculer l'ensemble Premier, on utilise les règles suivantes :

P1 Si ξ est un terminal a, alors Premier(ξ) = {a}.

P2 si ξ est un non-terminal X qui admet les productions X ::= ξ_1 | .... | ξ_n, alors Premier(ξ) = Premier (ξ_1) $\cup$ ... $\cup$ Premier (ξ_n)

P3 si ξ est de la forme α_1 ... α_n, alors Premier(ξ) = Premier(α_1), mais si epsilon est dans Premier(ξ_1), alors Premier(ξ) = Premier*(ξ_1) auquel on ajoute Premier(ξ_2) à moins que epsilon soit aussi dans Premier(ξ_2), alors on ajoute Premier*(ξ_2) et ...

Si epsilon est dans tous les Premier(ξ_k), on l'ajoute aussi dans Premier(ξ).

P4 si ξ est une étoile η^*, alors Premier(ξ) = Premier(η) $\cup$ ϵ.

Le calcul de l'ensemble Suivant se fait par ajouts successifs en considérant toutes les productions où X apparaît à droite :

S1 mettre la marque de fin de fichier dans Suivant(S) où S est l'axiome.

S2 si l'on a une production `Z ::=` α `X` β , alors on ajoute `Premier*(`β`)` à `Suivant(X)`. De plus, si epsilon est dans `Premier(`β`)`, alors on ajoute `Suivant(Z)` à `Suivant(X)`.

S3 si l'on a une production `Z ::=` α `X`, alors on ajoute `Suivant(Z)` à `Suivant(X`

On renvoie à la bibliographie pour les démonstrations.

Exercice 8 *En utilisant la grammaire ETF pour les expressions arithmétiques :*

```
<E> ::= <T> <E'>
<E'> ::= '+' <E> | '-' <E> | ϵ
<T> ::= <F> <T'>
<T'> ::= '*' <T> | '/' <T> | ϵ
<F> ::= '(' <E> ')' | '-' <F> | 'nombre' | 'identificateur '
```

vérifier le calcul des ensembles suivants :

```
Premier(E) = Premier(T) = Premier(F) = { ( , - , ident , nombre}
Premier(E') = {+ , - , ϵ }
Premier(T') = {* , / , ϵ }

Suivant(E') = Suivant(E) = { ) , eof}
Suivant(T') = Suivant(T) = {+ , - , ) , eof}
Suivant(F) = {+ , - , * , / , ) , eof }
```

En déduire que les ensembles directeurs des règles pour E', T', F sont :

pour E' : {+} , {-} , {) , eof}
pour T' : {} , {/} , {+ , -,) , eof}*
pour F : {(} ,{-} , {identificateur} , {nombre}.

13.8 Un analyseur d'expressions arithmétiques

Appliquons ceci à l'écriture d'un analyseur syntaxique d'expressions arithmétiques. L'expression à analyser est dans un fichier auquel on associe un flux qui est lu par un analyseur lexical. Chaque appel à l'analyseur lexical retourne une paire constituée de la classe et de la valeur du lexème. L'analyseur utilise la classe du lexème pour sélectionner la règle à utiliser. A chaque non-terminal on associe une fonction chargée d'analyser le texte pouvant être engendré par ce non-terminal. Si le non-terminal `X` a des règles de la forme `X ::=` ξ_1 `| ... |` ξ_n, où ξ_1 est de la forme $\alpha_1 \ldots \alpha_h$, ξ_2 de la forme $\beta_1 \ldots \beta_k$, ... alors la structure de sa fonction d'analyse, notée `analyse-X`, sera :

```
(lambda ()
  (case classeSuivante
    ((ensDirecteur de ksi-1)  (analyse-alpha-1) ...(analyse-alpha-h))

    ((ensDirecteur de ksi-2)  (analyse-beta-1) ... (analyse-beta-k))
     ...
    (else
     (*erreur*
      ''on attend un terminal dans la reunion des ens directeurs''))))
```

L'analyse d'un non-terminal consiste simplement à lire ce terminal pour passer au lexème suivant. Si l'appel de la fonction d'analyse de l'axiome a permis de lire tout le fichier, alors ce fichier est syntaxiquement correct et la fonction **analyseETF** retourne **#t** sinon on a un message d'erreur.

```
(define (analyseETF flux)
  (let ((lex (make-lex flux  '()
                    '((#\- . moins)(#\+ . plus)(#\* . mul)
                                 (#\\ . div)(#\( . lpar)( #\) . rpar)))

        (lexemeSuivant '?)
        (classeLexeme  '?))

    (letrec (
            (Lire-LexemeSuivant
              (lambda ()
                (begin (set! lexemeSuivant (lex))
                       (set! classeLexeme (car lexemeSuivant)))))

            (analyse-E
              (lambda ()
                (analyse-T)
                (analyse-E1)))

            (analyse-E1
              (lambda ()
                (case classeLexeme
                  ((plus moins) (Lire-LexemeSuivant) (analyse-E))
                  ((Rpar eof) '())
                  (else (*erreur* "on attend + , - , ) : "
                                  classeLexeme)))))

            (analyse-T
              (lambda ()
                (analyse-F)
                (analyse-T1)))

            (analyse-T1
              (lambda ()
                (case classeLexeme
                  ((mul div) (Lire-LexemeSuivant) (analyse-T))
                  ((plus moins  Rpar eof) '())
                  (else (*erreur* "on attend + , - , * , / , eof , ) : "
                                  classeLexeme)))))

            (analyse-F
              (lambda ()
                (case classeLexeme
                  ((Lpar)
                   (Lire-LexemeSuivant) (analyse-E) (Lire-LexemeSuivant))
                  ((identificateur entier) (Lire-LexemeSuivant))
```

```
                (else (*erreur* "on attend  ident , entier , ( , eof : "
                                        classeLexeme)))))
          )
     (Lire-lexemeSuivant)
     (analyse-E)
     (if (eq? 'eof  classeLexeme)
       #t
       (*erreur* ''fichier non enti\'erement analys\'e '' )))))
```

Exercice 9 *La structure de l'analyseur d'une grammaire* `LL(1)` *suit la structure des règles de production, aussi n'est-il pas très difficile de générer automatiquement le code de l'analyseur à partir de la grammaire. Définir une représentation d'une grammaire et écrire un générateur d'analyseur (on suppose les grammaires de classe* `LL(1)`*).*

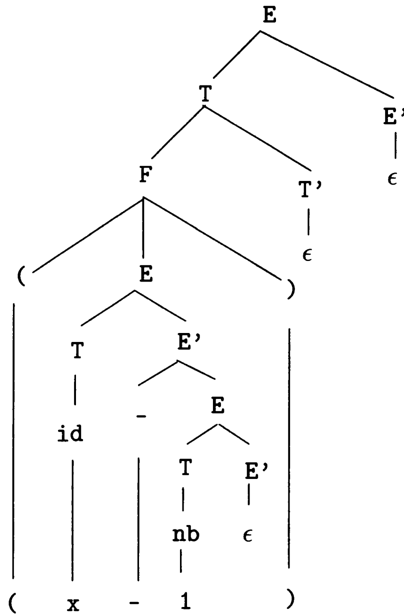
Exercice 10 *Ecrire un analyseur en utilisant la grammaire EBNF des expressions arithmétiques. Pour calculer l'ensemble directeur d'une expression $\{\xi\}$, on la considérera comme une abréviation des règles* `Z ::=` ϵ `|` ξ `Z`.

13.9 Syntaxe abstraite et grammaires attribuées

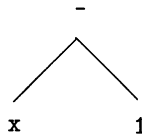
Dans la pratique, on ne se contente pas de la seule reconnaissance des mots syntaxiquement corrects. On profite en général de l'analyse syntaxique d'un mot d'un langage pour lui associer aussi une valeur.

Notion de syntaxe abstraite

Dans le cas d'un programme cette valeur peut être, par exemple, une représentation du programme sous forme d'un arbre dit de *syntaxe abstraite*. Prenons l'exemple de l'expression arithmétique `(x - 1)` : son arbre d'analyse avec la grammaire ETF précédente est :



En fait, ce n'est pas l'arbre d'analyse syntaxique qui est intéressant mais l'arbre qui donne la structure de l'expression. L'arbre d'analyse est anecdotique, car il dépend de la grammaire utilisée, alors que l'arbre de l'expression décrit sa structure intrinsèque, on l'appelle l'*arbre de syntaxe abstraite* :

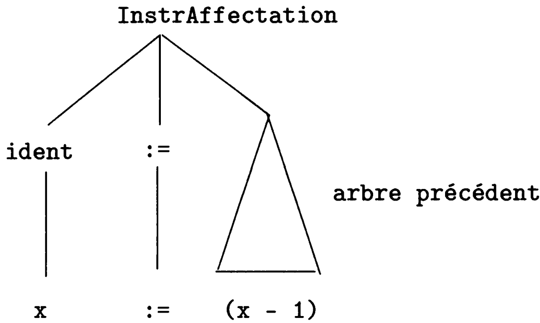


Dans cet arbre les feuilles sont des identificateurs ou des constantes. Les nœuds internes sont étiquetés par des opérateurs ; l'arbre représente la forme préfixe de l'expression. La connaissance de l'arbre de syntaxe abstraite facilite les manipulations sur l'expression, par exemple le calcul de sa valeur.

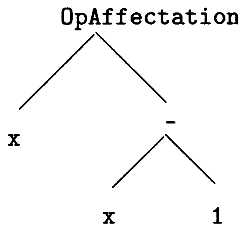
La syntaxe abstraite est une représentation arborescente d'une expression ou d'une instruction qui ne conserve que la structure et oublie ce qu'on appelle le «sucre syntaxique», alors que l'arbre d'analyse syntaxique est lié à la représentation textuelle. Précisons ce point avec l'exemple de l'instruction d'affectation dans un langage comme Pascal. L'affectation est définie par une règle de grammaire de la forme :

```
<InstrAffectation> ::= 'ident' ':=' <E>
```

Ainsi l'affectation `x := (x - 1)` aura pour arbre d'analyse :



et pour arbre de syntaxe abstraite :



qui contient la même information mais sans le sucre syntaxique comme les parenthèses. La formalisation de la notion de syntaxe abstraite sera donnée au chapitre 17.

Grammaires attribuées

Que ce soit le calcul d'un arbre de syntaxe abstraite ou le calcul de la valeur d'une expression arithmétique, on va montrer plus généralement comment on peut associer une valeur à un mot d'un langage défini par une grammaire. Cette valeur sera la valeur associée à l'axiome de la grammaire et sera calculée pendant l'analyse syntaxique. Pour le faire, on associera une valeur à chaque élément de la grammaire, cette valeur dépendra de la règle utilisée pour l'analyser et d'une fonction associée à chaque règle.

Définition 3 On appelle grammaire attribuée[6] la donnée pour chaque règle $X_i ::= \alpha_1 \ldots \quad \alpha_k$, d'une fonction f (on dit aussi un *attribut* ou une fonction sémantique) de k arguments.

La valeur associée à X_i, via une règle, est par définition $f(\$1, \ldots, \$k)$ où $j est la valeur associée à l'élément α_j. On se donne directement la valeur des terminaux (elle est souvent calculée par l'analyseur lexical).

[6] Il s'agit ici du cas particulier des attributs purement synthétisés, on définira la notion générale au §3 du chapitre suivant.

La valeur du mot analysé sera par définition celle de l'axiome. Pour faire ce calcul, il suffit de modifier légèrement la structure de chaque fonction d'analyse, plus précisément, dans la fonction d'analyse d'un non-terminal :

```
(lambda ()
  (case classeSuivante
    ((ensDirecteur de alpha)  (analyse-alpha-1) ...(analyse-alpha-h))
    ((ensDirecteur de beta)   (analyse-beta-1) ... (analyse-beta-k))
    ...
    (else
     (*erreur* on attend un terminal dans la reunion des ens directeurs))))
```

On remplace chaque séquence du type `(analyse-alpha-1)...(analyse-alpha-h` par un `let*`[7] de la forme :

```
(let* (($1 (analyse-alpha-1))
       ...
       ($h (analyse-alpha-h)))
  (f $1 ... $h))
```

On va appliquer ceci aux expressions arithmétiques.

Forme préfixe d'une expression arithmétique

La forme préfixe d'une expression arithmétique peut lui servir de syntaxe abstraite. La forme préfixe est définie par la grammaire suivante d'axiome `P-exp` :

```
<P-exp> ::=   '(' <op2>  <p-exp>  <p-exp> ')'
            | '('  '-'  <p-exp> ')'
            | 'identficateur'
            | 'nombre'

<op2>    ::=  '+' | '-' | '*' | '/'
```

Par exemple, la forme préfixe de l'expression `prixHT * (1 + TauxTVA / 100)` est `(* prixHT (+ 1 (/ TauxTVA 100)))`.

On utilise la grammaire suivante pour les expressions arithmétiques :

```
<E>   ::=    <T>  <E'>
<E'>  ::=    '+'  <E>  |  '-'   <E> | ϵ
<T>   ::=    <F>  <T'>
<T'>  ::=   '*'   <T>  |  '/' <T>  |  ϵ
<F>   ::=  '(' <E> ')' |  '-' <F>  |  'nombre' | 'identificateur'
```

Pour passer de la forme infixe à la forme préfixe, on associe à chaque règle une fonction sémantique de sorte que la valeur de l'axiome E soit la forme préfixée.

[7] Ce `let*` séquentiel est nécessité par la consommation du flux d'entrée.

- Pour les productions :

  <E'> ::= '+' <E> | ' -' <E>

  on prend la liste des valeurs, c'est-à-dire (f $1 $2) = (list $1 $2)

- Pour la production <E'> ::= ϵ, on prend (f $1) = ().

- Cherchons maintenant la fonction (f $1 $2) à associer avec la règle

  <E> ::= <T> <E'>.

  - Si E' n'utilise pas la production vide, sa valeur $2 sera de la forme ($\pm$ Exp-prefixee-E'). L'expression préfixée de E sera donc : ($\pm$ exp-prefixe-T Exp-prefixee-E'). Il faut donc utiliser l'expression (list (car $2) $1 (cadr $2)) pour (f $1 $2).
  - Si E' utilise la production vide, sa valeur est () et la valeur de E sera celle de T.

  Finalement la fonction f à associer à l'axiome E est donnée par :

```
(lambda ($1 $2)
  (if (null? $2)
    $1
    (list (car $2) $1 (cadr $2))))
```

- On procède de façon analogue pour les règles :

  <T> ::= <F> <T'>

  <T'> ::= '*' <T> | '/' <T> | ϵ

- Les fonctions sémantiques pour les productions de F sont plus immédiates. Pour les terminaux, on utilise essentiellement les valeurs données par l'analyse lexicale. Pour un nombre c'est sa valeur, pour un identificateur ou un caractère spécial, on utilise le symbole associé à sa chaîne.

Voici finalement la grammaire ETF avec, pour chaque règle, les fonctions à utiliser pour calculer la forme préfixe de l'expression arithmétique analysée :

```
<E>  ::= <T> <E'>          ; (lambda ($1 $2)
                               (if (null? $2) $1 (list (car $2) $1 (cadr $2)))

<E'> ::= '+' <E> | ' -'  <E>  ;  (lambda ($1 $2) (list $1 $2))

<E'> ::=  ε              ;  (lambda () '())

<T>  ::= <F>  <T'>         ; (lambda ($1 $2)
                               (if (null? $2) $1  (list (car $2) $1 (cadr $2))

<T'> ::= '*'  <T>  | '/' <T>  ;  (lambda ($1 $2) (list $1 $2))

<T'> ::= ε   ;  (lambda () '())

<F>  ::=   '('   <E>    ')'   ;  (lambda ($1 $2 $3)  $2)
```

```
<F>  ::=   '-'   <F>           ;  (lambda ($1 $2) (list $1 $2))

<F>  ::=  'nombre'              ;   (lambda ($1 ) $1)

<F>  ::=  'identificateur'      ;   (lambda ($1 ) (string->symbol $1))
```

Exercice 11 *Donner les fonctions sémantiques pour que l'on puisse retourner la valeur numérique des expressions (on suppose qu'il n'y a pas de variable dans l'expression).*

On en déduit immédiatement un traducteur `infixe->prefixe`:

```
(define (infixe->prefixe flux)
  (let ((lex (make-lex flux  '()
                       '((#\- . moins)(#\+ . plus)(#\* . mul)
                                      (#\\ . div)(#\( . lpar)(#\) . rpar))))
        (lexemeSuivant '?)
        (classeLexeme  '?)
        (valeurLexeme  '?))

    (letrec (
            (Lire-LexemeSuivant
              (lambda ()
                (begin (set! lexemeSuivant (lex))
                       (set! classeLexeme (car lexemeSuivant))
                       (set! valeurLexeme (cdr lexemeSuivant)))))

            (analyse-terminal
              (lambda ()
                (let ((valeur valeurLexeme))
                  (Lire-LexemeSuivant)
                  (if (number? valeur) valeur (string->symbol valeur)))))

            (analyse-E
              (lambda ()
                (let* (($1 (analyse-T))
                       ($2 (analyse-E1)))
                  (if (null? $2)
                    $1
                    (list (car $2) $1 (cadr $2))))))

            (analyse-E1
              (lambda ()
                (case classeLexeme
                  ((plus moins)(let* (($1 (analyse-terminal))
                                      ($2 (analyse-E)))
                                 (list $1 $2)))
                  ((Rpar eof) '())
                  (else (*erreur* "on attend  + , - , ) ,  eof : "
                                  classeLexeme)))))
```

```
        (analyse-T
          (lambda ()
            (let* (($1 (analyse-F))
                   ($2 (analyse-T1)))
              (if (null? $2)
                $1
                (list (car $2) $1 (cadr $2))))))

        (analyse-T1
          (lambda ()
            (case classeLexeme
              ((mul div) (let* (($1 (analyse-terminal))
                                ($2  (analyse-T)))
                           (list $1 $2)))
              ((plus moins  Rpar eof) '())
              (else (*erreur* "on attend + , - , * , / , eof , ) : "
                              classeLexeme)))))

        (analyse-F
          (lambda ()
            (case classeLexeme
              ((Lpar) (Lire-LexemeSuivant)
                      (begin1 (analyse-E) (Lire-LexemeSuivant)))
              ((identificateur entier) (analyse-terminal))
              (else (*erreur* "on attend  ident , nb , ( , eof : "
                              classeLexeme)))))
        )
    (Lire-lexemeSuivant)
    (let ((valeur (analyse-E)))
      (if (eq? 'eof  ClasseLexeme)
        valeur
        (*erreur* ``fichier non enti\`erement analys\'e `` ))))))
```

Si le flux est associé à un fichier contenant l'expression
`prixHT *(1 + TauxTVA / 100)` alors cette fonction retourne
`(* prixHT (+ 1 (/ TauxTVA 100)))`

Comme pour les analyseurs syntaxiques, il n'est pas très difficile d'en déduire un générateur automatique d'analyseur quand on se donne une grammaire `LL(1)` et des fonctions sémantiques. Il en existe même pour des classes plus générales de grammaires. On verra au chapitre suivant d'autres applications de l'idée de «décoration» d'un arbre par des attributs.

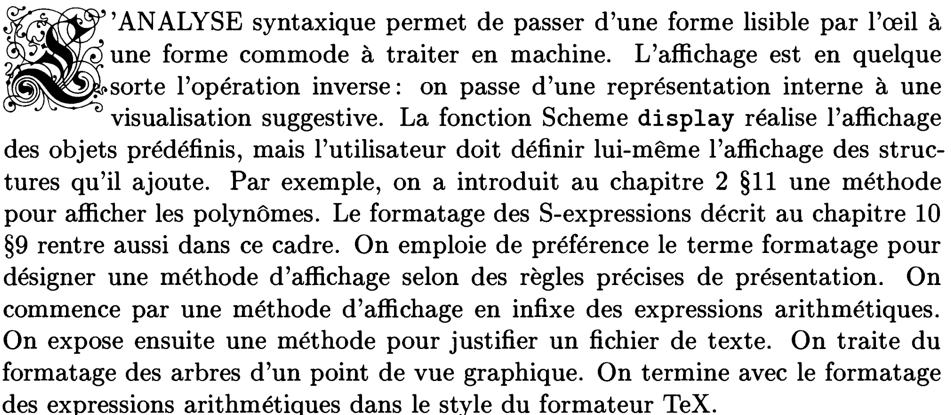
13.10 De la lecture

Pour les méthodes d'analyse lexicale et syntaxique, on renvoie aux classiques [FL88, ASU90].
Pour l'approche fonctionnelle de la reconnaissance des expressions régulières, on peut consulter [Bur75, WL93, CM95].

On n'a pas abordé l'aspect analyse syntaxique des langues naturelles, mais il est présenté, avec le code en Lisp, dans [Nor92].

Chapitre 14

Affichage et formatage



L'ANALYSE syntaxique permet de passer d'une forme lisible par l'œil à une forme commode à traiter en machine. L'affichage est en quelque sorte l'opération inverse : on passe d'une représentation interne à une visualisation suggestive. La fonction Scheme `display` réalise l'affichage des objets prédéfinis, mais l'utilisateur doit définir lui-même l'affichage des structures qu'il ajoute. Par exemple, on a introduit au chapitre 2 §11 une méthode pour afficher les polynômes. Le formatage des S-expressions décrit au chapitre 10 §9 rentre aussi dans ce cadre. On emploie de préférence le terme formatage pour désigner une méthode d'affichage selon des règles précises de présentation. On commence par une méthode d'affichage en infixe des expressions arithmétiques. On expose ensuite une méthode pour justifier un fichier de texte. On traite du formatage des arbres d'un point de vue graphique. On termine avec le formatage des expressions arithmétiques dans le style du formateur TeX.

14.1 Mise sous forme infixe des expressions arithmétiques

En mathématique, les expressions arithmétiques sont en général écrites sous la forme infixe, aussi peut-on avoir besoin de passer de la forme préfixe Scheme à la forme infixe. Cette transformation s'appelle parfois la décompilation car on y fait le travail inverse de l'analyse syntaxique. Heureusement la transformation est plus simple dans ce sens. Par exemple, il s'agit de transformer

`(- (* (* a b) (+ (- c) d)))` en `- (a * b * ( - c + d))`

Forme infixe complètement parenthésée

Pour accéder à l'opérateur et aux opérandes d'une expression préfixe on définit les fonctions suivantes (où l'on a convenu que l'opérateur moins unaire n'avait pas de premier opérande mais seulement un deuxième opérande) :

```
(define op   car)

(define (opd1 exp)
  (if (null? (cddr exp)) ;; cas du  - unaire
    '()
    (cadr exp)))

(define (opd2 exp)
  (if (null? (cddr exp))
    (cadr exp)
    (caddr exp)))
```

La fonction `prefixe->infixe-str` associe à une expression préfixe la chaîne représentant sa forme infixe.
Il suffit de faire un raisonnement par cas. Pour la clarté de l'expression infixe, on laisse un blanc de part et d'autre de chaque opérateur.

```
(define (prefixe->infixe-str exp)
  (cond ((number? exp) (number->string  exp))
        ((null? exp) "")
        ((symbol? exp) (symbol->string  exp))
        (else (string-append "(" (prefixe->infixe-str (opd1 exp))
                             " " (symbol->string (op exp))  " "
                             (prefixe->infixe-str (opd2 exp)) ")"))))

? (prefixe->infixe-str '(- (* (* a b) (+ (- c) d))))
"( - ((a * b) * (( - c) + d)))"
```

C'est juste, mais pas satisfaisant : les relations de priorité entre les opérateurs permettent de supprimer des parenthèses. Cette expression s'écrit aussi :
`- (a * b * ( - c + d))`.

Forme infixe avec priorités

On réécrit la fonction `prefixe->infixe-str` en déléguant à la fonction auxiliaire `operand->string` le calcul de la chaîne de chaque opérande. Cette fonction ne met pas systématiquement des parenthèses autour des opérandes. Si un opérande a même opérateur ou est prioritaire par rapport à celui de l'expression englobante, on ne le met pas entre des parenthèses.

```
(define (prefixe->infixe-str exp)
  (cond ((number? exp) (number->string  exp))
        ((null? exp) "")
        ((symbol? exp) (symbol->string  exp))
        (else (string-append (operand->string (opd1 exp) exp)
                             " " (symbol->string (op exp))  " "
                             (operand->string (opd2 exp) exp)))))

(define (operand->string opd exp)
  (if  (or (< (priorite exp)(priorite opd))
           (and (pair? opd)(eq? (op opd)(op exp))))
```

```
    (prefixe->infixe-str opd)
    (string-append "(" (prefixe->infixe-str opd) ")")))
```

On représente les priorités des expressions par des entiers :

```
(define (priorite exp)
  (if (pair? exp)
    (cond ((null? (cddr exp)) 3)
          ((memq (op exp) '(* /)) 2)
          (else 1))
    4))
```

Testons avec la même expression :

```
?(prefixe->infixe-str '(- (* (* a b) (+ (- c) d))))
" - (a * b * ( - c + d))"
```

14.2 Formatage de texte : centrage, justification...

Dans ce paragraphe on s'intéresse au cas d'un fichier de texte. On a vu au chapitre 10 §3 comment on peut afficher ligne par ligne un fichier de texte ; on étudie maintenant son formatage selon des styles de mise en page.

Les styles de formatage de texte

Le formatage d'un texte concerne la façon de présenter ce texte : le choix des caractéristiques de la police de caractères (famille, taille, grandeur, ...), de la marge, de la longueur de la ligne, du nombre de colonnes, du mode, ... C'est un sujet très complexe car il y a un nombre considérable de paramètres qui influent sur l'esthétique d'un texte. On distingue deux grandes familles de logiciels pour traiter cet aspect : les logiciels WYSIWYG[1] et les logiciels de traitement par balisage.
La première catégorie correspond à un mode d'utilisation interactif ; l'utilisateur indique en détail les actions à faire et elles sont immédiatement prises en compte au niveau de l'affichage.
La deuxième catégorie correspond à un mode d'utilisation où les instructions de formatage sont incluses dans le fichier à formater. Le logiciel traite ce fichier et génère un fichier formaté. L'exemple le plus connu est certainement le système TEX avec lequel est réalisé ce livre.
On se place dans le deuxième cas, le premier était, en grande partie, une question d'interface homme machine. On va considérer une forme très fruste, mais représentative du problème. On pourra agir sur la marge, la longueur des lignes et le mode de justification du texte :
- la marge gauche indique à combien de caractères du bord gauche on peut commencer à écrire,
- la longueur de la ligne est la longueur utilisable pour écrire,

[1] *What-You-See-Is-What-You-Get* : ce que vous voyez est ce que vous obtiendrez.

- il est prévu quatre modes de justification, on formate la description de chaque mode avec le mode concerné :

Justification à gauche

Chaque ligne de texte du fichier source est transformée en une
ligne du fichier destination mais elle commence contre la marge gauche
et les espaces entre les mots sont réduits à un blanc.
Si la ligne du fichier source est plus longue que celle du fichier destination, les mots en surplus sont écrits sur une nouvelle ligne.

Justification à droite

Chaque ligne de texte du fichier source est transformée en une
ligne du fichier destination mais elle se termine à l'extrémité
droite et les espaces entre les mots sont réduits à un blanc.
Si la ligne du fichier source est plus longue que celle du fichier destination alors
les mots en surplus sont écrits sur une nouvelle ligne.

Justification centrée

Chaque ligne de texte du fichier source est transformée en une
ligne du fichier destination mais elle
s'affiche centrée et les espaces entre les mots sont réduits à un blanc

Complètement justifié

On oublie la structure en ligne du fichier source, le texte s'affichera sur toute la longueur de la ligne avec un nombre variable (mais petit) d'espaces entre les mots pour les aligner à gauche et à droite. On ne traite pas le problème de la césure des mots. Si l'on a besoin de trop espacer les mots pour réaliser la justification complète, on se contente de la justification à gauche. Pour sauter des lignes dans ce mode, il faut en donner l'ordre dans le texte source.

Les paramètres

Pour programmer ce formateur on utilise les notions de *tampon* (*buffer* en anglais) et d'environnement de formatage. Le tampon est utilisé pour stocker les mots qui serviront à fabriquer une ligne du fichier de sortie. L'environnement représente l'ensemble des paramètres considérés : le mode, la longueur de la marge gauche, la longueur de la ligne. Pour changer ces divers paramètres on place dans le fichier d'entrée des commandes de formatage. Ces commandes seront toujours situées sur une ligne sans texte et une ligne de commandes commencera par un caractère spécial. Elles prennent effet à partir de la ligne suivante. Il y a quatre commandes possibles :

- `(mode nom-mode)` pour fixer le mode, les noms des modes possibles sont : `letf`, `right`, `center`, `fill`,
- `(margeG n)` fixe la marge gauche à la valeur `n`,
- `(LongLigne n)` fixe la longueur de la ligne à la valeur `n`,
- `(sauter n)` pour sauter `n` lignes (seule façon de sauter des lignes en mode `fill`).

Un exemple de formatage

Voici un exemple de formatage :

Fichier source

```
#(mode center) (LongLigne 56) (margeG 10)
Université de Nice Sophia Antipolis
#(sauter 2)(mode right)
Nice, le 20 novembre 1995
#(sauter 1)(mode left) (margeG 15)
Cher ami,
#(mode fill)(margeG 10)(sauter 1)
Voici un exemplaire du livre que je t'avais promis il y a quelques
semaines. Dans la perspective
d'une future traduction en anglais, je serais heureux
de recevoir tes remarques sur le fond comme sur la forme.
#(sauter 1)
Bonne lecture et bien amicalement.
#(mode left) (margeG 15)(sauter 2)
Jacques Chazarain
#(sauter 1)
Département d'Informatique
06108 NICE cedex 2
```

Fichier formaté

```
                     Université de Nice Sophia Antipolis


                                          Nice , le 20 novembre 1995

               Cher ami ,

          Voici  un exemplaire du livre que je t'avais promis il y
          a  quelques  semaines . Dans la perspective d'une future
          traduction  en  anglais  , je serais heureux de recevoir
          tes remarques sur le fond comme sur la forme .

          Bonne lecture et bien amicalement .


               Jacques Chazarain

               Département d'Informatique
               06108 NICE cedex 2
```

Modélisation avec des prototypes

La méthode consiste à lire les mots du fichier d'entrée et à les insérer dans le tampon. Quand le tampon comporte assez de mots pour formater une ligne, il est vidé et la ligne est écrite dans le fichier de sortie. Comme on n'a pas besoin de la notion d'héritage, on va représenter en Scheme le tampon et l'environnement par des prototypes.

L'*environnement* sera un prototype à trois champs : le mode, la marge gauche, la longueur de la ligne. On définit les accesseurs et modificateurs de ces champs :

```
(define (creer-env  mode   margeG   LongLigne)
  (lambda (message . arg)
    (case message
      ((mode) mode)
      ((margeG) margeG)
      ((longLigne) longLigne)
      ((set-mode!) (set! mode (car arg)))
      ((set-margeG!) (set! margeG (car arg)))
      ((set-longLigne!) (set! longLigne (car arg))))))
```

Le *tampon* sera aussi un prototype à trois champs : la liste des mots qu'il contient, le nombre de ces mots, la place restante dans la ligne. Quand le tampon est vide, la place disponible est la longueur de la ligne, quand on ajoute un mot, elle est diminuée de la longueur du mot + 1 (le +1 est dû à l'espace blanc minimum entre les mots). Pour simplifier, tous les signes de ponctuation sont aussi séparés par un espace de part et d'autre. On définit les accesseurs et modificateurs pour ces champs. On définit une méthode pour réinitialiser le tampon et une pour ajouter un mot :

```
(define (creer-tampon  Lmots  nbMots  placeRestante)
  (lambda (message . arg)
    (case message
      ((Lmots) Lmots)
      ((nbMots) nbMots)
      ((placeRestante) placeRestante)
      ((set-Lmots!)  (set! Lmots (car arg)))
      ((set-nbMots!) (set! nbMots (car arg)))
      ((set-placeRestante!) (set! placeRestante (car arg)))
      ((reset-tampon)
       (set! Lmots '())
       (set! nbMots 0)
       (set!  placeRestante (car arg)))
      ((ajouter-mot!)
       (set! nbMots (+ 1 nbMots))
       (set! Lmots (cons (car arg) Lmots))
       (set! placeRestante (- placeRestante
                              (string-length (car arg))
                              1))))))
```

Fonction principale

La fonction principale de formatage transforme le fichier d'entrée contenant le texte et les commandes en un fichier de texte formaté. Elle prend aussi en paramètre les valeurs de l'environnement initial. C'est une boucle, notée loop, qui saute les espaces et, selon les cas, exécute une ligne de commande ou formate une ligne.

```
(define (formatage  input-file  output-file  mode  margeG  longLigne)
  (call-with-input-file input-file
    (lambda (in-stream)
      (call-with-output-file output-file
        (lambda (out-stream)
          (let ((env    (creer-env mode  margeG  LongLigne))
                (tampon (creer-tampon '()  0  LongLigne)))
            (letrec ((loop
                      (lambda ()
                        (let ((char (sauter-espaces in-stream)))
                          (cond
                            ((eof-object? char) (display "termine"))
                            ((commande? char) (read-char in-stream)
                             (executer-commandes tampon env in-stream
                                                 out-stream)
                             (loop))
                            (else
                             (formater-ligne tampon env in-stream
                                             out-stream)
                             (loop)))))))
              (loop)))))))
```

On a choisi de faire commencer une ligne de commandes par le caractère # ; il est donc interdit dans le texte.

```
(define (commande? char)
  (char=? char #\#))
```

La fonction suivante saute les espaces dans le flux d'entrée et rend le premier caractère non espace :

```
(define  (sauter-espaces in-stream)
  (do ((char (peek-char in-stream) (peek-char in-stream)))
      ((or (eof-object? char) (not (char=? #\space char))) char)
    (read-char in-stream)))
```

La lecture du texte source se fait lexème par lexème. Un lexème est en général un mot, mais cela peut être aussi un caractère de ponctuation. La fonction lire-lexeme retourne un lexème (sous forme d'une chaîne) ou l'objet fin du fichier, ou la marque de fin de la ligne eoln ou enfin annonce une ligne de commandes.

```
(define (lire-lexeme in-stream)
  (letrec ((lire-mot
            (lambda (mot)
```

```
              (let ((char (peek-char  in-stream))
                (if (separateur? char)
                  mot
                  (lire-mot (string-append mot
                                 (string (read-char in-stream))))))))

    (sauter-espaces in-stream)
    (let ((char (read-char in-stream))
      (if (eof-object? char)
        char
        (case char
          ((#\newline)    'eoln)
          ((#\#)      'commande)
          ((#\, #\; #\:  #\.  #\!  #\? ) (string char))
          (else   (lire-mot (string char)))))))
```

Les caractères séparateurs sont : l'espace, le passage à la ligne, la marque de fin de fichier, les caractères de ponctuation et le caractère de début de ligne de commande.

```
(define (separateur? char)
  (or (memq char '(#\space #\newline #\, #\; #\: #\. #\! #\? #\#))
      (eof-object? char)))
```

Quand on rencontre une ligne de commandes, on commence par vider le tampon s'il ne l'est pas car une commande ne peut affecter que la ligne qui suit. On lit et exécute les commandes une à une ; chaque commande étant une liste on utilise `read` pour les lire. Quand on change la longueur de la ligne, il faut aussi modifier le champ `place-restante` du tampon.

```
(define (executer-commandes tampon env in-stream out-stream)
  (unless (null? (tampon 'Lmots tampon))
          (vider-tampon tampon env out-stream))
  (letrec ((loop
            (lambda (commande)
              (let ((nom-commande (car commande))
                    (arg-commande (cadr commande))
                    (car-suivant (sauter-espaces  in-stream)))
                (case nom-commande
                  ((sauter)    (sauter-lines arg-commande out-stream))
                  ((mode)      (env 'set-mode!      arg-commande))
                  ((margeG)    (env 'set-margeG!    arg-commande))
                  ((LongLigne) (env 'set-LongLigne! arg-commande)
                               (tampon 'set-placeRestante!
                                       arg-commande)))
                (if (or (eof-object? car-suivant)(eq? #\newline car-suivant
                  (read-char in-stream)
                  (loop (read in-stream)))))))

    (loop (read in-stream))))
```

Pour sauter `k` lignes dans un fichier, on utilise la fonction ;

```
(define (sauter-lines k stream)
  (do ((i 0 (+ 1 i)))
      ((= i k))
    (newline stream)))
```

Formatage d'une ligne

Le point essentiel est le formatage d'une ligne. En mode `fill`, on remplit le tampon tant qu'il reste de la place. Quand un mot ne peut plus rentrer, on vide le tampon dans le fichier puis on initialise le prochain tampon avec le mot résiduel. Dans les autres modes, on copie les mots d'une ligne dans le tampon puis on vide le tampon, selon le mode choisi, dans le fichier de sortie. Si la ligne déborde, on place le surplus dans le prochain tampon.

```
(define (formater-ligne tampon env in-stream out-stream)
  (letrec ((loop
            (lambda (lexeme)
              (cond
                ((eof-object? lexeme)
                 (vider-tampon tampon env out-stream))
                ((eq? 'eoln lexeme)
                 (if (eq? 'fill (env 'mode))
                   (loop (lire-lexeme in-stream))
                   (vider-tampon tampon env out-stream)))
                ((eq? 'commande lexeme)
                 (executer-commandes tampon env in-stream out-stream))
                ((<= (string-length lexeme)(tampon 'placeRestante))
                 (tampon 'ajouter-mot! lexeme)
                 (loop (lire-lexeme in-stream)))
                (else ;; si le tampon deborde, on le vide puis on verse
                 (vider-tampon tampon env out-stream);;le surplus dans
                 (tampon 'ajouter-mot! lexeme)))))  ;;le tampon suivant

    (loop  (lire-lexeme in-stream))))
```

Pour vider le tampon, on fabrique (à partir de la liste renversée des mots du tampon) une chaîne selon le mode courant et on l'écrit dans le fichier destination, puis on initialise le prochain tampon. En mode `fill` la place restante est augmentée de 1 car on n'ajoute pas d'espace après le dernier mot.

```
(define  (vider-tampon tampon env out-stream)
  (let ((margeG         (env 'margeG))
        (long-ligne     (env 'LongLigne))
        (mode           (env 'mode))
        (NbMots         (tampon 'NbMots))
        (Lmots          (reverse (tampon 'Lmots)))
        (place-restante (tampon 'placeRestante)))
    (case mode
      ((left)
       (ecrire-ligne (creer-ligne-left  margeG Lmots) out-stream))
      ((right)
```

```
       (ecrire-ligne (creer-ligne-right margeG Lmots place-restante)
                     out-stream))
      ((center)
       (ecrire-ligne (creer-ligne-center margeG Lmots place-restante)
                     out-stream))
      ((fill)
       (ecrire-ligne (creer-ligne-fill margeG Lmots (+ 1 place-restante)
                                       NbMots)
                     out-stream)))
    (tampon 'reset-Tampon long-ligne)))
```

Mode left

En mode left, la chaîne commence par un espace blanc égal à la marge et chaque lexème est séparé du suivant par un blanc :

```
(define (creer-ligne-left margeG  Lmots)
  (letrec ((creer-ligne
            (lambda (ligne Lmots)
              (if (null? Lmots)
                ligne
                (creer-ligne (string-append ligne (car Lmots) " ")
                             (cdr Lmots))))))
    (creer-ligne (creer-espace  margeG) Lmots)))
```

Mode right

En mode right, la chaîne commence par un espace blanc égal à la somme de la marge et de la place non utilisée dans la ligne :

```
(define (creer-ligne-right margeG  Lmots place-restante)
  (letrec ((creer-ligne
            (lambda (ligne Lmots)
              (if (null? Lmots)
                ligne
                (creer-ligne (string-append ligne  " " (car Lmots))
                             (cdr Lmots))))))
    (creer-ligne (creer-espace (+ margeG place-restante)) Lmots)))
```

Mode centré

En mode centré, la chaîne commence par un espace blanc égal à la somme de la marge et de la moitié de la place non utilisée dans la ligne :

```
(define (creer-ligne-center margeG  Lmots  place-restante)
  (letrec ((creer-ligne
            (lambda (ligne Lmots)
              (if (null? Lmots)
                ligne
                (creer-ligne (string-append ligne  " " (car Lmots))
                             (cdr Lmots))))))
```

```
    (let* ((placeRestante-gauche (quotient place-restante 2))
           (nb-espace-gauche (+ margeG placeRestante-gauche)))
      (creer-ligne (creer-espace nb-espace-gauche) Lmots))))
```

Mode fill

Le mode fill est plus complexe. On calcule le nombre d'espaces qu'il faut ajouter pour remplir la ligne avec les mots du tampon. Ce nombre s'obtient en divisant la place non utilisée par le nombre d'intervalles. Cette division a en général un reste qui sont des espaces que l'on répartit entre les premiers intervalles. Pour des raisons esthétiques, si un intervalle entre deux mots comporte plus de trois blancs, on passe en mode left. C'est souvent le cas dans la dernière ligne d'un paragraphe :

```
(define (creer-ligne-fill margeG  Lmots place-restante NbMots)
  (if (<  NbMots 2)     ;;pour moins de deux mots on passe en mode left
    (creer-ligne-left margeG  Lmots)
    (let ((nb-espaces   (+ 1 (quotient  place-restante   (- NbMots 1))))
          (reste-espaces (remainder place-restante  (- NbMots 1))))
      (if (<= 2 nb-espaces) ;;si trop d'espacement on passe en mode left
        (creer-ligne-left margeG  Lmots)
        (let ((espacement (creer-espace nb-espaces)))
          (letrec ((creer-ligne
                     (lambda (ligne Lmots i)
                       (cond ((=  NbMots i) ligne)
                             ((< i reste-espaces)
                              (creer-ligne (string-append ligne (car Lmots)
                                                          espacement " ")
                                           (cdr Lmots) (+ i 1)))
                             (else
                              (creer-ligne (string-append ligne (car Lmots)
                                                          espacement)
                                           (cdr Lmots) (+ i 1) ))))))

            (creer-ligne (creer-espace margeG) Lmots 0)))))))
```

Pour créer une chaîne de n espaces :

```
(define  (creer-espace  n)
  (make-string n #\space))
```

Pour écrire une ligne suivie d'un passage à la ligne :

```
(define (ecrire-ligne ligne stream)
  (display ligne stream)
  (newline stream))
```

Exercice 1 *1. En supposant donnée une fonction pour couper les mots, modifier ce programme pour permettre la césure des mots en mode fill.*

2. Il y a toujours un espace après un signe de ponctuation, mais certains signes, comme la virgule et le point, sont immédiatement derrière un mot, d'autres exigent un espace. Utiliser une table précisant les différents cas et intégrer ces règles dans cette méthode de formatage.

Structure du programme de formatage d'un fichier de texte

`(formatage input-file output-file mode margeG longLigne)`
Le formatage du fichier d'entrée est écrit dans le fichier de sortie.
Les valeurs initiales des paramètres de formatage sont passées en arguments.
Il y a quatre modes possibles : `left, right, center, fill.`
La marge gauche est un entier ≥ 0.
La longueur utile de la ligne est un entier ≥ 0.

> `(creer-env mode margeG LongLigne)`
> Création d'un prototype pour représenter l'environnement courant de formatage avec les valeurs des paramètres.
>
> `(creer-tampon Lmots nbMots placeRestante)`
> Création d'un prototype pour représenter le tampon courant avec les valeurs des paramètres.
>
> `(sauter-espaces in-stream)`
> Saute les espaces et rend le premier caractère non espace.
>
> `(commande?  char)`
> Prédicat pour tester si un caractère est le caractère qui signale le début d'une ligne de commandes.
>
> `(executer-commandes tampon env in-stream out-stream)`
> Pour lire et exécuter les commandes qui se présentent dans le fichier d'entrée.
>
> `(formater-ligne tampon env in-stream out-stream)`
> Fabrique, à partir de la liste des mots contenus dans le tampon, une ligne formatée selon l'environnement puis affiche cette ligne dans le fichier de sortie.
>
>> `(lire-lexeme in-stream)`
>> Rend le premier lexème lu. Un lexème est une suite maximale de caractères sans séparateur ou est un caractère de ponctuation.
>>
>>> `(separateur?  char)`
>>> Teste si un caractère est un signe de ponctuation, une marque de fin de ligne, de fin de fichier ou de début de commande.
>>
>> `(vider-tampon tampon env out-stream)`
>> Déclenche le transfert du contenu du tampon dans le fichier de sortie.
>>
>>> `(creer-ligne-left margeG Lmots)`
>>> Création d'une ligne justifiée à gauche sous forme d'une chaîne construite avec les mots de la liste Lmots.
>>>
>>> `(creer-ligne-right margeG Lmots place-restante)`
>>> Création d'une ligne justifiée à droite sous forme

d'une chaîne construite avec les mots de la liste Lmots.

`(creer-ligne-center margeG Lmots place-restante)`
Création d'une ligne centrée sous forme d'une chaîne construite avec les mots de la liste Lmots.

`(creer-ligne-fill margeG Lmots place-restante NbMots)`
Création d'une ligne justifiée sous forme d'une chaîne construite avec les mots de la liste Lmots.

> `(creer-space n)`
> Création d'une chaîne de `n` espaces.

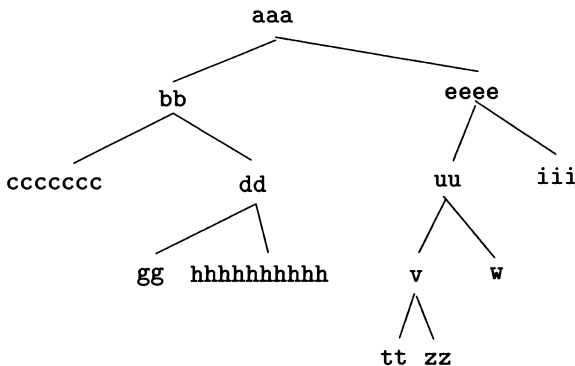
`(ecrire-ligne ligne-str stream)`
Ecriture de la chaîne dans le flux puis passage à la ligne.

14.3 Dessins d'arbres binaires et grammaires attribuées

On a donné au chapitre 7 §8 une méthode pour afficher un arbre binaire. Cette méthode a l'avantage de la simplicité mais l'arbre est représenté après rotation de 90° et les arcs reliant les sommets ne sont pas indiqués. On va présenter une méthode beaucoup plus puissante pour réaliser un affichage graphique complet d'un arbre binaire étiqueté. Par exemple, l'arbre binaire défini par la liste :

```
(aaa (bb ccccccc
         (dd gg hhhhhhhhhh))
     (eeee (uu  (v tt zz)
                w)
           iii))
```

sera affiché avec notre fonction `affiche-arbre` :

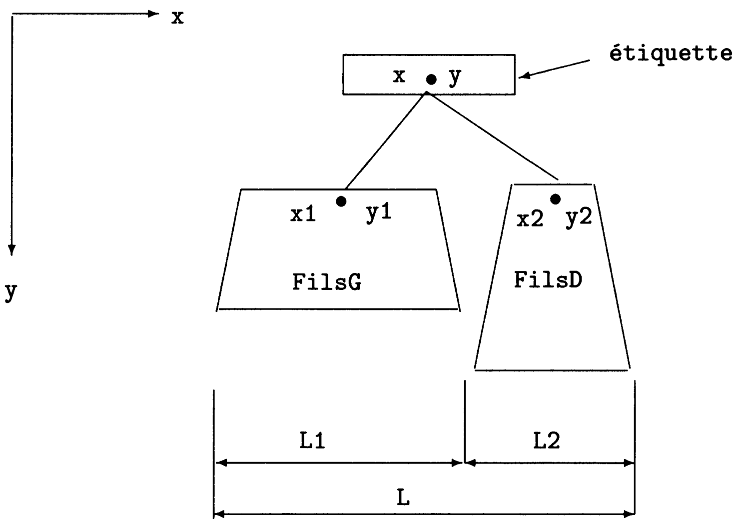


On suppose disposer des primitives graphiques suivantes :
`(move-to x y)` ;; positionne le crayon au point `x , y`
`(line-to x y)` ;; trace un segment du point courant au point `x , y`
`(draw-string str x y)` ;; affiche la chaîne `str` à partir de `x , y`
`(draw-string-center str x y)` ;; affiche la chaîne `str` centrée en `x , y`
`(string-width str)` ;; donne la longueur graphique de la chaîne `str`

Calcul des positions des nœuds

La difficulté est de calculer les positions des nœuds pour qu'il n'y ait pas de télescopage quelle que soit la largeur des étiquettes. Pour cela on doit écarter suffisamment les fils d'un même nœud ; cet écartement dépend de la largeur des sous-arbres respectifs. Le calcul des positions des nœuds se fait en deux temps :
- on commence par calculer les largeurs de tous les sous arbres
- ensuite, on déduit les positions des nœuds à partir de la position choisie pour la racine de l'arbre.

On trace les arêtes en joignant l'axe d'une étiquette aux axes des étiquettes des fils.



Calcul de la largeur totale L d'un arbre

Si l'arbre est réduit à une feuille, cette largeur est déterminée par l'encombrement horizontal de son étiquette. Pour ménager un espace horizontal entre deux étiquettes, on augmente la largeur d'une étiquette d'une quantité fixe `*ecartH*`. Aussi, la largeur d'une feuille de l'arbre sera donnée par l'expression :
`(+ (string-width etiquette) *ecartH*)`.

Dans le cas général, le calcul de la largeur `L` d'un arbre dépend des largeurs `L1` et `L2` de ses deux fils et le résultat est donné par la formule :

```
L = (max (+ L1 L2) (largeur etiquette))
```

On doit prendre le `max` avec la largeur de l'étiquette, au cas où l'étiquette serait plus large que les deux fils réunis.

Calcul de la position de chaque nœud

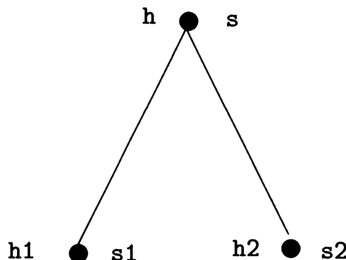
Appelons `x1 , y1` et `x2 , y2` les coordonnées des racines des deux fils d'un arbre de racine en `x , y`.
Le calcul de `y1` et `y2` est immédiat (noter que l'axe des y est dirigé vers le bas) :
`y1 = y + *ecartV*` et `y2 = y + *ecartV*`
Où `*ecartV*` est une constante qui représente l'écart vertical entre un nœud et ses fils. Pour calculer `x1` et `x2` on fait le choix (ce n'est pas le plus esthétique) de placer la racine dans l'axe de l'arbre. On trouve immédiatement :
`x1 = x - (L - L1)/2` et `x2 = x + (L - L1)/2`

Attributs synthétisés et attributs hérités

Pour faire ces calculs, on ajoute à chaque nœud de l'arbre les trois valeurs `L , x , y`. Ces valeurs associées à chaque nœud s'appellent des *attributs*. Il y a une importante différence entre l'attribut `L` et les attributs `x , y`. On constate que la largeur `L` ne dépend que des largeurs `L1` et `L2` des *fils* et de l'étiquette, on dit que L est un *attribut synthétisé*. Il se calcule en parcourant l'arbre de bas en haut, et est immédiat pour une feuille.
En revanche, le calcul de `x1 , x2` se fait à partir du `x` du *père* et des `Li`, on dit que ce sont des *attributs hérités*. Ils se calculent en parcourant l'arbre de haut en bas (après avoir calculé les `Li`). Le calcul, beaucoup plus simple, des `yi` montre que ce sont aussi des attributs hérités.

De façon plus générale, considérons un arbre (disons binaire pour simplifier les notations) où chaque nœud comporte un attribut `s` et un attribut `h` (`s` et `h` peuvent être des tuples de valeurs) qui vérifient en chaque nœud des relations de la forme :

$$s = F(s1, s2) \qquad h1 = G1(h, s1, s2) \qquad h2 = G2(h, s1, s2)$$



où F, $G1$, $G2$ sont des fonctions données (qui peuvent dépendre du nœud). Si le nœud est une feuille, on suppose donnée directement la valeur de `s`. Dans ces

conditions, on dit que les h sont des attributs hérités et que les s sont des attributs synthétisés. Bien entendu, on peut généraliser cette définition au cas de plusieurs attributs synthétisés et hérités en chaque nœud.

Les relations vérifiées par `L`, `x`, `y` montrent que l'on est bien dans ce cas de figure car on peut éliminer `L` en fonction de `L1` et `L2` dans l'expression des `xi`.
On a vu au chapitre 13 §7 une autre application de la notion d'arbre attribué : la transformation d'une expression infixe en une expression préfixe a utilisé des attributs synthétisés. L'étude des méthodes d'évaluation optimales de ces attributs est l'objet de la théorie dite des *grammaires attribuées*, on renvoie à la bibliographie pour cette théorie.

Calcul des attributs de chaque nœud

Pour faire ces calculs, on va utiliser notre extension objet SchemO définie au chapitre 11 §5.
On représente un arbre par un objet d'une classe `arbre` avec 6 champs :

```
(make-class  'arbre #f  'etiquette 'filsG 'filsD  'feuille? 'L 'x 'y)
```

Le champ `feuille?` est un booléen pour distinguer les feuilles des autres nœuds. Le calcul des attributs `L` de chaque nœud est réalisé par la méthode `calcul-L`. Si l'on a affaire à une feuille, le calcul est immédiat : on utilise une fonction graphique `string-width` qui donne la longueur de l'affichage d'une chaîne en points graphiques. Sinon on commence par calculer les `Li` des fils puis on utilise la formule donnant la valeur de `L`.

```
(defmethod  calcul-L  arbre  ()
  (let* ((etiquette (send self 'etiquette))
         (L-etiquette (+ (string-width etiquette) *ecartH*)))
    (if (send self 'feuille?)
      (send self 'set-L!  L-etiquette)
      (let ((Filsg (send self 'filsg))
            (Filsd (send self 'filsd)))
        (send Filsg 'calcul-L)
        (send Filsd 'calcul-L)
        (let ((L1 (send Filsg 'L))
              (L2 (send Filsd 'L)))
          (send self 'set-L! (max (+ L1 L2) L-etiquette)))))))
```

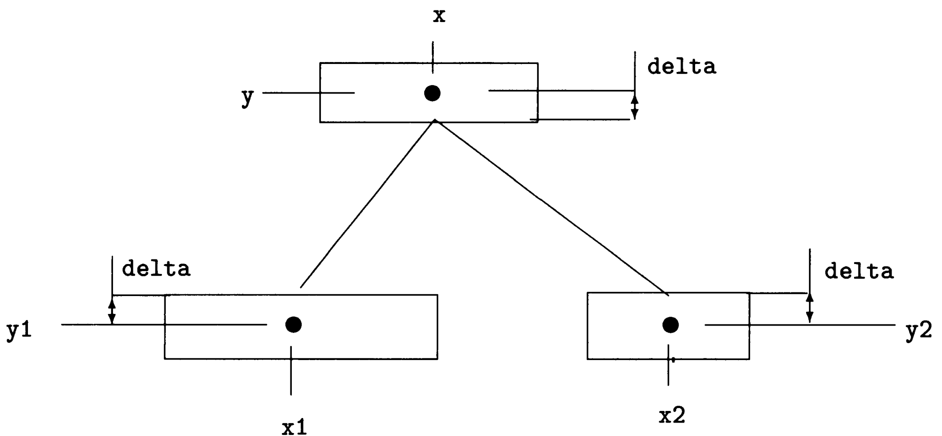
On calcule ensuite les attributs hérités `x` , `y` de tous les nœuds. On dit que le calcul des attributs est effectué en deux passes car on utilise deux parcours d'arbre. On se donne en paramètres les valeurs de `x` , `y` que l'on affecte aux attributs de la racine. Puis, si l'on n'est pas en une feuille, on utilise les formules précédentes pour calculer les valeurs des `xi` et `yi` des deux fils de l'arbre.

```
(defmethod calcul-x-y arbre  (x y)
  (send self 'set-x! x)
  (send self 'set-y! y)
  (unless (send self 'feuille?)
```

```
    (let ((L (send self 'L))
          (Fg (send self 'filsg))
          (Fd (send self 'filsd)))
      (let ((L1 (send Fg 'L))
            (L2 (send Fd 'L)))
        (let ((x1 (- x (quotient (- L L1) 2)))
              (x2 (+ x (quotient (- L L2) 2)))
              (y1 (+ y *ecartV*)))
          (send Fg 'calcul-x-y x1 y1)
          (send Fd 'calcul-x-y x2 y1))))))
```

Affichage de l'arbre

Une fois calculés tous les attributs, le dessin de l'arbre consiste à afficher l'étiquette de chaque nœud et à tracer les arêtes entre les nœuds. On affiche l'étiquette avec la fonction `draw-string-center` du système graphique, elle centre l'affichage d'une chaîne au point `x , y`. Pour ne pas chevaucher les étiquettes, les arêtes joignent le point `x , y + delta` aux points `x1 , y1 - delta` et `x2 , y2 - delta`, `delta` étant une constante de l'ordre de grandeur d'une hauteur de lettre.



On commence par afficher l'étiquette et les arêtes de la racine, puis on propage l'affichage aux deux fils.

```
(defmethod affiche-arbre-attribue  arbre  ()
  (let ((x (send self 'x))
        (y (send self 'y))
        (etiquette (send self 'etiquette)))
    (draw-string-center etiquette x y)
    (unless (send self 'feuille?)
      (let ((Fg (send self 'filsg))
            (Fd (send self 'filsd)))
        (let ((x1 (send Fg 'x))
```

```
            (x2 (send Fd 'x))
            (y1 (send Fd 'y)))
        (move-to  x  (+ y delta))
        (line-to  x1 (- y1 delta))
        (move-to  x  (+ y delta))
        (line-to  x2 (- y1 delta))
        (send Fg 'affiche-arbre-attribue)
        (send Fd 'affiche-arbre-attribue))))))
```

Il reste à transformer un arbre binaire en un objet arbre ayant ses attributs évalués. On se donne un arbre binaire au moyen de la représentation définie au chapitre 7 §7. La fonction `arbre2->objet-arbre` transforme un arbre binaire en un objet de la classe `arbre` mais sans calculer ses attributs numériques

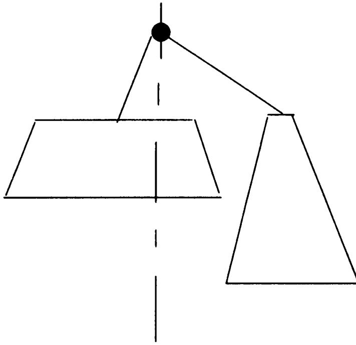
```
(define (arbre2->objet-arbre arbre)
  (let ((etiquette (symbol->string (arbre2:racine arbre))))
    (if (arbre2:feuille? arbre)
      (make-instance 'arbre 'etiquette etiquette 'feuille? #t)
      (let ((Filsg (arbre2->objet-arbre (arbre2:filsg arbre)))
            (Filsd (arbre2->objet-arbre (arbre2:filsd arbre))))
        (make-instance 'arbre 'etiquette  etiquette
                       'feuille?  #f   'filsg Filsg  'filsd Filsd)))))
```

La fonction principale `affiche-arbre` dessine un arbre binaire en plaçant sa racine au point `x` , `y`. On convertit l'arbre binaire en un objet arbre, puis on calcule l'attribut `L` de chaque nœud, puis les attributs `x` , `y` et enfin on affiche l'arbre attribué.

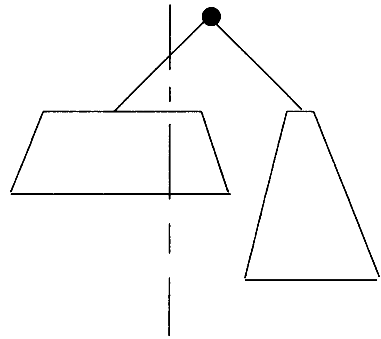
```
(define (affiche-arbre arbre x y)
  (let ((objet-arbre (arbre2->objet-arbre arbre)))
    (send objet-arbre  'calcul-L)
    (send objet-arbre  'calcul-x-y x y)
    (send objet-arbre  'affiche-arbre-attribue)))
```

Exercice 2 *Pour bien décomposer les opérations de calcul des attributs et les opérations d'affichage, on n'a pas hésité à multiplier les parcours d'arbres. Montrer que l'on peut regrouper le calcul des attributs hérités x , y et l'affichage au sein d'un même parcours.*

Exercice 3 *Adapter cette méthode d'affichage à un autre choix de visualisation. Au lieu de centrer chaque racine dans l'axe de son arbre, on demande que les arêtes issues d'un nœud aient des pentes symétriques.*



avec racine centrée



avec arêtes symétriques

Structure du programme d'affichage d'arbres

`(affiche-arbre arbre x y)`
Affiche un arbre binaire dans une fenêtre graphique avec l'étiquette de la racine centrée au point x , y.

> `(arbre2->objet-arbre arbre)`
> Convertit un arbre binaire en un objet de la classe arbre.
>
> > `arbre2:racine`
> > `arbre2:feuille?`
> > `arbre2:filsg`
> > `arbre2:filsd`
> > Les accesseurs de la structure d'arbre binaire.
>
> `calcul-L`
> Méthode pour calculer et affecter à chaque nœud la valeur de son attribut L.
>
> > `(string-width str)`
> > Longueur, en nombre de points, de l'affichage d'une chaîne.
> >
> > `*ecartH*`
> > Espace horizontal minimum entre deux étiquettes.
>
> `calcul-x-y`
> Méthode pour calculer et affecter à chaque nœud la valeur de ses attributs x , y.
>
> > `*ecartV*`
> > Espace vertical entre l'étiquette d'un nœud et celle de ses fils.

`affiche-arbre-attribue`
Méthode pour afficher un arbre dont on a auparavant calculé les attributs L , x , y.

> `(draw-string-center str x y)`
> Affiche une chaîne avec son centre en x , y.
>
> `(move-to x y)`
> Place le crayon au point x, y.
>
> `(line-to x y)`
> Trace un segment du point courant au point x , y.
>
> `delta`
> Constante indiquant le décalage vertical entre l'étiquette d'un nœud et l'origine des arêtes issues de ce nœud.

14.4 Formatage de formules : MiniTeX

Position du problème

L'affichage, ou plutôt le formatage, d'une formule mathématique est le fruit d'une longue tradition, aussi est-ce un sujet complexe et riche. Pour simplifier, on va se restreindre à une petite classe de formules, mais la méthode utilisée peut traiter les cas les plus complexes. Pour mettre le lecteur en appétit, voici quelques formules formatées avec notre système :

$$e = \left(\frac{a - b}{a + b}\right)^{\frac{\cos a}{1 + \tan b}}$$

$$n^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}} <= 2^{n^{n^{\cdot^{\cdot^{\cdot^{n}}}}}}$$

$$2^{\frac{1}{2}} = 1 + \frac{1}{2 + \frac{1}{2 + \frac{1}{2 + \frac{1}{2 + \frac{1}{\dots}}}}}$$

Méthode des boîtes

La méthode est basée sur la technique des boîtes introduite par D. Knuth à l'occasion de son fameux système de typographie informatique TEX. Une formule est réalisée par un assemblage de boîtes. Pour nous, les boîtes élémentaires correspondent à des chaînes alors que dans TEX c'est le caractère qui est la boîte indécomposable. Ce paragraphe peut donc être considéré comme la réalisation d'un mini (voire micro) TEX.

Considérons l'exemple de la formule $(cos\ x)^2$, il lui correspond une boîte circonscrite :

$$\boxed{(\ cos\ x\ )^2}$$

qui résulte de la juxtaposition des boîtes des constituants de cette formule :

$$\boxed{\boxed{(\ \boxed{cos}\ \boxed{x}\ )}^{\boxed{2}}}$$

Chaque boîte se positionne par rapport à une ligne de base, cette ligne correspond à la ligne sur laquelle l'écolier trace ses lettres. Par exemple, dans la formule suivante on a dessiné la ligne de base :

```
                                   cos a
                                 ---------
                                 1 + tan b
                      a - b
__________ e =      (-------)   ______________
                      a + b
```

Une boîte est un rectangle défini par cinq paramètres (attributs) :

- sa largeur L,

- son élévation e au-dessus de sa ligne de base,

- sa profondeur p en-dessous de sa ligne de base,

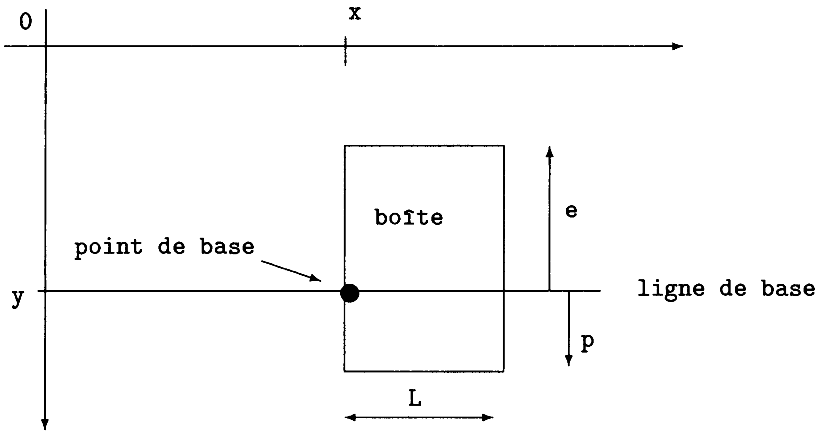
- l'abscisse x de son côté gauche dans un repère xOy,

- l'ordonnée y de sa ligne de base (l'axe des y étant dirigé vers le bas).

Le point de coordonnées x ,y s'appelle le point de base de la boîte.

Le problème est donc de calculer les paramètres de chaque boîte pour positionner l'affichage des constituants de la formule à afficher.

On fera usage des primitives graphiques indiquées dans la section précédente.



Caractéristiques d'une boîte

Langage de description des formules

Pour définir les formules à afficher, on se donne un petit langage d'expressions mathématiques préfixées :

```
<formule> ::= 'ident'| 'nombre'| '(' <op-binaire><formule><formule> '
              | '(' <op-unaire> <formule> ')' | '(' 'par' <formule> ')
<op-binaire> ::=  '+'  |  '-'  |  '*'  | '/'  | '='  | '<='  | '^'
<op-unaire> ::= 'sin'  | 'cos' | 'tan'  | 'log'
```

L'opérateur binaire `^` correspond à la fonction puissance et l'expression `(par formule)` signifie que l'on veut que formule soit entre parenthèses. Par exemple, la formule précédente est définie par l'expression :

```
(= e  (^ (par (/ (- a b)(+ a b)))
         (/ (cos a) (+ 1 (tan b)))))
```

Il s'agit maintenant de définir une fonction qui prend en entrée une formule et réalise son formatage selon les règles usuelles en mathématiques.
On va utiliser la programmation par objets pour au moins deux raisons :
- cela permet de regrouper le traitement de chaque type de formule,
- cela facilite l'extension à d'autres classes de formules.

Les classes de boîtes

A chaque type de boîte correspond une classe.

La classe `box`

Toutes les classes seront filles de la classe `box` qui contient les cinq champs qui définissent une boîte :

```
(make-class 'box #f  'e  'p 'L 'x 'y)
```

On aura parfois besoin de connaître la hauteur totale d'une boîte, aussi définit-on la méthode :

```
(defmethod H box ( )
  (+ (send self 'p)(send self 'e)))
```

Les champs se regroupent en deux catégories :
- les formules pour calculer les champs `e, p, L` montrent que ce sont des attributs synthétisés de l'arbre de l'expression,
- les formules pour calculer les champs `x, y` montrent que ce sont des attributs hérités.

La méthode de calcul des attributs est voisine de celle utilisée au paragraphe précédent. Dans une première passe on calcule les attributs synthétisés puis dans une deuxième les attributs hérités.
Les formules de calcul des attributs dépendent du type de l'expression contenu dans la boîte et seront donc décrites pour chaque classe. La seule chose commune à toutes les classes est d'affecter les valeurs des paramètres `x, y` aux attributs `x, y` de la boîte.

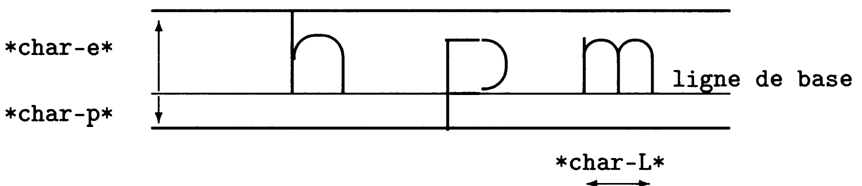
```
(defmethod calcul-x-y  box (x y)
  (send self 'set-x! x)
  (send self 'set-y! y))
```

La classe `box-string`

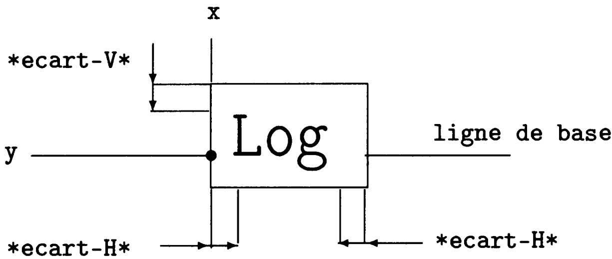
C'est la classe des boîtes élémentaires (c.-à-d. qui ne contiennent aucune boîte). Elle sert à représenter les chaînes (identificateur ou nombre) que l'on stocke dans son champ `str`.

```
(make-class 'box-string 'box 'str)
```

Pour calculer les attributs d'une telle boîte, il faut avoir des informations sur la police utilisée. Pour cela, on suppose données les constantes suivantes :
`*Char-e*` désigne l'élévation maximum d'un caractère,
`*Char-p*` désigne la profondeur maximum d'un caractère,
`*Char-L*` désigne la largeur maximum d'un caractère.



Pour traiter en une seule fois le problème des espaces horizontaux et verticaux entre les chaînes, la boîte associée à une chaîne s'écarte horizontalement d'un écart `*ecart-H*` et verticalement d'un écart `*ecart-V*` de la chaîne.



Le calcul des attributs synthétisés `L, e, p` est immédiat à partir des constantes précédentes et de la longueur de l'affichage de la chaîne `str`.

```
L = (+ *ecart-H* (largeur str) *ecart-H*))

e = (+ *Char-e* *ecart-V*) , p = (+ *Char-p* *ecart-V*)
```

D'où la méthode de calcul des attributs synthétisés :

```
(defmethod calcul-e-p-l box-string  ()
  (send self 'set-l!   (+ *ecart-H* (string-width (send self 'str))
                          *ecart-H*))
  (send self 'set-e!   (+ *Char-e* *ecart-V*))
  (send self 'set-p!   (+ *Char-p* *ecart-V*)))
```

On affecte aux attributs hérités `x, y` les valeurs paramètres `x, y` c'est donc un appel à la méthode de la classe mère `box`

```
(defmethod calcul-x-y  box-string  (x y)
  (send-next  self 'box-string 'calcul-x-y x y))
```

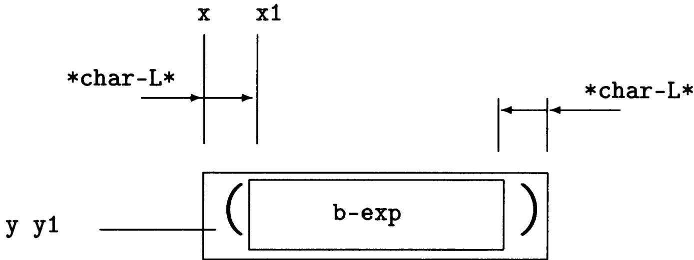
La méthode d'affichage consiste à afficher la chaîne à partir du point de coordonnées `x + *ecart-H* , y`.

```
(defmethod affiche box-string  ()
  (let ((x   (send self 'x))
        (y   (send self 'y))
        (str (send self 'str)))
    (draw-string  str (+ x *ecart-H*) y)))
```

La classe `box-parenthese`

```
(make-class 'box-parenthese 'box  'b-exp)
```

Elle sert à afficher une formule entre parenthèses. La boîte de la formule est contenue dans le champ `b-exp`.



Sa boîte se déduit de celle de son champ `b-exp` en l'allongeant aux deux bouts d'une longueur `*Char-L*` pour tenir compte des parenthèses :

```
L = (+ *Char-L* L1 *Char-L*)   ,  e = e1   ,  p = p1
```

D'où la méthode de calcul des attributs synthétisés :

```
(defmethod calcul-e-p-l box-parenthese  ()
  (let ((b-exp (send self 'b-exp)))
    (send b-exp 'calcul-e-p-l)
    (let ((L1 (send b-exp 'L))
          (e1 (send b-exp 'e))
          (p1 (send b-exp 'p)))
      (send self 'set-L!  (+ *Char-L* L1 *Char-L*))
      (send self 'set-e!  e1)
      (send self 'set-p!  p1))))
```

La valeur `x1` de l'expression incluse s'obtient en ajoutant `*Char-L*` à `x`, d'où le calcul des attributs hérités :

```
x1 = (+ x *Char-L*)   ,    y1 = y
```

```
(defmethod calcul-x-y  box-parenthese  (x y)
  (send-next  self  'box-parenthese 'calcul-x-y x y)
  (let ((b-exp (send self 'b-exp)))
    (send   b-exp 'calcul-x-y  (+ x *Char-L*) y)))
```

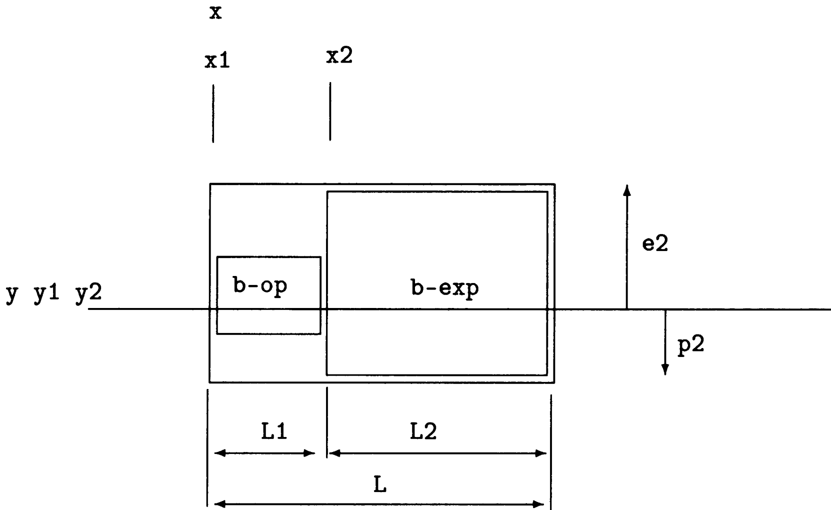
Pour afficher cette boîte, il suffit d'afficher la parenthèse gauche, puis l'expression puis la parenthèse droite :

```
(defmethod affiche  box-parenthese ()
  (let ((x (send self 'x))
        (y (send self 'y))
        (b-exp (send self 'b-exp)))
    (draw-string  "("  x y)
    (send b-exp 'affiche)
    (draw-string  ")"  (+ x *Char-L* (send b-exp 'L))  y)))
```

La classe `box-unaire`

Cette classe comporte deux champs propres : la boîte de l'opérateur unaire et celle de son opérande :

```
(make-class 'box-unaire 'box  'b-op  'b-exp)
```



Le calcul des attributs est clair sur la figure :
`L = (+ L1 L2)`
`e = e2` car l'élévation d'une boîte est toujours supérieure à celle d'une chaîne.
`p = p2` car la profondeur d'une boîte est toujours supérieure à celle d'une chaîne.
D'où la méthode de calcul des attributs synthétisés :

```
(defmethod calcul-e-p-l box-unaire  ()
  (let ((b-op   (send self 'b-op))
        (b-exp  (send self 'b-exp)))
    (send b-op   'calcul-e-p-l)
    (send b-exp  'calcul-e-p-l)
    (let ((L1 (send b-op 'L))
          (L2 (send b-exp  'L))
          (e2 (send b-exp  'e))
          (p2 (send b-exp  'p)))
      (send self  'set-L!  (+ L1 L2))
      (send self  'set-e!  e2)
      (send self  'set-p!  p2))))
```

Pour les attributs synthétisés on a immédiatement :
`x1 = x , x2 = (+ x L1) , y1 = y , y2 = y`

```
(defmethod calcul-x-y  box-unaire  (x y)
  (send-next  self 'box-unair 'calcul-x-y x y)
```

```
  (let ((b-op    (send self 'b-op))
        (b-exp   (send self 'b-exp )))
    (let ((L1 (send b-op 'L)))
      (send b-op   'calcul-x-y x y)
      (send b-exp  'calcul-x-y (+ x L1) y))))
```

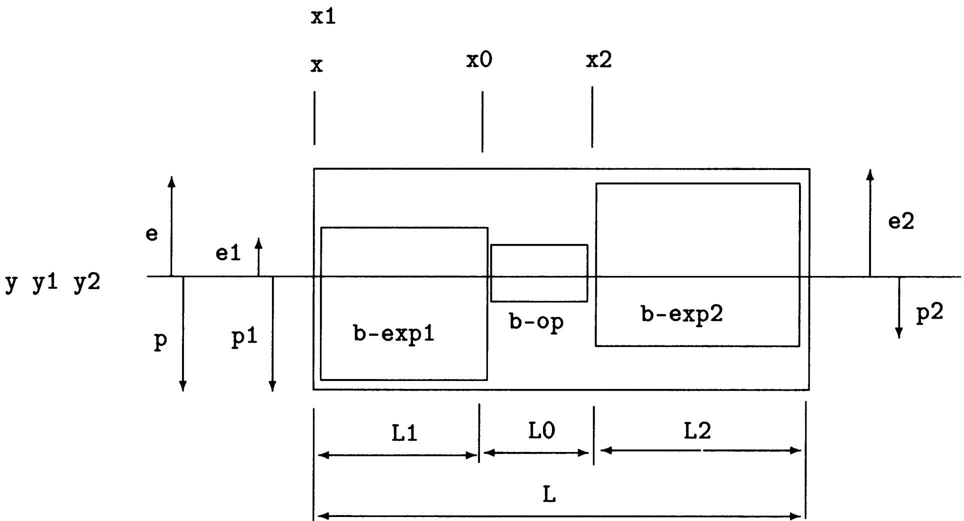
La méthode d'affichage consiste simplement à demander l'affichage de l'opérateur et de son opérande :

```
(defmethod affiche  box-unaire ()
  (let ((b-op   (send self 'b-op))
        (b-exp  (send self 'b-exp )))
    (send b-op   'affiche)
    (send b-exp  'affiche)))
```

La classe box-binaire

Cette classe comporte trois champs propres : la boîte de l'opérateur binaire, celle du premier opérande et celle du deuxième :

```
(make-class 'box-binaire 'box 'b-op 'b-exp1  'b-exp2)
```



L'élévation e est le maximum des élévations des boîtes des opérandes :
`e = (max e1 e2)`.
La profondeur p est le maximum des profondeurs des boîtes des opérandes :
`p = (max p1 p2)`.
La largeur est la somme des largeurs : `L = (+ L1 L0 L2)`.
D'où la méthode de calcul des attributs synthétisés :

```
(defmethod calcul-e-p-l box-binaire  ()
  (let ((b-op   (send self 'b-op))
        (b-exp1 (send self 'b-exp1))
        (b-exp2 (send self 'b-exp2)))
    (send b-op   'calcul-e-p-l)
    (send b-exp1  'calcul-e-p-l)
    (send b-exp2  'calcul-e-p-l)
    (let ((L0 (send b-op   'L))
          (L1 (send b-exp1 'L))
          (p1 (send b-exp1 'p))
          (e1 (send b-exp1 'e))
          (L2 (send b-exp2 'L))
          (p2 (send b-exp2 'p))
          (e2 (send b-exp2 'e)))
      (send self  'set-L!  (+ L0 L1 L2))
      (send self  'set-p!  (max p1 p2))
      (send self  'set-e!  (max e1 e2)))))
```

Le calcul des attributs hérités est immédiat :

```
x1 = x , x0 = (+ x L1) , x2 = (+ x L1 L0) ,  y1 = y0 = y2 = y
```

```
(defmethod calcul-x-y  box-binaire  (x y)
  (send-next  self  'box-binaire 'calcul-x-y x y)
  (let ((b-op   (send self 'b-op))
        (b-exp1 (send self 'b-exp1))
        (b-exp2 (send self 'b-exp2)))
    (let ((L0 (send b-op   'L))
          (L1 (send b-exp1 'L)))
      (send b-exp1  'calcul-x-y x y)
      (send b-op    'calcul-x-y (+ x L1) y)
      (send b-exp2  'calcul-x-y (+ x L0 L1) y))))
```

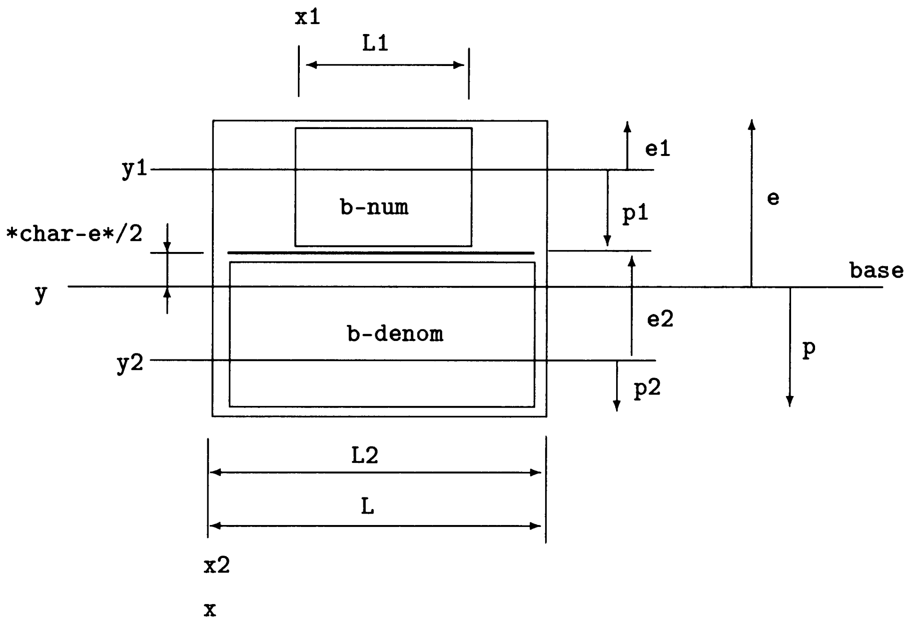
Comme d'habitude, l'affichage consiste à afficher chaque composante :

```
(defmethod affiche  box-binaire ()
  (let ((b-op   (send self 'b-op))
        (b-exp1 (send self 'b-exp1))
        (b-exp2 (send self 'b-exp2)))
    (send b-op   'affiche)
    (send b-exp1 'affiche)
    (send b-exp2 'affiche)))
```

La classe box-fraction

Cette classe comporte deux champs propres : la boîte du numérateur et celle du dénominateur :

```
(make-class 'box-fraction  'box  'b-num  'b-denom)
```



Le calcul des attributs est un peu plus complexe car on doit centrer les opérandes et la barre de fraction. De plus, la barre de fraction doit être un peu au-dessus de la ligne de base pour être alignée avec l'axe des opérateurs comme `+`, `-`, `...`, ce qui conduit à la décaler en hauteur d'une demi-hauteur de caractère.
La largeur L est le `max` des largeurs des opérandes : `L = (max L1 L2)`.
L'élévation `e` est égale à la hauteur du numérateur augmentée du décalage d'une demi-hauteur de caractère : `e = (+ H1 (quotient *Char-e* 2))`.
La profondeur p est égale à la hauteur du dénominateur diminuée d'une demi-hauteur de caractère : `p = (- H2 (quotient *Char-e* 2))`.
D'où la méthode de calcul des attributs synthétisés :

```
(defmethod calcul-e-p-l box-fraction ()
  (let ((b-num   (send self 'b-num))
        (b-denom (send self 'b-denom)))
    (send b-num   'calcul-e-p-l)
    (send b-denom 'calcul-e-p-l)
    (let ((L1 (send b-num   'L))
          (H1 (send b-num   'H))
          (L2 (send b-denom 'L))
          (H2 (send b-denom 'H)))
      (send self  'set-L!  (max L1 L2))
      (send self  'set-e!  (+  H1 (quotient *Char-e* 2)))
      (send self  'set-p!  (-  H2 (quotient *Char-e* 2))))))
```

Le décalage en hauteur complique aussi le calcul des ordonnées `y1` et `y2` :
`y1 = (- y p1 (quotient *Char-e* 2))`

y2 = (- (+ y e2) (quotient *Char-e* 2))

Enfin, le numérateur et le dénominateur doivent être centrés sur la barre de fraction, cela donne pour x1 et x2 :

x1 = (- (+ x (quotient L 2)) (quotient L1 2))

x2 = (- (+ x (quotient L 2)) (quotient L2 2))

D'où la méthode de calcul des attributs hérités :

```
(defmethod calcul-x-y  box-fraction  (x y)
  (send-next self 'box-fraction 'calcul-x-y x y)
  (let ((b-num   (send self 'b-num))
        (b-denom (send self 'b-denom))
        (L       (send self 'L)))
    (let ((L1 (send b-num   'L))
          (p1 (send b-num   'p))
          (L2 (send b-denom 'L))
          (e2 (send b-denom 'e)))
      (send b-num   'calcul-x-y (- (+ x (quotient L 2)) (quotient L1 2))
                                (- y p1 (quotient *Char-e* 2)))
      (send b-denom 'calcul-x-y (- (+ x (quotient L 2)) (quotient L2 2))
                                (- (+ y e2) (quotient *Char-e* 2))))))
```

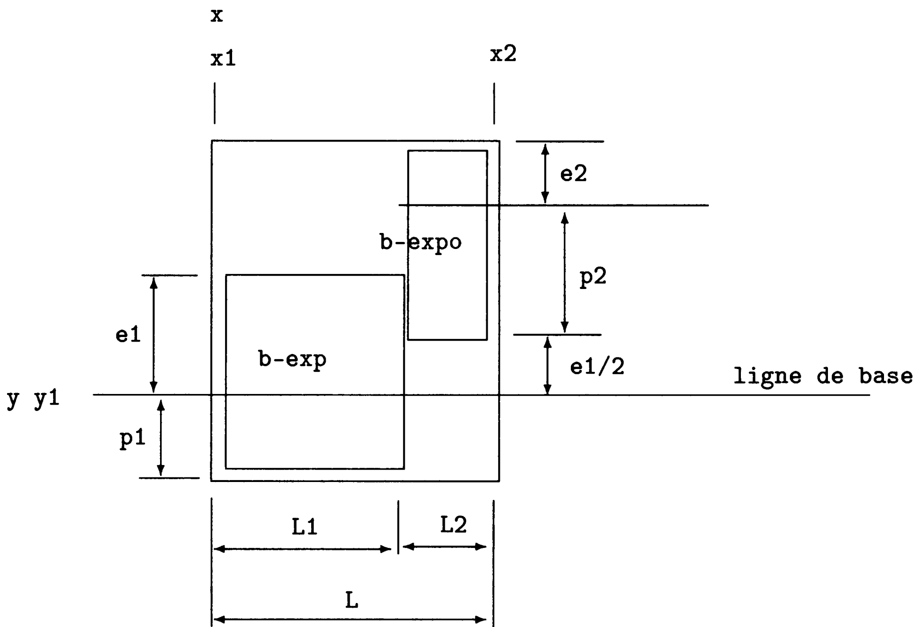
En plus de l'affichage du numérateur et du dénominateur, il faut tracer la barre de fraction. Elle va du point x , y - *Char-e*/2 au point x + L , y - *Char-e*/2.

```
(defmethod affiche  box-fraction ()
  (let ((b-num   (send self 'b-num))
        (b-denom (send self 'b-denom))
        (x           (send self 'x))
        (y           (send self 'y))
        (L           (send self 'L)))
    (send b-num   'affiche)
    (send b-denom 'affiche)
    (move-to x       (- y (quotient *Char-e* 2)))
    (line-to (+ x L) (- y (quotient *Char-e* 2)))
    ))
```

La classe box-puissance

Cette classe comporte deux champs propres : la boîte d'une formule et celle de son exposant.

```
(make-class 'box-puissance  'box 'b-exp 'b-expo)
```



L'exposant doit être situé à droite et décalé en hauteur mais il n'y pas de règle précise sur la valeur de ce décalage. On a choisi de placer le bas de l'exposant à mi-hauteur de l'expression. Il vient les expressions suivantes :
`L = (+ L1 L2)`
`e = (+ H2 (quotient e2 2))`
`p = p1`
D'où la méthode de calcul des attributs synthétisés :

```
(defmethod calcul-e-p-l box-puissance ()
  (let ((b-exp  (send self 'b-exp))
        (b-expo (send self 'b-expo)))
    (send b-exp    'calcul-e-p-l)
    (send b-expo   'calcul-e-p-l)
    (let ((L1 (send b-exp 'L))
          (p1 (send b-exp 'p))
          (e1 (send b-exp 'e))
          (L2 (send b-expo 'L)))
      (send self  'set-L! (+ L1 L2))
      (send self  'set-p!  p1)
      (send self  'set-e!  (+ (send b-expo 'H)(quotient e1 2))))))
```

Le point de base de la boîte complète est le même que celui de la formule dont on prend la puissance `x1 = x , y1 = y`
et la position de l'exposant entraine que `x2 = (+ x L1)` et
`y2 = (- y (quotient e1 2) p2))`.

D'où la méthode de calcul des attributs hérités :

```
(defmethod calcul-x-y  box-puissance  (x y)
  (send-next self 'box-puissance 'calcul-x-y x y)
  (let ((b-exp  (send self 'b-exp))
        (b-expo (send self 'b-expo)))
    (let ((L1 (send b-exp 'L))
          (e1 (send b-exp 'e))
          (p2 (send b-expo 'p)))
      (send  b-exp  'calcul-x-y x y)
      (send  b-expo 'calcul-x-y (+ x L1)
                                (- y (quotient e1 2) p2)))))
```

Enfin la méthode d'affichage est donnée par :

```
(defmethod affiche  box-puissance ()
  (let ((b-exp  (send self 'b-exp))
        (b-expo (send self 'b-expo)))
    (send b-exp   'affiche)
    (send b-expo  'affiche)))
```

Formatage de formules : MiniTeX

Appelons `MiniTeX` la fonction principale, elle affiche une formule dans une fenêtre en plaçant le point de base en `x , y`. En premier lieu, on convertit la formule en un objet boîte, puis l'on détermine ses attributs synthétisés, puis ses attributs hérités et enfin on l'affiche.

```
(define (MiniTeX formule x y)
  (let ((boite (formule->boite  formule)))
    (send boite  'calcul-e-p-l)
    (send boite  'calcul-x-y  x y)
    (send boite  'affiche)))
```

Même remarque que pour l'affichage d'arbres, on peut combiner dans la même passe le calcul des attributs hérités avec l'affichage. On a préféré le faire en deux temps car l'expression munie de tous ses attributs est une forme indépendante des fonctions d'affichage[2]. En particulier, elle peut être utilisée à d'autres fins, par exemple, pour générer du code pour un langage de description de page.

Il ne reste qu'à définir la fonction de conversion d'une formule en un objet boîte. On remplace les nombres ou les identificateurs par une boîte de chaîne. On associe à chaque type d'opérateur sa classe de boîte et l'on fait de même pour les opérandes. Il est évidemment très facile d'ajouter d'autres opérateurs ayant un style d'affichage déjà décrit.

```
(define (formule->boite  formule)
  (cond ((number? formule)
         (make-instance 'box-string 'str (number->string formule)))
        ((symbol? formule)
         (make-instance 'box-string 'str (symbol->string formule)))
```

[2]Dans un certain sens, cette expression joue le rôle du fichier .dvi généré par TEX.

```
        (else
         (case (car formule)
           ((+  - * =  <=)
            (make-instance 'box-binaire
                           'b-op (make-instance
                                  'box-string 'str
                                  (symbol->string (car formule)))
                           'b-exp1 (formule->boite  (cadr formule))
                           'b-exp2 (formule->boite  (caddr formule))))

           ((/)
            (make-instance 'box-fraction
                           'b-num   (formule->boite (cadr formule))
                           'b-denom (formule->boite (caddr formule))))

           ((sin cos tan log)
            (make-instance 'box-unaire
                           'b-op (make-instance
                                  'box-string 'str
                                  (symbol->string (car formule)))
                           'b-exp (formule->boite (cadr formule))))

           ((^)
            (make-instance 'box-puissance
                           'b-exp  (formule->boite (cadr formule))
                           'b-expo (formule->boite (caddr formule))))

           ((par)
            (make-instance 'box-parenthese
                           'b-exp (formule->boite (cadr formule))))

           (else (*erreur*  "operateur inconnu : " (car formule))))))
```

Pour obtenir les trois affichages donnés au début de ce paragraphe, il suffit d'appliquer `MiniTex` aux expressions suivantes :

```
(= e  (^ (par (/ (- a b)(+ a b))) (/ (cos a) (+ 1 (tan b)))))

(<= ( n (^ 2 (^ 2 (^ ... 2)))) (^ 2 (^ n (^ n (^ ... n)))) )

(= (^ 2 (/ 1 2))(+ 1 (/ 1 (+ 2 (/ 1 (+ 2 (/ 1 (+ 2 (/ 1 (+ 2 (/ 1 ...))))))))))
```

Pour les constantes globales on a utilisé les valeurs :

```
(define *Char-e* 10)
(define *Char-p* 2)
(define *Char-L* 5)
(define *écart-V* 2)
(define *écart-H* 3)
```

Remarque 1 Il est facile au lecteur de modifier ces constantes et certains calculs d'attributs pour obtenir un affichage plus à son goût. La programmation par objets

facilite aussi l'extension de MiniTeX à des objets mathématiques plus complexes : séries, intégrales, vecteurs, matrices, ...

Remarque 2 Pour éviter de rentrer dans les spécificités du système graphique utilisé, on n'a pas joué sur la taille des caractères. Mais il n'est pas difficile d'ajouter un attribut précisant cette taille et, par exemple, de réduire la taille des caractères d'un exposant.

Structure du programme MiniTeX

Définitions des classes utilisées :

```
(make-class 'box #f 'e 'p 'L 'x 'y)
     (make-class 'box-string 'box 'str)
     (make-class 'box-parenthese 'box 'b-exp)
     (make-class 'box-unaire 'box 'b-op 'b-exp)
     (make-class 'box-binaire 'box 'b-op 'b-exp1 'b-exp2)
     (make-class 'box-fraction 'box 'b-num 'b-denom)
     (make-class 'box-puissance 'box 'b-exp 'b-expo)
```

Fonctions, méthodes et constantes :

`(MiniTeX formule x y)`
Formate une formule en plaçant son point de base en `x` , `y`.

> `(formule->boite formule)`
> Transforme une formule en un objet d'une classe de boîte.
>
> `calcul-e-p-L`
> Méthode de calcul et d'affectation des attributs synthétisés : `e, p, L`
>
> > `(string-width str)`
> > Largeur de l'affichage d'une chaîne `str`.
> >
> > `*Char-e*`
> > Constante désignant l'élévation de police de caractères.
> >
> > `*Char-p*`
> > Constante désignant la profondeur de police de caractères.
> >
> > `*Char-L*`
> > Constante désignant la largeur maximum d'un caractère.
> >
> > `*ecart-V*`
> > Espace vertical minimal entre deux chaines.
> >
> > `*ecart-H*`
> > Espace horizontal minimal entre deux chaines.

`H`
Méthode donnant la hauteur totale d'une boîte.

`calcul-x-y`
Méthode de calcul et d'affectation des attributs hérités : `x, y`.

`affiche`
Méthode pour afficher un objet d'une classe de boîte.

`(draw-string str x y)`
Affiche une chaîne `str` à partir du point `x` , `y`.

`(move-to x y)`
Place le crayon graphique au point `x` , `y`.

`(line-to x y)`
Trace un segment du point courant au point `x` , `y`.

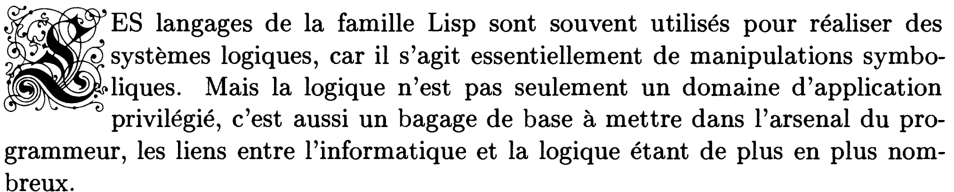
Les fonctions `string-width`, `draw-string`, `move-to` et `line-to` sont à définir en fonction du système graphique disponible avec Scheme.

14.5 De la lecture

Une autre approche de la justification de texte est donnée dans [SF90].
La théorie des grammaires attribuées est présentée dans [DJL88], elle est exposée aussi dans [FL88] en liaison avec ses applications à la compilation.
Le système TEX est décrit par son auteur dans [Knu84].

Chapitre 15

Calcul propositionnel



LES langages de la famille Lisp sont souvent utilisés pour réaliser des systèmes logiques, car il s'agit essentiellement de manipulations symboliques. Mais la logique n'est pas seulement un domaine d'application privilégié, c'est aussi un bagage de base à mettre dans l'arsenal du programmeur, les liens entre l'informatique et la logique étant de plus en plus nombreux.

On a toujours espéré pouvoir faire traiter par une machine des démonstrations de formules, ou bien modéliser le raisonnement d'un expert dans un domaine précis. Bien entendu, on sait que beaucoup de ces problèmes sont indécidables. Cependant, malgré cette limitation théorique, il y a des réussites. Elles peuvent concerner des sous-classes de problèmes pour lesquelles on a des algorithmes ou des heuristiques assez efficaces : calcul propositionnel, vérification de circuits digitaux, preuve de certaines classes de programmes Lisp, ou bien des systèmes interactifs où l'utilisateur guide la recherche.

On expose dans les chapitres 15 à 20 des éléments de logique utiles pour l'informaticien. Bien entendu, ce n'est pas un livre de logique, notre objectif est d'introduire les principales notions de base en liaison avec leur programmation. Notre expérience de l'enseignement de ces sujets nous a montré que la programmation systématique des concepts logiques est une approche bien acceptée par l'étudiant en informatique alors que les présentations uniquement mathématiques le rebutent souvent. On présente dans ce chapitre une première approche de la logique propositionnelle. Elle est basée sur la notion de valeur sémantique ; cette approche permet de faire le lien avec les circuits numériques. Le chapitre suivant présentera une autre approche basée sur la notion de preuve.

15.1 Langage du calcul propositionnel

Un pas considérable a été effectué en mathématique quand on est passé des manipulations numériques aux formules algébriques. Ce pas a aussi été franchi en logique

après des siècles de tâtonnement depuis Aristote. C'est le calcul booléen, du nom du logicien anglais G. Boole, qui permet d'algébriser certains raisonnements.

Connecteurs logiques

Dans la vie courante, on fait constamment des raisonnements du genre :

«je sais que mon manteau est dans l'entrée ou dans la penderie
et je ne le vois pas dans l'entrée
donc il doit être rangé dans la penderie»

Pour formaliser ce type de raisonnement, on utilise des connecteurs logiques et des identificateurs ou variables propositionnelles. Les principaux connecteurs logiques sont :

- $\vee$ qui signifie intuitivement *ou*
- $\wedge$ qui signifie intuitivement *et*
- $\neg$ qui signifie intuitivement *non*
- $\Rightarrow$ qui signifie intuitivement *implique*
- $\Leftrightarrow$ qui signifie intuitivement *équivalent*
- $\top$ qui signifie intuitivement le *vrai*
- $\perp$ qui signifie intuitivement le *faux*

Pour traduire le raisonnement précédent, on symbolise l'énoncé : «mon manteau est dans l'entrée» avec la variable e et l'énoncé : «mon manteau est dans la penderie» avec la variable p.
Reprenons cette phrase en soulignant les énoncés et en écrivant les connecteurs en gras :
«je sais que mon manteau est dans l'entrée **ou** dans la penderie
et je ne le vois **pas** dans l'entrée
donc il doit être rangé dans la penderie»

En remplaçant les parties soulignées et les conjonctions par les symboles correspondants, on obtient une formule qui représente la structure logique du raisonnement utilisé :

$$((e \vee p) \wedge (\neg e)) \Rightarrow p.$$

Exercice 1 *Jean, Pierre et Serge sont suspectés d'avoir commis un vol, on les interroge :*
- Pierre déclare "«Jean est coupable et Serge est innocent.»
- Serge déclare «Je suis innocent mais l'un au moins des autres est coupable.»
- Jean déclare «Si Pierre est coupable alors Serge l'est aussi.»
On introduit trois variables propositionnelles J, P, S *dont les significations intuitives sont respectivement : Jean est innocent, Pierre est innocent, Serge est innocent. Ecrire les formules logiques correspondant à chacune de ces dépositions.*

Grammaire du calcul propositionnel

Les formules propositionnelles ont une structure très voisine de celles des expressions arithmétiques, ce qui permet d'en donner une grammaire analogue à la grammaire ETF du chapitre 13 §4. On assimile :

- les connecteurs $\Rightarrow, \Leftrightarrow$ aux opérateurs $+, -$
- les connecteurs $\wedge, \vee$ aux opérateurs $*, /$
- le connecteur $\neg$ au moins unaire

Pour se dispenser de certaines parenthèses, on donne aux connecteurs les mêmes priorités que les opérateurs correspondants. Avec ces priorités, la formule booléenne précédente peut encore s'écrire de façon non ambiguë :

$$(e \vee p) \wedge \neg e \Rightarrow p$$

On est donc conduit à la grammaire suivante (l'axiome est noté `Prop`)

```
Prop    ::= Term '⇒' Prop | Term '⇔' Prop | Term
Term    ::= Facteur '∨' Term | Facteur '∧' Term | Facteur
Facteur ::= identificateur | '(' Prop ')' | '¬' Facteur | '⊤' | '⊥'.
```

Représentation en Scheme des formules propositionnelles

Pour manipuler en Scheme les formules propositionnelles, on a besoin de convenir d'une représentation *externe* (celle que l'on affiche) et d'une représentation *interne* (celle que l'on manipule).
En ce qui concerne la représentation externe, on fait les choix suivants :
- les parenthèses Scheme `( , )` servent de séparateurs,
- les connecteurs $\vee, \wedge, \Rightarrow, \Leftrightarrow, \neg\ \bot$,
top sont représentés respectivement par les symboles suivants : `V, &, =>, <=>, ~, F, T.`
- les identificateurs de variables propositionnelles, sont des symboles qui ne doivent pas s'écrire comme un connecteur
Par exemple, la formule précédente s'affichera :

```
( e V p ) & ~ e => p.
```

Exercice 2 *Adapter au calcul propositionnel l'analyseur lexical des expressions arithmétiques vu au chapitre 13 §3. On prendra comme table des mots-clés la a-liste*

```
(("V" . ou)("&" . et)("~" . non)("=>" . imp)
  ("<=>" . equiv)("F" . faux)("T" . vrai))
```

et comme table des symboles spéciaux la a-liste

```
(("(" . lpar) (")" . rpar)).
```

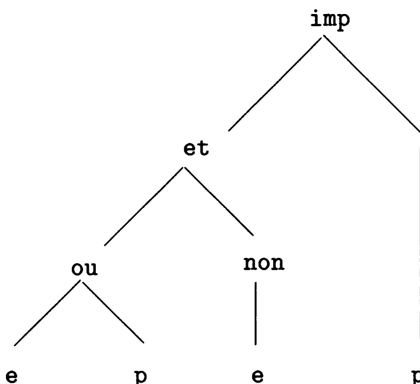
La représentation interne d'une formule est une expression préfixée analogue à celle utilisée pour les expressions arithmétiques. La grammaire de cette représentation interne est donnée par :

```
 <exp>      ::= '(' <connect2> <exp> <exp> ')' | '(' <connect1> <exp> ')'
              | <connect0> | ident
<connect2>::= 'ou' | 'et' | 'imp' | 'equiv'
<connect1>::= 'non'
<connect0>::= 'vrai' | 'faux'
```

Par exemple, la formule précédente aura comme représentation interne la liste :

```
 (imp (et (ou e p)(non e)) p)
```

On a choisi cette représentation car elle joue aussi le rôle de syntaxe abstraite, l'arbre de cette même formule est :



Pour manipuler cette représentation des formules, on définit quelques prédicats et fonctions d'accès.

```
(define *Lconnecteurs* '(et ou imp equiv non vrai faux))

(define (variable-propo? s)
  (and (symbol? s)
       (not (memq s *Lconnecteurs*))))
```

Une formule est composée si elle n'est pas réduite à une variable ou à l'un des deux connecteurs d'ordre 0 : `vrai, faux`.

```
(define compose-propo? pair?)
```

Pour ces formules on définit les accesseurs suivants :

```
 (define connecteur car) ; le connecteur de tête
 (define opd1 cadr)  ; le premier opérande
```

Pour ne pas déclencher d'erreur si l'on demande le deuxième opérande du connecteur unaire non, on utilise la fonction :

```
(define (opd2 formule)
  (and (not (null? (cddr formule)))
       (caddr formule)))
```

On a souvent besoin de calculer l'ensemble des variables qui apparaissent dans une formule. Si c'est une variable ou une constante, la réponse est immédiate. Si c'est une formule composée, on prend la réunion des variables qui apparaissent dans chacun des opérandes.

```
(define (listeVar-propo formule)
  (cond ((variable-propo? formule)(list formule))
        ((eq? formule 'vrai) '())
        ((eq? formule 'faux) '())
        (else (apply union (map listeVar-propo (cdr formule))))))
```

Exercice 3 *Adapter au calcul propositionnel le traducteur* **infixe->prefixe** *des expressions arithmétiques vu au chapitre 13 §7. Pour l'analyse syntaxique, il suffira de raisonner sur la valeur du lexème donnée par l'analyseur lexical.*

Exercice 4 *Adapter au calcul propositionnel l'afficheur d'expressions préfixes défini au chapitre 14 §1.*

15.2 Valeur sémantique d'une formule

Pour le moment, une formule du calcul propositionnel est une expression sans signification et donc sans valeur de vérité. Un objectif du calcul propositionnel est d'attribuer une valeur de vérité à une formule logique connaissant les valeurs de vérité affectées à chacune de ses variables. Il s'agit d'un problème d'évaluation semblable à celui des expressions arithmétiques, il faut donc se donner la sémantique des connecteurs, appelée *table de vérité*.

Fonctions sémantiques des connecteurs

Si l'on convient de représenter le `vrai` par 1 et le `faux` par 0, on définit les fonctions sémantiques associées aux connecteurs par :

X	Y	$\neg X$	$X \vee Y$	$X \wedge Y$	$X \Rightarrow Y$	$X \Leftrightarrow Y$	$\top$	$\bot$
0	0	1	0	0	1	1	1	0
0	1	1	1	0	1	0	1	0
1	0	0	1	0	0	0	1	0
1	1	0	1	1	1	1	1	0

Ce tableau traduit le sens usuel que nous accordons à ces connecteurs. Par exemple, le $\vee$ est vrai si au moins l'un de ses opérandes est vrai. La seule colonne non intuitive est celle du connecteur $\Rightarrow$ dont la valeur n'est 0 que dans le seul cas de $1 \Rightarrow 0$.
La représentation des valeurs de vérité par 0 et 1 permet de donner des formules pour chaque fonction sémantique :

- la fonction sémantique de $\neg$ est $1 - X$
- la fonction sémantique de $\vee$ est $max(X, Y)$

- la fonction sémantique de $\wedge$ est $min(X, Y)$
- la fonction sémantique de $\Leftrightarrow$ est $1 - |X - Y|$
- la fonction sémantique de $\Rightarrow$ est $max(1 - X, Y)$
- la fonction sémantique de $\top$ est la constante 1
- la fonction sémantique de $\perp$ est la constante 0

La représentation en Scheme de la fonction sémantique d'un connecteur est donc donnée par :

```
(define  (fonctionSemantique connecteur)
  (case connecteur
    ((vrai)  (lambda () 1))
    ((faux)  (lambda () 0))
    ((non)   (lambda (x)(- 1 x)))
    ((et)    min)
    ((ou)    max)
    ((imp)   (lambda (x y)(max (- 1 x) y)))
    ((equiv) (lambda (x y)(1- (abs (- x y)))))))
```

Remarque 1 Il y a $4^2 = 16$ fonctions distinctes de $\{0,1\} \times \{0,1\}$ dans $\{0,1\}$; les connecteurs binaires précédents n'épuisent donc pas tous les cas possibles. On verra un peu plus loin d'autres connecteurs binaires.

Valeur d'une formule pour une valuation

Pour calculer la valeur sémantique d'une formule, il faut se donner la valeur sémantique de chacune de ses variables. La donnée d'une valeur 0 ou 1 pour *chaque* variable s'appelle une *valuation*. Par exemple, avec la valuation v qui vaut 1 sur X et 0 sur Y, la formule $\neg X \vee Y$ a pour valeur 0. Plus généralement, la valeur d'une formule du type `(connect2 exp1 exp2)`, pour une valuation donnée, s'obtient en appliquant la fonction sémantique du connecteur aux valeurs (calculées récursivement) des opérandes.
Si l'on représente une valuation par une a-liste `((variable .  valeur)...)` , la valeur sémantique d'une formule est calculée par la fonction :

```
(define (valeurSemantique   exp  valuation)
  (cond ((variable-propo?   exp)(cdr (assq  exp valuation)))
        ((eq? 'vrai  exp) 1)
        ((eq? 'faux  exp) 0)
        (else (apply (fonctionSemantique (connecteur   exp))
                     (map (lambda (x)(valeurSemantique x valuation))
                          (cdr  exp))))))
```

```
? (valeurSemantique  '(imp (et (ou e p)(non e)) p)  '((e . 1)(p . 0)))
1       ; (e  ∨  p)  ∧  ¬  e  ⇒  p

? (valeurSemantique  '(imp (ou a b) faux)          '((a . 0)(b . 1)))
0      ; a  ∨  b   ⇒  p
```

Si une formule comporte n variables, il y a 2^n valuations possibles pour ces variables. Le tableau à 2^n lignes des valeurs sémantiques de la formule s'appelle sa *table de vérité.*

Par exemple, la table de vérité de la formule $\neg X \vee Y$ est :

X	Y	$\neg X \vee Y$
0	0	1
0	1	1
1	0	0
1	1	1

Exercice 5 *Ecrire la table de vérité de la formule* $(e \vee p) \wedge \neg e \Rightarrow p$.

15.3 Tautologies

Quelques définitions

Parmi les formules, il y a celles qui prennent toujours la valeur 1 quelle que soit la valuation considérée, autrement dit leur table de vérité ne comporte que des 1, on les appelle des *tautologies.*

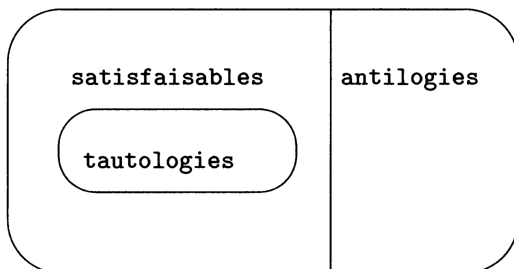
Par exemple, les formules $X \vee \neg X$ et $(e \vee p) \wedge \neg e \Rightarrow p$ sont des tautologies.

Pour indiquer qu'une formule φ est une tautologie, on utilise la notation $\models \varphi$.

La formule $X \wedge \neg X$ prend toujours la valeur sémantique 0, on dit que c'est une *antilogie*.

Une formule pour laquelle il existe une valuation qui lui donne la valeur 1 est dite *satisfaisable*.

On a une partition des formules en satisfaisables ou antilogies, les tautologies étant une partie des formules satisfaisables.



Ensemble des formules propositionnelles

Exercice 6 *Les formules suivantes sont-elles des tautologies ?*

$X \Rightarrow (Y \Rightarrow X)$

$(X \Rightarrow (Y \Rightarrow Z)) \Rightarrow ((X \Rightarrow Y) \Rightarrow (X \Rightarrow Z))$
$(X \Rightarrow Y) \Rightarrow \neg Y V X$

Exercice 7 *Donner une valuation qui satisfait les trois dépositions de l'exercice 1.*

Le prédicat `tautologie?` en Scheme

Pour vérifier qu'une formule est une tautologie, il suffit de construire sa table de vérité en interrompant la construction dès que l'on rencontre la valeur sémantique 0. Ecrivons, sur ce principe, un prédicat **tautologie?** qui rend **#t** si une formule est une tautologie.
La fonction locale **tauto-aux?** est une boucle qui génère les valuations successives et calcule les valeurs sémantiques correspondantes. Si l'on a généré les 2^n valuations (où n est le nombre de variables) sans rencontrer la valeur 0, on a bien une tautologie. En revanche, dès que l'on rencontre une valuation qui fournit la valeur sémantique 0, on interrompt la boucle, on affiche cette valuation et l'on renvoie la valeur **#f**.

```
(define (tautologie? formule)
  (let* ((Lvar (ListeVar-propo formule))
         (nb-var (length Lvar))
         (nb-valuation (expt 2 nb-var))
         (Liste0 (map (lambda (x) 0) Lvar))) ;; la liste (0 ... 0))
    (letrec ((tauto-aux?
               (lambda (i liste0ou1)
                 (let ((valuation (map cons Lvar liste0ou1)))
                   (cond ((= i nb-valuation) #t)
                         ((zero? (valeurSemantique formule valuation))
                          (*erreur*  " non satisfaite par : " valuation)
                          #f)
                         (else
                          (tauto-aux? (+ i 1)(Liste0ou1Suivante liste0ou1)))
                         )))))
      (tauto-aux 0 Liste0))))

? (tautologie? '(imp (et (ou e p)(non e)) p)) -> #t
? (tautologie? '(imp (ou X Y) (et X Y)))
ERREUR  non satisfaite par : ((y . 1) (x . 0))  -> #f
```

Pour construire ces listes de 0 ou 1, on utilise la fonction **Liste0ou1Suivante**. Elle prend en argument une telle liste et retourne la liste obtenue en ajoutant 1 à cette liste considérée comme l'écriture à l'envers[1] d'un nombre en binaire.

```
(define (Liste0ou1Suivante liste0ou1)
  (cond ((null? liste0ou1) '())
        ((= 0 (car liste0ou1))(cons 1 (cdr liste0ou1)))
        (else (cons 0 (Liste0ou1Suivante (cdr liste0ou1))))))

? (Liste0ou1Suivante '(0 0 0)) -> (1 0 0)
? (Liste0ou1Suivante '(1 0 1)) -> (0 1 1)
```

[1] C'est-à- dire en écrivant le chiffre des unités à gauche ...

Equivalence de formules

Si deux formules φ_1 et φ_2 prennent les mêmes valeurs pour toutes les valuations, on dit qu'elles sont *équivalentes*, on le note $\varphi_1 \equiv \varphi_2$.

Par exemple, la table de vérité de la formule $\neg X \vee Y$ montre qu'elle est équivalente à la formule $X \Rightarrow Y$.
La table de vérité du connecteur $\Leftrightarrow$ montre que deux formules φ_1 et φ_2 sont équivalentes si et seulement si la formule $\varphi_1 \Leftrightarrow \varphi_2$ est une tautologie.

En utilisant le prédicat `tautologie?`, on vérifie que l'on a les équivalences suivantes :
$X \vee X \equiv X \quad , \quad X \wedge X \equiv X \quad , \quad X \vee Y \equiv Y \vee X \quad , \quad X \wedge Y \equiv Y \wedge X$
$X \vee \top \equiv \top \quad , \quad X \vee \bot \equiv X \quad , \quad X \wedge \top \equiv X \quad , \quad X \wedge \bot \equiv \bot$
$\bot \equiv X \wedge \neg X \quad , \quad \top \equiv X \vee \neg X$
Associativité de $\vee$ et de $\wedge$:
$(X \vee Y) \vee Z \equiv X \vee (Y \vee Z) \quad , \quad (X \wedge Y) \wedge Z \equiv X \wedge (Y \wedge Z)$
Cette associativité permet d'écrire $X \vee Y \vee Z$ ou $X \wedge Y \wedge Z$ sans mettre de parenthèses.
On a la distributivité entre $\vee$ et $\wedge$:
$X \vee (Y \wedge Z) \equiv (X \vee Y) \wedge (X \vee Z) \quad , \quad X \wedge (Y \vee Z) \equiv (X \wedge Y) \vee (X \wedge Z)$.
Dualité de $\vee$ et $\wedge$:
$\neg(X \vee Y) \equiv (\neg X \wedge \neg Y) \quad , \quad \neg(X \wedge Y) \equiv (\neg X \vee \neg Y)$
Le connecteur $\neg$ est une involution : $\neg\neg X \equiv X$
$(\neg X \Rightarrow \neg Y) \equiv (Y \Rightarrow X)$
Définition de $\Leftrightarrow$: $(X \Leftrightarrow Y) \equiv (X \Rightarrow Y) \wedge (Y \Rightarrow X)$

Ces équivalences constituent les règles du calcul booléen quand on considère les formules modulo l'équivalence $\equiv$.

Si l'on calcule à équivalence près, on peut exprimer des connecteurs en fonctions d'autres. Par exemple, les équivalences $X \vee Y \equiv \neg X => Y$ et $X \wedge Y \equiv \neg(X \Rightarrow \neg Y)$ permettent d'éliminer partout les connecteurs $\vee$ et $\wedge$.

Exercice 8 *Utiliser le prédicat* `tautologie?` *pour aider l'inspecteur à résoudre cette énigme policière : après enquête, l'inspecteur est arrivé aux conclusions suivantes :*

- *Si Pierre n'a pas rencontré François c'est que Pierre ment ou que François est le coupable.*
- *Si François est le coupable alors Pierre n'a pas rencontré François et le crime a eu lieu après minuit.*
- *Si le crime a eu lieu après minuit alors François est le meurtrier ou Pierre ne ment pas.*

Peut-on en conclure que François est le coupable ?

15.4 Déduction sémantique

On a souvent besoin de relativiser la notion de tautologie par rapport à des hypothèses. De façon précise, on pose la :

Définition 1 On dit qu'un ensemble H de formules implique sémantiquement une formule φ si toute valuation valant 1 sur les formules de H vaut 1 sur φ.
Cette notion de déduction sémantique est notée $H \models \varphi$.

On remarque que lorsque l'ensemble H de formules est vide, on retrouve la notion de tautologie ; ce qui justifie l'emploi du symbole $\models$.
On peut alors énoncer le *théorème de la déduction sémantique* :

Théorème Soit H un ensemble de formules et φ et θ des formules. On a $H, \theta \models \varphi$ si et seulement si $H \models \theta \Rightarrow \varphi$.[2]

Preuve partie si Supposons $H, \theta \models \varphi$ et montrons que $H \models \theta \Rightarrow \varphi$.

Soit v une valuation valant 1 sur les formules de H. Pour montrer que $v(\theta \Rightarrow \varphi) = 1$, il suffit de vérifier que $v(\theta) = 1$ implique $v(\varphi) = 1$, ce qui est bien le cas grâce à l'hypothèse.

Preuve partie seulement si Supposons $H \models \theta \Rightarrow \varphi$ et montrons que $H, \theta \models \varphi$.

Soit v une valuation valant 1 sur les formules de H et sur θ, montrons que $v(\varphi) = 1$. Par hypothèse, on a $v(\theta \Rightarrow \varphi) = 1$ et comme $v(\theta) = 1$ on en déduit bien que $v(\varphi) = 1$.

On a donné cette démonstration car c'est un exemple typique de *méta raisonnement* en logique. La logique est l'étude du raisonnement et donc les théorèmes de logique sont des résultats sur le raisonnement. Mais, pour les démontrer on doit faire un raisonnement qui se place à un niveau *méta*, c'est-à-dire en dehors de la logique qui est l'objet de l'étude. Dans le cas du théorème précédent, on a fait un méta-raisonnement dont le principe est précisément le contenu intuitif du théorème objet à démontrer : pour démontrer que $A \Rightarrow B$, on suppose A et on prouve B !

La notion de déduction sémantique peut se ramener au cas d'une tautologie en faisant passer toutes les formules à droite. En effet, démontrons que :

$\theta_1, \ldots, \theta_n \models \varphi$ si et seulement si $\models \theta_1 \wedge \ldots \wedge \theta_n \Rightarrow \varphi$

La preuve consiste à remarquer qu'une valuation vaut 1 sur les formules $\theta_1, \ldots, \theta_n$ si et seulement si elle vaut 1 sur la formule $\theta_1 \wedge, \ldots, \wedge \theta_n$.

Exercice 9 *Montrer l'assertion suivante, dite du «modus ponens sémantique» : si l'on a $H \models \theta$ et $H \models \theta \Rightarrow \varphi$ alors on a $H \models \varphi$.*

[2] On a noté H, θ l'ajout de la formule θ à l'ensemble H.

15.5 Forme clausale

Il est parfois utile de transformer une formule propositionnelle en une formule équivalente ayant une structure particulière.

Une *clause* est une disjonction de littéraux et un *littéral* est soit une variable soit la négation d'une variable. Enfin, une formule est sous *forme clausale* si c'est une conjonction de clauses.
Une forme clausale est donc de la forme $(L_1 \vee \ldots \vee L_h) \wedge \ldots \wedge (N_1 \vee \ldots \vee N_k)$ où les L_i, N_j sont des littéraux.

Par exemple, la formule $(\neg X \vee Y \vee \neg Z)$ est une clause et la formule $(\neg X \vee Y) \wedge (X \vee \neg Y)$ est une forme clausale.

Pour tester si une formule propositionnelle est un littéral, on définit le prédicat :

```
(define (litteral-propo? formule)
  (or (variable-propo? formule)
      (and (compose-propo? formule)
           (eq? 'non (connecteur  formule))
           (variable-propo? (opd1 formule))))).
```

Principe de la transformation

L'objet de ce paragraphe est de montrer comment on peut passer d'une formule quelconque à une forme clausale équivalente. La transformation procède par étapes :

1ère étape On élimine les connecteurs $\Rightarrow, \Leftrightarrow, \top, \bot$ en utilisant les équivalences suivantes :

$$X \Rightarrow Y \equiv (\neg X \vee Y)$$

$$X \Leftrightarrow Y \equiv (\neg X \vee Y) \wedge (X \vee \neg Y)$$

$$\top \equiv X \vee \neg X$$

$$\bot \equiv X \wedge \neg X$$

2ème étape On rentre la négation à l'intérieur des formules pour qu'elle ne porte que sur des variables. Pour cela on utilise les équivalences suivantes :

$$\neg\neg X \equiv X$$

$$\neg(X \wedge Y) \equiv (\neg X \vee \neg Y)$$

$$\neg(X \vee Y) \equiv (\neg X \wedge \neg Y)$$

3ème étape Pour faire entrer les $\vee$ à l'intérieur des $\wedge$, on utilise la distributivité de $\vee$ par rapport à $\wedge$:

$$(X_1 \wedge X_2) \vee Y \equiv (X_1 \vee Y) \wedge (X_2 \vee Y)$$

$$X \vee (Y_1 \wedge Y_2) \equiv (X \vee Y_1) \wedge (X \vee Y_2)$$

Programmation de cette transformation

Voici maintenant les fonctions Scheme qui réalisent ces étapes successives, c'est un exemple typique de manipulations symboliques.

La première étape d'élimination de certains connecteurs est effectuée par la fonction `elimine-connecteur`, elle ne fait que mettre en œuvre les transformations indiquées.

```
(define (elimine-connecteur formule)
  (cond
    ((variable-propo? formule) formule)
    ((eq? 'vrai formule) '(ou x (non x)))
    ((eq? 'faux formule) '(et x (non x)))
    (else
     (let ((connect (connecteur  formule))
           (f1 (opd1 formule))
           (f2 (opd2 formule)))
       (let ((f1-reduite (elimine-connecteur f1))
             (f2-reduite (if (not (eq? 'non connect))
                             (elimine-connecteur f2))))
         (case connect
           ((non)    `(,connect  ,f1-reduite))
           ((et ou ) `(,connect  ,f1-reduite
                                 ,f2-reduite))
           ((imp)    `(ou  (non  ,f1-reduite
                                 ,f2-reduite)))
           ((equiv)  `(et
                        (ou (non ,f1-reduite)
                            ,f2-reduite)
                        (ou (non ,f2-reduite)
                            ,f1-reduite)))
           ))))))
```

```
 ? (elimine-connecteur '(imp (et (ou e p)(non e)) p))     ; (e ∨ p) ∧ ¬e ⇒  p
(ou (non (et (ou e p) (non e)) p))     ; ¬((e ∨ p) ∧ ¬e) ∨ p
```

La deuxième étape est effectuée par la fonction `rentre-non`. Si la formule est un littéral, il n'y a rien à faire. Si le connecteur de la formule est `non`, on utilise les transformations indiquées et sinon on effectue la transformation à l'intérieur de la formule.

```
(define (rentre-non formule)
  (if (litteral-propo? formule)
    formule
    (let ((connect (connecteur formule))
          (f1 (opd1 formule))
          (f2 (opd2 formule)))
      (if (eq? 'non connect)
        (let ((connect1 (connecteur  f1))
              (f11 (opd1 f1))
              (f12 (opd2 f1)))
```

```
        (case connect1
          ((non) (rentre-non f11))
          ((et) `(ou ,(rentre-non (list 'non f11))
                     ,(rentre-non (list 'non f12))))
          ((ou) `(et ,(rentre-non (list 'non f11))
                     ,(rentre-non (list 'non f12))))))
      `(,connect ,(rentre-non f1) ,(rentre-non f2))
      ))))

? (rentre-non  '(ou (non (et (ou e p) (non e))) p))
(ou (ou (et (non e) (non p)) e) p) ;      (¬e ∧ ¬p) ∨ e ∨ p
```

Avant de programmer la dernière étape, il faut convenir d'une représentation commode des formes clausales. Pour éviter de multiplier les connecteurs ou et les connecteurs `et`, on représente la forme clausale $(X_1 \vee \ldots \vee X_h) \wedge \ldots \wedge (Z_1 \vee \ldots V Z_k)$ par la liste de listes `((X1...Xh) ... (Z1...Zk))`.
Avec ce choix de représentation, la dernière transformation est réalisée par la fonction :

```
(define (forme-clausale formule)
  (if (litteral-propo? formule)
    (list (list formule))
    (case (connecteur formule)
      ((et) (append (forme-clausale (opd1 formule))
                    (forme-clausale (opd2 formule))))
      ((ou) (let((fc1 (forme-clausale (opd1 formule)))
                 (fc2 (forme-clausale (opd2 formule))))
              (append-map (lambda (x)
                            (map (lambda (y)(append x y))
                                 fc1))
                          fc2)))
      )))


? (forme-clausale '(ou (ou (et (non e) (non p)) e) p))
(p e (non e)) (p e (non p)))     ; (p ∨ e ∨ ¬e) ∧  (p ∨ e ∨ ¬p)
```

Comme les formules $e \vee \neg e$ et $p \vee \neg p$ sont des tautologies, la forme précédente permet de redémontrer que la formule initiale $(e \vee p) \wedge \neg e \Rightarrow p$ est une tautologie.

Il ne reste plus qu'à composer ces trois transformations pour mettre une formule quelconque sous forme clausale :

```
(define (MiseSousFormeClausale  formule)
  (forme-clausale (rentre-non (elimine-connecteur formule))))

? (MiseSousFormeClausale '(et (et a b) c))
((a) (b) (c))

? (MiseSousFormeClausale '(ou (et a b) (et c d)))
((c a) (c b) (d a) (d b))
```

15.6 Calcul booléen et circuits numériques

En électronique, les circuits numériques désignent des circuits à deux états : présence ou absence de courant. Si l'on modélise ces états par 1 et 0, alors le calcul booléen est aussi un calcul pour les circuits numériques.

Les connecteurs NAND et XOR

En utilisant les règles du calcul booléen on va montrer que l'on peut tout exprimer à l'aide d'un seul connecteur : le `NAND` ou «not and».
On définit le connecteur binaire `NAND`, que l'on notera $|$, par $(X|Y) = \neg(X \wedge Y)$.
On vérifie immédiatement les équivalences :

$\neg X \equiv X|X$
$X \vee Y \equiv (X|X)|(Y|Y)$
$X \wedge Y \equiv (X|Y)|(X|Y)$
$X \Rightarrow Y \equiv \neg X \vee Y \equiv ((X|X)|(X|X))|(Y|Y)$

Elles permettent d'éliminer les connecteurs $\neg, \vee, \wedge, \Rightarrow$ et par conséquent aussi les connecteurs $\Leftrightarrow, \top, \bot$.
La possibilité d'exprimer tous les connecteurs avec le `NAND` explique son importance dans les circuits numériques.
Un autre connecteur utile est le *ou exclusif* ; on le notera $+$, il est défini par

$$X + Y = (X \wedge \neg Y) \vee (\neg X \wedge Y)$$

Sa fonction sémantique vaut 1 quand un et seul de ses arguments vaut 1. On vérifie immédiatement que cette fonction sémantique est aussi la somme modulo 2 des valeurs des variables.

Exercice 10 *Ecrire les expressions Scheme qui représentent les fonctions sémantiques des connecteurs* `NAND` *et* `+`*. En déduire l'extension du prédicat* **`tautologie?`** *au cas de ces connecteurs.*

Exercice 11 *Un multiplexeur est un circuit qui permet de sélectionner une ligne dans 2^n lignes d'entrée $i_1, \ldots, i_{2^n}$ à l'aide de n lignes de contrôle $c_1, \ldots, c_n$. Pour $n = 1$, la sortie est définie par la formule :*

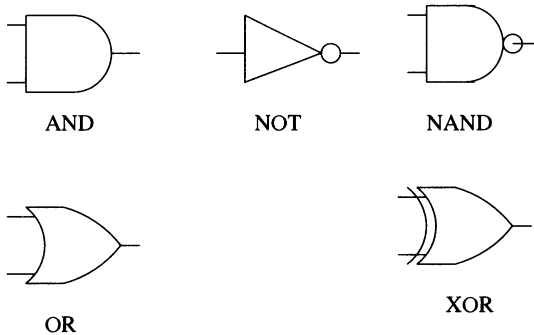
$$MUX(i_1, i_2, c_1) = (i_1 \ \wedge \ c_1) \ \vee \ (i_1 \ \wedge \ \neg c_1).$$

Prouver que tout connecteur binaire peut s'exprimer à l'aide des fonctions MUX , $\top$ et $\bot$.

Exercice 12 *(Pour algébriste) Vérifier que* `+` *définit sur le calcul booléen une opération commutative idempotente, et que les opérations* `+` *et $\wedge$ font du calcul booléen un anneau commutatif unitaire.*

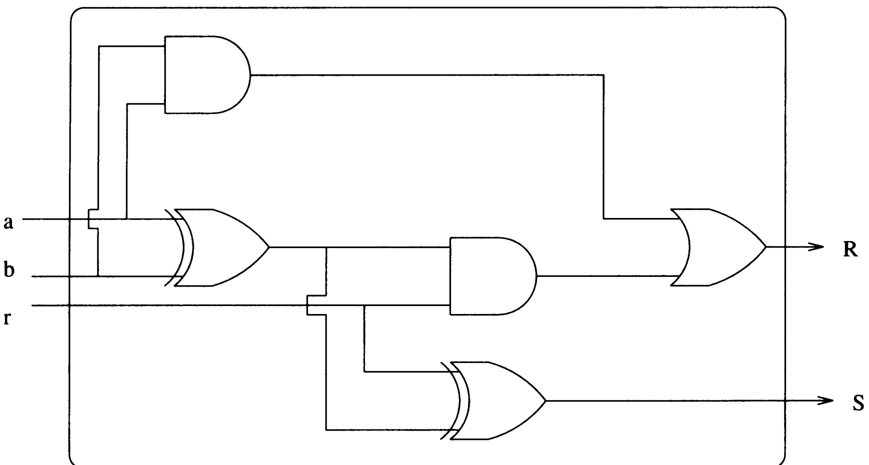
Additionneur 1 bit

Dans l'étude des circuits, la tradition est de noter `OR, AND, NOT, NAND, XOR` ce que nous avons notés $\vee, \wedge, \neg, |, +$. Dans les dessins de circuits numériques, on symbolise ces connecteurs par les diagrammes suivants :



Schemas des connecteurs en electronique

Les valeurs d'entrée 0 ou 1 sont placées sur les deux fils de gauche (l'unique fil dans le cas du `NOT`), le fil de droite porte la valeur de la fonction associée au connecteur. Le calcul booléen est utilisé pour prouver que des circuits digitaux réalisent bien les fonctionnalités escomptées. Par exemple, on va prouver que le circuit suivant réalise un additionneur 1 bit.



Additionneur 1 bit

Un additionneur 1 bit comporte trois entrées `a, b, r` et deux sorties `R, S`. Il doit réaliser les fonctions suivantes :

- `S` est la somme modulo 2 des trois entrées `a , b , r`,
- `R` est la valeur de la retenue de cette somme. On doit donc avoir :

```
S = (remainder  (+ a  b  r)  2)
R = (quotient   (+ a  b  r)  2)
```

Comme la retenue est égale à 1 dès qu'au moins deux entrées sont à 1, on peut écrire encore :
`R = (V (V (& a b)(& a r)) (& b r))`

Calculons les expressions de `S` et `R` définies par ce circuit. Par définition du ou exclusif, on trouve que `S = (+ (+ a b) r)`, ce qui est bien la somme modulo 2 des entrées.
Pour la retenue `R`, on trouve, d'après le schéma, l'expression
`R = (V (& a b) (& (+ a b) r)`. Il reste donc à vérifier que cette expression est équivalente à la spécification de `R`. Pour cela, on utilise notre fonction `tautologie?` :

```
? (tautologie? '(<=>    (V (& a b) (& (+ a b) r ))
                        (V (V (& a b)(& a r)) (& b r)) ))
#t
```

Ce circuit est bien correct. Dans la pratique on doit prouver des circuits beaucoup plus complexes, ce qui a motivé l'introduction de nouvelles méthodes de preuve de tautologie.

15.7 Méthode de Shannon

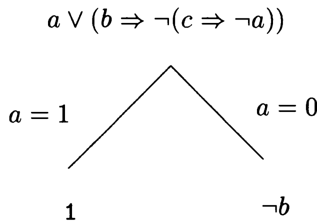
Les preuves de circuits nécessitent de manipuler des formules ayant un grand nombre de variables, aussi la méthode précédente explose rapidement. En effet, pour conclure qu'une formule à n variables est une tautologie, la définition du prédicat `tautologie?` montre que l'on doit toujours considérer les 2^n lignes de la table de vérité. On a cherché d'autres méthodes qui peuvent parfois conclure sans avoir à explorer tous les cas. Une de ces méthodes, due à Shannon, est basée sur l'étude de l'arbre des valeurs sémantiques.

Arbre de décision

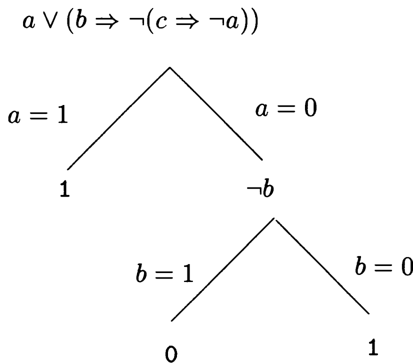
On représente par un arbre de décision la suite de tous les choix possibles de valeurs 0 ou 1 pour les variables. Expliquons sa construction dans le cas de la formule :

$$a \vee (b \Rightarrow \neg(c \Rightarrow \neg a))$$

On étudie ses valeurs sémantiques en supposant successivement que $a = 0$ ou 1 puis $b = 0$ ou 1 puis $c = 0$ ou 1.
On représente ces divers cas par un arbre dont les branches sont étiquetées par les valeurs attribuées aux variables :
- quand $a = 1$, on peut conclure que la formule vaut 1 sans avoir à considérer les valeurs de b et c car on sait que $X \vee Y$ vaut 1 dès que X ou Y vaut 1,
- quand $a = 0$, la définition de $\Rightarrow$ montre que $(c \Rightarrow \neg a)$ vaut 1, par conséquent la sous formule $b \Rightarrow \neg(c \Rightarrow \neg a)$ est équivalente à $\neg b$.
On dit que $\neg b$ est l'*évaluation partielle* de la valeur sémantique de la formule quand on sait que $a = 0$. On voit ainsi apparaître des simplifications dans l'exploration de l'arbre car la variable c a disparu. On peut imaginer des cas de simplification notables si c représente une «grosse» formule.



On continue la construction de cet arbre de décision avec les deux valeurs pour b. Quand b a pour valeur 1, la formule prend la valeur 0, ce qui montre que ce n'est pas une tautologie ; de plus la branche conduisant à cette feuille définit une valuation pour laquelle la formule vaut 0. On note que chaque feuille donne une valeur sémantique, on n'a donc pas besoin de considérer les valeurs pour c.



Evaluation partielle de formules

Pour programmer cette méthode, il faut se donner toutes les possibilités d'évaluations partielles des connecteurs. C'est l'objet de la fonction `eval-partielle` qui calcule une formule équivalente sachant qu'une variable a une valeur donnée. Si la formule est composée, on calcule la valeur partielle des opérandes puis l'on termine la réduction en discutant selon le connecteur de tête. Prenons par exemple le cas du connecteur ou, on calcule la valeur d'une formule partielle `(ou v1 v2)` selon les valeurs de `v1` et `v2` :

1. si `v1 = 1` ou `v2 = 1` la valeur est 1
2. si `v2 = 0` la valeur est celle de `v1`
3. si `v1 = 0` la valeur est celle de `v2`
4. sinon il n'y a plus de simplification, on rend la formule `(ou v1 v2)`

Notons que les formules partielles ne sont pas toujours des formules propositionnelles car elles peuvent contenir des valeurs 0 ou 1.

```
(define (eval-partielle formule variable valeur)
  (cond
    ((number? formule) formule)
    ((symbol?  formule)
     (if (eq? formule variable)  valeur formule))
    (else (let ((connect (connecteur formule))
                (f1 (opd1 formule))
                (f2 (opd2 formule)))
            (let ((v1 (eval-partielle f1 variable valeur))
                  (v2 (if f2 (eval-partielle f2 variable valeur))))
              (case connect
                ((non)(cond ((eq? v1 1) 0)
                             ((eq? v1 0) 1)
                             (else `(non ,v1))))
                ((et) (cond ((eq? v1 1) v2)
                            ((or (eq? v1 0)(eq? v2 0)) 0)
                            ((eq? v2 1) v1)
                            (else `(et ,v1 ,v2))))
                ((ou) (cond ((eq? v1 0) v2)
                            ((or (eq? v1 1)(eq? v2 1)) 1)
                            ((eq? v2 0) v1)
                            (else `(ou ,v1 ,v2))))
                ((imp) (cond ((eq? v1 1) v2)
                             ((or (eq? v1 0)(eq? v2 1)) 1)
                             ((eq? v2 0) `(non ,v1))
                             (else `(imp ,v1 ,v2))))
                ((equiv) (cond ((equal? v1 v2) 1)
                               ((eq? v1 1) v2)
                               ((eq? v2 1) v1)
                               (else `(equiv ,v1 ,v2))))
              ))))))
```

Appliquons cet évaluateur à la formule précédente :

```
? (eval-partielle '(ou a (imp b (non (imp c (non a)))))  'a 0)
(non b)

? (eval-partielle '(ou a (imp b (non (imp c (non a)))))  'a 1)
1
```

Exercice 13 *Ajouter le cas des connecteurs NAND et XOR.*

Un autre test de tautologie

Pour vérifier si une formule est une tautologie, on calcule ses évaluations partielles en fixant progressivement les valeurs de ses variables. Dès que deux évaluations partielles (relatives aux valeurs 0 et 1 d'une variable) donnent 1, on conclut à une tautologie.

```
(define (tautologie2? formule)
  (letrec ((tautologieAux?
            (lambda (forme-partielle  Lvar)
              (cond ((eq? 1 forme-partielle) #t)
                    ((eq? 0 forme-partielle) #f)
                    (else
                      (and (tautologieAux?
                             (eval-partielle forme-partielle (car Lvar) 1)
                             (cdr Lvar))
                           (tautologieAux?
                             (eval-partielle forme-partielle (car Lvar) 0)
                             (cdr Lvar))))
                    ))))
    (tautologieAux? formule (listeVar-propo formule))))
```

On vérifie que la formule précédente n'est pas une tautologie :

```
? (tautologie2?  '(ou a (imp b (non (imp c (non a)))))) -> #f
```

et on retrouve que la formule du premier paragraphe en est une

```
? (tautologie2?  '(imp (et (ou e p)(non e)) p)) -> #t
```

Si l'on préfère avoir une définition sous forme récursive terminale, il suffit d'utiliser une programmation par continuation en ajoutant dans la fonction `tautologieAux?` une continuation `k` en paramètre :

```
(define (tautologie3? formule)
  (letrec ((tautologieAux?
            (lambda (forme-partielle  Lvar k)
              (cond ((eq? 1 forme-partielle) (k #t))
                    ((eq? 0 forme-partielle) (k #f))
                    (else
                     (tautologieAux?
                       (eval-partielle forme-partielle  (car Lvar) 1)
                       (cdr Lvar)
                       (lambda (v1)
                         (tautologieAux?
                           (eval-partielle forme-partielle (car Lvar) 0)
                           (cdr Lvar)
                           (lambda (v2)(and v1 v2))))))
                   ))))
    (tautologieAux? formule (listeVar-propo formule) (lambda (x) x))))
```

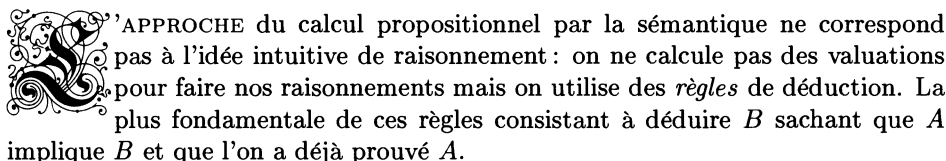
Des raffinements de cette méthode, connus sous le nom technique des BDD pour *Binary Decision Diagram*, permettent de traiter des circuits ayant des milliers de portes logiques. L'idée est d'essayer de diminuer la taille de l'arbre de décision en favorisant le partage des sous-expressions, on obtient alors un graphe acyclique.

15.8 De la lecture

Le calcul propositionnel est détaillé dans tout livre de logique, citons en quelques--uns plus spécialement destinés au lecteur informaticien [Gal86, Lal90, MW93]. Une présentation du calcul booléen faisant le lien avec les circuits digitaux se trouve dans [AU93]. Les principaux circuits numériques sont décrits dans [WH90]. La méthode dite "d'évaluation partielle" utilisée à la fin du chapitre est un cas particulier d'une méthode générale qui est exposée par exemple dans [JGS93].

Chapitre 16

Déduction naturelle et calcul des séquents



L'APPROCHE du calcul propositionnel par la sémantique ne correspond pas à l'idée intuitive de raisonnement : on ne calcule pas des valuations pour faire nos raisonnements mais on utilise des *règles* de déduction. La plus fondamentale de ces règles consistant à déduire B sachant que A implique B et que l'on a déjà prouvé A.

Avant de modéliser le calcul propositionnel par un système de règles, on va définir de façon très générale en quoi consiste un raisonnement : c'est la théorie des systèmes formels.

Il existe de nombreux systèmes formels pour décrire le calcul propositionnel. On décrit d'abord la *déduction naturelle* qui, comme son nom l'indique, se veut une modélisation proche de la pratique courante. On termine par le formalisme du *calcul des séquents* qui a l'avantage de s'implanter facilement.

16.1 Notion de systèmes formels

Définitions

La pratique des mathématiques (ou même la vie courante) nous a donné une idée intuitive de la notion de raisonnement. C'est une notion indépendante du domaine d'application considéré. En gros, un raisonnement consiste à déduire, en appliquant des règles, des énoncés nouveaux à partir d'énoncés déjà prouvés ou bien à partir d'énoncés admis. En logique, une expression suceptible de faire l'objet d'une preuve s'appelle un *jugement*. Pour définir la notion générale de preuve, on introduit la structure abstraite de *système formel*.

Définition 1 Un système formel F c'est la donnée :

- d'un langage L sur un alphabet A pour représenter les jugements à prouver,

- d'un ensemble, noté $AXIOMES$, de mots de L appelés les axiomes,
- d'un ensemble, noté $REGLES$, de règles ; une règle étant une *relation* sur L. On dit qu'une règle est n-aire si elle est définie par une partie de L^{n+1}.

On dit que les mots $H_1, \ldots, H_n$, C vérifient une règle n-aire $r \in REGLES$ si la relation $r(H_1, \ldots, H_n, C)$ est vraie.
L'usage est de l'écrire sous la forme d'une fraction $\frac{H_1, \ldots, H_n}{C}$ qui se lit : à partir des hypothèses $H_1, \ldots, H_n$ on peut conclure C.
Pour signifier qu'un axiome est un énoncé qui n'a pas besoin de justification, on l'écrit souvent sous forme d'une fraction sans numérateur.

Remarque 1 Pour ne pas compliquer, on a passé sous silence une restriction sur les ensembles $AXIOMES$ et $REGLES$. Ils doivent être des ensembles récursifs, c'est à dire pour lesquels on peut décider effectivement si un mot est dedans ou non.

Illustrons la notion de systèmes formels avec deux exemples.

Exemple 1

- On prend comme langage L les formules du calcul propositionnel,
- On prend comme axiomes les formules suivantes (appelées traditionnellement K et S) :

  K : $A \Rightarrow (B \Rightarrow A)$

  S : $(A \Rightarrow (B \Rightarrow C)) \Rightarrow ((A \Rightarrow B) \Rightarrow (A \Rightarrow C))$

- et pour règles, une seule règle (dite du modus ponens), citée dans l'introduction : $\frac{A \Rightarrow B \;,\; A}{B}$

On obtient un système formel, dit de Hilbert, pour la logique minimale ne comportant que le connecteur d'implication.

Il est important de noter que les variables A, B, C sont des *méta-variables,* c'est-à-dire qu'elles peuvent représenter n'importe quelle formule propositionnelle. De sorte que les formules K et S représentent en fait une infinité de formules. De même, la règle du modus ponens est bien une relation valable pour des instances quelconques de A et B.

Exemple 2

- On prend comme langage L l'ensemble des mots sur l'alphabet réduit à la parenthèse ouvrante et la parenthèse fermante,
- l'ensemble des axiomes est réduit au mot vide : $AXIOME = \{\epsilon\}$,
- on a deux règles définies par :

  $r_1 : \frac{A \;,\; B}{A\, B}$ $\qquad r_2 : \frac{A}{(A)}$

Ici la règle r_1 est binaire et r_2 est unaire.

Notion de preuve et de théorème

On est maintenant en mesure de définir la notion fondamentale de démonstration ou preuve dans un système formel F.

Définition 2 Une suite finie $\varphi_1, \ldots, \varphi_k$ de mots du langage L est une *démonstration* si chaque mot φ_j est soit un axiome, soit il existe une règle n-aire r et n mots $\varphi_{i_1}, \ldots, \varphi_{i_n}$ d'indices strictement plus petits que j et tels que la règle r soit vérifiée

$$r : \frac{\varphi_{i_1}, \ldots, \varphi_{i_n}}{\varphi_j}$$

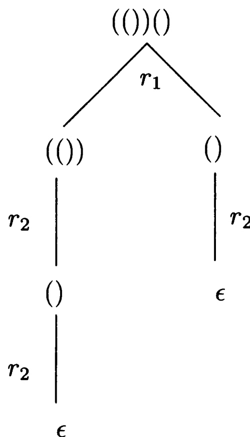
Définition 3 Un *théorème* d'un système formel est un mot qui apparaît dans une démonstration.

Cela traduit bien l'idée qu'un théorème est un axiome ou est démontré à partir d'éléments déjà prouvés.

On note $Th(F)$ l'ensemble des théorèmes d'un système F. Pour indiquer qu'une formule φ est un théorème, on écrit $\vdash_F \varphi$; on omet l'indice F quand il n'y a pas d'ambiguïté possible.

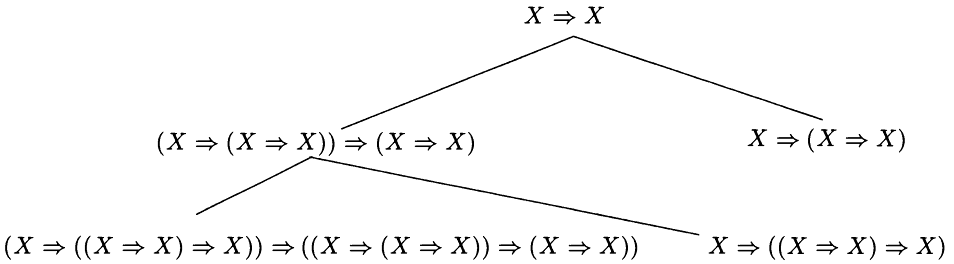
Dans le cas de l'exemple 2, la suite de formules : ϵ , () , (()) , (())() est une démonstration du mot (())(). En effet, () se déduit de ϵ par r_2, puis (()) se déduit de () par r_2 et enfin (())() se déduit de (()) et () par r_1. Il est assez facile de voir que les théorèmes de ce système sont les expressions bien parenthésées.

Il est souvent plus clair de représenter une démonstration d'un théorème par un *arbre de preuve*. C'est un arbre dont chaque nœud interne représente l'application d'une règle, les feuilles sont étiquetées par des axiomes et la racine est étiquetée par le théorème :



Bien entendu, la pratique des mathématiques nous a montré qu'il y a souvent plusieurs méthodes pour prouver un théorème, l'arbre de preuve n'est pas unique.

Le problème fondamental concernant un système formel c'est de pouvoir dire si un mot est un théorème ou non. Malheureusement, cette question est très souvent indécidable car il n'y a pas en général d'algorithme pour y répondre. Et même quand c'est décidable, il est souvent compliqué de trouver une démonstration. Par exemple, la formule $X \Rightarrow X$ est un théorème du système de l'exemple 1, en voici un arbre de preuve avec deux applications du modus ponens :



On reconnaît dans les feuilles de cet arbre des instances des axiomes K et S. La complication de la preuve d'une formule aussi simple que $X \Rightarrow X$ n'augure rien de bon pour des théorèmes plus complexes. Aussi, s'intéressera-t-on à d'autres systèmes formels pour définir les preuves en calcul propositionnel. En effet, on va voir comment des systèmes de règles différents peuvent conduire au même ensemble de théorèmes. Tout d'abord, on relativise la notion de théorème.

Définition 4 On dit qu'un jugement φ est un théorème sous un ensemble d'hypothèses H si c'est un théorème du système où l'on ajoute les formules de H aux axiomes, on le note $H \vdash \varphi$.

A partir des règles d'un système F, on peut définir des règles dérivées. Une règle n-aire r est *dérivée* si pour tout ensemble $\varphi_1, \ldots, \varphi_n, \psi$ de formules en relation par r, on peut prouver le théorème $\varphi_1, \ldots, \varphi_n \vdash_F \psi$.

Définition 5 Si deux ensembles de règles R_1 et R_2 sont tels que toute règle de l'un est une règle dérivée de l'autre, alors il est immédiat de vérifier que ces deux systèmes démontrent les mêmes théorèmes. On dit qu'ils sont *équivalents*.

Exercice 1 *Montrer que si l'on remplace dans l'exemple 2 la règle r_1 par la règle $\frac{A\ ,\ B}{(A\ B)}$, on obtient un système équivalent.*

Système MIU de Hofstadter

Pour permettre au lecteur de manipuler un système formel simple et non trivial, on extrait du livre de Hofstadter [Hof85] le système MIU.

Le langage de MIU est l'ensemble des mots sur les trois lettres M, I, U.
Un seul axiome : le mot MI.

Quatre règles :

$$r_1 : \frac{x}{xU} \qquad r_2 : \frac{Mx}{Mxx} \qquad r_3 : \frac{xIIIy}{xUy} \qquad r_4 : \frac{xUUy}{xy}$$

où x et y sont des méta-variables pouvant représenter n'importe quel mot du langage.
La règle 1 signifie que l'on peut toujours ajouter un U à la fin.
La règle 2 signifie que l'on peut répéter un suffixe de M.
La règle 3 signifie que l'on peut remplacer trois I consécutifs par un U.
La règle 4 signifie que l'on peut supprimer deux U consécutifs.
Notons que toutes les règles sont unaires, aussi les arbres de preuve seront filiformes.

Démontrons que $MIUI$ est un théorème de MIU.
On part de l'axiome :
MI
on utilise la règle 1
MIU
on utilise la règle 2
$MIUIU$
on utilise la règle 1
$MIUIUU$
on utilise la règle 4, pour obtenir enfin :
$MIUI$

Exercice 2 *On voit que les théorèmes de cette démonstration commencent tous par M. Montrer par récurrence sur la longueur d'une preuve, que c'est vrai pour tous les théorèmes de MIU.*

Exercice 3 *Le mot MUIU est-il un théorème de MIU ? Même question pour le mot MU (indication : raisonner sur le nombre de I présents dans un théorème).*

16.2 Déduction naturelle en logique propositionnelle

Après ce bref aperçu sur les systèmes formels, on va s'intéresser aux systèmes formels pour le calcul propositionnel. La recherche d'un ensemble de règles pour décrire notre raisonnement est un problème qui a intéressé les penseurs depuis l'antiquité : logique d'Aristote, lois de la scolastique du Moyen âge, ... On va utiliser une formalisation inventée par Gentzen [Sza70] et baptisée déduction naturelle «k»lassique : `NK`.

Le langage est constitué par les jugements de la forme $\Gamma \vdash \varphi$ où Γ désigne un *ensemble* fini de formules et φ une formule du calcul propositionnel. Intuitivement, un tel jugement se lit : on peut démontrer φ à partir des formules de Γ.

Règles de la déduction naturelle

Les axiomes sont les jugements $\Gamma \vdash \varphi$ pour lesquels φ est une formule aussi *dans* Γ, ce qui est en accord avec l'interprétation intuitive ci-Pessus.

Les règles se divisent en deux groupes : des règles qui permettent *d'introduire* un connecteur dans la partie droite de $\vdash$ et celles qui permettent *d'éliminer* un connecteur de cette partie droite.

Introduction de connecteur

Pour prouver $\varphi \wedge \psi$, il suffit de prouver φ et ψ :

$$\frac{\Gamma \vdash \varphi \qquad \Gamma \vdash \psi}{\Gamma \vdash \varphi \wedge \psi}$$

Pour prouver $\varphi \Rightarrow \psi$, il suffit de prouver qu'en supposant φ on en déduit ψ :

$$\frac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \Rightarrow \psi}$$

Pour prouver $\varphi \vee \psi$, il suffit de prouver φ ou de prouver ψ :

$$\frac{\Gamma \vdash \varphi}{\Gamma \vdash \varphi \vee \psi} \qquad \frac{\Gamma \vdash \psi}{\Gamma \vdash \varphi \vee \psi}$$

Si l'on peut prouver une formule et sa négation, alors on en déduit le faux :

$$\frac{\Gamma \vdash \varphi \qquad \Gamma \vdash \neg\varphi}{\Gamma \vdash \perp}$$

Elimination de connecteur

Si l'on a $\varphi \wedge \psi$ alors on a en particulier φ et ψ :

$$\frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \varphi} \qquad \frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \psi}$$

Si $\varphi \Rightarrow \psi$ et si l'on a prouvé φ alors on en déduit ψ (φ joue le rôle d'un lemme intermédiaire) :

$$\frac{\Gamma \vdash \varphi \Rightarrow \psi \qquad \Gamma \vdash \varphi}{\Gamma \vdash \psi}$$

Pour prouver θ sachant que l'on a $\varphi \vee \psi$ il suffit de prouver θ dans le cas où l'on suppose φ ainsi que dans le cas où l'on suppose ψ :

$$\frac{\Gamma \vdash \varphi \vee \psi \quad \Gamma, \varphi \vdash \theta \quad \Gamma, \psi \vdash \theta}{\Gamma \vdash \theta}$$

Si l'on peut prouver le faux en supposant que l'on a $\neg\varphi$, alors on a φ (raisonnement par l'absurde) :

$$\frac{\Gamma, \neg\varphi \vdash \perp}{\Gamma \vdash \varphi}$$

Le connecteur $\neg\varphi$ est une abréviation pour $\varphi \Rightarrow \perp$, aussi satisfait-il aux règles suivantes :

Si à partir de φ on démontre le faux, alors on a $\neg\varphi$:

$$\frac{\Gamma, \varphi \vdash \bot}{\Gamma \vdash \neg\varphi}$$

Si on peut prouver φ et $\neg\varphi$, alors on peut prouver n'importe quoi :

$$\frac{\Gamma \vdash \varphi \qquad \Gamma \vdash \neg\varphi}{\Gamma \vdash \psi}$$

Exemples de preuves

A titre d'exemple, prouvons le théorème $\vdash \varphi \Rightarrow \neg\neg\varphi$.
Pour conserver le sens des règles de déduction, on va en donner une preuve avec l'arbre de preuve ayant sa racine en bas :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\varphi, \neg\varphi \vdash \varphi} \qquad \dfrac{}{\varphi, \neg\varphi \vdash \neg\varphi}}{\varphi, \neg\varphi \vdash \bot}}{\varphi \vdash \neg\varphi \Rightarrow \bot}}{\varphi \vdash \neg\neg\varphi}}{\vdash \varphi \Rightarrow \neg\neg\varphi}$$

```
(axiomes)
(introduction du faux)
(introduction de ¬)
(définition de ¬)
(introduction de ⇒)
```

Cette démonstration n'a pas utilisé le raisonnement par l'absurde, en revanche la preuve de l'implication réciproque $\vdash \neg\neg\varphi \Rightarrow \varphi$ l'utilise :

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\neg\neg\varphi, \neg\varphi \vdash \neg\varphi} \qquad \dfrac{}{\neg\neg\varphi, \neg\varphi \vdash \neg\neg\varphi}}{\neg\neg\varphi, \neg\varphi \vdash \bot}}{\neg\neg\varphi \vdash \varphi}}{\vdash \neg\neg\varphi \Rightarrow \varphi}$$

```
(axiomes)
(introduction du faux)
(raisonnement par l'absurde)
(introduction de ⇒)
```

Compléments et notion de logique intuitionniste

Si l'on a une preuve de $\Gamma \vdash \varphi$, alors on a une preuve de $\Gamma, \psi \vdash \varphi$. En effet, il suffit de rajouter l'hypothèse ψ à chaque étape de la preuve de $\Gamma \vdash \varphi$; cela ne change pas la validité des règles à utiliser ni celle des axiomes utilisés.

Remarque 2 On peut donner une autre présentation des règles en ajoutant la règle :

$$\frac{\Gamma \vdash \varphi}{\Gamma, \psi \vdash \varphi}$$

et en réduisant la notion d'axiome aux jugements de la forme $\varphi \vdash \varphi$.

On en déduit que si l'on a une preuve du faux, alors on peut prouver n'importe quoi :
Si $\Gamma \vdash \perp$ alors $\Gamma, \varphi \vdash \perp$ et $\Gamma, \neg\varphi \vdash \perp$ d'où par la règle d'élimination du $\neg$, on déduit $\Gamma \vdash \psi$ pour n'importe quelle formule ψ. La règle

$$\frac{\Gamma \vdash \perp}{\Gamma \vdash \psi}$$

est donc une règle dérivée pour le système NK.
Si l'on remplace la règle du raisonnement par l'absurde par cette dernière règle, on obtient un système formel, moins puissant, qui s'appelle la *logique intuitionniste*. Gentzen note NJ ce système, il comporte des théorèmes en commun avec NK mais il n'y a pas ceux qui nécessitent la règle du raisonnement par l'absurde.

Exercice 4 *Prouver dans NK que l'on a le théorème $\vdash \varphi \vee \neg\varphi$, que l'on appelle le principe du tiers exclus. Prouver que la règle du raisonnement par l'absurde peut être remplacée par l'axiome du tiers exclus.*

Mais on peut montrer que $\vdash \varphi \vee \neg\varphi$ n'est pas un théorème dans la logique intuitionniste **NJ**. Cela reflète le caractère constructif de cette logique : pour prouver $\varphi \vee \neg\varphi$, il faut donner une preuve de φ ou une preuve de $\neg\varphi$.

On peut démontrer un théorème de *complétude* qui affirme que les théorèmes de NK sont exactement les tautologies du calcul propositionnel. Il est facile de montrer qu'un théorème est une tautologie car on vérifie immédiatement que les règles sont vraies sémantiquement (c'est-à-dire quand on remplace $\vdash$ par $\models$). La preuve de la réciproque est assez complexe.

Le système formel NK utilise des règles assez naturelles, mais la démonstration d'un théorème n'est pas toujours évidente. En particulier, les règles d'élimination demandent souvent de «l'imagination» car les formules utilisées dans les hypothèses ne sont pas toujours déjà présentes dans la conclusion. C'est typiquement le cas avec la règle d'élimination de $\Rightarrow$. Il s'agit de trouver une formule intermédiaire φ pour permettre de démontrer ψ. Cette méthode du lemme «astucieux» est monnaie courante en mathématiques.
La déduction naturelle sert souvent de base à la réalisation de systèmes *interactifs* de preuve. Pour construire un système de preuve *automatique*, on se base sur un autre système formel, également introduit par Gentzen, et appelé le calcul des séquents.

16.3 Calcul des séquents propositionnels

Le calcul des séquents est un système formel ayant une structure voisine de la déduction naturelle mais, contrairement à ce dernier, on peut éliminer des connecteurs à droite *et à gauche*. Le langage est constitué par les jugements de la forme $\Gamma \vdash \Delta$ où Γ et Δ désignent des ensembles finis de formules non tous les deux vides.

Les règles du calcul des séquents

Les *axiomes* sont les jugements $\Gamma \vdash \Delta$ pour lesquels les ensembles Γ et Δ ont une intersection non vide.

Les règles se divisent en deux groupes: des règles qui permettent d'éliminer un connecteur dans la partie gauche de $\vdash$ et celles qui permettent d'éliminer un connecteur dans la partie droite.

Elimination à gauche	**Elimination à droite**
$\dfrac{\Gamma, \varphi_1, \varphi_2 \vdash \Delta}{\Gamma, \varphi_1 \wedge \varphi_2 \vdash \Delta}$	$\dfrac{\Gamma \vdash \psi_1, \Delta \qquad \Gamma \vdash \psi_2, \Delta}{\Gamma \vdash \psi_1 \wedge \psi_2, \Delta}$
$\dfrac{\Gamma \vdash \varphi_1, \Delta \qquad \Gamma, \varphi_2 \vdash \Delta}{\Gamma, \varphi_1 \Rightarrow \varphi_2 \vdash \Delta}$	$\dfrac{\Gamma, \psi_1 \vdash \psi_2, \Delta}{\Gamma \vdash \psi_1 \Rightarrow \psi_2, \Delta}$
$\dfrac{\Gamma, \varphi_1 \vdash \Delta \qquad \Gamma, \varphi_2 \vdash \Delta}{\Gamma, \varphi_1 \vee \varphi_2 \vdash \Delta}$	$\dfrac{\Gamma \vdash \psi_1, \psi_2, \Delta}{\Gamma \vdash \psi_1 \vee \psi_2, \Delta}$
$\dfrac{\Gamma \vdash \varphi, \Delta}{\Gamma, \neg\varphi \vdash \Delta}$	$\dfrac{\Gamma, \psi \vdash \Delta}{\Gamma \vdash \neg\psi\Delta}$
$\dfrac{\Gamma \vdash \Delta}{\Gamma \vdash \bot, \Delta}$	$\dfrac{}{\Gamma, \bot \vdash \Delta}$

Le sens intuitif d'un séquent démontrable $\varphi_1, .., \varphi_m \vdash \psi_1, .., \psi_n$ est de représenter la tautologie $\varphi_1, .., \varphi_m \models \psi_1 \vee ... \vee \psi_n$. Mais il faudra prouver a posteriori qu'il en est bien ainsi.

Dans le calcul des séquents la preuve est guidée par les connecteurs à éliminer. On part du séquent à prouver et on utilise les règles qui permettent d'éliminer successivement tous les connecteurs. Si à la fin tous les séquents sont des axiomes, on a un théorème, sinon on sait que ce n'est pas un théorème.

Exemples de preuves

A titre d'exemple, prouvons le séquent $\vdash \neg(\varphi \wedge \psi) \Rightarrow (\neg\varphi \vee \neg\psi)$. Comme pour la déduction naturelle, il faut lire la preuve de bas en haut afin de garder la structure des règles utilisées. Le principe consiste à éliminer systématiquement tous les connecteurs logiques.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overline{\varphi, \psi \vdash \varphi} \qquad \overline{\varphi, \psi \vdash \psi}}{\varphi, \psi \vdash \varphi \wedge \psi}\ \wedge\ \texttt{droite}}{\psi \vdash \varphi \wedge \psi, \neg\varphi}\ \neg\ \texttt{droite}}{\vdash \varphi \wedge \psi, \neg\varphi, \neg\psi}\ \neg\ \texttt{droite}}{\neg(\varphi \wedge \psi) \vdash \neg\varphi, \neg\psi}\ \neg\ \texttt{gauche}}{\neg(\varphi \wedge \psi) \vdash \neg\varphi \vee \neg\psi}\ \vee\ \texttt{droite}}{\vdash \neg(\varphi \wedge \psi) \Rightarrow (\neg\varphi \vee \neg\psi)}\ \Rightarrow\ \texttt{droite}$$

(axiomes)

Essayons maintenant de prouver le séquent $\vdash (\neg\varphi \Rightarrow \psi) \Rightarrow (\psi \Rightarrow \varphi)$:

séquent irréductible

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\psi \vdash \varphi}{\psi \vdash \varphi}\ \texttt{axiome} \qquad \varphi, \psi \vdash \psi}{\varphi, \psi \vdash \varphi \wedge \psi}\ \wedge\ \texttt{à droite}}{\psi \vdash \varphi \wedge \psi, \neg\varphi}\ \neg\ \texttt{à droite}}{\vdash \varphi \wedge \psi, \neg\varphi, \neg\psi}\ \neg\ \texttt{à droite}}{\neg(\varphi \wedge \psi) \vdash \neg\varphi, \neg\psi}\ \neg\ \texttt{à gauche}}{\neg(\varphi \wedge \psi) \vdash \neg\varphi \vee \neg\psi}\ \vee\ \texttt{à droite}}{\vdash \neg(\varphi \wedge \psi) \Rightarrow (\neg\varphi \vee \psi)}\ \Rightarrow\ \texttt{à droite}$$

Ce n'est donc pas un théorème car on est ramené au séquent irréductible $\psi \vdash \varphi$ qui n'est pas un axiome.

Exercice 5 *Démontrer le séquent:*
$a \Rightarrow b \vee c, \neg c \Rightarrow a \wedge d, d \Rightarrow c \vee \neg b \vdash c$

Les règles du calcul des séquents possèdent une propriété fondamentale: toute formule apparaissant en conclusion d'une règle apparaît aussi dans une hypothèse. Autrement dit, on n'a pas besoin d'imagination, en revanche on a perdu pour certaines règles d'élimination à gauche le caractère «naturel». Les règles du calcul des séquents possèdent une symétrie gauche/droite. Notons que les règles d'élimination à droite sont semblables aux règles d'introduction en déduction naturelle mais on accepte un ensemble de formules à droite.

Remarque 3 L'application des règles binaires a tendance à faire diverger l'arbre de preuve, aussi une stratégie raisonnable est d'essayer d'éliminer en priorité les connecteurs associés à une règle unaire.

On démontre également un théorème de *complétude*: φ est une tautologie du calcul propositionnel si et seulement si le séquent $\vdash \varphi$ est démontrable. Plus généralement, on démontre (par récurrence sur le nombre de connecteurs) qu'un séquent $\varphi_1, .., \varphi_m \vdash \psi_1, .., \psi_n$ est un théorème du calcul des séquents si et seulement si on a $\varphi_1, .., \varphi_m \models \psi_1 \vee ... \vee \psi_n$. Ce résultat justifie l'interprétation intuitive des séquents.
Les théorèmes de complétude pour la déduction naturelle et pour le calcul des séquents permettent de conclure que $\Gamma \vdash \varphi$ est un théorème de la déduction naturelle si et seulement si c'est un théorème du calcul des séquents. Cette équivalence est difficile à démonter directement (voir références).

Exercice 6 *Compléter le système des séquents avec des règles pour les connecteurs* $\Rightarrow$ *et* $+$.

16.4 Implantation du calcul des séquents propositionnels

Le principe du moteur de preuve est le suivant. Pour prouver un séquent, on réduit les formules qui le composent en éliminant tous les connecteurs par application des règles. L'application d'une règle introduit de nouveaux séquents à prouver, aussi faudra-t-il considérer dès le départ la preuve d'une *liste* de séquents.
Un séquent comporte des formules à gauche et des formules à droite, de plus, parmi ces formules il y en a qui sont composées et d'autres réduites à des variables (on dira aussi *atomiques*). Pour éviter de répéter à gauche puis à droite le traitement des formules composées, on introduit la notion de *formule signée.* C'est simplement l'ajout à une formule de l'indication de son côté. Par exemple, on associe, à une formule f située à gauche, la formule signée (G f) et de même pour une formule à droite. D'où les fonctions de manipulation des formules signées:

```
(define partie-signe  car)
(define partie-formule cdr)
(define cons-signe&formule cons)
(define formule-atomique? variable-propo?)
```

Représentation des séquents

On est amené à représenter un séquent par un prototype à trois champs : la liste des formules atomiques à gauche, la liste des formules atomiques à droite et la liste des formules signées.
Un prototype séquent répondra aux messages suivants :
- `Lsigne&formule` : pour accéder à la liste des formules signées.
- `axiome?` : pour tester si un séquent est un axiome on se contente de regarder s'il y a une variable commune à gauche et à droite. Ce n'est pas une restriction car toutes les formules seront finalement réduites à des variables.
- `irreductible?` : pour tester si un séquent ne possède pas de formules composées.
Les règles du calcul des séquents conduisent à supprimer des formules dans un séquent et à en ajouter d'autres ; pour faciliter ces opérations on a les messages suivants :
- `supprimer-formule` : supprime la nième formule signée de la liste des formules signées.
- `ajouter-formule` : on ajoute une formule dont on donne le signe. Si elle est composée, on l'ajoute en queue de la liste des formules signées (la position en queue n'est pas importante pour le moment mais le sera quand on étendra ce système au chapitre 19). Et si elle n'est pas composée on la place en tête de la liste des formules atomiques de même côté.
Enfin, on a un message pour dupliquer un séquent et un pour l'afficher :
- `copier` : pour créer un nouveau séquent avec les mêmes champs.
- `affiche` : affiche les formules de gauche séparées par des espaces puis affiche $\vdash$ et les formules de droite. Les formules sont affichées en syntaxe abstraite, mais le lecteur ayant fait l'exercice 4 du chapitre 15 §1 pourra afficher les formules de façon plus agréable à l'œil.

```
(define (make-sequent LatomeG  LatomeD  Lsigne&formule)
  (lambda (message . Largts)
    (case message
      ((Lsigne&formule) Lsigne&formule)
      ((axiome?) (axiome? LatomeG LatomeD))
      ((irreductible? ) (null? Lsigne&formule))
      ((supprimer-formule) (set! Lsigne&formule
                              (remove-nth (car Largts) Lsigne&formule)))
      ((ajouter-formule)
       (let ((signe   (car Largts))
             (formule (cadr Largts)))
         (cond ((formule-compose? formule)
                (set! Lsigne&formule
                      (append  Lsigne&formule
                               (list (cons-signe&formule signe formule)))))
               ((eq? signe 'G) (set! LatomeG (cons formule  LatomeG)))
               (else (set! LatomeD (cons formule  LatomeD))))))
      ((copier-sequent) (make-sequent  LatomeG LatomeD  Lsigne&formule))
      ((affiche)
       (for-each (lambda (x)(display-all x "  "))
                 LatomeG)
       (for-each (lambda (signe&formule)
```

```
                (if (eq? 'G (partie-signe signe&formule))
                  (display-all "  " (partie-formule signe&formule))))
              Lsigne&formule)
    (display "|- ")
    (for-each (lambda (x)(display-all x "  "))
              LatomeD)
    (for-each (lambda (signe&formule)
                (if (eq? 'D (partie-signe signe&formule))
                  (display-all "  " (partie-formule signe&formule))))
              Lsigne&formule)
    ))))
```

On a utilisé deux fonctions auxiliaires, une pour tester si l'on avait la même variable des deux côtés et une autre pour supprimer le nième élément d'une liste.

```
(define (axiome? LatomeG LatomeD)
  (not (null? (intersection LatomeG LatomeD))))


(define (remove-nth No L)
  (if (zero? No)
    (cdr L)
    (cons (car L)(remove-nth (- No 1) (cdr L)))))
```

Fonction principale

La fonction principale `Demontrer-Lsequents?` est un prédicat qui vaut `#t` si tous les séquents d'une liste sont démontrables. Elle prend aussi en paramètre un booléen `trace?` pour savoir si l'on désire afficher le détail de la preuve ou donner seulement le résultat.
Son principe est le suivant : s'il n'y a plus de séquents à prouver c'est gagné, sinon on s'occupe du premier de la liste. Si c'est un axiome, il reste à prouver les autres séquents. S'il est irréductible, ce n'est pas la peine d'aller plus loin, il y a échec de la preuve. Dans le cas général, on choisit une formule composée du séquent et on ajoute, dans les séquents à prouver, le ou les séquents qui résultent de l'application de la règle à cette formule.

```
(define (Demontrer-Lsequents? Lsequents trace?)
  (cond
    ((null? Lsequents)(display "tout est demontre    ") #t)
    (else
      (let ((sequent (car Lsequents)))
        (affiche-cond trace? "on traite le sequent : " sequent)
        (cond
          ((sequent 'axiome?)
           (when trace? (display-alln "c'est un axiome : " #\newline))
           (Demontrer-Lsequents? (cdr Lsequents) trace?))
          ((sequent 'irreductible?)
           (affiche-cond trace? "le sequent :" sequent " est irreductible")
           (*abort* #f))
          (else (let ((Lnouveaux-sequents (traiter-sequent sequent trace?)))
```

```
                (Demontrer-Lsequents? (append Lnouveaux-sequents
                                              (cdr Lsequents)) trace?)))
        )))))
```

Le traitement d'un séquent consiste à choisir une formule composée du séquent et à lui appliquer la règle correspondante.

```
(define (traiter-sequent sequent trace?)
  (bind (signe formule  No) (choix-formule (sequent 'Lsigne&formule))
    (sequent 'supprimer-formule  No)
    (if (eq? signe 'G)
      (appliquer-regleG  sequent
                         (connecteur formule) formule
                         trace?)
      (appliquer-regleD  sequent
                         (connecteur formule) formule
                         trace?))
    ))
```

Stratégie pour le choix de la règle

On choisit en priorité les règles *unaires* pour essayer de retarder l'élargissement de l'arbre de preuve. Pour définir cette stratégie de choix, on associe à chaque connecteur, et selon son côté, un coefficient de priorité.

```
(define (priorite signe&formule)
  (let ((son-signe (partie-signe signe&formule))
        (son-connecteur (connecteur (partie-formule signe&formule))))
    (if (eq? 'G son-signe)
      (case son-connecteur
        ((non)  3)
        ((et)   2)
        ((ou)   1)
        ((imp)  1))
      (case son-connecteur
        ((non)  3)
        ((et)   1)
        ((ou)   2)
        ((imp)  2)))))
```

Le choix de la formule signée à traiter est réalisé par la fonction `choix-formule` qui retourne une liste constituée du signe, de la formule et de sa position dans la liste des formules signées.

```
(define (choix-formule  Lsigne&formule) ; -> (signe formule No)
  (let ((Lpriorite (map priorite Lsigne&formule)))
    (let ((prioriteMax   (apply max Lpriorite)))
      (let ((No (- (length Lpriorite)
                   (length (memq prioriteMax Lpriorite)))))
        (let ((signe&formule  (list-ref Lsigne&formule No)))
          (list (partie-signe signe&formule)
                (partie-formule signe&formule) No)))))
```

Application d'une règle

Une fois choisie la formule, son connecteur de tête détermine la règle à utiliser. Pour une règle unaire, on modifie les formules du séquent passé en paramètre. Pour une règle binaire, on crée une copie du séquent et l'on retourne le séquent et sa copie après modification des formules selon la description donnée par la règle. On a fait une fonction pour les règles de droite et une pour celles de gauche.

```
(define  (appliquer-regleG sequent connecteur  formule trace?)
  (when trace? (display-alln "regle du " connecteur "  a gauche" #\newline))
  (let ((f1 (opd1 formule))
        (f2 (opd2 formule)))
    (case connecteur
      ((non) (sequent 'ajouter-formule 'D f1) (list sequent))
      ((ou)  (let ((sequent-bis (sequent 'copier-sequent)))
               (sequent     'ajouter-formule 'G f1)
               (sequent-bis 'ajouter-formule 'G f2)
               (list sequent sequent-bis)))
      ((et)  (sequent  'ajouter-formule 'G f1)
             (sequent  'ajouter-formule 'G f2)
             (list sequent))
      ((imp) (let ((sequent-bis (sequent 'copier-sequent)))
               (sequent      'ajouter-formule 'D f1)
               (sequent-bis  'ajouter-formule 'G f2)
               (list sequent sequent-bis))))))

(define (appliquer-regleD sequent connecteur formule trace?)
  (when trace? (display-alln "regle du "  connecteur "  a droite" #\newline)
  (let ((f1 (opd1 formule))
        (f2 (opd2 formule)))
    (case connecteur
      ((non) (sequent 'ajouter-formule 'G f1) (list sequent))
      ((et)  (let ((sequent-bis (sequent 'copier-sequent)))
               (sequent        'ajouter-formule 'D f1)
               (sequent-bis    'ajouter-formule 'D f2)
               (list sequent sequent-bis)))
      ((ou)  (sequent 'ajouter-formule 'D f1)
             (sequent 'ajouter-formule 'D f2)
             (list sequent))
      ((imp) (sequent 'ajouter-formule 'G f1)
             (sequent 'ajouter-formule 'D f2)
             (list sequent)))))
```

La fonction `affiche-cond` réalise un affichage si la condition `trace?` vaut `#t` ; elle affiche une chaîne puis un séquent puis éventuellement d'autres chaînes :

```
(define (affiche-cond  trace? texte1  sequent . texte2)
  (when trace?
        (display-alln texte1)
        (sequent 'affiche)
        (apply  display-alln texte2)))
```

Pour prouver un séquent dont on donne la liste des formules de gauche et celles de droite, il est commode d'introduire la fonction :

```
(define (demontrer-sequent? LformuleG LformuleD trace?)
  (let ((LatomG (filtre formule-atomique? LformuleG))
        (LatomD (filtre formule-atomique? LformuleD))
        (Lsigne&formule
          (append (map (lambda (f)
                         (cons-signe&formule 'G f))
                       (filtre formule-compose? LformuleG))
                  (map (lambda (f)
                         (cons-signe&formule 'D f))
                       (filtre formule-compose? LformuleD)))))
    (Demontrer-Lsequents?
      (list (make-sequent LatomG LatomD Lsigne&formule))
      trace?)))
```

La fonction `filtre` a été définie au chapitre 3 §4.
En général, on veut démonter une formule φ, aussi on se ramène, grâce au théorème de complétude, à la preuve du séquent $\vdash \varphi$.

```
(define (demontrer-formule? formule trace?)
  (Demontrer-Lsequents?
    (list (make-sequent  '()  '()
                         (list (cons-signe&formule 'D formule))))
    trace?))
```

Enfin, on rappelle les fonctions utilisées pour manipuler les formules propositionnelles préfixées :

```
(define connecteur  car)

(define opd1 cadr)

(define (opd2 formule)
  (if (null? (cddr formule))
    #f
    (caddr formule)))

(define (formule-compose? s)
  (and (pair? s)(memq (connecteur s) '(et ou non imp))))
```

Quelques tests

Appliquons notre système à la preuve, en mode trace, de la formule :
$(a \vee b) \wedge (\neg c \Rightarrow \neg b) \wedge (a \Rightarrow c) \Rightarrow c$

```
? (demontrer-formule?
     '(imp (et (et (ou a b)(imp (non c)(non b)))(imp a c)) c) #t)
```

```
on traite le sequent :
|- (imp (et (et (ou a b) (imp (non c) (non b))) (imp a c)) c)
regle du imp  a droite

on traite le sequent :
(et (et (ou a b) (imp (non c) (non b))) (imp a c))   |- c
regle du et  a gauche

on traite le sequent :
(et (ou a b) (imp (non c) (non b)))   (imp a c)   |- c
regle du et  a gauche

on traite le sequent :
(imp a c)   (ou a b)   (imp (non c) (non b))   |- c
regle du imp  a gauche

on traite le sequent :
(ou a b)   (imp (non c) (non b))   |- c   a
regle du ou  a gauche

on traite le sequent :
a   (imp (non c) (non b))   |- c   a
c'est un axiome :

on traite le sequent :
b   (imp (non c) (non b))   |- c   a
regle du imp  a gauche

on traite le sequent :
b   |- c   a   (non c)
regle du non  a droite

on traite le sequent :
b   c   |- c   a
c'est un axiome :

on traite le sequent :
b   (non b)   |- c   a
regle du non  a gauche

on traite le sequent :
b   |- c   a   b
c'est un axiome :

on traite le sequent :
c   (ou a b)   (imp (non c) (non b))   |- c
c'est un axiome :

tout est demontre    #t
```

Essayons de prouver la formule $a \vee b \Rightarrow a \wedge b$ en mode trace :

```
? (demontrer-formule-propo? '(imp (ou a b)(et a b)) #t)

on traite le sequent :
|- (imp (ou a b) (et a b))
regle du imp  a droite

on traite le sequent :
(ou a b)   |- (et a b)
regle du et  a droite

on traite le sequent :
(ou a b)   |- a
regle du ou  a gauche

on traite le sequent :
a   |- a
c'est un axiome :

on traite le sequent :
b   |- a
le sequent :
b   |- a     est irreductible
#f
```

La preuve du séquent $a \Rightarrow b \vee c\ ,\ \neg c \Rightarrow a \wedge d\ ,\ d \vee c \vee \neg b\ \vdash\ c$ est donnée par :

```
? (demontrer-sequent?
  '((imp a (ou b c)) (imp (non c)(et a d)) (imp d (et c (non b)))) '(c) #f)
#t
```

On peut aussi donner une présentation de la logique intuitionniste en termes de calcul des séquents (voir références). Cela permet d'en déduire une méthode effective de preuve alors qu'on ne dispose pas de la technique des tables de vérité pour la logique intuitionniste.

Structure du système de preuve des séquents

`(make-sequent LatomG LatomD Lsigne&formule)`
Un séquent est représenté par un prototype construit par cette fonction.
Elle prend en argument les formules qui composent le séquent.

> `(cons-signe&formule signe formule)`
> Construit une formule signée dont on donne le signe (G ou D) et la formule.
>
> `(partie-signe formule-signee)`
> Le signe d'une formule signée.
>
> `(partie-formule formule-signee)`
> La partie formule d'une formule signée.

`formule-composee? formule)`
Pour distinguer les formules composées des variables.

`(axiome? LatomeG LatomeD)`
Pour tester si les deux listes ont une intersection non vide.

`(remove-nth No L)`
Pour supprimer l'élément d'indice `No` de la liste `L`.

`(demontrer-formule? formule trace?)`
Pour faire la preuve d'une formule propositionnelle, avec ou sans affichage des règles utilisées.

`(Demontrer-sequents? LformuleG LformuleD trace?)`
Même chose pour la preuve d'un séquent dont on donne les deux listes de formules.

`(filtre p? L)`
Rend la liste des éléments de la liste `L` qui satisfont au prédicat `p?`.

`(Demontrer-Lsequents? Lsequents trace?)`
Pour prouver une liste de séquents

`(traiter-sequent sequent trace?)`
On transforme un séquent en une liste d'un ou deux nouveaux séquents par application d'une règle.

`(affiche-cond trace? texte1 sequent . texte2)`
Réalise l'affichage des textes et du séquent si `trace?` est vrai.

`(choix-formule Lsigne&formule)`
Choix d'une formule signée prioritaire.

`(priorite signe&formule)`
Coefficient de priorité d'une formule signée pour une certaine stratégie.

`(appliquer-regleD sequent connecteur formule trace?)`
On applique au séquent la règle droite associée au connecteur.

`(appliquer-regleG sequent connecteur formule trace?)`
On applique au séquent la règle gauche associée au connecteur.

`(connect formule)`
Le connecteur d'une formule composée.

`(opd1 formule)`
Le premier opérande d'une formule composée.

`(opd2 formule)`
Le deuxième opérande d'une formule composée.

16.5 De la lecture

Pour tout ce chapitre on recommande le livre [Lal90].
Le système MIU est extrait de [Hof85].
Pour la déduction naturelle la thèse originelle de Gentzen est très lisible [Sza70].
Le livre de Gallier [Gal86] présente en détail différents calculs des séquents.
Le vieux traité de Kleene [Kle62] reste une référence de base pour ce chapitre.

Chapitre 17

Filtrage et réécriture

DANS ce chapitre on étudie diverses notions de filtrage. Le but du filtrage est d'indiquer si un objet donné est un cas particulier d'un modèle plus général. Une expression régulière sert aussi à représenter une classe de mots, mais dans le filtrage on utilise en plus des variables pour noter les liaisons à faire. On étudie successivement le filtrage de listes puis des termes. On en profite pour détailler la notion de terme ; c'est une notion souvent utilisée pour représenter la syntaxe abstraite des objets structurés. On termine par une introduction à la réécriture.

17.1 Filtrage de listes plates

Notion de motif

Consultons l'horaire des avions d'une compagnie aérienne. Pour chaque vol, on suppose qu'il comporte une ligne indiquant :

```
Numéro-vol   ville-départ   heure-départ   ville-arrivée  heure-arrivée
```

Si l'on cherche dans cet horaire les vols de NICE à PARIS, on «filtre» chaque ligne avec le motif suivant :

```
?No-vol          NICE            ?heure-depart          PARIS       ?heure-arrivee
```

Contrairement aux lignes de l'horaire, ce motif comporte des identificateurs préfixés par un point d'interrogation, ce sont des *variables de filtrage.* La consultation de l'horaire consiste à chercher des valeurs pour ces variables de façon à ce que le motif soit égal à une ligne de l'horaire. On trouve que les lignes suivantes sont à retenir :

```
403  NICE   8H50  PARIS    10H10
427  NICE  18H40  PARIS    20H
```

Mais une ligne comme :

```
450  NICE   15H    LONDRES   17H45
```

ne nous concerne pas car elle ne correspond pas à notre motif.
La présence de variables peut aussi permettre d'exprimer des relations d'égalité entre éléments de la liste. C'est le cas avec la répétition de la variable ?L dans le motif suivant :

```
(?X programme avec  ?L car le langage ?L est ?Y)
```

Ce motif peut représenter la liste :

```
(Alain programme avec  Scheme car le langage Scheme est elegant)
```

en convenant que `?X = Alain` et `?L = Scheme`. Mais ce motif ne peut pas représenter la liste :

```
(Alain programme avec  Scheme car le langage Lisp1.5 est obsolete)
```

car une variable doit représenter partout la même valeur.
Commençons avec une définition assez restrictive des motifs. On appelle *motif* une liste *plate* de symboles ou de nombres, les symboles préfixés par un point d'interrogation sont les *variables de filtrage*. Une *donnée* sera une liste *plate* sans variable.
Le but du filtrage est de calculer les valeurs à donner aux variables de filtrage pour qu'un motif soit égal à une donnée ou bien indiquer un échec. La liste des paires de `(variable . sa valeur)` s'appelle comme d'habitude un environnement.

Une fonction de filtrage de listes

On se donne un motif sous forme d'une liste `Lmotif` et une donnée sous forme d'une liste `Ldata`. La fonction `filtre-liste` retourne l'environnement qui permet d'égaler `Lmotif` et `Ldata`. La représentation d'une variable par un symbole commençant par `?` est purement conventionnelle, aussi ajoute-on le paramètre `var?`, c'est un prédicat pour distinguer les variables.

L'essentiel du filtrage est effectué par la fonction locale `filtre-aux` qui parcourt `Lmotif` en discutant selon la nature de ses éléments. Si l'on a une variable, la suite du travail est déléguée à la fonction `filtre-var`. Cette fonction regarde si l'on a déjà attribué une valeur à la variable. Si oui, il faut s'assurer de la cohérence, sinon on ajoute une nouvelle liaison, puis on traite le reste de la liste. Les cas d'échec sont :
- la présence de deux symboles distincts aux mêmes places dans `Lmotif` et `Ldata`,
- le besoin de donner des valeurs distinctes à une même variable,
- la non-égalité des longueurs des listes `Lmotif` et `Ldata`.
On représente l'environnement de filtrage par une a-liste :

```
(define (filtre-Liste Lmotif Ldata var?)
  (letrec ((filtre-aux
            (lambda (Lmotif Ldata env)
              (cond ((null? Lmotif)
                     (if (null? Ldata) env #f))
                    ((null? Ldata) #f)
                    ((var? (car Lmotif))
                     (filtre-var (car Lmotif) (car Ldata)
                                 (cdr Lmotif) (cdr Ldata) env))
                    (else
                     (if (eqv? (car Lmotif) (car Ldata))
                       (filtre-aux (cdr Lmotif)(cdr Ldata) env)
                       #f)))))

          (filtre-var
            (lambda (var data Lmotif Ldata env)
              (let ((valeur (lire-env var env)))
                (if valeur
                  (if (eqv? valeur  data)
                    (filtre-aux Lmotif Ldata env)
                    #f)
                  (filtre-aux Lmotif Ldata
                              (ajouter-env var data env))))))

          (lire-env
            (lambda (var env)
              (let ((pp (assq var env)))
                (if pp
                  (cdr pp)
                  #f))))

          (ajouter-env
            (lambda (var valeur env)
              (cons (cons var valeur) env))))

    (filtre-aux Lmotif Ldata '())))
```

Avec notre convention précédente pour les variables de filtrage, le prédicat `var?` s'écrit :

```
(define (var? x)
  (and (symbol? x)
       (char=? #\? (string-ref  (symbol->string x) 0))))
```

On teste notre filtre :

```
? (filtre-Liste '(?X programme avec  ?L car le langage ?L est ?Y)
      '(Alain programme avec  Scheme car le langage Scheme est elegant)
      var?)
((?y . elegant) (?L . scheme) (?x . alain))
```

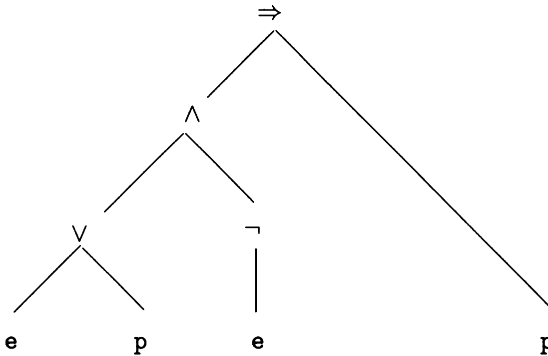
```
? (filtre-Liste '(?X programme avec  ?L car le langage ?L est ?Y)
      '(Alain programme avec  Scheme car le langage Lisp1.5 est obsolete)
      var?)
#f
```

On trouvera dans les références diverses extensions et applications de ce type de filtrage. On va étudier dans la suite une extension aux expressions structurées que l'on rencontre dans des domaines comme la transformation de programmes, la logique, le calcul formel, ...

17.2 Termes et syntaxe abstraite

Syntaxe abstraite

L'analyse syntaxique nous a conduit à faire la distinction entre l'expression textuelle et la structure interne d'un programme : c'est-à-dire la différence entre syntaxe concrète et syntaxe abstraite. On a la même différence entre la chaîne qui représente la formule $((e \vee p) \wedge (\neg e)) \Rightarrow p$ et l'arbre qui représente sa structure.



C'est un arbre étiqueté par des opérateurs, le nombre de fils d'un nœud est donné par l'arité du connecteur.
On a déjà rencontré cette notion de syntaxe abstraite à propos de la structure des expressions et instructions des langages de programmation. Elle intervient aussi dans la description de formules mathématiques et plus généralement dans tout domaine où l'on manipule des expressions symboliques structurées. On va en donner une définition générale — et donc un peu abstraite — avec la notion de *terme*.
On se donne :

- un ensemble de symboles fonctionnels : *Fonc*, avec un entier ≥ 0, appelé *arité*, associé à chaque symbole ;
- un ensemble de symboles de variables : *Var*.

Il est important de comprendre qu'il s'agit uniquement de se donner un cadre pour faire des manipulations symboliques. On parle de *symbole* de fonction et non de fonction car c'est une notion *purement syntaxique*, on ne se donne aucune fonction mais seulement des noms. L'arité modélise l'idée du nombre d'arguments d'une fonction. Un symbole de fonction d'arité 0 s'appelle une *constante.*

Les expressions bien formées que l'on peut construire s'appellent des termes, de façon précise on a la définition suivante :

Définition 1 Un terme t est soit une variable soit un symbole de fonction suivi d'un nombre de termes égal à son arité. On note $Termes(Fonc, Var)$ l'ensemble des termes ainsi construit :

$t \in Termes(Fonc, Var)$ si et seulement si $t \in Var$ ou bien $t = ft_1 \dots t_n$ où $f \in Fonc$ et $n =$ arité de f.

Les mots, sur l'alphabet $Var \cup Fonc$, qui représentent un terme peuvent être assimilés à des expressions préfixées. Voici deux exemples.

Exemple 1 La syntaxe abstraite du calcul propositionnel est représentée par les termes construits sur l'ensemble des symboles fonctionnels constitués par les connecteurs logiques avec leurs arités. La formule logique précédente correspond au terme $\Rightarrow \wedge \vee e\, p \neg e\, p$.

Cette écriture préfixée n'est pas toujours la plus lisible ! aussi, emploi-t-0n d'autres écritures. La notation préfixée et parenthésée à la Lisp permet de rendre visibles les arités des symboles de fonctions, le terme précédent s'écrit alors :

$(\Rightarrow\ (\wedge\ (\vee\ e\ p)\ (\neg\ e))\ p)$.

Exemple 2 Si l'on déclare que les symboles `instruction-while`, `instruction-affectation`, `op-inferieur`, `op-addition` sont d'arité 2 et que les nombres sont des constantes alors l'instruction :

`while x < 10 do x := x + 1` a pour syntaxe abstraite le terme :

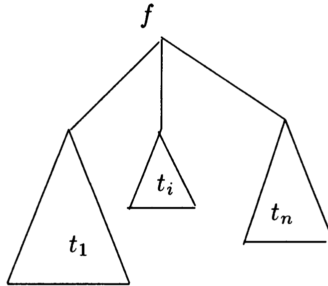
```
(instruction-while (inferieur  x 10)
                   (instruction-affectation  x (op-addition x 1)))
```

On peut aussi utiliser la notation mathématique avec des parenthèses et des virgules pour représenter un terme. Ainsi, le terme $ft_1 \dots t_n$ s'écrit encore $f(t_1, \dots, t_n)$ les $t_1, \dots, t_n$ s'appellent les fils du terme.

On peut associer de façon bijective un arbre étiqueté à un terme :

- à une variable x correspond l'arbre réduit à une feuille étiquetée par x,

- à un terme $t = ft_1 \dots t_n$ correspond l'arbre de racine étiquetée par f et dont les n fils sont les arbres associés aux t_i :



On note que les feuilles d'un tel arbre sont toujours des variables ou des constantes. Cette construction suit la définition récursive d'un terme ; la plupart des notions associées à un terme seront définies récursivement de cette manière.

C'est le cas de l'*ensemble des variables* d'un terme t, on le note ensemble-var(t) :
- si t est une variable alors ensemble-var$(t) = \{t\}$,
- si t est de la forme $ft_1 \ldots t_n$ alors
ensemble-var(t) =ensemble-var$(t_1) \cup \ldots \cup$ ensemble-var(t_n).
Noter que dans le cas d'une constante, cet ensemble est vide.

Un terme est dit *clos* si l'ensemble de ses variables est vide, il est donc construit uniquement avec des symboles de fonctions et de constantes. Il n'existe des termes clos que si l'ensemble des constantes est non vide.

Une représentation des termes en Scheme

Pour concrétiser un peu ces notions, représentons en Scheme la notion de terme. On choisit naturellement la représentation préfixée complètement parenthésée :
- une variable x sera représentée par son symbole `x`,
- si f est un symbole de fonction d'arité 3, g d'arité 2 et h d'arité 1 alors le terme $fxgyhhz$ est représenté par la S-expression `(f x (g y (h (h z))))`,
- si a est un nom de constante, on devrait représenter cette constante par la S-expression `(a)` car il y a 0 argument. Mais il est assez lourd d'écrire les constantes avec des parenthèses, aussi on les représentera par leur symbole. Mais maintenant il y risque de confusion avec la représentation des variables ! Pour l'éviter, on se donne des prédicats `terme:cte?` et `terme:var?` qui renseignent sur la nature d'un symbole.

Un terme qui n'est ni une variable ni une constante est dit composé ; il commence par un symbole de fonction suivi d'une suite de termes que l'on a appelés ses fils.

```
(define terme:compose?  pair?)
(define terme:fct       car)
(define terme:Lfils     cdr)
```

On construit un terme composé en se donnant un symbole de fonction et une liste de termes :

```
(define  (terme:cons  symb-fct Ltermes)
  (cons symb-fct Ltermes))
```

Ecrivons une fonction `terme:Lvar` qui calcule l'ensemble des variables d'un terme donné par sa représentation en Scheme. Pour éviter les répétitions, on utilise la fonction `union` pour ajouter les variables de chaque sous-terme.

```
(define  (terme:Lvar terme terme:var? terme:cte?)
  (cond ((terme:var? terme) (list terme))
        ((terme:cte? terme) '())
        (else (apply union (map (lambda (t)
                                  (terme:Lvar t  terme:var? terme:cte?))
                                (terme:Lfils terme))))))
```

Si l'on désire tester cette fonction, il faut se donner les prédicats `terme:var?` et `terme:cte?`; utilisons pour le test les définitions suivantes:

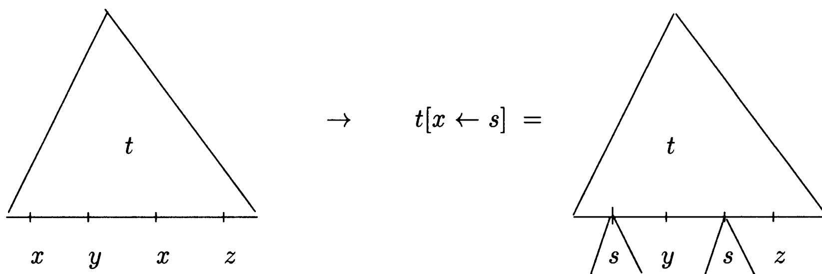
```
(define (terme:cte? s)
  (or (memq s '(a b c d))(number? s)))

(define  (terme:var? s)
  (and (symbol? s)(not (terme:cte? s))))

? (terme:Lvar  '(f x (g  a  (h (h x)))  y) terme:var? terme:cte) -> (x y)
```

17.3 Substitutions dans les termes

On veut définir sur les termes l'opération syntaxique très naturelle de changement de variable ou substitution. Il s'agit de remplacer dans un terme — disons t — toutes les occurrences d'une variable- – disons x — par un terme donné — disons s —. On note $t\,[x \leftarrow s]$ le résultat de cette substitution.

Définitions

Le résultat de la substitution $t\,[x \leftarrow s]$ est un terme défini inductivement par:
- si t est une variable y, alors si $y \neq x$ on ne fait rien et on rend y sinon on rend s,
- si $t = f\,t_1 \ldots t_n$ on effectue (en parallèle) la substitution dans chacun des fils du terme, soit $f\,t_1[x \leftarrow s] \ldots t_n[x \leftarrow s]$.



Par exemple :

```
(f x (g y (h (h x))) z))[x <- (h x)] = (f (h x) (g y (h (h (h x)))) z))
```

Plus généralement, on peut définir la substitution *simultanée* de plusieurs variables $x_1, \ldots, x_k$ par des termes $t_1, \ldots, t_k$; on la note $t[x_1 \leftarrow s_1, \ldots, x_k \leftarrow s_k]$.

Définition 2 On appelle *substitution* une application σ de l'ensemble des variables dans l'ensemble $Termes(Fonc, Var)$ qui ne modifie qu'un nombre fini de variables.

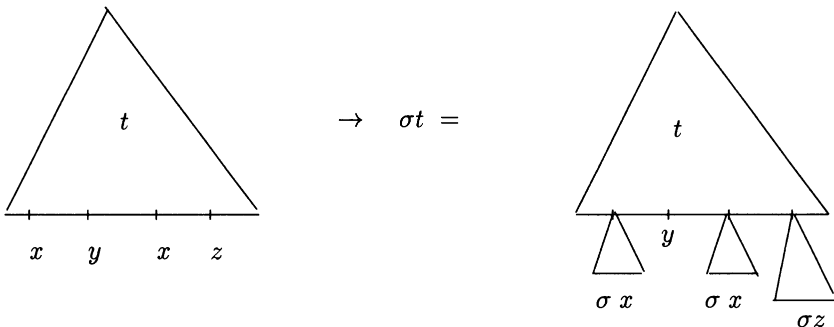
Une substitution est donc définie par la donnée d'un nombre fini de liaisons : $x_1 \to t_1, \ldots, x_k \to t_k$.
Le *domaine* d'une substitution σ est noté $Domaine(\sigma)$, c'est l'ensemble des variables qui sont effectivement modifiées par la substitution :

$$Domaine(\sigma) = \{x \in Var | x \neq \sigma(x)\}$$

On peut étendre aux termes l'application d'une substitution. La valeur de σ sur un terme t est un terme noté $\sigma\, t$ défini par :
- si t est une variable alors on rend $\sigma(x)$,
- sinon t est de la forme $f t_1 \ldots t_n$ et l'on rend le terme $f \sigma t_1 \ldots \sigma t_n$
(autrement dit, on rentre la substitution à l'intérieur du terme car une substitution ne s'applique qu'à des symboles de variables).

Par exemple, en prenant la substitution $\sigma = \{x \;\to\; (h\ x)\;,\; z \to (g\ x\ a)\}$, on trouve :

$$\sigma(f\ x\ (g\ y\ (h\ (h\ x)))\ z)) = (f\ (h\ x)\ (g\ y\ (h\ (h\ (h\ x))))\ (g\ x\ a)))$$



Un terme de la forme $\sigma\ t$ pour une substitution σ s'appelle une *instance* du terme t, car, en remplaçant des variables de t par des termes, on a particularisé le terme. Une substitution σ se prolonge donc en une application de $Termes(Fonc, Var)$ dans lui même, ce qui permet de *composer* les substitutions. La composée $\sigma_2 \circ \sigma_1$ des substitutions σ_2 et σ_1 est l'application définie par : $t \to \sigma_2\ (\sigma_1\ (t))$.

Implantation des substitutions

Pour remplacer dans un terme, une variable var par un terme s, on distingue trois cas selon que le terme est une variable, une constante ou un terme composé. Les deux premiers cas sont immédiats et le dernier consiste à substituer dans chacun des sous-termes.

```
(define (terme:substituerVar terme var s terme:var? terme:cte?)
  (cond ((terme:var? terme)
         (if (eq? terme var) s terme))
        ((terme:cte? terme) terme)
        (else (terme:cons
               (terme:fct terme)
               (map (lambda (t)
                      (terme:substituerVar t var s terme:var? terme:cte?))
                    (terme:Lfils terme))))))

? (terme:substituerVar '(f x (g y (h (h x))) z) 'x '(h x)
                       terme:var? terme:cte?)
(f (h x) (g y (h (h (h x)))) z)
```

Pour étendre au cas du remplacement simultané de plusieurs variables, on choisit de représenter une substitution par une a-liste ((var1 . terme1)...). La valeur d'une substitution sur une variable revient à consulter cette a-liste :

```
(define (valeur-subst sigma var)
  (let ((var.valeur (assq var sigma)))
    (if var.valeur
      (cdr var.valeur)
      var)))

? (valeur-subst '((x . (h x)) (y . y)(z . (g x a)))   'z) -> (g x a)
? (valeur-subst '((x . (h x)) (y . y)(z . (g x a))) 'u)   -> u
```

Le domaine d'une substitution est la liste des variables modifiées :

```
(define (domaine-subst sigma)
  (append-map (lambda (x.v)
                (let ((x (car x.v)))
                  (if (eq? x (valeur-subst sigma x))
                    '()
                    (list x))))
              sigma))


? (domaine-subst '((x . (h x)) (y . y)(z . (g x a)))) -> (x z)
```

On a expliqué comment l'application d'une substitution sigma s'étend aux termes

```
(define (terme:substituer terme sigma  terme:var? terme:cte?)
  ; applique la substitution sigma au terme
  (cond ((terme:var? terme)
```

```
         (let((valeur (valeur-subst sigma terme)))
            (if valeur
               valeur
               terme)))
        ((terme:cte? terme) terme)
        (else (terme:cons
                (terme:fct terme)
                (map (lambda (t)
                        (terme:substituer t sigma terme:var? terme:cte?))
                     (terme:Lfils terme))))))

? (terme:substituer  '(f x (g  y  (h (h x)))  z)
                     '((x . (h y))(z . (g x a)))
                     terme:var? terme:cte?)
(f (h y) (g y (h (h (h y)))) (g x a))
```

La composée de deux substitutions σ_1 et σ_2 ne peut modifier que des variables appartenant au domaine de l'une ou de l'autre. En parcourant l'union de ces domaines, on construit facilement une a-liste représentant la substitution composée $\sigma_2 \circ \sigma_1$.

```
(define (compose-subst  sigma1 sigma2 terme:var? terme:cte?)
  (map  (lambda (var)
          (cons var
                (terme:substituer (valeur-subst  sigma1 var)
                                  sigma2
                                  terme:var? terme:cte?)))
        (union (domaine-subst sigma1)(domaine-subst sigma2))))

? (compose-subst  '((x . (h b)) (y . (f x y z)))
                  '((x . (h x))(z . (g x a)))
                  terme:var? terme:cte?)
((z . (g x a)) (x . (h b)) (y . (f (h x) y (g x a))))
```

Egalité de deux termes

On termine ce paragraphe par un résultat un peu technique[1] sur l'égalité des termes. Est-ce que l'égalité des termes en tant que suite de symboles entraîne l'égalité structurelle? La réponse est oui, c'est le théorème suivant :

Théorème 1 Si deux termes $s = fs_1 \ldots s_m$ et $t = gt_1 \ldots t_n$ sont égaux en tant que mot sur l'alphabet $F \cup Var$, alors on a les égalités $f = g$ et $n = m$ et $s_1 = t_1, \ldots, s_m = t_m$.

Pour le prouver on commence par établir le

Lemme 1 Soit t un terme et u un mot sur l'alphabet $F \cup Var$, si le mot tu est un terme alors u est le mot vide.

[1] Le lecteur peut admettre ce résultat.

Preuve par récurrence sur la longueur $|t|$ du mot t.
Si $|t| = 1$ alors t est une variable ou une constante donc $t\,u$ ne peut être un terme que si u est vide.
Si $|t| > 1$ alors t est de la forme $t = ft_1 \ldots t_n$ et comme $t\,u$ est un terme, il est aussi de la forme $hs_1 \ldots s_k$, d'où l'égalité des mots $ft_1 \ldots t_n\; u = hs_1 \ldots s_k$. Donc $f = h$ et $n = k$. Si $t_i = s_i$ pour tout i c'est fini. Sinon, on va montrer que l'on obtient une contradiction. En effet, il existe alors un plus petit indice i tel que $t_i \neq s_i$ et, après simplification des préfixes communs, on en déduit l'égalité des mots $t_i \ldots t_n\; u = s_i \ldots s_k$. De cette égalité, on déduit $|t_i| \neq |s_i|$ car $t_i \neq s_i$. Disons que $|t_i| < |s_i|$, alors t_i est un préfixe de s_i et on peut écrire $s_i = t_i\, w$, alors l'hypothèse de récurrence implique que w est le mot vide ce qui est contradictoire avec $t_i \neq s_i$ CQFD.

Pour terminer la démonstration du théorème, revenons à l'égalité des mots : $fs_1 \ldots s_m$ et $gt_1 \ldots t_n$. La comparaison du premier symbole de chaque mot montre que $f = g$ et $m = n$. Si l'on n'avait pas $t_i = s_i$ pour tout i, alors on procède comme dans la preuve du lemme ; on se ramène à une égalité $s_i \ldots s_m = t_i \ldots t_m$ avec $t_i \neq s_i$, d'où l'on tire une contradiction grâce au lemme.

17.4 Filtrage de termes

Principe du filtrage de termes

Le filtrage est une opération très courante, elle est utilisée chaque fois que l'on applique un schéma général dans un cas particulier. Voici la définition :

Définition 3 Le *filtrage* est l'opération qui consiste à vérifier si un terme t est instance d'un autre terme s. Une substitution σ telle que t $= \sigma$s s'appelle un *filtre*.

D'où le problème du filtrage : étant donné deux termes s et t, trouver une substitution σ telle que $t = \sigma\; s$. Autrement dit, il s'agit de trouver les valeurs à donner aux variables de s pour que ce terme devienne égal à t.

Avant d'en étudier la théorie voici quelques exemples.

• Le mécanisme de `define-syntax` est basé sur ce type de filtrage.
Par exemple, pour appliquer la règle de définition de la macro `ifn` (cf. chapitre 9§7)

```
(ifn test e1 e2) --> (if test  e2 e1)
```

il faut trouver un terme Scheme qui soit filtré par le terme `(ifn test e1 e2)` où `test, e1, e2` jouent le rôle de variables. Ainsi, le terme
`(ifn (zero? x) (+ 1 x) (* 4 x))` est filtré si l'on considère la substitution : `test -> (zero? x) , e1 -> (+ 1 x) et e2 -> (* 4 x)` qui permet de le transformer en `(if (zero? x) (* 4 x) (+ 1 x))`.

• L'application d'une règle logique comme le modus ponens $\frac{A\;,\;A \Rightarrow B}{B}$ aux formules $X \Rightarrow (X \Rightarrow X)$ et $(X \Rightarrow (X \Rightarrow X)) \Rightarrow (X \Rightarrow X)$ pour en déduire $(X \Rightarrow X)$, consiste à instancier cette règle en prenant pour A la première formule et pour $A \Rightarrow B$ la seconde.

• Quand on simplifie la formule trigonométrique $sin^2(a+b) + cos^2(a+b)$ en 1, on utilise implicitement l'égalité plus générale : $sin^2(x) + cos^2(x) = 1$ car notre formule est une instance de cette dernière avec $x = a + b$.

Ces exemples montrent que le filtrage des termes est une opération tellement omniprésente qu'on l'applique parfois sans en être conscient.

Revenons à la théorie avec un premier résultat sur l'unicité :

Théorème 2 S'il existe un filtre σ tel que $t = \sigma\ s$, alors il est défini de façon *unique* sur l'ensemble des variables du terme s.

Preuve
En effet, soient deux substitutions σ_1 et σ_2 telles que $\sigma_1\ s = \sigma_2\ s$. Montrons par récurrence sur la longueur du terme s que $\sigma_1\ x = \sigma_2\ x$ pour les x qui sont des variables de s.
• Si $|s| = 1$ alors :
- soit s est une variable et on a bien $\sigma_1\ x = \sigma_2\ x$,
- soit s est une constante et on a aussi $\sigma_1\ x = \sigma_2\ x$ pour $x \in$ ensemble:var(s) car cet ensemble est vide.
• Si $|s| > 1$ alors $s = gs_1 \ldots s_m$ et l'égalité des termes $\sigma_1\ gs_1 \ldots s_m = \sigma_2\ gs_1 \ldots s_m$ implique $g\ \sigma_1 s_1 \ldots \sigma_1 s_m = g\ \sigma_2 s_1 \ldots \sigma_2 s_m$ et le théorème sur l'égalité implique $\sigma_1\ s_i = \sigma_2\ s_i$ pour tout i. Mais alors, l'hypothèse de récurrence montre que $\sigma_1 = \sigma_2$ sur ensemble:var(s_i) pour tout i. Donc ces substitutions coïncident sur la réunion de ces ensembles qui est l'ensemble des variables de s. CQFD.

Etudions maintenant l'*existence* d'un filtre σ tel que $t = \sigma\ s$. En général, il n'existe pas de filtre. Par exemple, c'est le cas si les termes s et t commencent par des symboles de fonction distincts. Un deuxième type d'obstruction se présente quand on doit donner des valeurs distinctes à la même variable.

Pour chercher s'il existe un tel filtre σ, on discute selon la nature du terme s :
• si s est une variable x, alors il suffit de prendre la substitution $x \to t$,
• si s est une constante c alors si $t \neq s$ il y échec et sinon toute substitution convient
• si $s = gs_1 \ldots s_m$ alors il y a échec si t n'est pas de la forme $t = gt_1 \ldots t_m$. S'il l'est, on est ramené à trouver un filtre commun à une suite de termes car on doit avoir :

$$\sigma\ s_i = t_i \quad \text{pour } i = 1 \ldots k.$$

Illustrons cette méthode avec les exemples suivants.

• Trouver un filtre σ tel que :

$$\sigma\ (f\ x\ (g\ y\ z)\ (h\ x)) = (f\ a\ (g\ (h\ x)\ b)\ (h\ a)).$$

En égalant les sous-termes respectifs, on est ramené à trouver σ tel que :
$\sigma\ x = a$ et
$\sigma\ (g\ y\ z) = (g\ (h\ x)\ b)$ et
$\sigma\ (h\ x) = (h\ a)$.

La première équation donne la substitution $x \to a$.
La deuxième ajoute les conditions $y \to (h\ x)\ ,\ z \to b$.
La troisième ajoute la condition $x \to a$ qui est cohérente avec la première.
D'où le filtre $\sigma\ :\ x \to a\ ,\ y \to (h\ x)\ ,\ z \to b$.

- Trouver un filtre σ tel que:

$$\sigma\ (f\ x\ (g\ y\ z)(h\ x)) = (f\ (h\ a)(g\ a\ b)(h\ (h\ y))).$$

On est ramené à trouver σ tel que:
$\sigma\ x = (h\ a)$ et
$\sigma\ (g\ y\ z) = (g\ a\ b)$ et
$\sigma\ (h\ x) = (h\ (h\ y))$.
La première équation donne la substitution $x \to (h\ a)$.
La deuxième ajoute les conditions $y \to a\ ,\ z \to b$.
La troisième ajoute la condition $x \to (h\ y)$ qui n'est pas cohérente avec la première.
Attention, ce n'est pas parce que $y \to a$ qu'il faut en déduire que $(h\ y)$ et $(h\ a)$ sont égaux, la substitution n'est appliquée qu'au membre de gauche.

Implantation du filtrage des termes

Pour trouver un filtre commun à une suite de termes $\sigma\ s_i = t_i$ pour $i = 1 \ldots k$. On résout successivement chaque problème de filtrage. Le premier filtrage $\sigma\ s_1 = t_1$ donne une substitution σ_1. Puis, on s'assure que le filtrage suivant $\sigma\ s_2 = t_2$ fournit une substitution σ_2 *cohérente* avec σ_1. On dit que deux substitutions sont cohérentes si elles fournissent les mêmes valeurs pour les variables communes. En réunissant les domaines de deux substitutions cohérentes, on en définit une troisième qui les prolonge toutes les deux.
Si pour chaque filtrage $\sigma\ s_i = t_i$ la substitution trouvée est cohérente avec l'ancienne, on continue avec le prolongement commun. Cette succession de filtrages est réalisée par la fonction `filtre-termes`, elle prend aussi en paramètre la substitution qu'il s'agit de prolonger et elle retourne un prolongement ou `#f` en cas d'échec.
Le filtrage d'un terme est effectué par la fonction `filtre-aux`, elle prend en paramètre une substitution pour pouvoir s'assurer de la cohérence de la solution trouvée. Dans le cas où le `terme1` est une variable on appelle la fonction `filtre-var` qui regarde si cette variable a déjà une valeur. Si oui, on vérifie la cohérence, sinon on ajoute la nouvelle liaison.
Si l'on appelle substitution ce que l'on appelait environnement au §1, on retrouve un algorithme assez proche du filtrage des listes.

```
(define (filtre-terme terme1 terme2 terme:var? terme:cte?)
  (letrec ((filtre-aux
            (lambda (terme1 terme2 sigma)
              (cond ((terme:cte? terme1)
                     (if (eqv? terme1 terme2) sigma #f))
                    ((terme:var? terme1)(filtre-var terme1 terme2 sigma))
                    ((terme:compose? terme2)
                     (if (eqv? (terme:fct terme1) (terme:fct terme2))
                       (filtre-termes (terme:Lfils terme1)
                                      (terme:Lfils terme2) sigma)
```

```
                    #f))
                  (else #f))))

        (filtre-var
          (lambda (var terme2 sigma)
            (let ((var.valeur (assq var sigma)))
              (if var.valeur
                (if (equal? (cdr  var.valeur)  terme2)
                  sigma
                  #f)
                (cons (cons var terme2) sigma)))))

        (filtre-termes
          (lambda (Ltermes1  Ltermes2  sigma)
            (if (null? Ltermes1)
              sigma
              (let ((sigma1 (filtre-aux (car Ltermes1)
                                        (car Ltermes2) sigma)))
                (if sigma1
                  (filtre-termes (cdr Ltermes1)(cdr Ltermes2) sigma1)
                  #f)))))
        )

    (filtre-aux terme1 terme2  '())))
```

On teste les exemples du début :

```
? (filtre-terme '(f x (g y z) (h x))  '(f a (g (h x) b) (h a))
                terme:var? terme:cte?)
((z . b) (y h x) (x . a))


? (filtre-terme '(f x (g y z) (h x))  '(f (h a)  (g a b) (h (h y)))
      terme:var? terme:cte?)
#f
```

Exercice 1 *Ajouter un traitement des cas d'échec, c'est-à-dire un message précisant la cause de l'échec.*

Remarque 1 On dit qu'un terme est *linéaire* s'il ne contient pas plusieurs fois la même variable. Dans ce cas particulier, l'échec pour incohérence disparaît puisque l'on ne rencontre jamais deux fois la même variable. Cela permet de réduire la fonction `filtre-var` à sa dernière ligne. Cette simplification explique pourquoi on fait cette hypothèse quand on veut augmenter l'efficacité du filtrage, par exemple les motifs dans `define-syntax` doivent être linéaires.

17.5 Réécriture de termes

Introduction à la réécriture

On est souvent amené à réduire une expression en utilisant des règles de simplification. Par exemple, l'expression arithmétique `(* a (+ (/ b 2) (* c 0)))` se réduit à `(* a (/ b 2))` en utilisant les égalités `(* y 0) = 0` et `(+ y 0) = y`.

On peut aussi vouloir transformer une expression. Par exemple, on élimine le connecteur $\Rightarrow$ d'une formule propositionnelle en remplaçant systématiquement le terme $(x \Rightarrow y)$ par $(\neg x \vee y)$. On a déjà rencontré un cas de réécriture d'expressions Scheme à l'occasion de `define-syntax`.
On peut aussi évaluer une expression en appliquant des règles de réduction. Détaillons ce dernier cas avec l'addition des entiers naturels.

Arithmétique de Peano

Supposons que nous voulions reconstruire la théorie des entiers à partir de la fonction successeur. On se donne le symbole de fonction S d'arité 1 et le symbole de constante 0. Les entiers seront les *termes* de la forme :
0 , $(S\ 0)$, $(S\ (S\ 0))$, $(S\ (S\ (S\ 0)))\ldots$ qui s'appellent : 0 , 1 , 2 , 3 ...
Ensuite, on définit la fonction d'addition $+$ à partir de la fonction successeur. Une définition récursive possible est donnée (en notation infixe) par

$$x + 0 = x$$
$$x + S(y) = S(x + y).$$

De même pour la multiplication, on pose :

$$x * 0 = 0$$
$$x * S(y) = x + x * y.$$

Ces définitions font partie des axiomes proposés par Peano pour fonder l'arithmétique (voir chapitre 19 §8). On les transforme en règles de réécriture par orientation de la gauche vers la droite, d'où, en revenant à la notation préfixe :

(r_1) $(+\ x\ 0) \rightarrow 0$
(r_2) $(+\ x\ (S\ y)) \rightarrow (S\ (+\ x\ y))$
(r_3) $(*\ x\ 0) \rightarrow 0$
(r_4) $(*\ x\ (S\ y)) \rightarrow (+\ x (*\ x\ y))$

Essayons de simplifier le terme $(+\ (S\ 0)(S\ 0))$. C'est une instance du terme $(+\ x\ (S\ y))$ pour $x = (S\ 0)$ et $y = 0$. Aussi, le simplifie-t-on par la règle r_2 en $(S\ (+\ (S\ 0)\ 0))$. Le sous-terme $(+\ (S\ 0)\ 0)$ est instance du terme $(+\ x\ 0)$ pour $x = (S\ 0)$. Cela permet de le simplifier par r_1 en $(S\ 0)$. Le terme complet se simplifie donc en $(S\ (S\ 0))$. On a bien réduit $(S\ (+\ (S\ 0)\ 0))$ en $(S\ (S\ 0))$. C'est la démonstration de l'égalité bien connue : $1 + 1 = 2$!

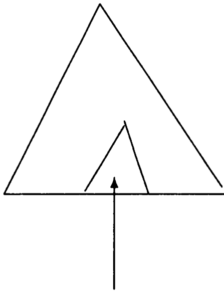
Définition de la réécriture

Ce dernier exemple illustre bien le mécanisme de la simplification ou réécriture d'un terme selon une règle. Voici la définition générale.
Une règle de réécriture s'écrit habituellement $l \to r$ où l et r sont des termes ; l est dit le membre gauche et r celui de droite.

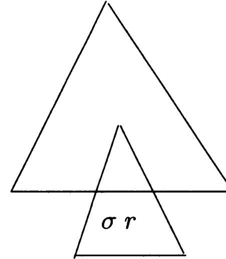
Définition 4 On dit qu'un terme t peut se réécrire par une règle $l \to r$ si un sous terme de t est égal à une instance $\sigma\ l$ du membre gauche l. Alors, t se réécrit en le terme obtenu en remplaçant ce sous-terme par le terme $\sigma\ r$.

Ainsi, la règle $(+\ x\ 0) \to 0$ a permis de réécrire le terme $(S\ (+\ (S\ 0)\ 0))$ car le sous-terme $(+\ (S\ 0)\ 0)$ est instance de la partie gauche $(+\ x\ 0)$.

On peut visualiser cette transformation par le schéma suivant.



Sous-terme de t égal à $\sigma\ l$



Terme t après remplacement du sous-terme par $\sigma\ r$

En pratique, on dispose souvent de plusieurs règles de réécriture. On appelle *système de réécriture* la donnée d'un nombre fini de règles $l_i \to r_i$. On dit qu'un terme t se réécrit par un système de réécriture R si l'une au moins des règles s'applique. On note $t \to_R t'$ une telle réduction. On peut enchaîner les réductions, on écrit $t \to_{R^+} t'$ si l'on passe de t à t' par un nombre fini et non nul de réécritures par R et on écrit $t \to_{R^*} t'$ si l'on passe de t à t' par un nombre fini (éventuellement aucune) de réécritures par R.

Définition 5 Un terme est dit *irréductible* pour R si aucune réduction ne s'applique au terme.

Ce type de réduction pose divers problèmes.

Problème de la terminaison Est-ce que l'on peut réduire indéfiniment un terme?

Considérons le système de deux règles :
$(f\ a\ x) \to (f\ x\ x)$
$(g\ a) \to a$

où f, g, a sont des symboles de fonctions d'arité respectives : 2, 1, 0.

Si l'on part du terme $(f\ (g\ a)\ (g\ a))$, il se réduit par la deuxième règle en $(f\ a\ (g\ a))$ et par la première on revient au terme de départ $(f\ (g\ a)(g\ a))$. Ce bouclage montre que, dans cet exemple, il y a une possibilité de réécrire indéfiniment un terme. Une condition suffisante pour assurer la terminaison de toute suite de réduction est de trouver un *ordre bien fondé* $\succ$ tel que si $t \rightarrow_R t'$ alors $t \succ t'$. (On rappelle qu'un ordre est bien fondé s'il n'existe pas de suite infinie et décroissante pour cet ordre.)

C'est aussi une condition nécessaire car, si on a terminaison, il suffit de considérer la relation de réécriture comme définissant un tel ordre.

Exercice 2 *Démontrer que le système réduit à l'unique règle* $(f\ (f\ x)) \rightarrow (f\ (g\ (f\ x)))$ *possède la propriété de terminaison. (Indication : considérer le nombre de symboles* f *adjacents).*

Problème de la confluence Si un terme t se réduit de deux façons en des termes irréductibles t_1 et t_2, a-t-on nécessairement $t_1 = t_2$?

On a souvent le choix entre plusieurs règles pour réduire un terme, aussi un terme peut avoir plusieurs formes irréductibles différentes. Quand la réécriture est utilisée comme procédé de calcul, on souhaite en général avoir unicité des formes réduites. Il existe des méthodes pour essayer de rendre confluent un système en lui ajoutant des règles convenables. Mais ce type de complétion ne réussit que pour des systèmes assez particuliers. On renvoie à la littérature sur le sujet pour tous ces problèmes.

Implantation de la réécriture

Il est facile de programmer un petit système de réécriture si l'on ne se préoccupe pas des questions d'efficacité. On se donne une liste de règles `Lregles` et on écrit une fonction `reduire` qui ramène un terme à une forme irréductible (si elle existe). Cette fonction utilise trois fonctions locales. La fonction `reecriture-regle` qui essaye d'appliquer une règle donnée à un terme et retourne le terme inchangé si la règle ne s'applique pas. La fonction `reecriture1` qui essaye chaque règle de la liste `Lregle` et dès qu'une règle s'applique on effectue la réduction. Enfin, la fonction `reecriture*` qui itère la fonction `reecriture1` tant que l'on peut réduire le terme.
Une règle est représentée par un doublet `(membre-gauche . membre-droit)` et un système de réécriture est donc représenté par une a-liste.

```
(define  (reduire terme  Lregles terme:var? terme:cte?)
  (letrec ((reecriture-regle
             (lambda (l r terme)
               (if (or (terme:var? terme)(terme:cte? terme))
                 terme
                 (let ((subst (filtre-terme l terme  terme:var? terme:cte?))
                   (if subst
                     (terme:substituer r  subst  terme:var? terme:cte?)
```

```
                    (terme:cons (terme:fct terme)
                                (map (lambda (x) (reecriture-regle l r x))
                                     (terme:Lfils terme)))))))
          (reecriture1
            (lambda (Lregle terme)
              (if (null? Lregle)
                terme
                (let* ((l (caar Lregle))
                       (r (cdar Lregle))
                       (terme1 (reecriture-regle l r terme)))
                  (if (equal? terme1 terme)
                    (reecriture1 (cdr Lregle) terme)
                    terme1)))))

          (reecriture*
            (lambda (terme)
              (let ((terme1 (reecriture1  Lregles terme)))
                (if (equal? terme terme1)
                  terme
                  (reecriture* terme1)))))
          )

    (reecriture* terme)))
```

17.6 Applications de la réécriture

Appliquons cette méthode de réduction à divers systèmes de réécriture.

Les entiers définis à partir de la fonction successeur

On reprend l'exemple des entiers de Peano donné au début de cette section. On définit quatre règles :

```
(define *Peano*
  '(((+ x 0) . x)
    ((+ x (S y)) . (S (+ x y)))
    ((* x 0) . 0)
    ((* x (S y)) . (+ x (* x y))) ))
```

Il faut définir aussi les prédicats pour reconnaître les constantes et les variables :

```
(define (terme:cte? x)
  (eq? x 0))

(define (terme:var? x)
  (and (symbol? x)
       (not (memq x '(+ S)))))
```

Vérifions que $1 + 1 = 2$ et que $2 * 2 = 4$:

```
? (reduire  '(+ (S 0) (S 0)) *Peano*  terme:var?  terme:cte?)
(s (s 0))

? (reduire  '(* (S (S 0)) (S (S 0)))  *Peano*  terme:var? terme:cte?)
(s (s (s (s 0))))
```

Mise sous forme clausale d'une formule propositionnelle préfixée

On a écrit au chap 15 §4 une fonction pour mettre toute formule propositionnelle sous forme clausale. Elle était basée sur un certain nombre de règles de réduction. Considérons ces règles comme des règles de réécriture et essayons de réduire les formules.

```
(define *regle-clausale*
  '(((imp x y) . (ou (non x) y))                      ; élimination de ⇒
    ((equiv x y) . (et (ou (non x) y)(ou x (non y)))) ; élimination de ⇔
    ((non (non x)) . x)                               ; élimination de ¬ ¬
    ((non (ou x y)) . (et (non x) (non y)))           ; rentrer ¬ dans un ∨
    ((non (et x y)) . (ou (non x) (non y)))           ; rentrer ¬ dans un ∧
    ((ou (et x1 x2) y) . (et (ou x1 y) (ou x2 y)))    ; rentrer ∨ dans un ∧
    ((ou x (et y1 y2)) . (et (ou x y1) (ou x y2)))))  ; rentrer ∨ dans un ∧
```

Les variables sont les symboles distincts des connecteurs :

```
(define (terme:var? x)
  (and (symbol? x)
       (not (memq x '(imp ou non et)))))
```

Pour alléger, on n'a pas traité les constantes $\top$ et $\bot$.

```
(define (terme:cte? x)
  #f)

? (reduire '(ou (et x y) z) *regle-clausale* terme:var? terme:cte?)
(et (ou x z) (ou y z))

? (reduire '(imp (et x y) (ou x y)) *regle-clausale* terme:var? terme:cte?)
(ou (ou (non x) (non y)) (ou x y))

? (reduire '(imp (ou x y) (et x y) ) *regle-clausale* terme:var? terme:cte?)
(et (et (ou (non x) x) (ou (non x) y)) (et (ou (non y) x) (ou (non y) y)))
```

Calcul formel

On a abordé le calcul formel avec les polynômes dans les compléments des chapitres 2 et 4. On considère maintenant la simplification et la dérivation d'expressions arithmétiques à deux variables x et y.
La dérivée en x d'une expression aussi simple que $(x+1)(y+1)$ met en jeu de nombreuses règles de simplification :

$$\begin{aligned} D_x(x+1)(y+1) &= (D_x(x+1))(y+1) + (x+1)(D_x(y+1)) \\ &= (1+0)(y+1) + (x+1)(0+0) \end{aligned}$$

$$= 1.(y+1) + (x+1).0$$
$$= (y+1) + 0$$
$$= y+1$$

Pour réaliser ce type de dérivation, on se donne quelques règles de simplification pour l'addition de 0 et la multiplication par 1 :

```
(define *simplification*
  '(((+ u 0) . u)
    ((+ 0 u) . u)
    ((* u 1) . u)
    ((* 1 u) . u)
    ((* u 0) . 0)
    ((* 0 u) . 0)))
```

On note D la dérivation par rapport à la variable x. Voici quelques règles pour dériver la somme ou le produit :

```
(define  *derivivation*
  '(((D (+ u v)) . (+ (D u)(D v)))
    ((D (* u v)) . (+ (* (D u) v)(* u (D v))))
    ((D 0) . 0)
    ((D 1) . 0)
    ((D y) . 0)
    ((D x) . 1)))
```

On réunit ces règles en un système de réécriture :

```
(define *regle-formelle* (append *simplification*  *derivation* ))
```

Les variables de filtrage sont les symboles distincts de +, *, D, x, y. Il est important de remarquer que les variables x et y sont des variables mathématiques mais non des variables de terme, ce sont même des constantes en tant que termes !

```
(define (terme:var? x)
  (and (symbol? x)
       (not (memq x '(x y + * D)))))

(define (terme:cte? s)
  (or (number? s)(eq? s 'x)(eq? s 'y)))
```

On essaye notre système :

```
? (reduire '(D (+ x y)) *regle-formelle* terme:var? terme:cte?)
1

? (reduire '(D (* (+ x 1)(+ 1 y)))  *regle-formelle* terme:var? terme:cte?)
(+ 1 y)

? (reduire '(D (* x x)) *regle-formelle* terme:var? terme:cte?)
(+ x x)
```

Cette approche du calcul formel par de simples règles de réécriture est trop limitée. Par exemple, on ne peut pas exprimer facilement que si `n1` et `n2` sont des nombres alors `(+ n1 n2)` est le nombre `n1+n2`. Il faudrait généraliser notre notion de règle en y incluant des *conditions d'application*. C'est l'approche suivie par le système Mathematica [Wol88] que l'on peut regarder comme étant un énorme système de réécriture conditionnelle.
Par ailleurs, la commutativité d'une opération comme l'addition ne peut être représentée par une règle du type `(+ x y)` $\rightarrow$ `(+ y x)` sous peine d'introduire une possibilité de bouclage dans les réductions. Aussi, est-on amené à définir un ordre total sur les termes pour avoir une forme réduite unique.
Un avantage de la réécriture réside dans son aspect *déclaratif* : on donne les règles en vrac, on peut donc facilement enrichir un système de règles. Cette souplesse est aussi une faiblesse car l'absence de déterminisme dans le choix de la règle à appliquer est une source d'inefficacité. Il y a cependant des techniques pour améliorer l'efficacité, l'une d'elles consiste à *indexer* les règles. Par exemple, au lieu de chercher de façon aveugle la première règle applicable, on se restreint aux règles dont les symboles de fonction apparaissent dans le terme à réduire.

Preuve de propriétés de fonctions

On peut souvent interpréter la définition d'une fonction comme la donnée de règles de simplification pour cette fonction. Par exemple, on a vu que la fonction **append** avec deux arguments peut être définie par :

```
(define (append2 L1 L2)
  (if (null? L1)
    L2
    (cons (car L1)
          (append2 (cdr L1) L2))))
```

Cette définition montre que l'on a les deux règles de simplification :

$$() \, @L_2 \;\rightarrow\; L_2 \qquad (a1)$$

$$(e \;\bullet\; L_1) \;@\; L_2 \;\rightarrow\; e \;\bullet\; (L_1 \;@\; L_2) \qquad (a2)$$

Pour alléger les notations, on a écrit les fonctions **append2** et **cons** en infixe, elles sont respectivement notées @ et $\bullet$.
Utilisons ces règles pour prouver que la fonction **append2** est *associative* :

$$(L_1 \;@\; L_2) \;@\; L_3 \;=\; L_1 \;@\; (L_2 \;@\; L_3) \qquad (assoc)$$

On prouve l'égalité $(assoc)$ par induction sur la liste L_1.

- Cas de base : $L1 = ()$. On remplace, dans les deux membres de $(assoc)$, L_1 par sa valeur :

  $$(() \;@\; L_2) \;@\; L_3 \;=_?\; () \;@\; (L_2 \;@\; L_3)$$

  On simplifie les deux membres avec la règle (a1), chaque membre se réduit à la même expression $(L_2 \;@\; L_3)$. Ils sont donc égaux.

- Induction avec $L_1 = e \bullet L$. En reportant l'expression de L_1 dans les deux membres de $(assoc)$, on est conduit à prouver l'égalité :

$$((e \bullet L) @ L_2) @ L_3 =_? (e \bullet L) @ (L_2 @ L_3) \qquad (*)$$

Le membre de gauche se réduit par deux applications de (a2) en :

$$e \bullet ((L @ L_2) @ L_3)$$

Le membre de droite se réduit par une application de (a2) en :

$$e \bullet (L @ (L_2 @ L_3))$$

L'hypothèse d'induction nous indique que l'égalité $(assoc)$ s'applique avec `L` à la place de `L1` :

$$((L @ L_2) @ L_3) = (L @ (L_2 @ L_3))$$

Par conséquent les deux membres de (*) sont bien égaux, ce qui termine la preuve de $(assoc)$.

Exercice 3 *Utiliser une définition naturelle de la fonction* **reverse** *pour démontrer que l'on a pour toute liste L :*

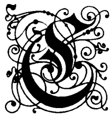
$$(reverse\ (reverse\ L)) = L$$

17.7 De la lecture

On trouvera des compléments sur le filtrage des listes dans [Que90, Nor92].
Pour les aspects théoriques sur les termes et la réécriture on renvoie à [Lal90, DJ90].
On trouvera un grand nombre de preuves combinant induction et réécriture dans [BM79].

Chapitre 18

Des systèmes experts à la programmation logique



E chapitre sert de transition entre la logique propositionnelle et la logique du premier ordre. On commence par étudier la méthode du chaînage arrière pour des systèmes de règles sans variable. Pour augmenter le pouvoir d'expression, on autorise ensuite la présence de variables. On considère d'abord le cas des formules dites datalog qui servent à raisonner sur les bases de données relationnelles. Enfin, on accepte aussi les symboles de fonctions ce qui conduit à la notion de programmation logique et au langage Prolog. La réalisation d'un moteur d'inférence dans le cas général nous conduit à présenter la théorie de l'unification entre les termes. On termine par des exemples qui illustrent certains aspects de la programmation en Prolog.

18.1 Systèmes experts d'ordre 0 et chaînage arrière

On peut souvent représenter les connaissances dans un domaine par des règles logiques de la forme : les propriétés θ_1 et ... et θ_n permettent d'inférer la propriété θ_0. Autrement dit, on se donne des formules $\theta_1 \wedge \ldots \wedge \theta_n \Rightarrow \theta_0$ où les θ_i sont atomiques. Une telle formule s'appelle une *clause de Horn*. C'est une clause, car on peut encore l'écrire $\neg\, \theta_1 \vee \ldots \vee \neg\, \theta_n \vee \theta_0$ et on dit que c'est une clause de Horn car il y a au plus un littéral sans négation.

Systèmes experts

Par exemple, on rencontre ce type de règle si l'on essaye de formaliser le raisonnement qui conduit à un diagnostic (diagnostic d'une maladie à partir d'une série de symptômes, diagnostic de panne de voiture, ...). De façon plus précise, on peut imaginer un système d'identification de plantes à partir de diverses observations.

Il pourra comporter des règles du type (on extrait ces règles d'un exemple de J.L. Laurière [Lau85]) :

SI fleur ET graine ALORS phanérogame

SI phanérogame ET graine-nue ALORS sapin

SI phanérogame ET 1-cotylédone ALORS monocotylédone

SI phanérogame ET 2-cotylédone ALORS dicotylédone

SI monocotylédone ET rhizome ALORS muguet

...

Notre but n'est pas de décrire les méthodes de représentation des connaissances mais d'étudier un mécanisme de déduction avec ce type de système, ce mécanisme s'appelle un *moteur d'inférence.*

On se donne un ensemble fini de *règles* de la forme $\theta_1 \wedge \ldots \wedge \theta_n \Rightarrow \theta_0$ qui représente les déductions habituelles d'un expert du domaine considéré.
On se donne un ensemble de *faits* constatés $\varphi_1 \ldots \varphi_k$. La donnée des règles et des faits s'appelle une *base de connaissance.* Il s'agit de voir si l'on peut en déduire une conjecture ou *but* β.

Un tel système s'appelle pompeusement un *système expert* ; on précise d'ordre 1 ou d'ordre 0 selon que les formules peuvent ou non comporter des variables. Dans ce paragraphe, on suppose qu'il n'y a pas de variable, on est donc dans le cadre du calcul propositionnel. Dans ce cas, on a déjà proposé plusieurs méthodes pour faire des preuves automatiques de formules. Ici, on va profiter de la structure particulière des formules pour obtenir des méthodes de preuve plus efficaces.

Chaînage arrière

Supposons que nous voulions démontrer un but à partir d'un système de règles et d'un ensemble de faits. Il y a deux grandes catégories de méthodes, celles qui procèdent par *chaînage arrière* et celles qui procèdent par *chaînage avant.* On va s'intéresser à la méthode par chaînage arrière car elle se généralisera assez naturellement au cas des formules avec variables.

La démonstration d'un fait β par la méthode du chaînage arrière consiste à regarder si β est un des faits connus alors c'est fini, sinon on sélectionne une règle qui conclut en β et l'on est ramené à prouver comme nouveaux buts les hypothèses de cette règle. S'il n'y a pas de telle règle, le but β n'est pas déductible. S'il y a plusieurs règles applicables, il faut une stratégie pour en choisir une à essayer en premier (par exemple la première !). Si l'on échoue dans la preuve de β, on pourra recommencer en essayant une autre règle applicable et non encore considérée (retour arrière).

Il est instructif d'implanter la méthode du chaînage arrière car elle sera à la base du langage Prolog qui est une généralisation au cas où les θ peuvent être des formules atomiques avec variables. On va étudier diverses implantations selon que l'on cherche une solution, toutes les solutions, ou les solutions au fur et à mesure de la demande. Ces méthodes de recherche rappelleront au lecteur les méthodes de recherche des solutions d'un puzzle décrites au chapitre 3 §10.

Pour implanter ces méthodes, on choisit de représenter les règles par des listes avec la conclusion en tête. On assimile les faits avec les règles sans hypothèses, un fait est donc représenté par une liste réduite à ce fait. On définit des fonctions pour accéder à la conclusion et à la liste des hypothèses d'une règle :

```
(define Conclusion car)

(define Lhypotheses cdr)
```

On définit un prédicat `demontre1?` qui prend en argument un but et une liste de règles. Il rend `#t` si l'on peut prouver le but à partir des règles. On a vu que la preuve d'un fait se ramène en général à celle d'une liste de sous-buts, aussi on utilise une fonction locale `demontreLbuts?` pour traiter ce cas. Quand il n'y a plus de but à prouver, on a réussi, sinon on cherche la première règle qui permet de prouver le premier but et on ajoute ses hypothèses dans la liste des buts à prouver. L'itérateur `some` permet de s'arrêter dès que l'on a trouvé une preuve.

```
(define (demontre1? but Lregles) ; le but est-il  démontrable ?
  (letrec ((demontreLButs?
             (lambda ( Lbuts )
               (if (null? Lbuts)
                 #t
                 (some (lambda (regle)
                         (if (equal? (car Lbuts) (Conclusion regle))
                           (demontreLbuts? (append (Lhypotheses regle)
                                                   (cdr Lbuts)))
                           #f))
                       Lregles)))))
    (demontreLButs?  (list but))))
```

Pour tester, on se donne des règles abstraites :

$d \Leftarrow b \wedge c$
$c \Leftarrow e \wedge f$
$e \Leftarrow b$
$f \Leftarrow e \wedge a$
$c \Leftarrow f \wedge a$
$h \Leftarrow g \wedge c$
$h \Leftarrow d \wedge e$
$g \Leftarrow h \wedge d$
$k \Leftarrow a \wedge b \wedge j$
$j \Leftarrow o$
$o \Leftarrow j$

et deux faits que l'on écrit comme étant des règles sans hypothèse :

$a \Leftarrow$
$b \Leftarrow$

Ce sens inhabituel pour écrire une implication est motivé par notre méthode de recherche par chaînage arrière : on consulte en premier la conclusion d'une règle

pour savoir si on peut l'utiliser. On utilisera ce style d'écriture dans tout ce chapitre.
Ce système est représenté par la liste:

```
(define *Lregles* '( (a) (b)  (d b c) (c e f) (e b) (f e a) (c f a)
                     (h g c) (h d e) (g h d) (k a b j) (j  o)(o j)))
```

On essaye de prouver des buts:

```
? (demontre1? 'd  *Lregles*)  -> #t
? (demontre1? 's  *Lregles*)  -> #f
```

Réponse avec liste des règles utilisées

Pour expliciter le raisonnement utilisé, on va donner une deuxième version qui retourne la liste des règles employées quand il y a succès.
Maintenant, la fonction locale demontreLButs? accumule dans son deuxième paramètre Lregles-utilisees la liste des règles utilisées. La dernière règle utilisée est ajoutée en fin de la liste Lregles-utilisees avec la fonction append1.

```
(define (demontre2? but Lregles) ; -> liste des règles si démontrable
  (letrec ((demontreLButs?
            (lambda (Lbuts Lregles-utilisees)
              (if (null? Lbuts)
                Lregles-utilisees
                (some (lambda (regle)
                        (if (equal? (car Lbuts) (Conclusion regle))
                          (demontreLbuts? (append (Lhypotheses regle)
                                                  (cdr Lbuts))
                                          (append1  Lregles-utilisees regle)
                          #f))
                      Lregles)))))
     (demontreLButs?  (list but) '())))
```

Pour ajouter un élément en fin d'une liste, on utilise la fonction:

```
(define (append1 L s)
  (append L (list s)))
```

Cherchons une preuve du but d:

```
? (demontre2? 'd  *Lregles*)
 -> ((d b c) (b) (c e f) (e b) (b) (f e a) (e b) (b) (a))
```

Qui se lit:
pour prouver d on utilise la règle $d \Leftarrow b \wedge c$,
pour prouver b on utilise le fait $b \Leftarrow$,
pour prouver c on utilise la règle $c \Leftarrow e \wedge f$,
....

Exercice 1 *Ecrire uen fonction qui affiche une preuve de cette façon.*

Notons que, dans certains cas, la preuve d'un fait peut boucler. C'est trivialement le cas avec :

```
? (demontre? 'j *Lregles*) -> boucle
```

car la preuve de j se ramène à celle de o qui à son tour se ramène à celle de j. La preuve de h boucle pour des raisons moins immédiates :

```
? (demontre? 'h  *Lregles*) -> bouclage
```

L'ensemble de toutes les preuves d'un but

En général, il y a plusieurs façons de choisir les règles pour prouver un but. La version suivante permet d'exhiber, à la demande, toutes les preuves. Quand on a prouvé tous les buts, on affiche la solution trouvée puis on demande à l'utilisateur s'il souhaite voir une autre solution. S'il répond oui, on renvoie `#f` en valeur pour forcer la recherche[1] d'une autre solution.

```
(define (demontre3? but Lregles) ; les solutions sur demande
  (letrec ((demontreLButs?
            (lambda (Lbuts Lregles-utilisees)
              (if (null? Lbuts)
                (Voir-solution Lregles-utilisees)
                (some (lambda (regle)
                        (if (equal? (car Lbuts) (Conclusion regle))
                          (demontreLbuts? (append (Lhypotheses regle)
                                                  (cdr Lbuts))
                                          (append1  Lregles-utilisees regle))
                          #f))
                      Lregles)))))
    (demontreLButs?  (list but) '())))

(define (Voir-solution  solution)
  (display-alln (reverse solution))
  (display "autre solution o/n ? : ")
  (if (eq? 'o (read))
    #f
    #t))

? (demontre3? 'd  *Lregles*)
((d b c) (b) (c e f) (e b) (b) (f e a) (e b) (b) (a))
autre solution o/n ? : o
((d b c) (b) (c f a) (f e a) (e b) (b) (a) (a))
autre solution o/n ? : o
#f
```

On peut aussi retourner la liste de toutes les solutions, on place le symbole `fin` pour marquer la séparation entre les solutions. On utilise l'itérateur `append-map` pour concaténer l'ensemble des solutions trouvées.

[1] On a emprunté cette idée au livre de Norvig [Nor92].

```
(define (demontre4? but Lregles) ;  toutes les solutions ou bouclage
  (letrec ((demontreLButs?
            (lambda (Lbuts Lregles-utilisees)
              (if (null? Lbuts)
                (append  Lregles-utilisees '(fin))
                (append-map (lambda (regle)
                              (if (equal? (car Lbuts) (Conclusion regle))
                                (demontreLbuts? (append (Lhypotheses regle)
                                                        (cdr Lbuts))
                                                (append1  Lregles-utilisees
                                                          regle))
                                '()))
                            Lregles)))))
    (demontreLButs?  (list but) '())))

? (demontre4? 'd  *Lregles*)
((d b c) (b) (c e f) (e b) (b) (f e a) (e b) (b) (a) fin
 (d b c) (b) (c f a) (f e a) (e b) (b) (a) (a) fin)
```

Dispositif anti-bouclage

On a constaté que l'on pouvait boucler, c'est évidemment le cas si, pour démontrer un but, on doit en démontrer d'autres dont l'un redemande que l'on prouve l'un des buts que l'on essayait de prouver. Pour parer à cet inconvénient, on va stocker dans un paramètre, noté `LbutsEnCours`, la liste des buts en cours de preuve. On répartit le travail entre deux fonctions locales : `demontreBut?`, `demontreLbuts`. La fonction `demontreBut?` sert à chercher la preuve d'un but. Si on constate que ce but appartient à la liste `LbutsEnCours`, c'est le signe d'un bouclage et l'on rend `#f` pour forcer la recherche d'une autre solution.

Pour prouver une liste de buts, on fait appel à la fonction `demontreLbuts?`. Si l'on réussit à prouver le premier but, on le supprime de la liste `LbutsEnCours` et on rappelle `demontreLbuts?` sur les buts restants.

```
(define (demontre5? but Lregles) ; -> une solution s'il en existe
    (letrec ((demontreBut?
              (lambda (but  Lregles-utilisees  LbutsEnCours)
                (cond
                  ((member but LbutsEnCours)  ;; si bouclage ...
                   (display "bouclage detecte  ") #f)
                  (else
                   (some
                     (lambda (regle)
                       (when (eq? but  (Conclusion regle))
                             (demontreLbuts? (Lhypotheses regle)
                                             (append1 Lregles-utilisees
                                                      regle)
                                             (cons but LbutsEnCours))))
                     Lregles)))))
```

```
          (demontreLbuts?
            (lambda (Lbuts  Lregles-utilisees  LbutsEnCours)
              (if (null? Lbuts)
                Lregles-utilisees ;; cas de succès
                (let ((but (car Lbuts)))
                  (let ((preuveBut (demontreBut? but Lregles-utilisees
                                                     LbutsEnCours)))
                    (when preuveBut
                    ;; si le but est prouvé
                        (demontreLbuts? (cdr Lbuts) preuveBut
                                        (remove but LbutsEnCours))
                        ;; on le supprime des buts en cours
                        )))))))
      (demontreBut? but '() '() )))

? (demontre5? 'd  *Lregles*)
-> ((d b c) (b) (c e f) (e b) (f e a) (a))

? (demontre5? 'j  *Lregles*)      bouclage detecte
->  #f

? (demontre5? 'h  *Lregles*)      bouclage detecte
-> ((h d e) (d b c) (b) (c e f) (e b) (b) (f e a) (e b) (b) (a) (e b) (b))
```

On remarquera que les bouclages pour les buts j et h sont d'espèces différentes. Pour le but h, le bouclage cachait une solution que l'on n'essayait pas alors que pour j, il n'y avait pas de solution.

Exercice 2 *On constate que certaines preuves refont plusieurs fois la démonstration d'un même fait (la preuve de h demande deux fois la preuve de e). Donner une nouvelle version de la fonction* **demontre?** *qui évite cet inconvénient.*

Cette série de moteurs d'inférence ne fait qu'effleurer le sujet, on a besoin dans les applications de beaucoup d'autres extensions. Par exemple, si l'on échoue dans la preuve d'un but, on peut demander à l'utilisateur s'il désire l'ajouter dans la base des faits. Dans certains domaines — comme la médecine —, la conclusion d'une règle n'est pas certaine mais on peut lui affecter un coefficient de vraisemblance. On doit alors combiner l'application des règles avec des techniques probabilistes. Ce modèle a été utilisé dans un système expert très connu, MYCIN (voir [Lau85]), qui est une aide au diagnostic de maladies infectieuses. A titre indicatif, voici une règle typique de ce système :

> SI la coloration de l'organisme est gram-positive
>
> ET la morphologie de l'organisme est coque
>
> ET la conformation de croissance de l'organisme est agglutination
>
> ALORS l'identité de l'organisme est staphylocoque avec coefficient de vraisemblance 0.7

Exercice 3 *On peut aussi attaquer le problème par la méthode dite du* chaînage avant. *Elle consiste à engendrer systématiquement tous les buts démontrables à partir des faits connus puis tous les buts démontrables à partir de l'ensemble obtenu... On continue jusqu'à trouver le but cherché ou constater que le processus stationne. Implanter cette méthode.*

18.2 Systèmes de règles avec variables

Pour écrire des règles ayant une portée générale, il est commode d'introduire des variables. On présente cette extension en commençant avec un exemple tiré de la mythologie grecque.
Considérons une base de données familiales pour certains dieux grecs. On déclare les faits suivants :

mere-de($Gaia, Cronos$) ; cette écriture se lit : $Gaia$ est la mère de $Cronos$
mere-de($Rhea, Zeus$)
mere-de($Rhea, Hades$)
pere-de($Zeus, Pollux$) ; cette écriture se lit : $Zeus$ est le père de $Pollux$
pere-de($Ouranos, Cronos$)
pere-de($Cronos, Zeus$)
pere-de($Zeus, Helene$)
pere-de($Zeus, Castor$)

On a également des règles générales sur les liens de parenté. On dit qu'un père, ou une mère, est un parent de ses enfants. Par exemple, on a parent-de($Leda, Pollux$) car mere-de($Leda, Pollux$). Au lieu d'écrire la relation parent-de dans chaque cas particulier, on en donne une définition générale en utilisant des *variables* x, y :

parent-de(x, y) $\Leftarrow$ pere-de(x, y)
parent-de(x, y) $\Leftarrow$ mere-de(x, y)

Deux personnes ayant un même parent sont frères (on ne fait pas la distinction entre frère et demi-frère). Au lieu de faire la liste de tous les frères on se donne une règle qui définit la relation frère :

$$\text{frere}(y, z) \Leftarrow \text{parent-de}(x, y) \wedge \text{parent-de}(x, z)$$

On peut alors vérifier que frere($Castor, Pollux$) car ils ont pour père $Zeus$.
Introduisons une dernière relation de parenté, un grand-parent est le parent d'un parent :

$$\text{gd-parent-de}(i, k) \Leftarrow \text{parent-de}(i, j) \wedge \text{parent-de}(j, k)$$

Notons que les noms des variables d'une règle sont sans importance, on a utilisé ici les variables i, j, k.
On veut un système permettant de répondre à des questions du genre : est-ce que $Cronos$ est un grand-parent de $Pollux$? On va adapter la méthode du chaînage arrière à ce cas.

Essayons de démontrer que l'on a la relation gd-parent-de($Cronos, Pollux$).

Comme on ne trouve aucun fait de cette nature, on essaye d'appliquer une règle. Une seule règle conclut sur la relation gd-parent-de :

$$\text{gd-parent-de}(i, k) \Leftarrow \text{parent-de}(i, j) \wedge \text{parent-de}(j, k)$$

Pour l'utiliser il faut identifier sa conclusion gd-parent-de(i, k) avec notre but gd-parent-de$(Cronos, Pollux)$. Il s'agit d'un simple problème de filtrage, on doit prendre $i = Cronos$ et $k = Pollux$. Ensuite, il faut prouver les deux nouveaux buts :

$$\text{parent-de}(Cronos, j) \quad \text{et} \quad \text{parent-de}(j, Pollux)$$

On a deux règles qui concluent avec parent-de, essayons la première pour prouver parent-de$(Cronos, j)$, on doit égaler parent-de$(Cronos, j)$ et parent-de(x, y).

On voit apparaître ici un nouveau problème : trouver des valeurs pour les variables j, x, y pour que parent-de$(Cronos, j)$ = parent-de(x, y) ? Ce n'est pas un problème de filtrage car on peut instancier à droite et à gauche, il s'agit d'une opération appelée *unification*. On cherche une substitution σ telle que :

$$\sigma\, \text{parent-de}(Cronos, j) = \sigma\, \text{parent-de}(x, y).$$

Dans ce cas trivial, on a une solution évidente (et non unique) $x = Cronos$, $j = y$. On est donc conduit à prouver pere-de$(Cronos, y)$. On trouve une preuve avec $y = Zeus$. Il reste enfin à prouver parent-de$(j, Pollux)$ avec $j = y = Zeus$, c'est-à-dire parent-de$(Zeus, Pollux)$ ce qui est un fait et termine la démonstration de gd-parent-de$(Cronos, Pollux)$. Noter que l'on a propagé au deuxième but la valeur trouvée pour j en prouvant le premier but.

Exercice 4 *Chercher par cette méthode si l'on a frere$(Zeus, Hades)$ ou si gd-parent-de$(Ouranos, Helene)$?*

De façon générale, on considère des règles de la forme $\theta_0 \Leftarrow \theta_1 \wedge \ldots \wedge \theta_n$ où les θ_i peuvent être des relations entre des variables, c'est à dire de la forme $p(x_1, \ldots, x_n)$ où p est le nom de la relation et les x_j des variables. En revanche, les faits ne comportent pas de variable et sont de la forme $q(a_1, ..., a_m)$ où les a_j sont des constantes. Un tel ensemble de règles forme un système appelé datalog. Cette situation est fréquente dans les bases de données : les faits représentent des tableaux de données et les règles servent à effectuer des déductions à partir de ces données. Par exemple, on peut se donner les tableaux des horaires des avions entre deux villes et se poser une question du genre : comment aller en avion d'une ville V1 à une autre ville V2, quitte à utiliser des correspondances?

Pour définir un moteur d'inférence général, on doit commencer par étudier la théorie de l'unification.

18.3 Unification dans les termes

L'unification est une opération qui intervient dans de nombreux domaines liés aux manipulations symboliques. Aussi, on la traite dans son cadre naturel qui est la théorie des expressions symboliques ou termes. On va exposer le théorème de l'unificateur principal, il a son origine dans les travaux de J. Robinson sur les méthodes de résolution en logique.

Définition de l'unification

L'unification est une opération voisine du filtrage mais elle est symétrique pour les deux termes concernés.

Définition 1 Un *unificateur* de deux termes t_1 et t_2 est une substitution σ telle que $\sigma\, t_1 \;=\; \sigma\, t_2$.

On peut interpréter σ comme étant une solution de l'équation $t_1 \;=_?\; t_2$ car elle fournit des valeurs à donner aux variables des termes pour qu'ils soient syntaxiquement égaux.

Par exemple, considérons l'équation :

$$f(h(a\ ,\ x)\ ,\ g(g(z))) =_? f(y\ ,\ g(x)) \qquad (*)$$

où x, y, z sont des variables, f; , h des symboles de fonction d'arité 2, g est d'arité 1 et a une constante. D'après l'égalité de deux termes, on est ramené à deux équations simultanées (plus simples) :

$$h(a\ ,\ x) =_? y$$
$$g(g(z)) =_? g(x)$$

La première donne la valeur de y en fonction de x, la seconde se décompose à son tour en $g(z) =_? x$, d'où la valeur de x. On a donc trouvé la substitution $\sigma : x \to g(z) \quad , \quad y \to h(a, g(z))$. On vérifie bien que si l'on applique σ aux deux termes membres de l'équation (*), on retrouve le même terme.

Existence, unicité d'un unificateur?

Si, dans la substitution précédente, on remplace la variable z par un terme quelconque on a toujours un unificateur. Plus généralement, si l'on a $\sigma\ t_1 \;=\; \sigma\ t_2$ alors on a encore $\rho\sigma\ t_1 \;=\; \rho\sigma\ t_2$ pour toute substitution ρ. Il n'y a donc pas unicité d'un unificateur quand il existe.

Il n'existe pas toujours un unificateur entre deux termes. Par exemple, cherchons à unifier les termes :

$$f(x\ ,\ h(a\ ,\ x)) \text{ et } f(g(y)\ ,\ h(y\ ,\ g(b)))$$

où a et b sont des constantes. En égalant les sous termes, on doit satisfaire aux deux égalités :

$$x = g(y)$$
$$h(a\ ,\ x) = h(y\ ,\ g(b))$$

La deuxième égalité donne $y = a$ et $x = g(b)$, d'où un échec car on ne peut avoir à la fois $x = g(a)$ et $x = g(b)$.

On a constaté qu'il n'y avait pas unicité d'un unificateur quand il existait, mais on va voir qu'il y en a un «plus général» en un certain sens.

Définition 2 On dit que σ est un *unificateur principal* de t_1 et t_2 si tout autre unificateur τ est de la forme $\tau = \rho\,\sigma$ pour une certaine substitution ρ.

La notion d'unificateur principal permet d'avoir un résultat d'unicité.

Proposition 1 Si deux termes t_1 et t_2 admettent un unificateur principal, alors il est unique à une permutation près sur les noms de ses variables.

En effet, supposons que σ_1 et σ_2 soient unificateurs principaux de t_1 et t_2. Alors, par définition, il existe des substitutions ρ_1 et ρ_2 telles que $\sigma_1 = \rho_1\,\sigma_2$ et $\sigma_2 = \rho_2\,\sigma_1$, d'où l'on tire $\sigma_1 = \rho_1\rho_2\,\sigma_1$ et $\sigma_2 = \rho_2\rho_1\,\sigma_2$, ce qui implique facilement que ρ_1 et ρ_2 sont des bijections réciproques entre les ensembles de variables de σ_1 et σ_2.

Exercice 5 *Unifier les termes :*

$f(x\,,\,h(h(f(a\,,\,u)\,,\,h(v\,,\,u))\,,\,x))$ *et* $f(h(y\,,\,z)\,,\,h(x\,,\,h(f(a\,,\,t)\,,\,h(b\,,\,w))))$

où u, v, w, t *sont aussi des variables.*

Même question pour les termes $f(x\,,\,g(x))$ *et* $f(g(y)\,,\,y)$.

Calcul de l'unificateur principal

Pour étudier l'existence d'un unificateur principal, on commence par discuter le cas très simple où l'un des termes est une variable $x =_? t$. Il y a trois sous-cas :

- Si le terme t est égal à la variable x, alors la substitution identité est unificateur principal.
- Si le terme t contient la variable x alors il ne peut y avoir d'unificateur σ car on aurait à la fois $\sigma\,x = \sigma\,t$ et $|\sigma\,x| < |\sigma\,t|$ puisque $\sigma\,x$ est un sous-terme strict de $\sigma\,t$. La détection de ce cas s'appelle le *test d'occurrence*, il est parfois omis dans certaines implantations pour en augmenter l'efficacité au détriment de la fiabilité.
- Si t ne contient pas d'occurrence de x, alors il est immédiat de vérifier que la substitution $x \to t$ est unificateur principal.

Pour traiter le cas général, l'exemple du début montre qu'il est indispensable de considérer des systèmes. En effet, l'équation $f\ s_1 \ldots s_n =_? f\ t_1 \ldots t_n$ équivaut à trouver une substitution qui satisfait au système de n équations :

$$s_1 =_? t_1, \ldots, s_n =_? t_n.$$

Notons Φ_n un tel système et appelons $Unifier(s, t)$ un unificateur principal de s et t avec, par convention, la valeur `#f` s'il n'existe pas. La recherche d'un unificateur principal pour le système Φ_n est basée sur le théorème suivant

Théorème Soit $\{s_1 =_? t_1, \ldots, s_n =_? t_n\} = \Phi_n$ un système d'équations. Posons $\sigma_1 = Unifier(s_1\,,\,t_1)$ si $\sigma_1 =$ `#f` il y a échec sinon ;
posons $\sigma_2 = Unifier(\sigma_1 s_2\,,\,\sigma_1 t_2)$ si $\sigma_2 =$ `#f` il y a échec sinon ;
posons $\sigma_3 = Unifier(\sigma_2\sigma_1 s_3\,,\,\sigma_2\sigma_1 t_3)$ si $\sigma_3 =$ `#f` il y a échec, sinon
...

posons $\sigma_n = Unifier(\sigma_{n-1} \ldots \sigma_1 s_n \ , \ \sigma_{n-1} \ldots \sigma_1 t_n)$ si $\sigma_n =$ **#f** il y a échec sinon la composée des substitutions $\sigma = \sigma_n \ldots \sigma_1$ est unificateur principal de Φ_n.

Preuve Il est immédiat de vérifier que σ est un unificateur car on a par construction $\sigma_i \ldots \sigma_1 s_i = \sigma_i \ldots \sigma_1 t_i$ pour $i = 1 \ldots n$.

Démontrons par récurrence sur n qu'il est principal.
Si $n = 1$, c'est la définition de $\sigma_1 = \sigma$.
Si $n > 1$, soit τ un unificateur de Φ_n, il s'agit de montrer qu'il existe ρ tel que $\tau = \rho\sigma$.
Comme τ est unificateur de Φ_n, il est en particulier unificateur de Φ_{n-1}, donc, par hypothèse de récurrence, il existe ρ' telle que $\tau = \rho'\sigma_{n-1} \ldots \sigma_1$. Par ailleurs, τ est aussi unificateur de $s_n =_? t_n$ donc $\rho'\sigma_{n-1} \ldots \sigma_1 s_n = \rho'\sigma_{n-1} \ldots \sigma_1 t_n$. Mais, par définition de l'unificateur principal σ_n, il existe ρ" telle que $\rho' = \rho"\sigma_n$, ce qui montre que bien que $\tau = \rho"\sigma$.

Implantation de la fonction `unifier`

Ecrivons une fonction `unifier` qui calcule l'unificateur principal des termes `t1` et `t2` ou rend `#f` en cas d'échec.
La fonction locale `unificateur` calcule l'unificateur principal des termes `t1` et `t2` en considérant quatre cas :

- si `t1` est une variable, on applique la méthode décrite avant le théorème
- si `t2` est une variable, on se ramène au cas précédent en échangeant les termes
- si l'un des termes est une constante, il y a échec s'ils ne sont pas égaux
- si ce sont des termes composés avec le même symbole de tête, on unifie la liste des sous-termes avec la fonction `unifListe` sinon il y a échec.

La fonction `unifListe` unifie une liste de termes en appliquant la méthode du théorème. Comme toutes les fonctions sur les termes, la fonction `unifier` prend aussi en paramètre les prédicats `var?` et `cte?`.

```
(define (unifier t1 t2 var? cte?)
  (letrec (
          (unificateur
            (lambda (t1 t2)
              (cond ((var? t1)
                     (if (eq? t1 t2)
                       '()
                       (if (memq t1 (terme:Lvar t2 var? cte?))
                         #f
                         (cons (cons t1 t2) '()))))
                    ((var? t2)(unificateur t2 t1))
                    ((or (cte? t1)(cte? t2))
                     (if (eq? t1 t2)
                       '()
                       #f))
                    ((eq? (terme:fct t1)(terme:fct t2))
```

```
                     (unifListe (terme:Lfils t1)(terme:Lfils t2) '()))
                    (else #f))))

          (unifListe
            (lambda (Ltermes1 Ltermes2  sigma)
              (if  (null? Ltermes1 )
                sigma
                (let ((sigma1 (unificateur (car Ltermes1)
                                           (car Ltermes2))))
                  (if sigma1
                    (unifListe
                      (map (lambda (terme)
                             (terme:substituer terme sigma1  var? cte?))
                           (cdr Ltermes1))
                      (map (lambda (terme)
                             (terme:substituer terme sigma1  var? cte?))
                           (cdr Ltermes2 ))
                      (compose-subst  sigma sigma1 var? cte?))
                    #f)))))
          )
    (unificateur t1 t2)))
```

Pour tester, on se donne une définition de `cte?` et `var?`:

```
(define (cte? s)
  (memq s '(a b c)))

(define (var? s)
  (and (symbol? s) (not (cte? s))))

? (unifier '(f (h a x) (g (g z))) '(f y (g x)) var? cte?)
((x g z) (y h a (g z)))

? (unifier '(f x (h a x)) '(f (g y) (h y (g b))) var? cte?) -> #f

? (unifier '(f x (g x)) '(f (g y) y) var? cte?) -> #f

? (unifier '(f x y) '(f y z) var? cte?) -> ((x . z) (y . z))
```

Exercice 6 *Prévoir, en cas d'échec, l'affichage d'un message indiquant la raison de l'échec.*

18.4 Implantation d'un moteur d'inférence d'ordre 1

On est maintenant en mesure d'écrire un moteur d'inférence pour un système de règles *avec des variables.* De façon précise, on considère des règles :

$$\theta_0 \Leftarrow \theta_1 \wedge \ldots \wedge \theta_n \quad \text{et des faits} \quad \theta_i$$

où les formules θ sont de la forme $p(t_1, \ldots, t_n)$, où p est un nom de relation et les t_k sont des termes ; une telle formule logique θ est dite *atomique.*

Pour commencer, on suppose que les buts à prouver sont des formules θ sans variable. Une règle est représentée par une liste de telles formules avec la conclusion placée en tête.
Une formule atomique sera représentée de façon préfixe ainsi que les termes qui la composent.
Par exemple, les relations de parenté des dieux grecs seront représentées par la liste de règles :

```
(define *Dieux-Grecs*
  '(((mere-de  Gaia Cronos) )
    ((mere-de  Rhea Zeus))
    ((mere-de  Rhea Hades))
    ((pere-de  Zeus Pollux))
    ((pere-de  Ouranos Cronos) )
    ((pere-de  Cronos  Zeus))
    ((pere-de  Zeus Helene))
    ((pere-de  Zeus  Castor))
    ((parent-de ?x  ?y)   (pere-de ?x ?y))
    ((parent-de ?x  ?y)   (mere-de ?x ?y))
    ((gd-parent-de ?i ?k) (parent-de ?i ?j) (parent-de ?j ?k))
    ((frere ?y  ?z) (parent-de ?x ?y) (parent-de ?x ?z)) ))
```

Pour distinguer les variables des autres symboles, chaque variable commence par un ?. Ce qui conduit à définir les prédicats :

```
(define (var? s)
  (and (symbol? s)
       (char=? #\? (string-ref (symbol->string s) 0))))

(define (cte? s)
  (or (number? s)
      (and (symbol? s)
           (not (var? s)))))
```

Recherche d'une première solution

Commençons par la recherche d'une première démonstration d'un but. Le principe est le même que celui du premier moteur du §1 pour les règles sans variable. On le rappelle :

```
(define (demontre1? but Lregles)
  (letrec ((demontreLButs?
             (lambda ( Lbuts )
               (if (null? Lbuts)
                 #t
                 (some (lambda (regle)
                         (if (equal? (car Lbuts) (Conclusion regle))
                           (demontreLbuts? (append (Lhypotheses regle)
```

```
                                                   (cdr Lbuts)))
                          #f))
                      Lregles)))))
    (demontreLButs? (list but))))
```

La modification consiste essentiellement à remplacer le test d'égalité entre un but et une conclusion de règle par un test d'unification. De plus, on a vu que l'on doit propager aux nouveaux buts les valeurs données aux variables par l'unificateur trouvé. Pour cela, on utilise la fonction `sublis` du chapitre 2 §8 : elle applique la substitution trouvée à la liste des sous-buts. D'où une fonction `Prouve1` qui cherche à démontrer un but à partir d'une liste de règles.

```
(define (Prouve1 but  Lregles var? cte?) ; le but est-il prouvable ?
  (letrec ((ProuveLbuts
            (lambda (Lbuts)
              (if (null? Lbuts)
                #t
                (some (lambda (regle)
                        (let ((sigma1 (unifier (car Lbuts) ;***
                                               (conclusion  regle)
                                               var? cte?)))
                          (if  sigma1
                            (ProuveLbuts
                              (sublis  sigma1 (append (Lhypotheses  regle)
                                                      (cdr Lbuts))))
                            #f)))
                      Lregles)))))
    (ProuveLbuts (list but))))

? (Prouve1 '(gd-parent-de Cronos   Pollux) *Dieux-Grecs* var? cte?) -> #t
? (Prouve1 '(gd-parent-de Ouranos  Helene) *Dieux-Grecs* var? cte?) -> #f
? (Prouve1 '(frere Zeus  Hades)            *Dieux-Grecs* var? cte?) -> #t
? (Prouve1 '(frere Zeus  Rhea)             *Dieux-Grecs* var? cte?) -> #f
```

Ce programme est naturel mais pas très efficace. On peut l'améliorer en limitant la recherche de la règle à utiliser aux règles qui commencent par le même prédicat que le but. Pour cela, on définit une fonction `choix-Lregles` qui sélectionne une sous liste de la liste de toutes les règles en filtrant celles qui ont un symbole de tête donné :

```
(define (choix-Lregles Lregles tete)
  (filtre (lambda (regle) (eq? tete (caar regle))) Lregles))
```

D'où une version améliorée de `Prouve1` :

```
(define (Prouve1bis but  Lregles var? cte?)
  (letrec ((ProuveLbuts
            (lambda (Lbuts)
              (if (null? Lbuts)
                #t
                (some (lambda (regle)
```

```
                    (let ((sigma1 (unifier (car Lbuts)
                                           (conclusion  regle)
                                           var? cte?)))
                      (if  sigma1
                        (ProuveLbuts
                          (sublis sigma1 (append (Lhypotheses  regle)
                                                 (cdr Lbuts))))
                        #f)))
                  (choix-Lregles Lregles (car (car Lbuts)))))))

    (ProuveLbuts (list but))))
```

Buts avec variables

La présence des variables dans le but va nous permettre d'élargir considérablement le champ d'application de ces systèmes de règles. Quand on considère un but contenant des variables, on peut se demander si l'on peut donner des valeurs à ses variables pour que le but soit prouvable. Par exemple, détaillons la preuve du but :

```
(gd-parent-de Cronos ?x)
```

La seule règle de même tête que ce but est :

```
((gd-parent-de ?i ?k) (parent-de ?i ?j) (parent-de ?j ?k))
```

L'unification donne `?i -> Cronos` et `?x -> ?k`, ensuite on considère les deux nouveaux buts : `(parent-de Cronos ?j)` et `(parent-de ?j ?k)`. Le premier but peut utiliser deux règles, on essaye la première :

```
((parent-de ?x ?y) (pere-de ?x ?y))
```

L'unification donne `?x -> Cronos` et `?j -> ?y`. Mais ici, il y a confusion entre la variable `?x` du but initial et la variable `?x` utilisée dans cette dernière règle. Comme les noms des variables d'une règle sont sans importance, on recommence après avoir *renommé* ces variables :

```
((parent-de ?u ?v) (pere-de ?u ?v))
```

L'unification donne maintenant `?u -> Cronos` et `?j -> ?v`, et l'on est amené à prouver le but : `(pere-de Cronos ?v)`. On trouve immédiatement une solution avec `?v -> Zeus`, et en *propageant* dans le deuxième but les valeurs trouvées, il reste à montrer `(parent-de Zeus ?k)` car `?j -> ?v -> Zeus`. On trouve, par exemple `?k -> Pollux`. Et en se rappelant que `?x -> ?k`, on a finalement trouvé une solution de `(gd-parent-de Cronos ?x)` avec `?x -> Pollux`.

Comment modifier la fonction de preuve pour obtenir les valeurs des variables pour lesquelles un but est prouvable ? Il suffit essentiellement de composer les unificateurs utilisés pour démontrer le but. Pour composer ces substitutions, on les accumule en tant que deuxième argument de la fonction `prouveLbuts`. Quand il n'y a plus de buts, au lieu de rendre `#t`, on rend l'application de la substitution trouvée sur le but initial. C'est le principe de la fonction `prouve2` :

```
(define (prouve2 but  Lregles var? cte?)
  (letrec ((prouveLbuts
            (lambda (Lbuts sigma)
```

```
              (if (null? Lbuts)
                (if sigma (sublis sigma but) #f)
                (some (lambda (regle)
                       (let ((sigma1 (unifier (car Lbuts)
                                              (conclusion regle)
                                              var? cte?)))
                        (if  sigma1
                          (prouveLbuts
                            (sublis  sigma1 (append (Lhypotheses  regle)
                                                     (cdr Lbuts)))
                            (compose-subst sigma sigma1 var?  cte?))
                          #f)))
                     (choix-Lregles Lregles (car (car Lbuts)))))))

    (prouveLbuts (list but) '())))

? (Prouve2 '(gd-parent-de Cronos  ?x) *Dieux-Grecs* var? cte?)
(gd-parent-de cronos pollux)
```

En comparant ce résultat avec le but, on déduit la valeur `?x = Pollux` mais on souhaiterait un affichage plus explicite des valeurs trouvées. Mais, il y a plus grave, on n'a pas procédé au renommage des variables des règles utilisées ce qui peut introduire des confusions. Enfin, au lieu de calculer par composition la substitution finale pour l'appliquer au but, on peut se contenter d'appliquer, à chaque étape, la substitution trouvée aux variables du but. Intégrons ces améliorations.

Pour renommer une règle, on utilise la fonction `renomme`. Elle calcule l'ensemble des variables d'une règle, puis fabrique une substitution qui consiste à associer à chaque variable un nouveau nom et on applique cette substitution à la règle. La création d'une nouvelle variable est réalisée avec le générateur du chapitre 5 §7.

```
(define genVar (creer-genVar "?"))

? (genVar) --> ?1

(define (renomme regle var? cte?)
  (let* ((Lvar-regle (apply union
                            (map (lambda (terme)
                                   (terme:Lvar terme var? cte?))
                                 regle)))
         (renommage  (map (lambda (var)(cons var (genVar)))
                          Lvar-regle)))
    (sublis renommage regle)))

? (renomme '((parent-de ?x ?y) (pere-de ?x ?y))
((parent-de ?2 ?3) (pere-de ?2 ?3))
```

La liste des valeurs des variables du but est passée en deuxième argument de `prouveLbuts` et chaque nouvelle substitution trouvée est appliquée à cette liste. D'où la fonction `prouve3`:

```
(define (prouve3 but Lregles var? cte?); -> une substitution solution ou #f
  (let ((Lvar-but (terme:Lvar but var? cte?)))
    (letrec ((prouveLbuts
               (lambda (Lbuts Lvaleur)
                 (if (null? Lbuts)
                   (affiche-solution Lvar-but Lvaleur)
                   (some (lambda (regle)
                           (let* ((regle-bis (renomme regle var? cte?))
                                  (sigma1 (unifier (car Lbuts)
                                                   (conclusion regle-bis)
                                                   var? cte?)))
                             (if  sigma1
                               (prouveLbuts
                                 (sublis sigma1 (append (Lhypotheses regle-bis
                                                         (cdr lbuts)))
                                 (sublis  sigma1 Lvaleur))
                               #f)))
                         (choix-Lregles  Lregles (car (car Lbuts))))))))

      (prouveLbuts (list but) Lvar-but))))
```

Il reste à écrire la fonction d'affichage d'une solution. Pour chaque variable du but, on affiche la valeur trouvée et, s'il n'y a pas de variable, on affiche la liste vide :

```
(define (affiche-solution Lvar Lvaleur)
  (if (null? Lvaleur)
    '()
    (for-each (lambda (var val)
                (display-alln var " = " val))
              Lvar Lvaleur))
 #t)
```

Interrogeons notre base de données mythologiques :

```
? (prouve3 '(gd-parent-de Helene ?x) *Dieux-Grecs* var? cte?)
#f

? (prouve3 '(gd-parent-de ?y ?x)   *Dieux-Grecs* var? cte?)
?x = zeus
?y = ouranos
#t
```

Toutes les solutions à la demande

Le résultat est satisfaisant, mais on aimerait obtenir l'affichage de *toutes* les solutions possibles. Il suffit de reprendre la méthode développée au §1 pour la fonction `demontre4` : on construit la liste des substitutions solutions. Mais en cas de bouclage cette liste peut être infinie, aussi préfère-t-on afficher les solutions sur demande de l'utilisateur. On adapte la méthode utilisée dans la fonction `demontre3`, elle consiste simplement à remplacer dans `prouve3` la fonction d'affichage par une

fonction **autre-solution?**. Cette dernière fonction affiche la solution trouvée et demande ensuite à l'utilisateur s'il désire en chercher une autre.

```
(define (prouve4 but Lregles var? cte?);les substitutions sur demande ou #f
  (let ((Lvar-but (terme:Lvar but var? cte?)))
    (letrec ((prouveLbuts
               (lambda (Lbuts Lvaleur)
                 (if (null? Lbuts)
                   (autre-solution? Lvar-but Lvaleur)
                   (some (lambda (regle)
                           (let* ((regle-bis (renomme regle var? cte?))
                                  (sigma1 (unifier (car Lbuts)
                                                   (conclusion regle-bis)
                                                   var? cte?)))
                             (if  sigma1
                               (prouveLbuts
                                 (sublis sigma1
                                         (append (Lhypotheses regle-bis)
                                         (cdr lbuts)))
                                 (sublis  sigma1 Lvaleur))
                               #f)))
                         (choix-Lregles  Lregles (car (car Lbuts))))))))
      (prouveLbuts (list but) Lvar-but))))
```

```
(define (autre-solution? Lvar Lvaleur)
  (cond ((null? Lvar) #t)
        (Lvaleur  (affiche-solution Lvar Lvaleur)
                  (display  "autre solution o/n : ")
                  (not (eq? 'o   (read))))
        (else #f)))
```

```
? (Prouve4 '(gd-parent-de cronos ?x)   *Dieux-Grecs* var? cte?)
?x = pollux
autre solution o/n : o
?x = helene
autre solution o/n : n
#t
```

18.5 Programmation logique et Prolog

La fonction **prouve4** donne un moyen de calcul des valeurs à donner aux variables du but pour satisfaire à des conditions. Cet aspect méthode de calcul a conduit à considérer ce formalisme comme un langage de programmation. C'est le langage *Prolog* créé à l'origine par A. Colmerauer pour des questions d'analyse des langages naturels.

Un programme logique est simplement la description de relations logiques par un

ensemble Γ de formules $C_1, \ldots, C_N$ où les C_i sont des clauses de Horn de la forme $\theta_0 \Leftarrow \theta_1 \wedge \ldots \wedge \theta_n$ ou de la forme $\theta \Leftarrow$.
On se donne un but à démontrer qui est une formule atomique B et l'on cherche à calculer les valeurs à donner aux variables de B pour que les instances correspondantes soient déductibles de Γ. Autrement dit, on cherche les substitutions σ sur les variables de B telles que l'on puisse déduire σB de Γ. C'est exactement ce que fait la fonction `prouve4` que l'on rebaptise `prolog`:

```
(define prolog prouve4)
```

Nous allons présenter brièvement ce style de programmation à l'aide de quelques exemples.

Calcul sur les entiers de Peano

On a introduit au chapitre 17 §5 les entiers de Peano, ce sont des *termes* construits avec les symboles de fonction S et 0 d'arité respective 1 et 0. L'addition + est définie par les deux égalités:

$$x + 0 = x$$
$$x + S(y) = S(x + y)$$

On représente en Prolog, l'addition par une *relation* ternaire $add(x, y, z)$ qui est vraie quand $z = x + y$. Les deux égalités précédentes s'écrivent sous forme de deux règles:

$add(x, 0, x) <=$

La deuxième égalité se traduit, en utilisant une troisième variable $z = x + y$, par $add(x, S(y), S(z)) <= add(x, y, z)$.

D'où la représentation en Scheme de ces règles:

```
(define *Peano* '(((add ?x 0 ?x ))
                  ((add ?x (S ?y) (S ?z)) (add ?x ?y ?z))))
```

Si l'on désire calculer la somme de `(S 0)` et `(S (S (S 0)))`, on demande:

```
? (prolog '(add (S 0) (S (S (S 0))) ?z) *Peano* var? cte?)
?z = (s (s (s (s 0))))
```

Mais il y a mieux, Prolog est un langage *relationnel*, on peut l'interroger pour connaître les valeurs à donner aux arguments pour obtenir un résultat donné. Par exemple, si on demande comment réaliser $x + y = 3$, Prolog nous affiche les 4 solutions possibles:

```
? (prolog '(add ?x ?y (S (S (S 0)))) *Peano* var? cte?)
?y = 0
?x = (s (s (s 0)))
autre solution o/n ? o
?y = (s 0)
?x = (s (s 0))
```

```
autre solution o/n ? o
?y = (s (s 0))
?x = (s 0)
autre solution o/n ? o
?y = (s (s (s 0)))
?x = 0
autre solution o/n ? o
#f
```

Si l'on cherche un entier x tel que $x + x = 3$, on aura un échec :

```
? (prolog  '(add ?x ?x (S (S (S 0)))) *Peano* var? cte?) -> #f
```

On a fait l'hypothèse $x + 0 = x$, mais on ignore si $0 + x = x$, interrogeons Prolog à ce sujet :

```
? (prolog  '(add  0 ?x ?x) *Peano* var? cte?)
?x = 0
autre solution o/n ? o
?x = (s 0)
autre solution o/n ? o
?x = (s (s 0))
autre solution o/n ? o
?x = (s (s (s 0)))
...
```

Les réponses sont instructives : ce sont les entiers successifs. Cela signifie que l'égalité $0 + x = x$ est vraie pour chaque entier de Peano. Il ne faut pas en conclure que l'on peut déduire l'égalité $0 + x = x$ des deux règles. Il s'agit d'une distinction subtile entre théorème équationnel et théorème inductif, on renvoie à la fin du chapitre suivant pour plus de détails.

On dit que Prolog est un langage *déclaratif*, car on donne les relations qui définissent le problème mais on ne dit pas comment on doit résoudre le problème. C'est en partie exact, mais il faut nuancer. Par exemple, l'ordre des règles n'est *pas* indifférent à cause de la stratégie de recherche des solutions en profondeur d'abord. Pour le voir, permutons les deux règles qui définissent `*Peano*` ; alors à la question :

```
? (prolog  '(add  0 ?x ?x) *Peano* var? cte?)
```

il y a bouclage, car le moteur utilise indéfiniment la nouvelle première règle.

Manipulation de listes en Prolog

Naturellement, le langage Prolog permet la définition de relations sur les listes. On représente les listes par des *termes* construits à l'aide du symbole de fonction *cons* [2], d'arité 2 et de symboles de constantes a, b, c ... dont la constante *nil* pour représenter la liste vide. Par exemple, la liste `(a b c)` sera représentée par le *terme* $(cons\ a\,(cons\ b\,(cons\ c\ nil)))$.

[2] A ne pas confondre avec la fonction `cons` de Scheme.

Ecrivons un programme Prolog pour représenter la relation $member?(x\,,\,L\,,\,oui/non$ où le troisième argument vaudra *oui* si x appartient à la liste L et *non* sinon. On utilise trois règles :

$member?(x\,,\,nil\,,\,non) \Leftarrow$; *non* car la liste vide ne contient rien
$member?(x\,,\,(cons\ x\ L)\,,\,oui) \Leftarrow$; *oui* car x est en tête de la liste L
$member?(x\,,\,(cons\ y\ L)\,,\,reponse) \Leftarrow member?(x\,,\,L\,,\,reponse)$; récursion

Pour tester, on représente ces règles en Scheme :

```
(define *member*
  '(((member? ?x nil non))
    ((member? ?x (cons ?x ?L) oui))
    ((member? ?x (cons ?y ?L) ?rep) (member? ?x  ?L ?rep))))

? (prolog '(member? a  (cons b (cons a nil)) ?rep)*member* var? cte?)
?rep = oui

? (prolog '(member? ?x (cons b (cons a nil)) oui) *member* var? cte?)
?x = b
autre solution o/n ? o
?x = a
autre solution o/n ? o
#f
```

Exercice 7 *Ecrire sur le même principe la relation de concaténation de deux listes.*

On peut combiner plusieurs ensembles de règles. Définissons la relation $intersection?(L1\,,\,L2\,,\,oui/non)$ où le troisième argument vaudra *oui* si les listes $L1$ et $L2$ ont une intersection non vide. On va utiliser la relation *member?* pour définir *intersection?* :

$intersection?(nil\,,\,L2\,,\,non) \Leftarrow$
$intersection?((cons\ x\ L1)\,,\,L2\,,\,oui) \Leftarrow member(x\,,\,L2\,,\,oui)$
$intersection?((cons\ x\ L1)\,,\,L2\,,\,reponse) \Leftarrow member(x\,,\,L2\,,\,non)$
$\wedge\ intersection?(L1\,,\,L2\,,\,reponse)$

Soit en Scheme :

```
(define *intersection*
  '(((inter? nil ?L non))
    ((inter? (cons ?x ?L1) ?L2 oui) (member? ?x ?L2 oui))
    ((inter? (cons ?x ?L1) ?L2 ?rep) (member? ?x ?L2 non)
                                     (inter? ?L1 ?L2 ?rep))))
```

Avant de tester, on doit réunir les deux ensembles de règles :

```
(define *list* (append  *member*   *intersection*))

? (prolog '(inter? (cons a (cons b nil)) (cons c (cons d nil)) ?rep)
           *list*  var?  cte?)
rep? = non
```

Calcul des séquents en Prolog

La programmation logique est bien adaptée à l'implantation de systèmes formels. En effet, une règle d'inférence $\frac{H_1,\ldots,H_n}{C}$ peut se représenter par une clause de Horn $C \Leftarrow H_1 \wedge \ldots \wedge H_n$ si la classe des formules H_i rentre dans le cadre des formules atomiques.

Illustrons cette approche en implantant en Prolog le calcul des séquents propositionnels. Pour exposer le principe, on suppose que les formules ne contiennent que les connecteurs $\Rightarrow$ et $\neg$. Il s'agit de définir une relation de démontrabilité pour les séquents.

On spécifie une relation ternaire *demontrable?*$(\Gamma, \Delta,$ *oui/non*$)$ dont le troisième argument est égal à *oui* quand le séquent $\{\Gamma \vdash \Delta\}$ est démontrable et à *non* sinon.

Un séquent est un axiome, si les listes de formules à gauche et à droite ont une intersection non vide :

demontrable?$(\Gamma, \Delta,$ *oui*$) \Leftarrow$ *inter?*$(\Gamma, \Delta,$ *oui*$)$

On rappelle les deux règles pour les connecteurs $\Rightarrow$ et $\neg$:

$$\frac{\Gamma \vdash \varphi_1, \Delta \qquad \Gamma, \varphi_2 \vdash \Delta}{\Gamma, \varphi_1 \Rightarrow \varphi_2 \vdash \Delta} \qquad\qquad \frac{\Gamma, \varphi_1 \vdash \varphi_2, \Delta}{\Gamma \vdash \varphi_1 \Rightarrow \varphi_2, \Delta}$$

$$\frac{\Gamma \vdash \varphi, \Delta}{\Gamma, \neg\varphi \vdash \Delta} \qquad\qquad \frac{\Gamma, \varphi \vdash \Delta}{\Gamma \vdash \neg\varphi \Delta}$$

Elles se traduisent immédiatement en les clauses suivantes :

$$\begin{array}{l} \textit{demontrable?}((\textit{cons}\,(\textit{imp}\ \varphi_1\ \varphi_2)\,\Gamma), \Delta, \textit{oui}) \\ \qquad\qquad \Leftarrow \textit{demontrable?}(\Gamma, (\textit{cons}\ \varphi_1\ \Delta), \textit{oui}) \\ \qquad\qquad\quad \wedge\ \ \textit{demontrable?}((\textit{cons}\ \varphi_2\ \Gamma), \Delta, \textit{oui}) \end{array}$$

$$\begin{array}{l} \textit{demontrable?}(\Gamma, (\textit{cons}\,(\textit{imp}\ \varphi_1\ \varphi_2), \Delta, \textit{oui}) \\ \qquad\qquad \Leftarrow \textit{demontrable?}((\textit{cons}\ \varphi_1\ \Gamma), (\textit{cons}\ \varphi_2\ \Delta), \textit{oui})) \end{array}$$

$$\begin{array}{l} \textit{demontrable?}(\Gamma, (\textit{cons}\,(\textit{non}\ \varphi)\,\Delta), \textit{oui}) \\ \qquad\qquad \Leftarrow \textit{demontrable?}((\textit{cons}\ \varphi\ \Gamma), \Delta, \textit{oui})) \end{array}$$

$$\begin{array}{l} \textit{demontrable?}((\textit{cons}\,(\textit{non}\,\varphi)\,\Gamma), \Delta, \textit{oui}) \\ \qquad\qquad \Leftarrow \textit{demontrable?}(\Gamma, (\textit{cons}\ \varphi\ \Delta), \textit{oui}) \end{array}$$

Enfin, si aucune de ces règles ne s'applique, le séquent n'est pas démontrable. La traduction immédiate de ces règles en un programme Prolog est une belle illustration du caractère déclaratif de ce langage.

Pour tester ce système, on combine les règles précédentes avec les règles des listes :

```
(define *regle-sequent*    ;; axiome des séquents
  '(((demontrable? ?Gamma ?Delta oui)
     (inter? ?Gamma ?Delta oui))
    ((demontrable? (cons (imp ?phi1 ?phi2) ?Gamma) ?Delta  oui)
     (demontrable? ?Gamma (cons ?phi1 ?Delta) oui)
     (demontrable? (cons ?phi2 ?Gamma) ?Delta oui))
```

```
    ((demontrable? ?Gamma (cons (imp ?phi1 ?phi2) ?Delta)  oui)
     (demontrable? (cons ?phi1 ?Gamma) (cons  ?phi2  ?Delta)  oui))
    ((demontrable? ?Gamma (cons (non ?phi) ?Delta) oui)
     (demontrable? (cons ?phi ?Gamma)  ?Delta oui))
    ((demontrable? (cons (non ?phi) ?Gamma) ?Delta oui)
     (demontrable? ?Gamma  (cons ?phi ?Delta) oui))
    ((demontrable? ?Gamma ?Delta non))))

(define *sequent*  (append *regle-sequent* *list*))
```

On teste avec des exemples très simples, car l'efficacité n'est pas la qualité première de notre petit moteur Prolog.
La formule $a \Rightarrow \neg\neg a$ est-elle démontrable?

```
? (prolog '(demontrable? nil (cons(imp a (neg (neg a))) nil) ?rep)
          *sequent* var?  cte?)
rep? =  oui
```

La formule $(a \Rightarrow b) \Rightarrow a$ est-elle démontrable?

```
? (prolog '(demontrable? nil (cons (imp (imp a  b) a) nil) ?rep)
          *sequent*  var?  cte?)
rep? =  non
```

L'élégance et la clarté de ce programme montre l'excellente adéquation de ce type de langage à l'implantation de systèmes pour la déduction logique.

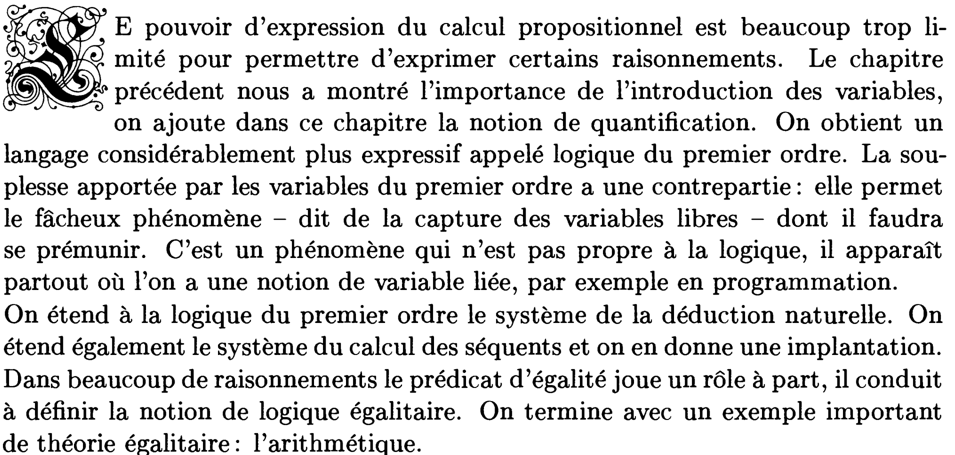
On n'a fait qu'effleurer les principes de la programmation en Prolog. Il ne faut pas confondre programmation logique et Prolog. Prolog est un langage qui utilise une stratégie de recherche des solutions d'un programme logique. Aussi, pour des raisons d'efficacité, le moteur Prolog s'écarte parfois de ses bases logiques. Citons le problème de l'instruction coupe-choix (ou cut) qui permet d'éliminer certaines parties de l'arbre de recherche d'une preuve. Ce coupe-choix sert, par exemple, à définir une pseudo opération de négation. Il s'agit de la *négation par l'échec* qui est distincte de la négation logique. On dit avoir prouvé la négation d'une formule si la preuve de cette formule a échoué. On renvoie le lecteur intéressé à la littérature sur ce langage.

18.6 De la lecture

Pour les systèmes experts, on renvoie aux livres de Laurière et de Voyer [Lau85, Voy87].
L'aspect base de données est détaillé dans [ASS89, ML95].
On trouvera une présentation plus complète de l'implantation de Prolog en Lisp dans [Boi88, Nor92].
Pour le langage Prolog, on conseille la lecture de [GKPvC85, SS93].
Pour les aspects logiques, on renvoie comme d'habitude au livre de Lalement [Lal90].

Chapitre 19

Logique du premier ordre



LE pouvoir d'expression du calcul propositionnel est beaucoup trop limité pour permettre d'exprimer certains raisonnements. Le chapitre précédent nous a montré l'importance de l'introduction des variables, on ajoute dans ce chapitre la notion de quantification. On obtient un langage considérablement plus expressif appelé logique du premier ordre. La souplesse apportée par les variables du premier ordre a une contrepartie : elle permet le fâcheux phénomène – dit de la capture des variables libres – dont il faudra se prémunir. C'est un phénomène qui n'est pas propre à la logique, il apparaît partout où l'on a une notion de variable liée, par exemple en programmation.
On étend à la logique du premier ordre le système de la déduction naturelle. On étend également le système du calcul des séquents et on en donne une implantation. Dans beaucoup de raisonnements le prédicat d'égalité joue un rôle à part, il conduit à définir la notion de logique égalitaire. On termine avec un exemple important de théorie égalitaire : l'arithmétique.

19.1 Langages du premier ordre

Formules du premier ordre

La pratique des mathématiques élémentaires nous a familiarisé avec une utilisation intuitive des quantificateurs et des variables. Par exemple, pour dire qu'un entier $p > 1$ est un nombre premier, on exprime qu'il n'admet pas d'autres diviseurs que 1 et lui-même. C'est la formule :

$$\forall\, m\, (m\, \mathit{divise}\, p \;\Rightarrow\; (m = 1 \,\vee\, m = p)) \tag{1}$$

Intuitivement, l'expression $m\, \mathit{divise}\, p$ est vraie quand l'entier m divise exactement l'entier p, autrement dit le symbole *divise* est le nom d'un prédicat binaire.
La formule (1) se lit :

pour tout m on a : si m divise p alors m est égal à 1 ou m est égal à p.

On dit que m est une variable quantifiée par le *quantificateur universel* $\forall$. Considérons cette formule comme une définition de la propriété $premier(p)$.
Si l'on veut exprimer qu'il existe toujours un nombre premier p supérieur à tout nombre donné N, on écrit :

$$\forall N \, \exists \, p \, (premier(p) \, \wedge \, (p > N)) \tag{2}$$

Le symbole $\exists$ de (2) est appelé *quantificateur existentiel.* Cette formule se lit : pour tout entier N il existe au moins un entier p qui soit premier et plus grand que N.
On peut aussi avoir besoin de symboles de fonction, par exemple exprimons que si un nombre n est premier, alors son successeur, désigné par le terme $S\,n$, ne l'est pas :

$$\forall n \, (premier(n) \;\Rightarrow\; \neg\,(premier(S\,n)))$$

L'utilisation des quantificateurs n'est pas réservée aux mathématiques, considérons le prédicat $parle(x\,,\,L)$ signifiant intuitivement que l'individu x parle la langue L. Alors, on a les traduction suivantes :
- «tout le monde parle au moins une langue» : $\forall x \, \exists L \; parle(x\,,\,L)$
- «il existe au moins une langue parlée par tout le monde» : $\exists L \, \forall x \; parle(x\,,\,L)$
on voit ici que si l'on permute un quantificateur $\forall$ avec un quantificateur $\exists$, le sens change complètement.
- «deux individus ont toujours une langue commune» :

$$\forall x \forall y \, \exists L \; (parle(x, L) \;\wedge\; parle(y, L))$$

On voit ici que l'on peut permuter les deux quantificateurs de même nature $\forall x$ et $\forall y$ sans modifier le sens, aussi abrège-t-on parfois une suite de quantificateurs de même nature $\forall x_1, \ldots, \forall x_n$ par $\forall x_1, \ldots, x_n$.

En revanche, ces quantificateurs ne permettent pas d'exprimer des notions comme : «je pense qu'il y a beaucoup d'individus qui parlent la langue L».

Remarque 1 Il est important de réaliser que toutes ces formules sont des objets syntaxiques, ce n'est que pour en faciliter la lecture que nous avons attribué un sens intuitif aux divers noms de prédicats.
Comme pour le calcul propositionnel, la preuve d'une formule se fera à l'aide d'un système de règles et sera indépendante de l'éventuel contenu intuitif qu'elle peut représenter.
Indiquons aussi que l'on peut définir une notion de valeur de vérité sémantique pour une formule du premier ordre mais c'est un aspect moins important pour le programmeur ; on renvoie à la littérature sur le sujet.

Avant de donner les règles de déduction pour utiliser les quantificateurs, on commence par préciser la notion de langage du premier ordre.

Définition d'un langage du premier ordre

Pour définir un langage du premier ordre, il faut d'abord se donner l'ensemble des symboles de base ou *alphabet.* Il est constitué par :

- un ensemble dénombrable Var de variables (dites variables du premier ordre),
- un ensemble F de symboles de fonctions avec leurs arités, on peut donc définir l'ensemble des termes $Termes(Var\,,\,F)$,
- les connecteurs propositionnels : $\wedge$, $\vee, \Rightarrow \ldots$
- un ensemble R de symboles de prédicats (ou de relations) avec leurs arités,
- deux quantificateurs : $\forall$ et $\exists$

On dit *un* langage du premier ordre car les ensembles F et R ne sont pas fixés, ils dépendent du domaine à étudier.

Définition des formules du premier ordre

Les formules du premier ordre les plus simples s'appellent les *formules atomiques.* Ce sont les formules de la forme $p(t_1, \ldots, t_n)$ où p est un symbole de prédicat d'arité n et les $t_1, \ldots, t_n$ sont des termes. Par exemple, $premier\,(S\,n)$ est une formule atomique utilisant le prédicat unaire *premier* et le terme $S\,n$. On a vu au chapitre précédent que les formules atomiques étaient les briques de base des clauses de Horn.

Les *formules du premier ordre* sont construites à partir des formules atomiques par combinaison à l'aide de connecteurs propositionnels ou par quantification.

Définition 1 Les formules du premier ordre sont les formules atomiques et les formules de la forme $\varphi_1 \circ \varphi_2\,, \quad \neg\,\varphi, \quad \forall\,x\;\varphi, \quad \exists\,x\;\varphi$, où φ, φ_1, φ_2 sont[1] des formules du premier ordre.

Par exemple, la formule suivante est du premier ordre :

$$\forall\,x \forall\,y \exists\,L\;(parle(x, L)\;\wedge\;parle(y, L))$$

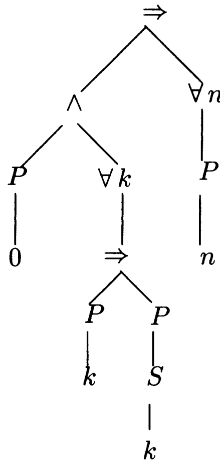
car elle est de la forme $\forall\,x\;\forall\,y\;\exists\,L\;\varphi$ avec $\varphi\;=\;\varphi_1 \wedge \varphi_2$ où φ_1, φ_2 sont les formules atomiques $\varphi_1\;=\;parle(x\,,\,L)$ et $\varphi_2\;=\;parle(y\,,\,L)$.

Le principe du raisonnement par récurrence — pour prouver qu'une propriété φ qui dépend de l'entier n est vraie pour tout n — s'exprime par la formule du premier ordre :

$$P(0) \wedge \forall\,k\,(P(k)\;\Rightarrow\;P(S\,k))\;\Rightarrow\;\forall\,n\;P(n) \tag{3}$$

Si l'on considère la quantification par rapport à une variable comme une opération unaire, on peut étendre aux formules du premier ordre la syntaxe abstraite des formules propositionnelles et des termes, ainsi la formule précédente a pour arbre :

[1] On a désigné par $\circ$ l'un quelconque des connecteurs binaires du calcul propositionnel.



Pour limiter l'utilisation des parenthèses, la quantification est prioritaire sur les connecteurs propositionnels. Par exemple, la formule :
$(\forall x\, p(x,y)) \Rightarrow (\exists y\, q(x,y))$ s'écrit plus simplement $\forall x\, p(x,y) \Rightarrow \exists y\, q(x,y)$.

On dit logique du *premier ordre* car les variables quantifiées sont dans l'ensemble Var, on ne quantifie pas sur les symboles de prédicats ou de fonctions. Mais, si l'on voulait exprimer que le raisonnement par récurrence est utilisable pour n'importe quel prédicat P, on serait amené à quantifier aussi sur le symbole de prédicat P :

$$\forall P\ (\ P(0) \wedge \forall k\, (P(k) \Rightarrow P(S\, k)) \quad \Rightarrow \quad \forall n\, P(n)\ )$$

Une telle quantification n'est pas permise en logique du premier ordre ; elle le devient si l'on se place dans une logique plus générale dite du *deuxième ordre* qui admet des variables de prédicats.

Exercice 1 *Soit f une application d'un ensemble A dans un ensemble B. Traduire par des formules logiques les propriétés d'injectivité ou de surjectivité de f.*

19.2 Une représentation en Scheme des formules du premier ordre

On a déjà choisi des représentations pour les formules propositionnelles (chapitre 16), pour les termes (chapitre 17), pour les formules atomiques (chapitre 18) aussi, reste-t-il à préciser la représentation pour les formules quantifiées.
On représente une formule quantifiée $Q\ x\ \psi$ par une liste à trois éléments :
- le premier est le nom du quantificateur : `tout` si $Q = \forall$ et `ex` si $Q = \exists$,
- le deuxième est le symbole qui représente la variable quantifiée,
- le dernier est le champ du quantificateur, c'est-à-dire ici la représentation de la formule ψ.

Pour distinguer dans nos exemples les symboles qui représentent un prédicat de ceux qui représentent des symboles de fonctions, on se donne a priori une liste de quelques symboles de prédicats.

```
(define   *Lpredicats*  '(p q r))

(define  (predicat? x)
  (memq x *Lpredicats*))
```

Pour distinguer les formules atomiques et les formules quantifiées, on définit les fonctions :

```
(define  (formule-atomique? f)
  (and (pair? f)(predicat? (car f))))

(define  (formule-quantifiee? f)
  (and (pair? f)(memq (car f) '(tout ex))))
```

Pour accéder aux divers champs d'une formule, on définit les accesseurs suivants :

```
(define  connecteur           car)  ; connecteur d'une formule composée
(define  quantificateur       car)  ; son quantificateur
(define  champQuantificateur  caddr); champ d'une formule quantifiée
(define  varQuantifiee        cadr) ; variable  quantifiée
```

19.3 Occurrences libres ou liées de variables

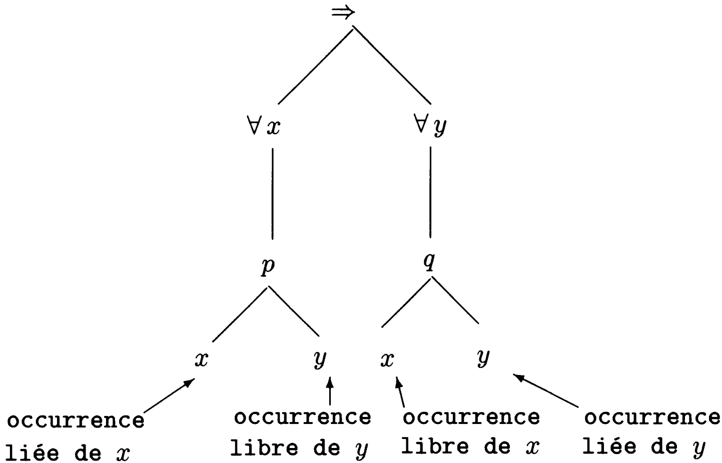
L'arbre d'une formule permet de définir le *champ d'un quantificateur* comme étant la sous-formule dont l'arbre est fils du nœud étiqueté par ce quantificateur. Par exemple, dans la formule (3) le champ du quantificateur $\forall k$ est la formule $P(k) \Rightarrow P(S\,k)$.

Définition 2 On dit qu'une *occurrence* d'une variable x dans une formule est *liée* si elle est dans le champ d'un quantificateur portant sur cette variable, sinon on dit que cette *occurrence est libre.*

Considérons la formule :

$$\forall x\, p(x\,,\, y) \;\Rightarrow\; \exists y\; q(x\,,\, y) \tag{4}$$

où p et q sont des prédicats binaires, indiquons sur son arbre la nature de chaque occurrence de variable.



On définit l'ensemble des *variables libres* d'une formule φ comme étant l'ensemble des variables ayant au moins une occurrence libre dans φ ; on le note $VarLib(\varphi)$. Une formule sans variable libre est dite *close*. Par exemple, l'ensemble $VarLib$ de la formule (4) est $\{x, y\}$ et la formule (3) est close.

Le calcul de la liste des variables libres d'une formule se fait en distinguant les trois types de formules : atomique, quantifiée ou composée. Dans le cas atomique, on fait usage de la fonction `terme:Lvar` du chapitre 17 §2 ce qui nécessite de passer en paramètres les prédicats `var?` et `cte?`.

```
(define  (formule:LvarLibre formule  var? cte?)
  (cond
    ((formule-atomique? formule)
     (apply union  (map (lambda (terme)(terme:Lvar  terme var? cte?))
                        (cdr formule))))
    ((formule-quantifiee? formule)
     (remove (varQuantifiee formule)
             (formule:LvarLibre (champQuantificateur formule) var? cte?)))
    (else
     (apply union  (map (lambda (form)(formule:LvarLibre form var? cte?))
                        (cdr formule))))  ))
```

```
? (formule:LvarLibre '(imp (tout x (p x y z)) (ex y (q y z))) var? cte?)
(z y)
```

Pour tester, on s'est donné une définition des fonctions `var?` et `cte?` :

```
(define (var? s)
  (and (symbol? s) (not (cte? s))))

(define (cte? s)
  (or (number? s)(memq s '(a b c d))))
```

Le calcul des variables liées est plus simple car une formule atomique n'en a pas :

```
(define  (formule:LvarLiee formule)
  (cond
    ((formule-atomique? formule) '())
    ((formule-quantifiee? formule)
     (cons (varQuantifiee formule)
           (formule:LvarLiee (champQuantificateur formule))))
    (else
     (apply union  (map  formule:LvarLiee
                         (cdr formule))))))

? (formule:LvarLiee '(imp (tout x (p x y z)) (ex y (q y z))) )
(y x)
```

Il y a une analogie — en fait c'est plus qu'une analogie (voir chapitre 20) — entre occurrence libre d'une variable et variable globale en programmation, et une analogie entre occurrence liée et variable locale. L'idée est que les occurrences de variables liées servent à désigner des *positions.* Par conséquent, le nom attribué à une telle variable n'est pas essentiel alors que pour les occurrences libres on ne peut pas changer de nom sans changer la signification de la formule. Par exemple, on considère que la formule (4) possède le même sens que la formule suivante :

$$\forall u\, p(u\,,\, y) \;\Rightarrow\; \exists v\, q(x\,,\, v)$$

Attention ! Il ne faut pas donner à u le nom de y car manifestement la formule :

$$\forall y\, p(y\,,\, y) \;\Rightarrow\; \exists v\, q(x\,,\, v)$$

n'a plus le même sens. On dit qu'il y a eu un *phénomène de capture* de l'occurrence libre de y par le quantificateur $\forall$.
On a la même distinction en mathématiques avec la notion de variable *muette.* Par exemple, dans la formule qui calcule le maximum en x d'une fonction F de deux variables : $\max_x F(x,y)$; la variable x est *muette* car elle est liée par le *max*, alors que y se comporte comme une variable libre. On peut donner à cette variable muette n'importe quel nom sauf y :

$$\max_x F(x,y) \;=\; \max_z F(z,y) \;\neq\; \max_y F(y,y)$$

19.4 Substitutions en logique du premier ordre

Comme pour les termes, on va définir la notion de *substitution* dans une formule, mais comme le nom des variables liées n'est pas essentiel, les substitutions ne concerneront que les occurrences *libres.*

Substitutions générales

On note $\varphi\,[x \leftarrow s]$ la formule obtenue en remplaçant toutes les occurrences *libres* de la variable x par le terme s. Pour s'assurer que l'on obtient bien encore une formule, en voici une définition plus précise qui suit la structure de la définition du concept de formule :

- si φ est atomique et donc de la forme $p(t_1, \ldots, t_n)$, alors on substitue dans chaque terme et on retourne $p(t_1[x \leftarrow s], \ldots, t_n[x \leftarrow s])$;
- si φ est une formule composée $\varphi_1 \circ \varphi_2$, alors on substitue dans chaque formule et on retourne $\varphi_1 [x \leftarrow s] \circ \varphi_2 [x \leftarrow s]$;
- si φ est une formule quantifiée de la forme $Q\, y\, \psi$, où Q désigne l'un des deux quantificateurs, alors il y a deux cas :
  - si $y = x$ alors il n'y a aucune occurrence libre de x dans φ donc on ne fait rien et on retourne φ,
  - si $x \neq y$ alors on substitue dans ψ et on retourne $Q\, y\, \psi[x \leftarrow s]$.

Quand une formule φ possède x comme variable libre, il est parfois plus parlant de la noter $\varphi(x)$ et par extension le résultat de la substitution du terme s à x sera noté $\varphi(s)$.

Ecrivons une fonction **`formule:substituerVar`** pour substituer un terme à une variable dans une formule. Notons que dans le cas des formules atomiques, on est ramené au cas des termes, aussi utilise-t-on la fonction `terme:substituer`. Pour les formules quantifiées, on substitue dans le champ du quantificateur si la variable de quantification est distincte de la variable à substituer.

```
(define  (formule:substituerVar formule var terme  terme:var? terme:cte?)
  (cond
    ((formule-atomique? formule)
     (terme:substituerVar formule var terme terme:var? terme:cte?))
    ((formule-quantifiee? formule)
     (let ((varQuantif (varQuantifiee formule)))
       (if (eq? var varQuantif)
         formule
         (list (quantificateur formule)
               varQuantif
               (formule:substituerVar  (champQuantificateur formule)
                                       var  terme
                                       terme:var? terme:cte?)))))
    (else
     (cons (connecteur formule)
           (map (lambda (f)
                  (formule:substituerVar f var terme terme:var? terme:cte?))
                (cdr formule))))))
```

Calculons $(\forall\, x\, p(x\,,\, y) \;\Rightarrow\; \exists\, y\; q(x\,,\, y))[y \leftarrow f(u)]$:

```
? (formule:substituerVar  '(imp (tout x (p x y))(ex y (q x y)))
                           'y  '(f u)   var?  cte?)
(imp (tout x (p x (f u))) (ex y (q x y)))
```

Exercice 2 *Plus généralement, définir l'action d'une substitution σ sur une formule φ.*

Capture et substitution admissible

Malheureusement, cette notion de substitution n'est pas assez restrictive car elle peut donner lieu à des phénomènes de capture. Pour comprendre le problème, reprenons l'exemple du *max* en mathématiques.
On définit la fonction G par $G(y) = \max_x F(x, y)$. Si l'on substitue à la variable y le terme $2 * u$, on trouve bien $G(2 * u) = \max_x F(x, 2 * u)$. Mais si l'on substitue à y le terme $2 * x$, alors il est faux que l'ait encore
$G(2 * x) = \max_x F(x, 2 * x)$. Ici la variable *libre* x du terme $2 * x$ a été capturée par le *max*.

On a exactement le même phénomène en logique.
Notons $\varphi(y)$ la formule $\exists x\ (x = 1 + y)$ alors si l'on substitue le terme $f(u)$ à l'occurrence libre de y, on obtient $\varphi(f(u)) = \exists x\ (x = 1 + f(u))$. Mais si l'on avait substitué le terme $f(x)$, on aurait eu une capture de la variable libre x de $f(x)$ par le quantificateur $\exists x$ et $\varphi(fx)) \neq \exists x\ (x = 1 + f(x))$.

Pour éviter ce phénomène, on introduit la notion de *substitution admissible*, c'est-à-dire sans capture. On dit qu'une substitution $[x \leftarrow s]$ est *admissible* pour une formule si après substitution aucune variable libre du terme s n'est dans le champ d'un quantificateur pour cette variable. De façon précise, la substitution de la variable x par le terme s dans une formule φ est admissible quand :

- φ est atomique,
- φ est de la forme $\varphi_1 \circ \varphi_2$ et la substitution est admissible dans les φ_i,
- φ est de la forme $Q\,x\,\psi$ (car il n'y pas de substitution à faire !),
- φ est de la forme $Q\,y\,\psi$ avec $x \neq y$, alors la substitution est admissible si elle l'est pour ψ et si y n'est pas variable libre du terme s.

La traduction en Scheme de cette définition est immédiate, le prédicat `subst-admissible?` rend `#t` si la substitution de `var` par un `terme` dans `formule` est admissible :

```
(define  (subst-admissible? formule var terme var? cte?)
  (cond
    ((formule-atomique? formule) #t)
    ((formule-quantifiee? formule)
     (and  (not (memq (varQuantifiee formule) (terme:Lvar terme var? cte?)))
           (subst-admissible? (champQuantificateur formule)
                              var terme var? cte?)))
    (else
     (every (lambda (f) (subst-admissible? f var terme var? cte?))
            (cdr formule)))))

? (subst-admissible?  '(-> (tout x (p x y)) (ex y (q x y)))  'x  'z var? cte?)
#t
? (subst-admissible?  '(-> (tout x (p x y)) (ex y (q x y)))  'x  'y var? cte?)
#f
```

Dorénavant, quand on appliquera une substitution à une formule du premier ordre, on supposera toujours que la substitution est admissible.

On est maintenant en mesure d'étendre à la logique du premier ordre les systèmes formels introduits pour le calcul propositionnel.

19.5 Déduction naturelle en logique du 1er ordre

On commence par étendre le système de la déduction naturelle. On ajoute aux règles déjà données au chapitre 16 pour les connecteurs propositionnels, les règles relatives aux deux quantificateurs.

Règles pour les quantificateurs

Il y a une règle d'introduction et une règle d'élimination pour chaque quantificateur.

Introduction

$$\frac{\Gamma \vdash \varphi\,;\; x \notin \mathit{VarLibre}(\Gamma)}{\Gamma \vdash \forall x\, \varphi}$$

$$\frac{\Gamma \vdash \varphi[x \leftarrow s]}{\Gamma \vdash \exists x\, \varphi}$$

Elimination

$$\frac{\Gamma \vdash \forall x\, \varphi}{\Gamma \vdash \varphi[x \leftarrow s]}$$

$$\frac{\Gamma \vdash \exists x\, \varphi \qquad \Delta, \varphi \vdash \psi;\; x \notin \mathit{VarLibre}(\Delta, \psi)}{\Gamma, \Delta \vdash \psi}$$

Commentons ces quatre règles. La présence de variables a nécessité l'ajout de conditions extra-ogiques pour qu'elles soient applicables.

- Introduction de $\forall$ (ou règle de généralisation).

  La notation $x \notin \mathit{VarLibre}(\Gamma)$ signifie que x n'est variable libre d'aucune des formules de Γ. Cette condition correspond à l'idée de variable *générique*. Autrement dit, si l'on a prouvé $\varphi(x)$ où x est une variable *quelconque* alors on a prouvé $\varphi(x)$ pour tout x. En revanche, si l'on supprimait l'hypothèse sur x, on aurait une règle ne correspondant pas à l'idée de variable générique. Par exemple, on peut toujours prouver que $x \geq 0$ en supposant que $x \geq 0$, mais il est faux d'en déduire que pour toute variable y on a $y \geq 0$; autrement dit le jugement $x \geq 0 \vdash x \geq 0$ n'entraîne pas le jugement $x \geq 0 \vdash \forall x\,(x \geq 0)$.

- Elimination de $\forall$ (ou règle d'instanciation).

  Rappelons que l'hypothèse d'admissibilité est toujours faite quand on doit effectuer une substitution dans une formule du 1er ordre. Cette règle exprime que si l'on a prouvé $\varphi(x)$ pour tout x, alors on aura en particulier prouvé $\varphi(s)$ pour un terme s quelconque.

- Introduction de $\exists$.

  Le fait d'avoir prouvé $\varphi(s)$ pour un certain terme s montre qu'il existe au moins une valeur de x pour laquelle $\varphi(x)$ est prouvable.

- Elimination de $\exists$ (ou règle d'instanciation).

  Le quantificateur $\exists$ étant une espèce de disjonction infinie, on retrouve une formulation indirecte analogue à la règle du $\vee$ en calcul propositionnel.

Selon les règles utilisées pour la négation, on obtient un système de *déduction naturelle intuitionniste ou classique.*

Exemples de preuves

Illustrons l'utilisation de ces règles avec quelques exemples simples. Comme d'habitude il faut lire la recherche de la preuve de bas en haut. On part de la formule à démontrer et l'on essaye de remonter aux axiomes en appliquant des règles.
Le premier exemple traduit l'idée qu'il ne sert à rien de quantifier sur une variable qui n'apparaît pas dans une formule :

si $x \notin VarLibre(\Gamma)$ alors on a : $\Gamma \vdash \forall x\, \varphi \Rightarrow \varphi$ et $\Gamma \vdash \varphi \Rightarrow \forall x\, \varphi$

Démonstration de $\Gamma \vdash \forall x\, \varphi \Rightarrow \varphi$:

$$\dfrac{\dfrac{\Gamma, \forall x\, \varphi \vdash \forall x\, \varphi \quad \texttt{axiome}}{\Gamma, \forall x\, \varphi \vdash \varphi}\ \texttt{instanciation avec } s = x}{\Gamma \vdash \forall x\, \varphi \Rightarrow \varphi}\ \texttt{introduction de} \Rightarrow$$

Cette démonstration n'a pas fait usage de l'hypothèse sur x, mais elle sera utilisée dans celle de $\Gamma \vdash \varphi \Rightarrow \forall x\, \varphi$:

$$\dfrac{\dfrac{\Gamma, \varphi \vdash \varphi \quad \texttt{axiome}}{\Gamma, \varphi \vdash \forall x\, \varphi}\ \texttt{généralisation possible grâce à l'hypothèse sur } x}{\Gamma \vdash \varphi \Rightarrow \forall x\, \varphi}\ \texttt{introduction de} \Rightarrow$$

Exercice 3 *Prouver que si* $x \notin VarLib(\psi)$ *on a* $\vdash (\forall x\, \varphi \circ \psi) \Rightarrow \forall x\, (\varphi \circ \psi)$.

Le deuxième exemple va montrer que l'on peut effectivement changer le nom d'une variable liée. On a le théorème :

$$\Gamma \vdash \forall x\, \varphi \Rightarrow \forall y\, \varphi[x \leftarrow y] \quad \text{où} \quad y \notin VarLibre(\Gamma, \varphi).$$

En voici une preuve :

$$\dfrac{\dfrac{\dfrac{\Gamma, \forall x\, \varphi \vdash \forall x\, \varphi}{\Gamma, \forall x\, \varphi \vdash \varphi[x \leftarrow y]}}{\Gamma, \forall x\, \varphi \vdash \forall y\, \varphi[x \leftarrow y]}}{\Gamma \vdash \forall x\, \varphi \Rightarrow \forall y\, \varphi[x \leftarrow y]}$$

axiome

instanciation, car $[x \leftarrow y]$ **est admissible**

généralisation, car $y \notin \mathit{VarLibre}(\Gamma, \varphi)$

introduction de $\Rightarrow$

Exercice 4 *Prouver le jugement :* $\vdash \exists x\, \varphi \Rightarrow \neg \forall x\, \neg \varphi$.

Avec notre dernier exemple on montre qu'en logique classique, on peut exprimer un quantificateur en fonction de l'autre. On a les théorèmes :

$\vdash \neg \forall x\, \neg \varphi \Rightarrow \exists x\, \varphi$ et $\vdash \neg \exists x\, \neg \varphi \Rightarrow \forall x\, \varphi$

Voici la preuve du premier théorème :

axiome

$\neg \forall x \neg \varphi \vdash \neg \forall x \neg \varphi$

———————————— définition de $\neg$

$\neg \forall x \neg \varphi \vdash \forall x \neg \varphi \Rightarrow \bot$

———————————— exercice 3

$\neg \forall x \neg \varphi \vdash \forall x (\neg \varphi \Rightarrow \bot)$

———————————— instanciation

$\neg \forall x \neg \varphi \vdash (\neg \varphi \Rightarrow \bot)$

———————————— définition de $\neg$

axiome

$\neg \forall x \neg \varphi \vdash \neg \neg \varphi$ $\qquad\qquad$ $\neg \varphi \vdash \neg \varphi$

———————————————————————— élimination de $\neg$

$\neg \forall x \neg \varphi , \neg \varphi \vdash \bot$

—————————— raisonnement par l'absurde en logique classique

$\neg \forall x \neg \varphi \vdash \varphi$

—————————— introduction de $\exists$ (avec $y = x$)

$\neg \forall x \neg \varphi \vdash \exists x \, \varphi$

—————————— introduction de $\Rightarrow$

$\vdash \neg \forall x \neg \varphi \Rightarrow \exists x \, \varphi$

19.6 Calcul des séquents du premier ordre

Le formalisme des séquents nous a permis de donner une implantation du calcul propositionnel, qu'en est-il pour la logique du premier ordre?

Ce formalisme se généralise effectivement à la logique du premier ordre, ainsi pour la logique classique, on a des règles d'élimination à droite et à gauche pour chaque quantificateur.

Règles pour les quantificateurs

Elimination à gauche **Elimination à droite**

$$\frac{\Gamma\,,\,\varphi[x \leftarrow s]\,,\forall x\,\varphi \vdash \Delta}{\Gamma\,,\forall x\,\varphi \vdash \Delta} \qquad \frac{\Gamma \vdash \varphi[x \leftarrow y]\,,\,\Delta\,; \quad y \notin \mathit{VarLibre}(\Gamma,\Delta,\forall x\,\varphi)}{\Gamma \vdash \forall x\,\varphi\,,\,\Delta}$$

$$\frac{\Gamma\,,\,\varphi[x \leftarrow y] \vdash \Delta\,; \quad y \notin \mathit{VarLibre}(\Gamma,\Delta,\exists x\,\varphi)}{\Gamma\,,\,\exists x\,\varphi \vdash \Delta} \qquad \frac{\Gamma \vdash \varphi[x \leftarrow s]\,,\,\exists x\,\varphi\,,\,\Delta}{\Gamma \vdash \exists x\,\varphi\Delta}$$

On a toujours des règles symétriques, mais on a en plus des hypothèses extra-logiques. Pour chaque règle on a supposé implicitement que *toutes* les substitutions $[x \leftarrow s]$ et $[x \leftarrow y]$ sont *admissibles.*

Contrairement au cas propositionnel, les règles d'élimination à gauche de $\forall$ et à droite de $\exists$ portent mal leur nom car on les garde dans le séquent ! Par ailleurs, le choix du terme s à utiliser n'est pas automatique. On se doute que la preuve automatique d'une formule n'est plus assurée. En fait, on peut démontrer qu'il n'*existe aucun algorithme* pour décider si une formule générale du premier ordre est un théorème ou non. Néanmoins, on peut donner des méthodes qui réussissent «souvent» quitte à demander à l'utilisateur de les aider pour certains choix critiques.

Exemples de preuves

Avant d'essayer de programmer une telle méthode, il est instructif de faire à la main quelques preuves pour voir les nouveaux phénomènes introduits par les quantificateurs.

A titre de comparaison, refaisons la preuve du jugement $\vdash \neg\forall x\,\neg\varphi \;\Rightarrow\; \exists x\,\varphi$ dans le formalisme des séquents. On se laisse guider par les connecteurs ou quantificateurs à éliminer.

	`axiome`
$\varphi \vdash \varphi$	`élimination de` $\exists$ `à droite (l'identité` $[x \leftarrow x]$ `est évidemment admissible)`
$\varphi \vdash \exists x\, \varphi$	`élimination de` $\neg$ `à droite`
$\vdash \neg\varphi, \exists x\, \varphi$	`élimination de` $\forall$ `à droite` (x `n'est pas libre dans` $\exists x\, \varphi$)
$\vdash \forall x\, \neg\varphi\,, \exists x\, \varphi$	`élimination de` $\neg$ `à gauche`
$\neg\forall x\, \neg\varphi \vdash \exists x\, \varphi$	`élimination de` $\Rightarrow$ `à droite`
$\vdash \neg\forall x\, \neg\varphi \;\Rightarrow\; \exists x\, \varphi$	

Donnons maintenant un exemple où, selon l'ordre dans lequel on choisit d'appliquer les règles d'élimination des quantificateurs, on arrive rapidement à une preuve ou non. Pour pouvoir commenter pas à pas le processus de recherche d'une preuve, on écrit ici la preuve de *haut en bas* : on part de la formule à prouver et l'on écrit en *dessous* les étapes successives à prouver.

On veut prouver le séquent :

$$\exists x \forall y\, p(x,y) \;\vdash\; \forall y\, \exists x\, p(x,y)$$

On peut éliminer $\forall y$ à droite car y n'est pas libre dans l'hypothèse de gauche :

$$\exists x \forall y\, p(x,y) \;\vdash\; \exists x\, p(x,y)$$

Ensuite on a le choix entre éliminer $\exists$ à droite ou à gauche. Essayons à droite, avec un terme s_1 à préciser ultérieurement :

$$\exists x \forall y\, p(x,y) \;\vdash\; p(s_1,y)\,,\; \exists x\, p(x,y)$$

Maintenant, pour éliminer $\exists x$ à gauche, il faut supposer que $x \notin \mathit{VarLib}(s_1)$:

$$\forall y\, p(x,y) \;\vdash\; p(s_1,y)\,,\; \exists x\, p(x,y)$$

On élimine $\forall y$ à gauche, avec un terme s_2 qu'il faudra préciser :

$$p(x,s_2)\,,\; \forall y\, p(x,y) \;\vdash\; p(s_1,y)\,,\; \exists x\, p(x,y)$$

Pour faire apparaître un axiome, on est conduit à égaler les formules atomiques $p(x\,,\,s_2)$ et $p(s_1\,,\,y)$. Pour cela, il faut trouver les valeurs à donner aux termes s_1 et s_2 de façon à avoir l'égalité syntaxique. Ici, une réponse est évidente, il faut prendre $s_1 = x$ et $s_2 = y$. Mais c'est contradictoire avec la condition $x \notin VarLib(s_1)$. On ne peut pas encore faire apparaître d'axiome à ce niveau. On applique à nouveau la règle d'élimination du $\exists$ à droite en introduisant un nouveau terme inconnu s_3 :

$$p(x\,,\,s_2)\,,\ \forall y\, p(x,y)\ \vdash\ p(s_1\,,\,y)\,,\ p(s_3\,,\,y)\,,\ \exists x\, p(x,y)$$

On essaye maintenant d'égaler syntaxiquement $p(x\,,\,s_2)$ et $p(s_3\,,\,y)$. Il vient $s_3 = x$ et $s_2 = y$ ce qui est acceptable puisqu'il n'y a pas de condition sur les termes s_2 et s_3. On a donc montré qu'avec ce choix on avait un axiome, ce qui termine la démonstration du séquent.

Stratégie de preuve

Notons que l'on a pu continuer après le premier échec, car l'élimination de $\exists$ à droite n'a pas fait disparaître la formule concernée de la partie droite, et permet donc de refaire plus tard une nouvelle élimination de $\exists$ à droite.

Il est intéressant de remarquer que l'on obtient une preuve plus directe si l'on choisit d'appliquer en priorité les règles d'élimination du $\forall$ à droite ou du $\exists$ à gauche car elles nécessitent une hypothèse sur la variable concernée. Avec cette nouvelle stratégie, on obtient la preuve suivante :

$$\exists x \forall y\, p(x,y)\ \vdash\ \forall y\, \exists x\, p(x,y)$$

Elimination de $\forall y$ à droite :

$$\exists x \forall y\, p(x,y)\ \vdash\ \exists x\, p(x,y)$$

On choisit en priorité d'éliminer le $\exists$ à gauche :

$$\forall y\, p(x,y)\ \vdash\ \exists x\, p(x,y)$$

Le choix entre l'élimination de $\exists x$ à droite ou de $\forall y$ à gauche est indifférent. Faisons en même temps les deux éliminations :

$$p(x\,,\,s_2)\,,\ \forall y\, p(x,y)\ \vdash\ p(s_1\,,\,y)\,,\ \exists x\, p(x,y)$$

Comme on n'a aucune condition sur les termes s_1 et s_2, il est maintenant légitime de prendre $s_1 = x$ et $s_2 = y$ pour obtenir un axiome.

Il est important d'avoir bien assimilé les preuves précédentes avant de lire le paragraphe concernant l'implantation.

Remarque 2 Il ne faut pas en conclure que cette stratégie réussit toujours. En général elle donne de bons résultats mais, dans certains cas, elle peut boucler. Ceci est lié à un résultat que nous avons déjà signalé : on sait qu'il n'y *pas* d'algorithme permettant de décider par oui ou par non si une formule est démontrable. Avec

l'implantation qui va suivre, on est assuré que si notre système répond `#t`, la formule considérée est un théorème ; mais elle peut boucler sans que cela implique que la formule soit fausse.

Exercice 5 *Les séquents suivants sont-ils démontrables ?*

1. $\forall y\, p(x,x) \vdash \forall x\, \exists y\, p(x,y)$
2. $\forall x\, p(x) \vee \forall y\, q(y) \vdash \exists z\, (p(z) \vee q(z))$
3. $\exists x\, p(x) \vdash p(y)$
4. $p(a), \forall x\, (p(x) \Rightarrow q(f(x))) \vdash \exists z\, q(z)$
5. $\forall x\, p(x, f(x)), \forall x y z\, (p(x,y) \wedge p(y,z) \Rightarrow q(x,z)) \vdash \forall z\, q(f(f(z)))$

19.7 Implantation du calcul des séquents classiques

On conserve la représentation d'un séquent par un prototype, décrite au chapitre 16. Et même mieux, on ne change pas la fonction de création de séquents ! En effet, l'absence de typage nous permet de réutiliser les trois champs d'un séquent dans le cas des formules du premier ordre. Le champ `LatomG` correspond à la liste des formules atomiques situées à gauche, de même pour le champ `LatomD`. Enfin, les formules signées sont constituées d'un signe et d'une formule du premier ordre non atomique. Ceci nous permet d'utiliser le travail déjà fait pour les connecteurs propositionnels. Il reste à le compléter par les règles relatives aux quantificateurs.

Contraintes pour l'utilisation des règles

La principale difficulté technique est la gestion des contraintes qui conditionnent l'utilisation des règles avec quantificateurs. On s'inspire d'une méthode implantée en ML par L. Paulson [Pau91].

- Pour les règles d'élimination de $\exists$ à droite et $\forall$ à gauche, le terme s introduit sera représenté par une méta-variable notée `?sj`. En effet, au moment où l'on introduit un tel terme, on ignore la valeur qu'il faudra lui donner pour que la preuve réussisse. Ce n'est qu'au moment où l'on essaye de trouver un axiome en égalant deux formules atomiques situées de part et d'autre de $\vdash$ que l'on peut être amené à expliciter ce type de termes. La notion de méta-variable permet de distinguer ces variables des variables du premier ordre des formules.

  Par ailleurs, pour satisfaire la condition d'admissibilité des substitutions du type $\varphi[x \leftarrow s]$, le terme s représenté par `?sj` ne doit pas contenir de variable pouvant être capturée par un quantificateur de φ. On satisfait à cette condition en excluant de s les variables liées de φ. On stocke ce type de conditions pour chaque méta-variable `?sj` dans une variable globale `*conditions*`.

- Pour les règles d'élimination de $\forall$ à droite et $\exists$ à gauche, on s'assure que la variable y introduite satisfait à la condition en générant une variable non déjà présente dans une formule. En particulier, cette variable ne devra pas apparaître dans un des termes représentés par les méta-variables déjà introduites, ce qui conduit à ajouter cette condition dans la liste des conditions pour ces méta-variables.

Pour représenter l'ensemble des variables ne devant pas apparaître dans les termes représentés par les méta-variables, la variable `*conditions*` a pour valeur une a-liste de la forme `((?s1 . Liste-de-variables-interdites-dans-s1) ...)`. Cette a-liste sera actualisée après chaque application d'une règle. Elle servira à vérifier si les valeurs des méta-variables, trouvées en égalant deux formules atomiques, satisfont aux conditions sur les variables des termes qu'elles représentent.

Reconnaissance d'un axiome

L'autre point nouveau est la reconnaissance des axiomes : il s'agit de voir si l'on peut trouver de part et d'autre du signe $\vdash$ des formules atomiques égales. En général, les formules atomiques contiennent des méta-variables, aussi le problème se ramène à essayer de donner aux méta-variables concernées des valeurs qui rendent identiques les deux prédicats. C'est un problème typique d'unification. Voyons un petit exemple.

Il s'agit de trouver les valeurs à donner aux méta-variables `?s1` et `?s2` pour que les formules atomiques suivantes soient égales :

```
(p ?s1 x2 (f ?s2)) = (p (f x1) ?s2 (f x2))
```

On doit donc unifier, en considérant les `?si` comme des variables et les variables du premier ordre `xj` comme des constantes, on les appelle des méta-constantes ! Il vient immédiatement les égalités :

`?s1 = (f x1)` et `?s2 = x2`

Il reste ensuite à vérifier si ces valeurs sont compatibles avec les conditions sur les `?si`. Par exemple, si `x1` est une variable interdite pour `?s1`, alors il y a échec dans l'égalisation des formules atomiques. Mais si les conditions sur les variables sont satisfaites, alors dorénavant les valeurs de `?s1` et `?s2` seront déterminées.
On stocke au fur et à mesure les liaisons trouvées dans une substitution globale `*metaSigma*`, que l'on actualisera chaque fois que l'on aura calculé les valeurs de nouvelles méta-variables.

Implantation du prédicat `axiome?`

Commençons par détailler le cas des axiomes. Pour tester si un séquent est un axiome, on cherche s'il y a une formule atomique à gauche et une à droite qui peuvent être rendues égales. C'est le prédicat `axiome?` qui remplace celui du chapitre 16 :

```
(define  (axiome? LatomG LatomD)
  (if  (or (null?  LatomG)(null?  LatomD))
```

```
    #f
    (some
      (lambda (atomG)
        (some (lambda (atomD)
                (Atome-egalisable? atomG atomD))
              LatomD))
      LatomG)))
```

Pour vérifier si deux formules sont égalisables, on commence par essayer de les unifier en ayant auparavant remplacé les méta-variables déjà calculées par leurs valeurs. Si on peut les unifier, on vérifie ensuite que la valeur trouvée pour chaque méta-variable est bien compatible avec l'éventuelle condition donnée par `*conditions*`. Si c'est le cas, on a bien un axiome et l'on augmente la substitution `*metaSigma*` des valeurs trouvées.

```
(define (Atome-egalisable? atomG atomD) ;; nouveau
  (let ((unificateur
         (unifier  (terme:substituer atomG *metaSigma*
                                     MetaVar? MetaCte?)
                   (terme:substituer atomD *metaSigma*
                                     MetaVar? MetaCte?)
                   MetaVar? MetaCte?)))

    (if (and unificateur (verifier-conditions unificateur))
      (begin (set! *metaSigma*
                   (compose-subst *metaSigma* unificateur
                                  MetaVar? MetaCte?))
             #t)
      #f)))
```

La vérification des conditions consiste à s'assurer que les expressions trouvées pour les méta-variables ne comportent pas de variables interdites.

```
(define (verifier-conditions unificateur)
  (every (lambda (var.val)
           (let ((var (car var.val))
                 (terme (cdr var.val)))
             (let ((var.Lvar-interdites (assq var *conditions*)))
               (if var.Lvar-interdites
                 (null? (intersection  (cdr var.Lvar-interdites)
                                       (terme:Lvar terme  Var?  Cte?)))

                 #t))))
         unificateur))
```

Variables logiques et méta-variables

L'utilisation, dans le même programme, de plusieurs espèces de variables explique pourquoi on avait prévu de passer en paramètre de certaines fonctions sur les termes, les prédicats de reconnaissance des variables et des constantes.
Les méta-variables sont des symboles qui commencent par un `?`.

```
(define (MetaVar? x)
  (and  (symbol? x)
        (eq? #\? (string-ref (symbol->string x) 0))))
```

On génère des méta-variables avec le générateur du chapitre 5 §7 :

```
(define genMetaVar (creer-genVar "?s"))

? (genMetaVar) -> ?s1
```

Une variable du premier ordre est un symbole qui n'est ni une méta-variable ni une constante :

```
(define (var? x)
  (and  (symbol? x)
        (not (memq x *Lconstantes*))
        (not (MetaVar? x))))
```

Les symboles de constantes du premier ordre sont les éléments d'une liste donnée par l'utilisateur ; pour les exemples qui suivront on prend la liste :

```
(define *Lconstantes* '(a b c d))
```

On a aussi besoin d'un générateur de variables du premier ordre :

```
(define  genVar (creer-genVar "x"))

? (genVar) -> x1
```

Une constante sera un élément de la liste `*Lconstantes*` ou une méta-variable :

```
(define  (cte? x)
  (or  (memq x *lconstantes*)
       (MetaVar? x)))
```

Enfin, une méta-constante est un symbole qui n'est pas une méta-variable :

```
(define  (MetaCte? x)
  (and  (symbol? x)
        (not (MetaVar? x))))
```

Règles pour l'élimination des quantificateurs

Il reste à traiter le cas des nouvelles règles associées aux quantificateurs. Ce qui conduit à modifier la fonction `traiter-sequent` du chapitre 16 en lui faisant appeler les nouvelles fonctions `appliquer-regle1G` et `appliquer-regle1D` au lieu de `appliquer-regleG` et `appliquer-regleD`.

```
(define (traiter-sequent sequent trace?)
  (bind (signe formule  No) (choix-formule (sequent 'Lsigne&formule))
    (sequent 'supprimer-formule  No)
    (if (eq? signe 'G)
```

```
    (appliquer-regle1G  sequent              ;;modification
                        (connecteur formule) formule
                        trace?)
    (appliquer-regle1D  sequent              ;;modification
                        (connecteur formule) formule
                        trace?))
  ))
```

Pour décider si une formule est composée, on doit modifier le prédicat `formule-compose?` du chapitre 16.

```
(define (formule-compose? s)
  (and (pair? s)(memq (connecteur s) '(et ou non imp tout ex))))
```

Si une formule composée commence par un connecteur propositionnel, on applique les fonctions définies au chapitre 16, sinon on discute selon la nature du quantificateur. La reconnaissance du cas propositionnel est faite par la fonction :

```
(define (propositionnel? connecteur)
  (memq connecteur '(et ou imp non)))
```

On rappelle la règle d'élimination du quantificateur universel à gauche :

$$\frac{\Gamma,\, \varphi[x \leftarrow s],\, \forall x\, \varphi \vdash \Delta}{\Gamma,\, \forall x\, \varphi \vdash \Delta}$$

Elle nous conduit à effectuer les opérations suivantes :

- on génère une nouvelle méta-variable pour représenter le terme à substituer ;
- on calcule la liste des variables liées dans le champ du quantificateur et, pour que la substitution soit admissible, on ajoute dans `*conditions*` la paire constituée de la nouvelle méta-variable et de la liste de variables à éviter ;
- on ajoute au séquent la formule obtenue en substituant, dans le champ du quantificateur, la méta-variable à la variable quantifiée ;
- enfin, on replace en queue des formules signées la formule qui vient d'être traitée.

On rappelle la règle d'élimination du quantificateur existentiel à gauche :

$$\frac{\Gamma,\, \varphi[x \leftarrow y] \vdash \Delta\,;\quad y \notin \mathit{VarLibre}(\Gamma, \Delta, \exists x\, \varphi)}{\Gamma,\, \exists x\, \varphi \vdash \Delta}$$

Elle nous conduit à effectuer les opérations suivantes :

- on génère une nouvelle variable y du premier ordre ;
- pour mériter le nom de nouvelle, elle ne devra pas apparaître dans l'expression des méta-variables existantes, aussi on l'ajoute comme variable interdite dans toutes les méta-variables actuelles ;

- enfin, on ajoute au séquent la formule obtenue en substituant, dans le champ du quantificateur, la nouvelle variable à la variable quantifiée existentiellement.

Tout ceci est réalisé par la fonction :

```
(define (appliquer-regle1G sequent connecteur  formule  trace?)
  (if (propositionnel? connecteur)
    (appliquer-regleG  sequent connecteur  formule  trace?)
    (let ((varQuantif (VarQuantifiee formule))
          (formule1 (champQuantificateur formule)))
      (when trace?
            (display-alln "regle du "  connecteur "  a gauche" #\newline))
      (case connecteur
        ((tout)
         (let ((Nlle-MetaVar (genMetaVar))
               (LvarInterdite (formule:LvarLiee formule1)))
           (ajouter-MetaVar-condition Nlle-MetaVar  LvarInterdite)
           (sequent 'ajouter-formule 'G
                    (formule:substituerVar formule1 varQuantif Nlle-MetaVar
                                           var? cte?))
           (sequent 'ajouter-formule 'G formule)
           (list sequent)))

        ((ex)
         (let ((Nlle-Var (genVar)))
           (ajouter-Var-condition Nlle-Var)
           (sequent 'ajouter-formule 'G
                    (formule:substituerVar formule1 varQuantif Nlle-Var
                                           var? cte?))
           (list sequent)))  ))))
```

On a utilisé les deux fonctions auxiliaires suivantes pour ajouter des nouvelles conditions dans `*conditions*` :

```
(define (ajouter-Var-condition var)
  (set! *conditions*
        (map (lambda (pp)(cons (car pp)(cons var (cdr pp))))
             *conditions*)))

(define (ajouter-MetaVar-condition metaVar Lvar-interdites)
  (set! *conditions*
        (cons (cons  metaVar  Lvar-interdites) *conditions*)))
```

On a la fonction analogue pour l'élimination à droite :

```
(define (appliquer-regle1D  sequent connecteur formule trace?)
  (if (propositionnel? connecteur)
    (appliquer-regleD  sequent connecteur  formule  trace?)
    (let ((varQuantif (VarQuantifiee formule))
          (formule1 (champQuantificateur formule)))
      (when trace?
```

```
          (display-alln "regle du "  connecteur "  a droite" #\newline))
      (case connecteur
        ((ex)
         (let ((Nlle-MetaVar (genMetaVar))
               (LvarInterdite (formule:LvarLiee formule1)))
           (ajouter-MetaVar-condition Nlle-MetaVar LvarInterdite)
           (sequent 'ajouter-formule 'D
                    (formule:substituerVar formule1 varQuantif Nlle-MetaVar
                                           var? cte?))
           (sequent 'ajouter-formule 'D     formule)
           (list sequent)))

        ((tout)
         (let ((Nlle-Var (genVar)))
           (ajouter-Var-condition Nlle-Var)
           (sequent 'ajouter-formule 'D
                    (formule:substituerVar formule1 varQuantif Nlle-Var
                                           var? cte?))
           (list sequent)))  ))))
```

On doit aussi modifier les priorités pour tenir compte des quantificateurs, on remplace la fonction `priorite` du chapitre 16 par :

```
(define (priorite signe&formule)
  (let ((son-signe (partie-signe signe&formule))
        (son-connecteur (connecteur (partie-formule signe&formule))))
    (if (eq? 'G son-signe)
      (case son-connecteur
        ((non)  5)
        ((et)   4)
        ((ou)   3)
        ((imp)  3)
        ((ex)   2)
        ((tout) 1))
      (case son-connecteur
        ((non)  5)
        ((et)   3)
        ((ou)   4)
        ((imp)  4)
        ((ex)   1)
        ((tout) 2) ))))
```

Les valeurs données aux quantificateurs tiennent compte de la stratégie de choix des règles décrite au paragraphe précédent.
Pour tester si une formule du premier ordre est démontrable, on appelle la fonction `demontrer-formule?` du chapitre 16 après avoir initialisé les variables globales `*metaSigma*` et `*conditions*`. On aurait pu remplacer ces deux variables globales par des paramètres mais cela aurait obligé à modifier trop de fonctions du chapitre 16.

```
(define (demontrer-formule-1erOrdre? formule trace?)
```

```
  (set! *metaSigma*  '())
  (set! *conditions* '())
  (demontrer-formule? formule trace?))
```

On ne présente pas la structure du système de preuve des séquents d'ordre1 car elle est pratiquement identique à celle des séquents propositionnels.

Quelques tests

Pour tester notre système, il faut d'abord charger le fichier correspondant à la preuve des séquents propositionnels, puis *toutes* les fonctions qui précèdent dans ce chapitre.

Essayons de prouver la formule $\exists y \, \forall x \, p(x,y) \;\Rightarrow\; \forall x \, \exists y \, p(x,y)$:

```
? (demontrer-formule-1erOrdre? '(imp (ex y (tout x (p x y)))
                                      (tout x (ex y (p x y)))) #t)


on traite le sequent :
|-   (imp (ex y (tout x (p x y))) (tout x (ex y (p x y))))
regle du imp  a droite

on traite le sequent :
  (ex y (tout x (p x y)))|-   (tout x (ex y (p x y)))
regle du ex  a gauche

on traite le sequent :
  (tout x (p x x1))|-   (tout x (ex y (p x y)))
regle du tout  a droite

on traite le sequent :
  (tout x (p x x1))|-   (ex y (p x2 y))
regle du tout  a gauche

on traite le sequent :
(p ?s1 x1)    (tout x (p x x1))|-   (ex y (p x2 y))
regle du ex  a droite

on traite le sequent :
(p ?s1 x1)    (tout x (p x x1))|- (p x2 ?s2)    (ex y (p x2 y))
c'est un axiome :

tout est demontré    #t
```

Considérons successivement le cas des formules :

$$\neg \exists x \neg p(x) \;\Rightarrow\; \forall y \, p(y)$$

$$\forall x \, \exists y \, p(x,y) \;\Rightarrow\; \exists y \, \forall x \, p(x,y)$$

$$\forall x\, p(x,x) \Rightarrow \forall y\, \exists x\, p(x,y)$$

```
? (demontrer-formule-1erOrdre?
   '(imp (non (ex x (non (p x)))) (tout y (p y)) #f) -> #t

? (demontrer-formule-1erOrdre?
   '(imp (tout x (ex y (p x y))) (ex y (tout x (p x y))))#f) -> boucle

? (demontrer-formule-1erOrdre?
   '(imp (tout x (p x x)) (tout y (ex x (p x y)))) #f) ->#t
```

On peut aussi donner directement un séquent à prouver, pour démontrer :

$$p(a)\,,\, \forall x\, (p(x) \Rightarrow q(f(x))) \vdash \exists z\, q(z)$$

on utilise la fonction `Demontrer-sequent?` :

```
? (Demontrer-sequent?
   '((p a) (tout x  (imp (p x) (q (f x)))))
   '((ex z (q z)))
   #f)
#t
```

Il est intéressant de remarquer que si l'on demande la valeur de `*metaSigma*` à la fin de cette preuve, on apprend que la variable existentielle z a été remplacée par le terme $(f\ a)$.

On peut obtenir la preuve d'un séquent complexe comme :

$$\forall x\, p(x, f(x))\,,\, \forall x\, y\, z\, (p(x,y) \wedge p(y,z) \Rightarrow q(x,z)) \vdash \forall z\, q(f(f(z)))$$

```
? (Demontrer-sequent?
   '((tout x (p x (f x)))
     (tout x (tout y (tout z (imp  (et (p x y) (p y z))(q x z))))))
   '((tout z (q z (f (f z))))
   #f)
#t
```

Lien avec la recherche d'une preuve avec Prolog

Le principe du chaînage arrière utilisé en Prolog peut se retrouver à partir de cette implantation du calcul des séquents. Chaque règle Prolog :

$$\theta_0 \Leftarrow \theta_1 \wedge \ldots \wedge \theta_n$$

est en fait implicitement quantifiée universellement, elle correspond à la formule du premier ordre

$$\forall x_1, \ldots, x_k\, (\theta_1 \wedge \ldots \wedge \theta_n \Rightarrow \theta_0)$$

Un but p correspond à une formule quantifiée existentiellement $\exists x_1, \ldots, x_k\ p$. Appelons Γ la base de règles et $p(x)$ le but à prouver[2]. Il s'agit de démontrer le jugement :

$$\Gamma \vdash \exists x\, p(x)$$

En éliminant $\exists x$, on est ramené à trouver un terme s (sc sera une valeur pour x) tel que :

$$\Gamma \vdash p([x \leftarrow s])$$

Il y a deux cas :

- soit on trouve dans Γ une formule atomique (un fait) de la forme $\forall z\ p(z)$, alors on élimine $\forall z$ en remplaçant z par un terme t et l'on est amené à unifier s et t pour exhiber un axiome,
- soit on trouve dans Γ une règle de la forme $\forall y\ (h_1 \wedge \ldots \wedge h_n \Rightarrow p)$ alors on élimine $\forall y$ par un terme u, et il vient le séquent :

  $$\Gamma,\ (h_1\ \wedge \ldots\ \wedge h_n \Rightarrow p)[y\ \leftarrow\ u]\ \vdash\ p([x\ \leftarrow\ s])$$

  Puis après élimination de l'implication à gauche, on est ramené à prouver les deux séquents :

  $$\Gamma,\ p([y \leftarrow u]) \vdash p([x \leftarrow s]) \quad \text{et} \quad \Gamma \vdash (h_1 \wedge \ldots \wedge h_n)[y \leftarrow u],\ p([x \leftarrow s])$$

  Pour le premier, on trouve un axiome si l'on peut unifier les termes u et s ; ensuite le second correspond à la preuve des hypothèses de la règle utilisée. On reconnaît la méthode utilisée pour le moteur Prolog. Notons que l'on n'a utilisé que des règles appartenant au calcul des séquents intuitionnistes.

19.8 Logiques égalitaires

Beaucoup de théories, et surtout en mathématiques, font jouer un rôle crucial au prédicat d'égalité. Pour modéliser en logique les raisonnements égalitaires, on introduit la notion de *logique égalitaire*.

Règles de la logique égalitaire

Une logique égalitaire est une logique qui comporte parmi ses prédicats un prédicat binaire, souvent noté $=$, qui est réflexif, symétrique et transitif :

$$\frac{}{\vdash x = x} \qquad \frac{\Gamma \vdash x = y}{\Gamma \vdash y = x} \qquad \frac{\Gamma \vdash x = y\ ,\ \Gamma \vdash y = z}{\Gamma \vdash x = z}$$

De plus, pour tout symbole de fonction f d'arité n on a :

[2] Pour alléger les notations, on ne considère qu'une seule variable x dans le but.

$$\frac{\Gamma \vdash x_1 = y_1, \ldots, \Gamma \vdash x_n = y_n}{\Gamma \vdash f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)}$$

et pour tout symbole de relation p d'arité n on a :

$$\frac{\Gamma \vdash x_1 = y_1, \ldots, \Gamma \vdash x_n = y_n}{\Gamma \vdash p(x_1, \ldots, x_n) \Rightarrow p(y_1, \ldots, y_n)}$$

Ces dernières règles traduisent l'idée que l'on peut remplacer des égaux par des égaux sans rien changer à la prouvabilité d'une formule. La formulation est identique en déduction naturelle et en calcul des séquents.

On écrit $\Gamma \vdash_= \varphi$ pour dire que la formule φ est démontrable sous les hypothèses Γ et les règles supplémentaires précédentes. On omet souvent l'indice $=$ quand il n'y a pas de confusion possible.

Voici deux exemples typiques de théorie égalitaire.

La théorie des groupes

C'est une théorie égalitaire basée sur un langage contenant un symbole de fonction binaire noté $\circ$, appelé la loi de composition du groupe, et un symbole de constante noté e, appelé élément neutre. Cette théorie contient les trois axiomes suivants :

$\forall x, y, z \quad x \circ (y \circ z) = (x \circ y) \circ z$; associativité de $\circ$ écrit en infixe
$\forall x \quad x \circ e = e \circ x = e$; neutralité de e
$\forall x\, \exists x' \quad x \circ x' = x' \circ x = e$; chaque élément x admet un inverse x'

Si l'on appelle G ces trois formules, un théorème de la théorie des groupes est une formule φ de ce langage démontrable en logique égalitaire sous les hypothèses G :

$$G \vdash_= \varphi$$

Par exemple, l'implication :

$$\forall x, y, z \quad (y \circ x = z \circ x \Rightarrow y = z)$$

est un théorème de la théorie des groupes. Pour alléger les notations, on écrit sa démonstration dans le style mathématique usuel ; le lecteur pourra la transcrire dans le formalisme de la déduction naturelle.
Pour prouver cette formule, on la démontre pour des variables x, y, z quelconques (élimination des quantificateurs universels). Pour démontrer l'implication $y \circ x = z \circ x \Rightarrow y = z$, on suppose $y \circ x = z \circ x$ et on montre $y = z$ (élimination de $\Rightarrow$). Etant donné x, le troisième axiome montre (élimination des quantificateurs $\forall$ et $\exists$) l'existence d'un x tel que $x \circ x' = e$. On peut composer à droite l'égalité $y \circ x = z \circ x$ par x', il vient l'égalité $(y \circ x) \circ x' = (z \circ x) \circ x'$ (logique égalitaire). Le premier axiome d'associativité permet d'en déduire l'égalité $y \circ (x \circ x)' = z \circ (x \circ x')$. Enfin, le choix de x' assure que l'on peut remplacer $x \circ x'$ par e d'où, en tenant compte de la neutralité de e, l'égalité cherchée $y = z$.

Exercice 6 *En déduire l'unicité de l'élément inverse.*

Le deuxième exemple est plus important pour le lecteur informaticien.

L'arithmétique de Peano

Il s'agit d'une théorie égalitaire dont le langage contient les symboles de fonction suivants :

- deux symboles d'arité 2 notés $+$ et $*$
- un symbole d'arité 1 noté S (la fonction successeur)
- un symbole de constante noté 0

On définit les axiomes suivants :

$$
\begin{array}{lll}
(P1) & \forall x & \neg (Sx = 0) \\
(P2) & \forall x, y & Sx = Sy \Rightarrow x = y \\
(P3) & \forall x & x + 0 = x \\
(P4) & \forall x, y & x + Sy = S(x + y) \\
(P5) & \forall x & x * 0 = 0 \\
(P6) & \forall x, y & x * Sy = x + x * y
\end{array}
$$

Le premier axiome dit que 0 n'est pas un successeur et le second signifie que S est injective.
Les quatre axiomes suivants sont des égalités déjà introduites au chapitre 17 pour l'addition et la multiplication des entiers de Peano.
On peut démontrer avec ces axiomes des égalités du genre :

$$S0 + S0 = S(S0))$$

On a prouvé ce type d'égalité au chapitre 17 avec la technique de la réécriture, mais la réécriture n'est qu'une implantation (partielle) du raisonnement égalitaire. Elle est partielle, car l'orientation des règles de réécriture fait que l'on n'utilise les égalités que dans un sens. La réécriture reste néanmoins l'un des meilleurs outils pour faire de la déduction automatique dans les théories égalitaires.
Pour le moment, on ignore si l'addition est commutative, et même si plus simplement on a :

$$\forall x \quad 0 + x = x$$

En fait, on peut prouver que cette formule ne découle *pas* des axiomes précédents ; il nous manque le célèbre axiome du *raisonnement par récurrence.* Il s'énonce :

$$(P7) \quad \varphi(0) \wedge \forall x\, (\varphi(x) \Rightarrow \varphi(Sx)) \quad \Rightarrow \quad \forall x\, \varphi(x)$$

où φ est une formule du langage ayant au moins une variable libre x. Ce n'est donc pas un axiome mais une infinité (dénombrable) d'axiomes, un par formule φ possible.
Démontrons maintenant que l'on a $\forall x \quad 0 + x = x$. Il suffit d'appliquer le raisonnement par récurrence en prenant pour φ la formule $0 + x = x$.
Pour pouvoir utiliser l'implication de (P7), on doit prouver que l'on a :

$$\varphi(0) \quad \text{et que} \quad \forall x\, (\varphi(x) \Rightarrow \varphi(Sx))$$

Ce sont les deux étapes habituelles du raisonnement par récurrence.
La première découle de l'axiome (P3) car elle s'écrit $0 + 0 = 0$.

Pour prouver la deuxième, on suppose $0+x=x$ et on doit en déduire $0+Sx=Sx$. Mais par (P4) on obtient $0+S(x)=S(0+x)$ et comme on a supposé $0+x=x$, on en conclut bien que $0+S(x)=S(x)$.

Exercice 7 *Prouver par récurrence les égalités :*
$\forall x, y \quad x+y=y+x$
$\forall x, y, z \quad x+(y+z)=(x+y)+z$

L'ensemble P des axiomes (P1) à (P7) s'appelle les axiomes de Peano. Les formules φ que l'on peut démontrer $P \vdash_= \varphi$, s'appellent les théorèmes de l'arithmétique de Peano.

Le choix des axiomes permet de montrer que tous les théorèmes de l'arithmétique de Peano sont des théorèmes vrais pour les entiers usuels **N**. Mais la réciproque est-elle vraie ? Est-ce que toute formule arithmétique vraie pour les entiers **N** est démontrable à partir des axiomes de Peano ? La réponse est *non* ! On peut penser qu'il suffit d'ajouter de nouveaux axiomes pour élargir le champ des théorèmes. Mais on démontre qu'aucune extension de ce système ne permettra de démontrer tous les théorèmes vrais sur les entiers **N**. C'est l'un des célèbres résultats d'incomplétude démontré par le logicien K. Gödel.
De plus, on démontre que l'arithmétique de Peano est indécidable. C'est-à-dire qu'il n'existe pas d'algorithme pour décider si une formule φ est ou n'est pas démontrable dans ce système. C'est sur ces mauvaises nouvelles pour le programmeur que l'on termine cette courte introduction à la logique.

19.9 De la lecture

Quelques livres de logique pour lecteurs informaticiens : [Gal86, Mar89, Lal90, MW93].

Chapitre 20

Introduction au lambda calcul

N dit parfois qu'un langage fonctionnel n'est que du sucre syntaxique sur le lambda calcul. Dans ce chapitre on va donner une initiation au lambda calcul et montrer son lien avec la programmation.
Le lambda calcul a été créé par le logicien A. Church dans les années 30 pour des questions de fondement en logique mathématique. Ce n'est que bien plus tard que ce formalisme s'est révélé être parfaitement adapté à l'étude des mécanismes d'évaluation en programmation. Il s'agit d'un système avec un nombre réduit de règles mais ayant un grand pouvoir d'expression : il permet de coder toutes les fonctions récursives. Le lambda calcul considère une fonction comme une méthode de calcul, contrairement à l'approche traditionnelle des mathématiques où une fonction est vue comme un ensemble de points en relation.
On distingue deux aspects du lambda calcul : le lambda calcul pur et le lambda calcul typé ; chacun de ces aspects peut servir de point de départ pour représenter l'autre. On commencera par le lambda calcul pur, le cas typé sera ensuite présenté comme une sous-théorie. Le lambda calcul pur permet d'étudier les différentes stratégies d'évaluation d'une expression. Le lambda calcul typé donne un cadre pour étudier le typage des expressions. On donnera une méthode pour inférer le type d'une lambda expression et l'on montrera le lien entre l'inférence de type et la déduction naturelle.

20.1 Définition du lambda calcul

Syntaxe concrète

La syntaxe concrète des lambda termes est définie par la donnée d'un ensemble *Var* de variables, d'une opération *d'abstraction* désignée par le symbole λ et d'une opération *d'application*. Plus précisément, on pose :

Définition 1 On dit que t est un lambda terme si :
- $t = x$ où x est dans l'ensemble de variables Var,

ou

- $t = \lambda\, x\, .\, M$ où x est dans Var et M est un lambda terme,

ou

- $t = (M\ N)$ où M et N sont des lambda termes.

Exemples de lambda termes :

- $(x\ x)$
- $\lambda\, x\, .\, \lambda y\, .\, (x\ (y\ z))$
- $\lambda x\, .\, ((x\ y)(x\ x))$
- $((\lambda\, x\, .\, (x\, x))(\lambda\, x\, .\, (x\, x)))$

Exercice 1 *Définir les lambda termes par une grammaire sur l'alphabet :* $\{Var\, ,\, (\, ,\,)\, ,\, .\}$

Intuitivement, l'abstraction $\lambda\, x\, .\, M$ représente la fonction $x \rightarrow M$ et $(M\ N)$ représente l'application de M à l'argument N.

Remarque 1 La notation $\lambda\ x\, .\, M$ pour désigner une fonction anonyme est à l'origine de la notion de lambda expression en Scheme.

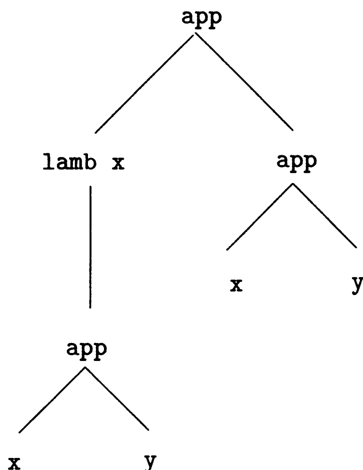
Notons que l'abstraction porte toujours sur une seule variable. Si l'on a besoin de représenter des fonctions de plusieurs variables $x_1, \ldots, x_n \rightarrow M$, on utilisera la curryfication pour se ramener au cas d'un variable : $\lambda\, x_1\, .\, (\lambda\, x_2\, .\, \ldots (\lambda\, x_n\, .\, M) \ldots)$. On verra que l'idée intuitive de fonction est trop restrictive mais c'est néanmoins un bon fil conducteur.

Pour simplifier les écritures, on utilisera parfois les conventions suivantes :
- l'abstraction est associative à droite : $\lambda x\, .\, \lambda y\, .\, \lambda z\, .\, M$ peut aussi se noter $\lambda\, x\, y\, z\, .\, M$,
- l'application est associative à gauche : $((M\ N)\ P)$ peut aussi se noter $M\, N\, P$.

Par exemple, le terme $\lambda\, x\, .\, \lambda y\, .\, ((x\ y)\ z)$ peut aussi s'écrire $\lambda\, x\, y\, .\, x\, y\, z$.

Syntaxe abstraite des lambda termes

La syntaxe abstraite peut être décrite avec la notion usuelle de terme. On se donne un symbole de fonction binaire, l'application, notée `app` et un lieur, `lamb`, qui à chaque variable `x` associe un symbole de fonction unaire `lamb x`. Ainsi, le lambda terme $(\lambda\, x\, .\, (x\ y)(x\ x))$ a pour arbre de syntaxe abstraite :



Représentation en Scheme des lambda termes

Pour définir une représentation des lambda termes en Scheme, on convient de désigner par $\overline{M}$ la représentation du lambda terme M. Alors :

- une variable x est représentée par le symbole `x`,
- une abstraction $\lambda x \,.\, M$ est représentée par la liste `(lamb x` $\overline{M}$`)`,
- une application $(M\ N)$ est représentée par la liste `(`$\overline{M}\ \ \overline{M}$`)`.

```
(define  (lambda:variable? s)
  (symbol? s))
```

Pour l'abstraction on a un prédicat, deux accesseurs et un constructeur :

```
(define (abstraction? lambdaTerme)
  (and (pair? lambdaTerme)(eq? 'lamb (car lambdaTerme))))

(define  (varAbstrac  abstr) ;
  (cadr abstr))

(define  (corpsAbstrac abstr)
  (caddr abstr))

(define (make-abstraction var corps)
  (list 'lamb var corps))
```

Pour l'application on a un prédicat, deux accesseurs et un constructeur :

```
(define (application?  lambdaTerme)
  (and (pair? lambdaTerme) (= 2 (length lambdaTerme))))

(define (operator app)
  (car app))
```

```
(define (operand app)
  (cadr app))

(define  (make-application operator operand)
  (list operator operand))
```

Le lieur λ se comporte exactement comme un quantificateur en logique : on a les mêmes notions de champ, d'occurrence libre et d'occurrence liée. Un lambda terme *clos*, c'est-à-dire sans occurrence libre de variable, s'appelle un *combinateur*.

```
(define (Lambda:Libres lambdaTerme)
  (cond ((lambda:variable?  lambdaTerme)(list lambdaTerme))
        ((application? lambdaTerme)
         (union (Lambda:Libres (operator lambdaTerme))
                (Lambda:Libres (operand lambdaTerme))))
        (else
         (remove (varAbstrac lambdaTerme)
                 (Lambda:Libres (corpsAbstrac lambdaTerme))))))

? (Lambda:Libres '( (lamb x  (x y)) (z y) ))  -> ( z y)


(define (lambda:Liees lambdaTerme)
  (cond ((lambda:variable?  lambdaTerme) '())
        ((application? lambdaTerme)
         (union (lambda:Liees (operator lambdaTerme))
                (lambda:Liees (operand lambdaTerme))))
        (else
         (cons (varAbstrac lambdaTerme)
               (lambda:Liees (corpsAbstrac lambdaTerme))))))

? (lambda:Liees '( (lamb x  (x y)) (z y) )) -> (x)
```

Exercice 2 *En utilisant les techniques des chapitres 13 et 14, écrire un analyseur syntaxique et un afficheur pour les lambda termes.*

20.2 Bêta réduction

Le lambda calcul est, comme son nom l'indique, un *calcul* ; il repose essentiellement sur la notion de substitution. L'opération de base s'appelle la *bêta réduction.* Mais avant de définir cette opération, on doit prendre des précautions avec le phénomène sournois de la capture de variable au moment d'une substitution.

Alpha conversion

Pour évaluer un lambda terme, le mécanisme de base est fondé sur la notion de substitution. Comme en logique du premier ordre, on peut définir le lambda terme $M[x \leftarrow N]$ résultant du remplacement dans M de toutes les occurrences *libres*

d'une variable x par le lambda terme N. Mais on a vu que la notion de lieur amène aussi le phénomène de capture. Par exemple, la substitution :

$$(\lambda\, x\,.\,(x\; y)[y \leftarrow\; (z\; x)] = (\lambda\, x\,.\,(x\; (z\; x))$$

entraîne la capture par λ de la variable x qui était libre dans $(z\; x)$. Pour éviter cette capture, on est amené à renommer les variables liées ce qui conduit à considérer comme identiques deux lambda termes qui se correspondent par un tel renommage. Après renommage de la variable liée x en $x0$, la substitution :

$$(\lambda\, x0\,.\,(x0\; y)[y \leftarrow\; (z\; x)] = (\lambda\, x\,.\,(x0\; (z\; x))$$

s'effectue sans capture.
Ce renommage s'appelle une *alpha conversion.*

Définition 2 L'alpha conversion d'une abstraction $\lambda\, x\,.\,M$ est définie par : $\lambda\, x0\,.\,M[x \leftarrow\; x0]$ où $x0$ n'est pas une variable libre de M.

On dit que deux lambda termes M et N sont égaux modulo alpha, s'ils sont égaux ou bien si l'on passe de l'un à l'autre par une suite d'alpha conversions (éventuellement dans des sous-termes) ; on le note $M\; =_\alpha\; N$.
Dorénavant, on considérera les lambda termes modulo alpha conversion, autrement dit, on se place dans l'espace quotient des lambda termes modulo $=_\alpha$. On définit un lambda terme par la donnée d'un représentant de sa classe.

Définition 3 La substitution sera dite une *lambda substitution* $M[x \leftarrow\; N]$ si elle est faite après un renommage éventuel de M pour éviter la capture des variables libres de N.

Bien entendu, on vérifie que la classe du lambda terme résultant de la substitution est indépendante du choix des renommages de M et de N. En langage plus mathématique, on dit que la lambda substitution passe au quotient.

Il est facile de programmer la lambda substitution, il suffit de s'assurer que pour substituer dans une abstraction, on renomme la variable liée si elle est libre dans le terme à substituer :

```
(define (Lambda:Substituer exp var terme)
  (cond ((lambda:variable?  exp)(if (eq? exp var) terme exp))
        ((application? exp)
         (make-application (Lambda:Substituer (operator exp) var terme)
                           (Lambda:Substituer (operand exp)  var terme)))
        (else
         (if (not (member var (Lambda:Libres exp)))
           exp                    ;; pas d'occurrence libre on ne fait rien
           (let ((var0  (varAbstrac exp))
                 (corps (corpsAbstrac exp)))
             (if (memq var0 (Lambda:Libres  terme));; si capture  on renomme
               (let* ((newVar0 (renomme  var0 (Lambda:Libres terme) ))
                      (new-corps (Lambda:Substituer corps var0 newVar0)))
                 (make-abstraction newVar0
                                   (Lambda:Substituer new-corps var terme)))
               (make-abstraction var0            ;; si absence de capture
                                 (Lambda:Substituer corps var terme))))))))
```

On a utilisé une fonction `renomme`; cette fonction ajoute, si besoin, un suffixe entier à une variable pour qu'elle ne soit pas dans la liste `ListeVar`:

```
(define (renomme  var ListeVar)
  (let ((suffixer (lambda (symb j)
                    (string->symbol (string-append (symbol->string symb)
                                                   (number->string j))))))
    (letrec ((renommeAux
              (lambda (j)
                (let ((varj (suffixer var j)))
                  (if (member varj ListeVar)
                    (renommeAux (1+ j))
                    varj)))))
      (renommeAux 0))))

? (renomme  'x  '(x x0 x1 x3))
x2
```

Guidé par l'analogie entre d'une part, abstraction et fonction anonyme et d'autre part, application et appel de fonction, on va définir l'opération de base sur les lambda termes.

Définition de la bêta réduction

L'application d'une abstraction à un terme s'appelle un *redex* (contraction de *red*ucible *ex*pression)

$$((\lambda x\,.\,M)\;N)$$

L'analogie avec l'appel de fonction nous conduit à définir le lambda terme résultant comme étant ce que l'on obtient après remplacement de toutes les occurrences libres de x dans M par l'argument N. D'où la

Définition 4 On appelle β-réduction la transformation

$$((\lambda x\,.\,M)\;N)\;\rightarrow_\beta\; M[x \leftarrow N]$$

On montre qu'elle est bien définie pour les lambda termes à alpha conversion près.

Exemples:

- $\lambda x\,.\,x\;x)(\lambda y\,.\,y))\;\rightarrow_\beta\;(\lambda y\,.\,y)(\lambda y\,.\,y))\;\rightarrow_\beta \lambda y\,.\,y)$
- $((\lambda x\,.\,(\lambda y\,.\,x\;y))\;u\;y)\;\rightarrow_\beta\;(\lambda y0\,.\,(u\;y)\;y0))$
  (noter le renommage en $y0$ de la variable liée y)

Pour reconnaître un redex, on utilise le prédicat `redex?`:

```
(define (redex? terme)
  (and (application? terme)
       (abstraction? (operator terme))))
```

La bêta réduction d'un redex se ramène à une substitution:

```
(define  (beta-reduc-redex redex)
  (let* ((abs (operator redex))
         (opd (operand redex))
         (var (varAbstrac abs))
         (corps (corpsAbstrac abs)))
    (Lambda:Substituer corps var opd)))

? (beta-reduc-redex  '((lamb x  (lamb y  (x y)))  (u y )))
(lamb y0 ((u y) y0))
```

Soient deux lambda termes M et N, si l'on peut passer de M à N par une succession de bêta réductions (éventuellement dans des sous-termes de M), on écrit $M \rightarrow *_\beta N$ et si l'on peut passer de l'un à l'autre par une suite de bêta réductions dans des sens quelconques, on écrit $M =_\beta N$.

Remarque 2 On considère parfois en plus la η−réduction. Un η−redex est une abstraction de la forme $\lambda x . (M\ x)$ où x n'est pas une variable libre du terme M. La η−réduction consiste à le transformer en le terme M :

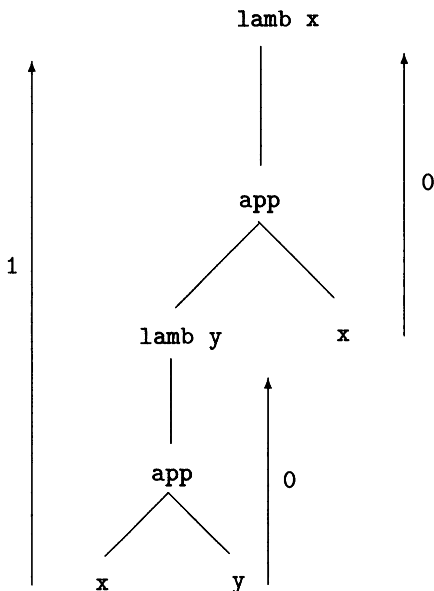
$$(\lambda x . M\, x) \rightarrow_\eta M \quad , \quad x \text{ non libre dans } M$$

Cette réduction consiste intuitivement à assimiler une fonction $x \rightarrow f(x)$ avec f.

Notation de de Bruijn

Le phénomène de capture vient du fait que l'on utilise des noms pour représenter des liaisons. Aussi, N. de Bruijn a proposé une notation pour représenter de façon canonique la classe d'équivalence modulo alpha d'un lambda terme.
Considérons un lambda terme sans variable libre, chaque variable liée sert en fait à repérer le λ qui la lie. On peut retrouver cette information en remplaçant chaque variable par un entier qui indique le nombre de λ à remonter dans son arbre de syntaxe pour arriver à celui qui la lie.

Par exemple, le terme $(\lambda x . ((\lambda y . x\ y)\, x)))$ a pour arbre :



On a représenté par des flèches les liaisons entre une variable et son lieur λ.
En remplaçant dans le corps de chaque abstraction les variables par ces hauteurs de liaisons, on obtient pour ce terme la forme : $(\lambda\,.\,((\lambda\,.\,1\ 0)\,0))$

Voici une fonction pour faire la traduction en notation de de Bruijn :

```
(define  (terme->deBruijn  lambda-terme env-hauteur)
  (cond ((lambda:variable?  lambda-terme)
         (cdr (assoc lambda-terme env-hauteur)))
        ((application? lambda-terme)
         (make-application (terme->deBruijn (operator lambda-terme)
                                                        env-hauteur)
                           (terme->deBruijn (operand lambda-terme)
                                                       env-hauteur)))
        (else
         (let ((var    (varAbstrac   lambda-terme))
               (corps (corpsAbstrac lambda-terme)))
           (list 'lamb
                 (terme->deBruijn  corps (ajouter-lamb var env-hauteur)))))))
```

Elle prend en paramètre un environnement qui associe à chaque variable sa hauteur de liaison. Quand on traverse une abstraction, il faut ajouter à l'environnement une nouvelle variable et incrémenter de 1 les hauteurs des anciennes. C'est la fonction :

```
(define  (ajouter-lamb var env)
  (acons var 0
         (map  (lambda (pp)(cons (car pp) (1+ (cdr pp))))
               env)))
```

```
? (terme->deBruijn  '(lamb x ((lamb y (x y)) x)) '())
(lamb ((lamb 1 0) 0))

? (terme->deBruijn  '(lamb x (lamb y (( x y)(lamb y (( x y) y))))) '())
(lamb (lamb ((1 0) (lamb ((2 0) 0)))))
```

Cette représentation est tout à fait indiquée pour servir de représentation interne car on n'a plus besoin de renommer. Mais elle est moins parlante pour le lecteur humain, aussi conservera-t-on la notation traditionnelle.

Remarque 3 Quand il y a des variables libres, on peut se ramener au cas des termes clos en considérant que les variables libres sont en fait liées par un λ qui n'est pas explicité.

20.3 Stratégies de réduction

Notion générale de réduction

On étend la bêta réduction en permettant de l'effectuer dans un sous-terme. On peut décrire cette notion de β–réduction par des règles d'inférence. On a un axiome et trois règles :

$$\frac{}{((\lambda x\,.\,M)\;N) \to M[x \leftarrow N]} \quad (\beta)$$

$$\frac{M \;\to\; M_1}{(M\,N) \;\to\; (M_1\,N)} \qquad \frac{N \;\to\; N_1}{(M\,N) \;\to\; (M\,N_1)} \qquad \frac{M \;\to\; M_1}{(\lambda x\,.\,M) \;\to\; (\lambda x\,.\,M_1)}$$

Un lambda terme sans redex est dit *irréductible*. Une suite de β-réductions de M en un terme N: $M \to *_\beta \; N$ s'appelle une *réduction* de M. Ces notions suggèrent les questions suivantes :

- est-ce que tout terme peut se réduire en un terme irréductible?
- si un terme peut se réduire en un terme irréductible, ce dernier est-il unique?

Pour réduire un terme, il suffit de réduire tous ses redex. Mais ce n'est pas si simple, car la réduction d'un redex peut en faire apparaître de nouveaux. Par ailleurs, quand il y a plusieurs redex, on doit choisir celui à réduire en premier, ce choix s'appelle une *stratégie de réduction.*

Commençons par faire quelques expériences.

• Considérons toutes les suites de réductions pour le terme $T1 = (\lambda x\,.\,(\lambda y\,.\,(x\;y)\;u)\;z)$ On souligne le redex à réduire :

$$\underline{(\lambda x\,.\,(\lambda y\,.\,(x\;y)\;u)\;z)} \;\to_\beta\; \underline{(\lambda y\,.\,(z\;y)\,u)} \;\to_\beta\; (zu)$$

Mais on aurait pu commencer par réduire à l'intérieur :

$$(\lambda x . \underline{(\lambda y . ((x\ y)\ u)\ z)} \to_\beta \underline{\lambda x . ((x\ u)\ z)} \to_\beta (zu)$$

On constate ici qu'il y a une forme irréductible et qu'elle ne dépend pas de la stratégie de réduction.

• Appelons Δ le terme $((\lambda y . y\ y)\ (\lambda y . y\ y))$. Il n'a qu'un seul redex, on trouve $\Delta \to_\beta \Delta$, ce qui permet de le réduire indéfiniment, donc Δ n'admet pas de forme irréductible.

• Posons $T2 = ((\lambda x . z)\ \Delta)$. Si l'on réduit le redex extérieur, on obtient une forme irréductible $\underline{((\lambda x . z)\ \Delta)} \to_\beta z$.

En revanche, on peut réduire indéfiniment Δ :

$$((\lambda x . z)\ \underline{\Delta}) \to_\beta ((\lambda x . z)\ \underline{\Delta}) \to_\beta \dots$$

Ces exemples conduisent à poser :

Définition 5 Un lambda terme est *normalisable* s'il admet une suite de bêta réductions qui termine. Un lambda terme est *fortement normalisable* si toutes ses suites de bêta réductions terminent.

On peut donc dire que $T1$ est fortement normalisable, que $T2$ est normalisable et que Δ n'est pas normalisable.
On a vu que les formes réduites du terme $T1$ étaient identiques, ceci est général car on démontre le théorème dit de confluence :

Théorème 1 Toutes les réductions d'un lambda terme qui terminent, aboutissent au même terme irréductible.

La question qui se pose ensuite est : si un terme est normalisable, est-ce qu'il existe une stratégie pour choisir les réductions à effectuer de façon à aboutir à la forme irréductible ?

Réduction normale et réduction par valeur

On définit une stratégie par la donnée de la position de redex à réduire en premier. Un redex est dit extérieur s'il n'est pas sous terme d'un autre redex, un redex est dit *normal* si c'est le plus à gauche des extérieurs. La stratégie qui consiste à toujours réduire le redex normal s'appelle la *réduction normale*. Pour décrire cette stratégie, il est plus clair d'indiquer l'ordre dans lequel on cherche à appliquer les règles de réduction d'une *application* : on réduit en priorité le redex de tête ; s'il n'y en a pas, on applique la même stratégie dans l'opérateur et sinon dans l'opérande :

$$\frac{}{((\lambda x . M)\ N) \to M[x \leftarrow N]}$$

$$\frac{M \to M_1}{(M\ N) \to (M_1\ N)}$$

$$\frac{N \rightarrow N_1}{(M\,N) \rightarrow (M\,N_1)}$$

L'importance de la stratégie normale provient du théorème suivant :

Théorème 2 La réduction normale appliquée à un terme normalisable aboutit toujours à la forme irréductible du terme.

C'est la stratégie qui a été utilisée pour réduire le terme $T2$. Une autre stratégie qui consiste à réduire en priorité les arguments avant le redex de tête s'appelle la *réduction par valeur* :

$$\frac{N \rightarrow N_1}{(M\,N) \rightarrow (M\,N_1)}$$

$$\frac{M \rightarrow M_1}{(M\,N) \rightarrow (M_1\,N)}$$

$$\frac{}{((\lambda x\,.\,M)\,N) \rightarrow M[x \leftarrow N]}$$

On peut évidemment imaginer de nombreuses autres stratégies. On ignore s'il existe une stratégie optimale, c'est-à-dire qui termine toujours pour un terme normalisable et qui fait, en un certain sens, un nombre minimal de réductions.

Implantation des deux stratégies

Pour comparer ces stratégies sur des exemples plus compliqués, on commence par les implanter. Pour ces deux stratégies, on définit une fonction de réduction en *un coup*. Cette fonction rend le terme réduit ou **#f** s'il est irréductible. Ces fonctions sont la transcription des règles précédentes.

```
(define (reduc1-normale terme)
  (cond ((lambda:variable?  terme) #f)
        ((abstraction? terme)
         (let ((reduc-corps (reduc1-normale (corpsAbstrac terme))))
           (if reduc-corps
             (make-abstraction (varAbstrac terme) reduc-corps)
             #f)))
        ((redex? terme)(beta-reduc-redex terme))
        (else (let ((reduc-operator (reduc1-normale (operator terme))))
                (if reduc-operator
                  (make-application reduc-operator (operand terme))
                  (let ((reduc-operande (reduc1-normale (operand terme))))
                    (if reduc-operande
                      (make-application (operator terme) reduc-operande)
                      #f))))))))

(define (reduc1-valeur terme)
  (cond ((lambda:variable?  terme) #f)
```

```
        ((abstraction? terme)
         (let ((reduc-corps (reduc1-valeur (corpsAbstrac terme))))
           (if reduc-corps
             (make-abstraction (varAbstrac terme) reduc-corps)
             #f)))
        ((application? terme)
         (let ((reduc-operande (reduc1-valeur (operand terme))))
           (if reduc-operande
             (make-application (operator terme) reduc-operande)
             (let ((reduc-operator (reduc1-valeur (operator terme))))
               (if reduc-operator
                 (make-application reduc-operator (operand terme))
                 (if (redex? terme)
                   (beta-reduc-redex terme)
                   #f))))))))
```

Pour itérer ces réductions jusqu'à une forme irréductible (ou diverger), on définit une fonction `iterer-reduction` qui prend en paramètre une méthode de réduction en un coup et un booléen `trace?` pour indiquer si l'on souhaite afficher les états intermédiaires :

```
(define (iterer-reduction reduction1 terme trace?)
  (letrec ((loop (lambda (terme)
                   (when trace? (display-alln "--> " terme))
                   (let ((terme1 (reduction1 terme)))
                     (if terme1
                       (loop  terme1)
                       terme)))))
    (loop terme)))
```

D'où nos deux stratégies de réduction :

```
(define  (reduction-normale terme trace?)
  (iterer-reduction reduc1-normale  terme trace?))

(define  (reduction-valeur terme trace?)
  (iterer-reduction reduc1-valeur  terme trace?))
```

Tests

Essayons d'abord avec nos deux exemples précédents :

```
(define T1 '((lamb x  ((lamb y (x y)) u))  z))

? (reduction-normale  T1 #t)
--> ((lamb x ((lamb y (x y)) u)) z)
--> ((lamb y (z y)) u)
--> (z u)
(z u)
```

```
? (reduction-valeur T1  #t)
--> ((lamb x ((lamb y (x y)) u)) z)
--> ((lamb x (x u)) z)
--> (z u)
(z u)

? (define delta '((lamb x (x x))(lamb x (x x))))

? (reduction-valeur   `(,delta ,delta) #t) -> boucle
? (reduction-normale  `(,delta ,delta) #t) -> boucle

? (define T2  `((lamb x   z)  ,delta))

? (reduction-normale T2 #t)
--> ((lamb x z) ((lamb x (x x)) (lamb x (x x))))
--> z
z

? (reduction-valeur T2 #t) -> boucle
```

La réduction normale a l'avantage de toujours trouver la forme irréductible d'un terme normalisable, mais elle a l'inconvénient d'être souvent plus longue. Comparons les réductions du terme T3 défini par :

```
(define T3   '((lamb x ((x x) (x x)))((lamb x x) z))

? (reduction-valeur  T3   #t)
--> ((lamb x ((x x) (x x))) ((lamb x x) z))
--> ((lamb x ((x x) (x x))) z)
--> ((z z) (z z))
((z z) (z z))

? (reduction-normale  T3  #t)
--> ((lamb x ((x x) (x x))) ((lamb x x) z))
--> ((((lamb x x) z) ((lamb x x) z)) (((lamb x x) z) ((lamb x x) z)))
--> ((z ((lamb x x) z)) (((lamb x x) z) ((lamb x x) z)))
--> ((z z) (((lamb x x) z) ((lamb x x) z)))
--> ((z z) (z ((lamb x x) z)))
--> ((z z) (z z))
((z z) (z z))
```

Comme l'argument est dupliqué dans le corps de la fonction, la réduction normale évalue chacune de ces copies, alors que la réduction par valeur évalue une seule fois l'argument.

Phénomène du funarg en Lisp

Le phénomène de capture d'une variable libre est bien connu des programmeurs Lisp. Ce phénomène — dit du funarg[1] descendant — apparaissait dans les premiers

[1] Le terme "funarg" est une contraction de "*functional argument*".

Lisp en liaison dynamique quand on passait une fonction en argument d'une autre fonction (voir aussi le chapitre 21 §1). Par exemple, évaluons l'expression suivante par β-réduction *sans effectuer* de renommage :

```
(let ((y 'a))
  (((lambda (f)(lambda (y)(f y))) (lambda (x)(list y x))) 'b))
```

Pour appliquer `(lambda (f)(lambda (y)(f y))` à `(lambda (x)(list y x))`, on remplace `f` par cette expression, il vient :
`(lambda (y)((lambda (x)(list y x)) y))` qui, appliquée à `b`, donne
`((lambda (x)(list 'b x)) u))` qui se réduit en `(b b)`.

Alors que le résultat donné par Scheme est :

```
? (let ((y 'a))
    (((lambda (f)(lambda (y)(f y))) (lambda (x)(list y x))) 'b))
(a b)
```

Car au moment de la définition de `(lambda (x)(list y x))`, la variable libre `y` vaut `a` et au moment de l'appel de `(lambda (y)(f y))`, la variable `y` vaut `b`. Mais il n'y pas eu confusion, car Scheme avait sauvegardé la liaison `y=b` dans la fermeture de `(lambda (x)(list y x))`. Ce mécanisme de la liaison statique est réalisé aussi en lambda calcul grâce à la bêta réduction.

En remplaçant le `let` par une application de lambda, un lambda terme analogue à cette expression Lisp est donné par :

```
'((lamb y (((lamb f (lamb y (f y)))  (Lamb x (y x))) b)) a)
```

La réduction de ce terme donne bien l'analogue de l'expression calculée par Scheme

```
? (reduction-valeur '((lamb y (((lamb f (lamb y (f y)))
                                 (lamb x (y x))) b)) a) #f)
(a b)
```

20.4 Représentation des données en lambda calcul

Avec l'abstraction et l'application, le lambda calcul ressemble à un langage de programmation, mais il manque un point essentiel : sur quelles données calcule--t-on ?
Des structures de données comme les booléens, les entiers, les listes, ... sont en fait des structures abstraites sur lesquelles on peut faire des opérations satisfaisant à certaines règles. On va montrer comment représenter de telles structures à l'intérieur du lambda calcul et plus précisément à l'aide de termes clos appelés aussi combinateurs. On écrira les combinateurs en majuscule pour les distinguer des objets classiques de même nom.

Représentation des booléens

On choisit deux combinateurs, non bêta équivalents, pour représenter les booléens «vrai» et «faux», ces choix sont reliés par un combinateur de négation «non» qui réalise l'échange entre le vrai et le faux.

VRAI = $\lambda x\,.\,\lambda y\,.\,x$; intuitivement c'est la première projection
FAUX = $\lambda x\,.\,\lambda y\,.\,y$; intuitivement c'est la deuxième projection
NON = $\lambda b\,.\,((b\;FAUX)\;VRAI)$

Ces choix peuvent sembler arbitraires, l'essentiel est que l'on ait les relations suivantes :

$(NON\;VRAI) =_\beta FAUX$ et $(NON\;FAUX) =_\beta VRAI$

Remarque 4 Il est important de noter qu'il n'y pas de notion de définition en lambda calcul. Quand on donne un nom à un combinateur, c'est simplement une *abréviation* qui est expansée en sa valeur quand on l'utilise.

Pour vérifier ces réductions, on utilise notre implantation du lambda calcul.

```
(define  VRAI  '(lamb x (lamb y x)))
(define  FAUX  '(lamb x (lamb y y)))
(define  NON   `(lamb b ((b ,FAUX) ,VRAI)))
```

Pour faire les réductions, on utilise de préférence la réduction normale, on pose :

```
(define (reduire terme)
  (reduction-normale terme #f))

? (reduire  `(,NON ,VRAI))  -> (lamb x (lamb y y)) = FAUX
? (reduire  `(,NON ,FAUX))  -> (lamb x (lamb y x)) = VRAI
```

On définit un connecteur ternaire SI par :

$$SI = \lambda b\,.\,(\lambda e1\,.\,(\lambda e2\,.\,((b\;e1)\;e2)))$$

Et l'on vérifie que l'on a bien :

$(((SI\;VRAI)\;x)\;y)\;=_\beta\;x$
$(((SI\;FAUX)\;x)\;y)\;=_\beta\;y$

```
(define SI  '(lamb test (lamb e1 (lamb e2  ((test e1) e2)))))

? (reduire  `(((,SI ,VRAI)  e1)  e2))  -> e1
? (reduire  `(((,SI ,FAUX)  e1)  e2))  -> e2
```

Exercice 3 *Essayer de trouver une représentation des connecteurs* $\wedge$ *et* $\vee$ *à partir de SI.*

On peut donc représenter par des lambda termes tout le calcul booléen.

Représentation des entiers

Il s'agit de donner des lambda termes pour les entiers 0, 1, 2, ... et pour la fonction successeur en faisant en sorte que le successeur de 0 soit 1, ...
On pose :

$\overline{0} = \lambda f . \lambda x . x$
$\overline{1} = \lambda f . \lambda x . (f\ x)$; on applique f une fois sur x
$\overline{2} = \lambda f . \lambda x . (f\ (f\ x))$; on applique f deux fois sur x
...
De façon générale, le lambda terme associé à l'entier n est noté $\overline{n}$ et est défini par $\overline{n} = \lambda f . \lambda x . (f \ldots (f\ x) \ldots)$; on applique n fois f sur x.

La fonction successeur consiste à appliquer f une fois de plus :

$$SUCC = \lambda n . \lambda f . \lambda x . (f\ (n\ f)\ x))$$

On vérifie, par exemple, que $(SUCC \quad UN) =_\beta DEUX$.

L'addition peut se définir par le lambda terme :

$$ADD = \lambda m . \lambda n . \lambda f . \lambda x . ((m\ f)\ ((n\ f)\ x))$$

Et on vérifie avec notre implantation que

$$((ADD\ UN)\ UN) =_\beta DEUX.$$

L'intérêt de cette représentation des entiers, c'est qu'elle permet de coder toutes les fonctions calculables. En effet, on peut démontrer le théorème suivant :

Théorème 3 A toute fonction calculable f de $\mathbf{N}$ dans $\mathbf{N}$ on peut associer un lambda terme $\overline{f}$ tel que pour tout entier n l'entier $f(n)$ est représenté par le lambda terme $(\overline{f} \quad \overline{n})$.

Représentation des listes

La structure de liste consiste à se donner : le constructeur `cons`, la constante `liste-vide`, les accesseurs `car` et `cdr`, et le prédicat `vide?` de façon à vérifier les relations habituelles. On définit les lambda termes :

$CONS = \lambda x . \lambda y . \lambda f . ((f\ x)\ y)$
$CAR = \lambda L . (L \quad VRAI)$
$CDR = \lambda L . (L \quad FAUX)$
$VIDE = \lambda L . VRAI$
$VIDE? = \lambda L . (L\ (\lambda a . (\lambda d . FAUX)))$

Pour vérifier les égalités :

$(CAR \quad ((CONS \quad x) \quad y)) =_\beta x$
$(CDR \quad ((CONS \quad x) \quad y)) =_\beta y$
$(VIDE? \quad VIDE) \quad =_\beta VRAI$
$(VIDE?\ ((CONS \quad x) \quad y)) =_\beta FAUX$

On utilise notre représentation en Scheme, on écrit C-ONS, C-AR et C-DR à la place de CONS, CAR, CDR pour ne pas écraser les fonctions prédéfinies de même nom.

```
(define  C-ONS  '(lamb x (lamb y (lamb f ((f x) y)))))
(define  C-AR   `(lamb L (L ,vrai)))
(define  C-DR   `(lamb L (L ,faux)))

? (reduire  `(,C-AR ((,C-ONS u) v)))  -> u
? (reduire  `(,C-DR ((,C-ONS u) v)))  -> v
```

Pour définir les listes, on se donne une représentation de la liste vide :

```
(define  VIDE  `(lamb x ,VRAI))
```

avec le prédicat

```
(define  VIDE? `(lamb L (L (lamb a (lamb d ,FAUX)))))
```

On peut achever la vérification de la cohérence de ces représentations :

```
? (reduire  `(,C-AR ((,C-ONS u) ,VIDE))) -> u
? (reduire  `(,C-DR ((,C-ONS u) ,VIDE))) -> (lamb x (lamb x (lamb y x))) = VIDE
? (reduire  `(,VIDE?  ,VIDE))            -> (lamb x (lamb y x)) = VRAI
? (reduire  `(,VIDE?  ((,C-ONS a ) ,VIDE))) -> (lamb x (lamb y y)) = FAUX
```

Avec les listes, se pose immédiatement la question : peut-on définir des fonctions récursives en lambda calcul ?

20.5 Combinateur Y, récursion et domaines

Le théorème 3 nous assure que l'on peut représenter par un lambda terme toutes les fonctions programmables. Cherchons quel peut être l'analogue du mécanisme de la définition récursive alors qu'il n'y pas de façon de nommer un terme en lambda calcul. Pour cela, on analyse la définition d'une fonction récursive en Scheme ; prenons l'exemple de la fonction `dernier`. Elle retourne le dernier élément d'une liste non vide.

```
(define dernier
  (lambda (L)
    (if  (null? (cdr L)) (car L) (dernier (cdr L)))))
```

Cette définition entraîne l'égalité :

`dernier = (Phi dernier)`   (*)

où la fonction `Phi` est l'abstraction de la définition de `dernier` :

```
(define Phi
  (lambda (f)
    (lambda (L)
      (if  (null? (cdr L)) (car L) (f (cdr L))))))
```

Le combinateur Y

L'égalité (*) s'interprète en disant que la fonction `dernier` est point fixe de `Phi`, ce qui conduit à se poser le problème plus général :

Problème : étant donné un lambda terme M, existe-t-il un lambda terme N qui en soit un point fixe au sens suivant : $(M\ N) =_\beta N$?

La réponse est oui et on peut même donner une formule pour calculer un tel point fixe. Supposons que nous ayons un lambda terme Fix tel que pour tout terme M on ait :

$$(Fix\ M) =_\beta (M\ (Fix\ M)) \qquad (**)$$

alors le terme $N = (Fix\ M)$ fournit un point fixe de M.

Le combinateur suivant, dû à H. Curry et traditionnellement noté Y, peut jouer le rôle de Fix. On pose :

$$Y = \lambda f\,.\,((\lambda x\,.\,(f\,(x\,x)))(\lambda x\,.\,(f\,(x\,x))))$$

Montrons que Y vérifie la condition (**) :

$$\begin{aligned}(Y\ M) &\to_\beta ((\lambda x\,.\,(M\,(x\,x)))(\lambda x\,.\,(M\,(x\,x))))\\ &\to_\beta (M\ ((\lambda x\,.\,(M\,(x\,x)))(\lambda x\,.\,(M\,(x\,x)))))\end{aligned}$$

et comme la première réduction entraîne que

$$((\lambda x\,.\,(M\,(x\,x)))(\lambda x\,.\,(M\,(x\,x)))) =_\beta (Y\ M)$$

on a finalement $(Y\ M) =_\beta (M\ (Y\ M))$.

Revenons à la recherche d'une solution de (*), le combinateur Y nous conduit à définir le lambda terme $DERNIER$ par $(Y\ \ PHI)$ où

$$PHI = (\lambda f\,.\,\lambda L\,.\,(SI\ ((((VIDE?\ (CDR\ L))(CAR\ L))(f\ (CDR\ L)))))).$$

Vérifions, à l'aide de notre implantation, que $DERNIER$ calcule effectivement le dernier élément d'une liste. On pose :

```
(define Y  '(lamb f ((lamb g (f (g g)))(lamb g (f (g g))))))

(define DERNIER `(,Y (lamb f  (lamb L (((,si (,VIDE? (,C-DR L)))
                                         (,C-AR L))
                                        (f (,C-DR L))))))).
```

L'application de $DERNIER$ au terme qui représente la liste $(a\,b\,c)$ donne :

```
? (reduire  `(,DERNIER  ((,C-ONS a)((,C-ONS b) ((,C-ONS c) ,VIDE)))))
c
```

Flots en lambda calcul

Le lambda calcul nous permet aussi de définir des listes infinies ou flots. Par exemple, la liste infinie de a `(a a a a ...)` doit satisfaire la condition de point fixe :

$$LISTE-DE-A \ = \ ((CONS\, a) \quad LISTE-DE-A)$$

Ce qui conduit à définir le lambda terme $LISTE-DE-A$ par :

$$LISTE-DE-A = (Y\ (\lambda L\,.\ ((CONS\, a)\ L))).$$

On vérifie que si l'on pose :

```
(define LISTE-DE-A '(,Y (lamb L ((,C-ONS a) L))))
```

on trouve

```
? (reduction  '(, C-AR (,C-DR (,C-DR  ,LISTE-DE-A)))
a
```

Approximations successives d'une fonction

Essayons de dissiper le mystère qui entoure le lien entre Y et définition récursive. On part de la définition de la fonction `dernier` en Scheme :

```
(define dernier
  (lambda (L)
    (if  (null? (cdr L)) (car L) (dernier (cdr L)))))
```

Si l'on ne s'intéresse qu'aux listes de longueur 1, on peut se contenter de la fonction `dernier-1` :

```
(define dernier-1
  (lambda (L) (if (null? (cdr L)) (car L) 'indefinie)))
```

Si l'on ne s'intéresse qu'aux listes de longueur $\leq$ 2, on peut se contenter de la fonction `dernier-2` :

```
(define dernier-2
  (lambda (L) (if (null? (cdr L)) (car L) (dernier-1 (cdr L)))) )
```

...

Et plus généralement, si l'on ne s'intéresse qu'aux listes de longueur $\leq n$, on peut se contenter de la fonction `dernier-n` :

```
(define dernier-n
  (lambda (L) (if (null? (cdr L)) (car L) (dernier-n-1 (cdr L)))))
```

Comme toute liste a une certaine longueur, on peut se contenter de cette suite de fonctions pour calculer le dernier élément d'une liste.
Pour donner une définition plus compacte de cette suite de fonctions, on utilise à nouveau la fonctionnelle `Phi`. On a :

```
dernier-n = (Phi  dernier-n-1)
          = (Phi (Phi ... (Phi dernier-0)...)
```

où `dernier-0` est la fonction indéfinie partout `(lambda (L) 'indefinie)`.

Tout est donc basé sur une itération de la fonction `Phi`. C'est précisément ce que réalise le combinateur Y car l'égalité $(Y\ M) =_\beta (M\ (Y\ M))$ implique :

$$(Y\ PHI) =_\beta (PHI\ (Y\ PHI)) =_\beta (PHI\ (PHI \ldots (Y\,PHI) \ldots).$$

Cette recopie indéfinie du terme PHI est basée en dernier ressort sur la propriété d'auto-reproduction du lambda terme $((\lambda\, x\,.\,(x\ x) \quad \lambda\, x\,.\,(x\ x))$

Le lecteur attentif n'a pas manqué de faire le rapprochement entre la suite des fonctions `dernier-k` et la méthode des approximations successives présentée au chapitre 3 §4. On avait indiqué que la méthode des approximations successives était un outil puissant pour chercher un éventuel point fixe d'une fonction. C'est exactement ce que l'on a fait ici pour trouver un point fixe de la fonctionnelle `Phi`. Au lieu de considérer des valeurs numériques, on a considéré des valeurs fonctions ce qui oblige à préciser la notion de convergence.

Chacune des fonctions `dernier-k` est une fonction des listes non vides dans les S-expressions augmentées de la valeur indéfinie. En notant LISTES* l'ensemble des listes non vides, on a :

`dernier-k :   LISTES*` $\rightarrow$ `S-EXPRESSIONS` $\cup$ `indefinie`

Cette suite de fonctions `dernier-k` est croissante pour la relation d'ordre suivante sur les fonctions.
On dit que $f \prec g$ si g est définie quand f l'est et égale à f en ces points, soit en bref : $f \prec g$ si g est une extension de f.
Comme la fonction `dernier-n` est une extension de `dernier-n-1`, on a bien la propriété de croissance de cette suite.
Cette relation d'ordre jouit d'une propriété particulière : toute suite croissante f_n admet une borne supérieure. La preuve est immédiate, définissons la fonction f par $f(x) = f_n(x)$ s'il existe un n pour lequel f_n est définie en x et $f(x) = indefinie$ sinon. Il est clair que f est la borne supérieure des f_n.

Dans le cas de la suite `dernier-n`, la borne supérieure n'est autre que la fonction `dernier`.

Domaines et théorème du plus petit point fixe

Ce type de construction intervient dans d'autres secteurs de l'informatique, ce qui a donné naissance à la théorie des *domaines*. On en présente le minimum nécessaire à l'énoncé du célèbre théorème du plus petit point fixe.

Définition 6 On appelle *domaine* un ensemble muni d'une relation d'ordre (notée $\prec$) telle que :
- il existe un élément minimal (on le note $\perp$),
- toute suite croissante x_n admet une borne supérieure (c'est-à-dire un plus petit majorant).

L'ensemble des fonctions $LISTES* \rightarrow S-EXPRESSIONS \cup \perp$ avec la relation d'ordre précédente est un domaine dont l'élément minimum est la fonction indéfinie partout.

Une fonction d'un domaine D_1 dans un autre D_2 est dite *croissante* si pour tous les points on a $x_1 \prec x_2 \Rightarrow F(x_1) \prec F(x_2)$.

Une fonction f croissante est dite *continue* si pour toute suite croissante (x_n) on a $f(sup_n\, x_n) = sup_n\, f(x_n)$.

On peut alors énoncer le théorème du plus petit point fixe.

Théorème 4 Soit f une fonction croissante et continue d'un domaine D dans lui-même. Alors la suite des itérés $x_n = f(x_{n-1})$, $x_0 = \perp$ est croissante et sa borne supérieure est le plus petit point fixe de f.

Preuve

Montrons par récurrence sur n que $x_n < x_{n+1}$.

Pour $n = 0$, on a $x_0 = \perp \prec x_1$ car $\perp$ est minimal.

On suppose que l'on a $x_{n-1} \prec x_n$, on en déduit, en appliquant la fonction croissante f,

$x_n = f(x_{n-1}) \prec f(x_n) = x_{n+1}$.

Soit $x = sup_n\, x_n$, montrons que x est point fixe de f. Comme f est continue on a $f(x) = f(sup_n\, x_n) = sup_n\, f(x_n) = sup_n\, x_{n+1} = x$.

Soit y un point fixe de f, montrons que $x \prec y$. Il suffit de prouver par récurrence que pour tout n $x_n \prec y$. C'est vrai pour $n = 0$ car x_0 est minimal. Si $x_n \prec y$, en appliquant f, il vient $x_{n+1} = f(x_n) \prec f(y) = y$.

Enfin ce plus petit point fixe est unique. En effet, si x' était un autre point fixe minimal, le raisonnement précédent montre que $x < x'$ et donc $x = x'$.

Lambda calcul et Scheme

Le lambda calcul fournit un langage de programmation complet, bien que basé sur deux constructions : l'abstraction et l'application. Cette simplicité et cette généralité en font un outil idéal pour étudier de façon théorique les mécanismes d'évaluation. On est maintenant en mesure d'expliquer en quel sens un langage fonctionnel est du sucre syntaxique autour du lambda calcul. Plaçons dans un tableau un fragment de Scheme et les analogues en lambda calcul :

Scheme	**Lambda calcul**
symbole `x`	variable x
booléens `#t` et `#f`	combinateurs $VRAI$ et $FAUX$
`(if test e1 e2)`	combinateur SI
entier n	lambda terme $\overline{n}$
`car, cdr, cons, '(), null?`	CAR, CDR, CONS, VIDE, VIDE?
`(lambda (x) corps)`	$\lambda x\,.\,corps$
`(let ((x e)) corps)`	$((\lambda x\,.\,corps)\ e)$
`(letrec ((f (lambda (x) corps))) e)`	$((Y\ (\lambda f\,.\,(\lambda x\,.\,corps)))\ e)$
évaluation	réduction

Scheme et l'appel par valeur

En ce qui concerne la réduction, il faut être plus précis : quelle est la stratégie d'évaluation utilisée par Scheme?

A l'appel d'une fonction `(f e)`, Scheme évalue les expressions `f` et `e` puis réalise l'application. Il s'agit donc de la réduction par valeur. Mais il y a une petite différence avec la réduction par valeur en lambda calcul. En Scheme, on ne réduit pas le corps d'une lambda expression. Autrement dit, la règle de réduction suivante (parfois appelée ξ−réduction) n'est pas utilisée.

$$\frac{M \to M_1}{\lambda x\,.\,M \to \lambda x\,.\,M_1} \quad (\xi)$$

Par exemple, on constate que l'évaluation de :

```
? (lambda (x) (display "coucou") (+ x 1))
#[procedure]
```

ne provoque pas l'affichage de coucou !

On peut modéliser en lambda calcul ce type de réduction, on le nomme l'*appel par valeur*. Pour l'implanter, il suffit de changer une ligne dans la fonction `reduc1-valeur` : dans le cas d'une abstraction, on retourne `#f` [2].

Il est intéressant de noter que l'on peut aussi utiliser en Scheme un combinateur de point fixe pour définir une fonction récursive. L'appel par valeur conduit à utiliser une variante du combinateur Y car la réduction par valeur d'un terme du type $(Y\ M)$ ne termine pas. On remplace dans Y le sous-terme $(f\ (g\ g))$ par le terme $\lambda x\,.\,((f\ (g\ g))\ x)$ qui lui est η−équivalent. On appelle $Y0$ le combinateur

[2] On a l'analogue pour la réduction normale, c'est l'appel par nom. On l'étudiera au prochain chapitre.

obtenu. On vérifie immédiatement que la réduction faible par valeur d'un terme du type $(Y0\,M)$ termine en un terme η−équivalent à $M\,(Y0\,M)$.
Sa définition en Scheme est donnée par :

```
(define YO
  (lambda (f)
    ((lambda (g) (lambda (x)((f (g g)) x)))
     (lambda (g) (lambda (x)((f (g g)) x))))))
```

Voici une nouvelle définition de la fonction factorielle à l'aide de `YO` :

```
(define fac
  (YO (lambda (f)
        (lambda (x)
          (if (zero? x)
            1
            (* x (f (- x 1))))))))
```

On vérifie bien que :

```
? (fac 4)
24
```

20.6 Lambda calcul typé

L'interprétation intuitive d'une abstraction f comme une fonction n'est pas tout à fait exacte. En lambda calcul, il est parfaitement licite de considérer l'application $(f\ f)$ d'une abstraction sur elle-même. Ceci n'est pas compatible avec l'interprétation fonctionnelle, car une fonction applique un domaine de définition A dans un domaine B d'arrivée : $A \rightarrow B$. Pour que f puisse être aussi argument, il faut que l'on ait $A \approx A \rightarrow B$ ce qui est impossible pour des raisons de cardinalité dès que B a plus d'un élément[3].
Pour rapprocher le lambda calcul de notre intuition des fonctions, on ajoute une notion de typage pour les termes. On commence par considérer une notion de types réduite au minimum.

Lambda termes typés

On se donne un ensemble fini T_0 de noms de types qui représente les types de base. On définit l'ensemble T des types par : a est un type si a est dans T_0 ou bien si a est de la forme $(b \rightarrow c)$ où b et c sont dans T.

Par exemple, si T_0 est composé des noms de types *entier* et *bool* alors $((bool \rightarrow entier) \rightarrow (entier \rightarrow entier))$ est un type.

Autrement dit, les types sont les termes construits sur un ensemble de constantes T_0 avec pour seul constructeur, le symbole binaire $\rightarrow$.

Maintenant, on peut définir les *lambda termes typés* de types dans T.

[3]Mais cela peut être possible si l'on considère des sous-ensembles particuliers de fonctions de $A \rightarrow B$.

Définition 7 On se donne pour tout type a dans T un ensemble, non vide, de lambda variables Var_a. On définit, pour tout type a, l'ensemble Λ_a des lambda termes de types a par :
- une variable de type a est dans Λ_a,
- si M est dans $\Lambda_{b \to a}$ et N dans Λ_b, alors $(M\ N)$ est dans Λ_a ,
- si x est dans Var_b et M dans Λ_c alors $\lambda x . M$ est dans $\Lambda_{b \to c}$.

On introduit la notation $M : a$ pour signifier que M est un lambda terme typé de type a. Avec cette notation on peut redéfinir les termes typés par les trois règles de déduction suivantes :

$$\frac{x \in Var_a}{x : a} \qquad \frac{M : b \to a \quad N : b}{(M\ N) : a} \qquad \frac{x \in Var_b \quad M : c}{\lambda x . M : b \to c} \qquad (*)$$

Tout lambda terme ne peut être typé, par exemple le lambda terme $(M\ M)$ n'est pas typable. En effet, s'il avait un type a, alors la deuxième règle montre que l'on devrait avoir l'existence d'un type b tel que $b \to a = b$ ce qui est une égalité impossible entre termes. En particulier, le combinateur de point fixe Y n'est pas typable.

Conversions entre termes typés

Il est immédiat de vérifier que la substitution $M[x \leftarrow N]$ est bien définie dans les lambda termes typés quand on impose à x et à N d'avoir le même type. Il s'ensuit que les définitions de alpha conversion et bêta réduction s'étendent immédiatement au lambda calcul typé.

- *Alpha conversion* : $\lambda x . M \to_\alpha \lambda x_0 . M[x \leftarrow x_0]$ où x_0 n'est pas variable libre de M.

  (Ici x et x_0 sont des variables de même type pour faire la substitution $x \leftarrow x_0$.)

- *Bêta réduction* : $((\lambda x . M)\ N) \to_\beta M[x \leftarrow N]$

  (ici N et x ont nécessairement le même type)

Le fait d'être typé est une restriction très forte sur les lambda termes. On démontre le théorème suivant :

Théorème 5 Tout terme de ce lambda calcul typé est fortement normalisable.

Types simples

On a envie de dire que le lambda terme $\lambda x . x$ est de type $a \to a$ *quel que soit* le type a. Ce n'est pas possible avec notre système de types, car on ne dispose pas de *variables* de type. Ce qui conduit à considérer la notion de *types simples* comme étant les termes, *avec variables,* construits sur les constantes et le constructeur $\to$ précédents. Si l'on note α_0, α_1 ,... les variables de type, alors par exemple $((bool \to \alpha_0) \to (\alpha_1 \to entier))$ est un type simple. On notera τ_1, τ_2, ... les expressions de type simple. Attention, ne pas confondre les variables du lambda calcul, notées $x, y, ...$ avec les variables de type notées α_0, α_1 !

On va donner une définition du lambda calcul typé de sorte que $\lambda x.x$ soit bien de type $\alpha_0 \rightarrow \alpha_0$. Comme on a ajouté des variables de type, on va avoir besoin de la notion d'environnement de type. Un environnement de type, c'est la donnée du type d'un nombre fini de variables du lambda calcul : $\{x_1 : \tau_1, \ldots, x_n : \tau_n\}$; on notera souvent Γ un tel environnement.

Règles de typage

Pour définir les lambda termes typés et leurs types, on généralise les règles (*) en considérant aussi un environnement de typage pour les variables libres des lambda termes.
On définit le jugement suivant : le lambda terme M est de type τ dans l'environnement $\{x_1 : \tau_1, \ldots, x_n : \tau_n\}$. Ce jugement sera noté comme un séquent, avec à gauche l'environnement (qui joue le rôle d'hypothèse) et à droite le lambda terme et son type :

$$\{x_1 : \tau_1, \ldots, x_n : \tau_n\} \vdash M : \tau$$

où $\{x_1, \ldots, x_n\}$ est un ensemble de variables distinctes comprenant l'ensemble des variables libres de M.
La notation $\Gamma, x : \tau$ désigne l'environnement Γ étendu par la liaison $x : \tau$.

Voici les règles de déduction qui permettent de prouver un tel jugement :

$$\frac{}{\{x_1 : \tau_1, \ldots, xj : \tau_j, \ldots, x_n : \tau_n\} \vdash x_j : \tau_j} \quad \text{; axiome pour une variable}$$

$$\frac{\Gamma \vdash M : \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash N : \tau_1}{\Gamma \vdash (M\, N) : \tau_2} \quad \text{; règle pour l'application}$$

$$\frac{\Gamma, x : \tau_1 \vdash M : \tau_2}{\Gamma \vdash \lambda x.M : \tau_1 \rightarrow \tau_2} \quad \text{; règle pour l'abstraction}$$

La substitution $M[x \leftarrow N]$ est toujours faite sous la condition que la variable x et le terme N soient de même type. L'alpha conversion et la bêta réduction restent définies de la même façon.
On a la stabilité du type par bêta réduction, plus précisément on démontre, par induction sur la structure des lambda termes, la proposition suivante :

Proposition 1 Si M est un lambda terme de type τ, c'est-à-dire si $\Gamma \vdash M : \tau$ et si $M \rightarrow *_\beta N$ alors $\Gamma \vdash N : \tau$.

A titre d'exemple d'application des règles, voici la preuve du typage du terme $\lambda x.(f\ (g\, x))$. On prouve le jugement :

$$\{g : \alpha_1 \rightarrow \alpha_2\, ,\, f : \alpha_2 \rightarrow \alpha_3\} \vdash \lambda x.(f\ (g\, x)) : \alpha_1 \rightarrow \alpha_3.$$

Le choix de la règle à utiliser est déterminé par la structure du lambda terme. On dit que la preuve est *dirigée par la syntaxe*.

$$\dfrac{\dfrac{\text{axiome}}{f : \alpha_2 \to \alpha_3 \vdash f : \alpha_2 \to \alpha_3} \quad \dfrac{\dfrac{\text{axiome}}{g : \alpha_1 \to \alpha_2 \vdash g : \alpha_1 \to \alpha_2} \quad \dfrac{\text{axiome}}{x : \alpha_1 \vdash x : \alpha_1}}{g : \alpha_1 \to \alpha_2 , x : \alpha_1 \vdash (g\, x) : \alpha_2}}{\dfrac{g : \alpha_1 \to \alpha_2 , x : \alpha_1 , f : \alpha_2 \to \alpha_3 , x : \alpha_1 \vdash (f\,(g\, x)) : \alpha_3}{g : \alpha_1 \to \alpha_2 , x : \alpha_1 , f : \alpha_2 \to \alpha_3 \vdash \lambda x.(f\,(g\,x)) : \alpha_1 \to \alpha_3}}$$

Cette preuve correspond à la notion de *vérification de type*. Dans beaucoup de langages on déclare le type des variables et on laisse le soin au compilateur de vérifier la cohérence de l'ensemble. C'est la phase dite de vérification de *sémantique statique*.
On peut se poser un problème plus complexe : étant donné un lambda terme, peut-on trouver un environnement de type pour ses variables libres de sorte que le lambda terme soit bien typé dans cet environnement ? C'est le problème de l'*inférence de type*.

20.7 Inférence de type

De façon générale, si l'on se donne un lambda terme M, on peut se demander s'il existe un environnement de type Γ pour ses variables libres et un type τ tel que le jugement $\Gamma \vdash M : \tau$ soit prouvable. L'inférence de type consiste à trouver un tel couple (Γ , τ), on l'appelle un *typage* de M.

Pour commencer, montrons qu'il n'y a pas unicité du typage pour un terme M. Etant donné une substitution σ sur les *variables de type* et un environnement de typage $\Gamma = \{x_1 : \tau_1 \ldots, x_n : \tau_n\}$, on définit l'application de σ sur Γ par :
$\sigma\,\Gamma = \{x_1 : \sigma\,\tau_1 \ldots, x_n : \sigma\,\tau_n\}$.

Avec cette notation, on peut prouver, par induction sur la structure des lambda termes, la

Proposition 2 Etant donnée une substitution σ sur les variables de type et un lambda terme polymorphe M tel que $\Gamma \vdash M : \tau$ alors on a aussi $\sigma\,\Gamma \vdash M : \sigma\,\tau$.

De façon analogue à la notion de type principal de deux termes (Chapitre 17), on pose la définition ci-après :

Définition 8 Un typage (Γ , τ) du terme M est dit *principal* si pour tout autre typage (Γ' , τ') de M, il existe une substitution σ telle que $(\Gamma' , \tau') = (\sigma\,\Gamma , \sigma\,\tau)$.

On va démontrer le résultat suivant de Hindley et Milner :

Théorème 6 Si le lambda terme M admet un typage, alors il admet un typage principal et il est unique à un renommage près des variables de type.

Preuve
La démonstration de ce théorème est assez longue, mais elle est importante car elle fournit un algorithme de calcul d'un type principal.

• *Preuve de l'unicité.*
Si (Γ_1 , τ_1) et (Γ_2 , τ_2) sont deux types principaux, il existe des substitutions $\sigma 1$ et $\sigma 2$ telles que :

$$\tau_2 = \sigma_1 \tau_1 , \Gamma_1 = \sigma_2 \Gamma_2 \quad \text{et} \quad \tau_1 = \sigma_2 \tau_2 , \Gamma_2 = \sigma_1 \Gamma_1 .$$

D'où l'on tire :

$$\tau_2 = (\sigma_1 \sigma_2) \tau_2 , \Gamma_2 = (\sigma_1 \sigma_2) \Gamma_2 \quad \text{et} \quad \tau_1 = (\sigma_2 \sigma 1) \tau_1 , \Gamma_1 = (\sigma_2 \sigma 1) \Gamma_1$$

D'où l'on déduit que :

$$\sigma_1 \sigma_2 = Identite \text{ sur } Var(\tau_2 , \Gamma_2) \quad \text{et} \quad \sigma_2 \sigma_1 = Identite \text{ sur } Var(\tau_1 , \Gamma_1)$$

Ce qui implique que les substitutions σ_1 et σ_2 sont des permutations entre les variables de types de (Γ_1 , τ_1) et (Γ_2 , τ_2).

• *Preuve de l'existence*
On peut remplacer la recherche d'un environnement de typage Γ par la recherche d'une substitution. En effet, tout environnement de type $\{x_1 : \tau_1, \ldots, x_n : \tau_n\}$ est de la forme $\sigma \Gamma_0$ avec $\Gamma_0 = \{x_1 : \alpha_1, \ldots, x_n : \alpha_n\}$ où les α_j sont des *variables* de types deux à deux distinctes. Le problème de l'*existence* revient donc à chercher une substitution σ et un type τ tels que $(\sigma \Gamma_0 , \tau)$ soit un typage principal.
On procède par induction sur la structure des lambda termes. Il y a trois cas à considérer : le cas de base où le terme est réduit à une variable, les cas d'une abstraction et celui d'une application.

i) Cas où M est une variable x

Dans ce cas on prend pour σ l'identité et pour type une variable de type car il est clair que le typage $\quad x : \alpha_0 \vdash x : \alpha_0 \quad$ est principal.

ii) Cas où M est une abstraction $(\lambda x . N)$

On cherche un couple (σ , τ) tel que $\quad \sigma \Gamma_0 \vdash \lambda x . N : \tau \quad$ où Γ_0 est un environnement pour les variables libres de $(\lambda x . N)$.

La preuve du typage de M doit finir par la règle d'abstraction ; comme M est typable, on en déduit que N doit l'être aussi. Donc l'hypothèse d'induction s'applique à N. Par conséquent, il existe un type principal pour N, c'est-à-dire un couple (σ_1 , τ_1) tel que :

$$\sigma_1 \Gamma_0 , x : \sigma_1 \alpha_0 \vdash N : \tau_1$$

et la règle d'abstraction implique :

$$\sigma_1 \Gamma_0 \vdash \lambda x . N : (\sigma_1 \alpha_0) \rightarrow \tau_1$$

D'où le typage défini par : $\sigma = \sigma_1$, $\tau = (\sigma_1 \alpha_0) \rightarrow \tau_1$

Par construction, c'est un typage de $\lambda x . N$ et il est principal car tout typage principal de $\lambda x . N$ doit nécessairement découler d'un typage de N par la règle d'abstraction.

Illustrons cette méthode avec deux exemples simples.

Exemple1 : typage principal de $\lambda x . x$.

Le typage principal de la variable x étant $x : \alpha_0 \vdash x : \alpha_0$,
on en déduit que le typage principal de $\lambda x . x$ est : $\vdash \lambda x . x : (\alpha_0 \rightarrow \alpha_0)$.

Exemple2 : typage principal de $\lambda x . y$.

On a vu qu'un typage principal de la variable y est $y : \alpha_0 \vdash y : \alpha_0$
et a fortiori $y : \alpha_0 , x : \alpha_1 \vdash y : \alpha_0$
d'où l'on déduit qu'un typage principal de $\lambda x . y$ est :
$y : \alpha_0 \vdash \lambda x . y : (\alpha_1 \rightarrow \alpha_0)$

iii) Cas où M est une application $(P\, Q)$

On cherche un couple (σ , τ) tel que $\sigma \Gamma_0 \vdash (P\ Q) : \tau$ où Γ_0 est un environnement pour les variables libres de P et de Q.

Comme $(P\, Q)$ est typable, la règle de typage d'une application :

$$\frac{\Gamma \vdash P : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash Q : \tau_1}{\Gamma \vdash (P\, Q) : \tau_2}$$

implique que les termes P et Q sont typables et donc l'hypothèse d'induction s'applique à P et Q. Par conséquent, il existe pour P un type principal défini par un couple (σ' , τ') tel que :

$$\sigma' \Gamma_0 \vdash P : \tau'$$

et pour Q un type principal définit par un couple (σ'' , τ'') tel que

$$\sigma'' \Gamma_0 \vdash Q : \tau''$$

Mais pour pouvoir utiliser la règle de l'application, on doit avoir le même environnement Γ. Cet environnement sera donc instance de $\sigma' \Gamma_0$ et de $\sigma'' \Gamma_0$. Aussi, après avoir calculé le couple (σ' , τ'), on cherche un type principal pour Q en partant de l'environnement $\sigma' \Gamma_0$: d'où un couple (η_1 , θ_1) tel que :

$$\eta_1 \sigma' \Gamma_0 \vdash Q : \theta_1$$

Et pour P on a encore, d'après la proposition précédente,

$$\eta_1 \sigma' \Gamma_0 \vdash P : \eta_1 \tau'$$

La règle de typage de l'application impose que les types $\eta_1\,\tau'$ et θ_1 doivent être de la forme $\tau_1 \to \tau_2$ et τ_1, on cherche donc une substitution ρ telle que :

$$\rho\,(\eta_1\,\tau') = \tau_1 \to \tau_2 \quad \text{et} \quad \rho\,\theta_1 = \tau_1$$

Soit α_j une nouvelle variable de type, on peut toujours supposer que τ_2 est de la forme $\rho\,\alpha_j$, d'où l'équation :

$$\rho\,(\eta_1\,\tau') = \rho\,(\theta_1 \to \alpha_j)$$

Comme on cherche la substitution ρ la plus générale, ρ doit être unificateur principal des termes $\eta_1\,\tau'$ et $(\theta_1 \to \alpha_j)$. Si un tel unificateur principal ρ existe, alors il vient $\tau_2 = \rho\,\alpha_j$, d'où finalement le typage de $(P\;Q)$:

$$\rho\,\eta_1\,\sigma_2\,\Gamma_0 \vdash (P\;Q) : \rho\,\alpha_j$$

On obtient donc σ par composition des trois substitutions ρ , η_1 et σ_2.

Illustrons cette méthode avec deux exemples simples.

Exemple1 : typage du terme $(x\;y)$.

On part des typages principaux pour les variables x et y :

$$x : \alpha_0 \vdash x : \alpha_0 \quad \text{et} \quad y : \alpha_1 \vdash y : \alpha_1$$

La condition $\rho\,\alpha_0 = \rho\,(\alpha_1; \to \alpha_j)$ donne pour ρ l'unificateur principal $\rho\,\alpha_0 = (\alpha_1 \to \alpha_j)$. D'où le typage principal de $(x\;y)$:

$$x : (\alpha_1 \to \alpha_j)\,,\; y : \alpha_1 \vdash (x\;y) : \alpha_j$$

Exemple2 : typage du terme $(\lambda\,x\,.\,(x\;y))$.

Pour calculer le type principal du corps $(x\;y)$ de l'abstraction, on utilise l'exemple précédent :

$$x : (\alpha_1 \to \alpha_j)\,,\; y : \alpha_1 \vdash (x\;y) : \alpha_j$$

D'où le typage

$$y : \alpha_1 \vdash (\lambda\,x\,.\,(x\;y)) : ((\alpha_1 \to \alpha_j) \to \alpha_j)$$

Extension du typage et langage ML

Si l'on se reporte au tableau comparatif entre un fragment de Scheme et le lambda calcul, on est conduit à se poser la question suivante : peut-on inférer un type pour les expressions Scheme ? La réponse est oui, *si* l'on impose des restrictions sur l'usage de certaines expressions. Par exemple, les deux branches du `if` devront avoir le même type, les listes auront tous leurs termes de même type, ... Ce sont des restrictions assez fortes, aussi aboutit-on à un *autre* langage qui s'appelle ML. C'est un langage fonctionnel qui concilie la sécurité du typage des expressions avec l'absence de déclarations de type. Signalons qu'en ML le typage du `let` est plus général que celui que l'on peut déduire par application d'une lambda expression. On renvoie aux ouvrages sur ML pour les détails.

20.8 Implantation de l'inférence de type

On va suivre pas à pas la méthode utilisée dans la démonstration précédente.
On représente un typage (σ , τ) par une liste où la substitution σ est représentée par une a-liste `((type-var . type-terme)...)` et le type τ par un terme de type. Un environnement de typage est une a-liste ($(x_i \ . \ \tau_i)\ldots$).
La fonction `typage` calcule le typage principal (σ , τ) d'un lambda terme, en cas d'échec on rend le couple `(#f #f)`.

```
(define (typage type-env lambda-terme)
  (cond ((lambda:variable?  lambda-terme)
         (list '() (LireEnv  lambda-terme type-env)))
        ((abstraction? lambda-terme)
         (typage-abst type-env lambda-terme))
        (else (typage-app type-env lambda-terme))))
```

• Dans le cas d'une *variable*, il suffit de lire son type dans l'environnement de type.

```
(define (LireEnv   var type-env)
  (cdr (assoc var type-env)))
```

• Dans le cas d'une *abstraction* `(lamb var corps)`, on fait appel à la fonction auxiliaire `typage-abst`. Cette fonction calcule le type du `corps` dans un environnement de type augmenté par la liaison entre la `var` de l'abstraction et une *nouvelle* variable de type générée par `genVar`. Comme un typage est une liste à deux éléments, il est commode d'utiliser le `let*` destructurant `bind-let*` défini par une macro au chapitre 9 §6.

```
(define (typage-abst type-env abstr)
  (bind-let*  ((alpha (genVar))
               ((sigma type-corps)
                  (typage (acons (varAbstrac abstr) alpha type-env)
                                 (corpsAbstrac abstr))))
    (if type-corps
      (let ((type-var (valeur-subst sigma alpha)))
        (list  sigma  `(-> ,type-var  ,type-corps))
      (list '#f '#f))))
```

La fonction `genVar` génère des variables de préfixe «alpha», elle est définie par :

```
(define genVar (creer-genVar "alpha"))
```

où `creer-genVar` est introduit au chapitre 5 §7.

• Dans le cas d'une *application* `(P Q)`, on utilise la fonction auxiliaire `typage-app`. Cette fonction calcule successivement le typage de `P` puis celui de `Q` et unifie les résultats pour être dans les conditions de la règle de typage d'une application. On fait appel à la fonction `unifier` du chapitre 18 §3 ce qui oblige à définir des prédicats `var-type?`, `cte-type?` pour distinguer les variables de type et les constantes de type.

```
(define (typage-app type-env app)
  (bind-let* ((op   (operator app))
              (opd  (operand app))
              ((sigma1 type-op)(typage type-env op))
              ((sigma2 type-opd)(typage (sublis sigma1 type-env) opd))
              (alpha (genVar))
              (sigma (unifier `(-> ,type-opd ,alpha)
                              (sublis sigma2  type-op)
                              var-type? cte-type?)))
    (if sigma
      (list (compose-subst sigma
                           (compose-subst sigma2 sigma1
                                          var-type? cte-type?)
                     var-type? cte-type?)
            (sublis sigma alpha))
      (list #f #f))))


(define (var-type? x)
  (and (symbol? x)
       (string=? "alpha" (substring (symbol->string x) 0 5))))

(define (cte-type? x)
  (and (symbol? x)(not (variable? x))))
```

Pour utiliser les fonctions `compose-subst` et `unifier`, on a besoin de charger les fonctions sur les termes du chapitre 17 §2 et §3.
Enfin, on peut ajouter une fonction qui affiche le typage sous forme d'un séquent :

```
(define (afficheTypage lambda-terme)
  (let ((type-env0 (initialise-type-env  lambda-terme)))
    (bind (sigma son-type) (typage type-env0 lambda-terme)
      (if son-type
        (display-alln
          (sublis sigma type-env0)  "  I-  "  lambda-terme " : " son-type)
        (display-alln lambda-terme "  non typable")))))
```

Cette fonction fait appel à `initialise-type-env` pour créer un environnement de type dans lequel chaque variable libre du lambda terme a pour type une nouvelle variable de type. On en profite aussi pour réinitialiser la fonction `genVar`.

```
(define (initialise-type-env  lambda-terme)
  (set! genVar (creer-genVar "alpha"))
  (let ((Lvar (lambda:Libres lambda-terme)))
    (map (lambda (x)(cons x (genVar))) LVar)))
```

Quelques tests

```
? (afficheTypage '(lamb x (lamb y x)))
()  |-  (lamb x (lamb y x)) : (-> alpha1 (-> alpha2 alpha1))
```

```
? (afficheTypage '(lamb x (lamb y y)))
()  |-  (lamb x (lamb y y)) : (-> alpha1 (-> alpha2 alpha2))

? (afficheTypage '(x x)) --> (x x)  non typable

? (afficheTypage '(lamb x (y x)))
((y . ( -> alpha2 alpha3))) |-  (lamb x (y x)) : (-> alpha2 alpha3)

? (afficheTypage '((f x)(g x)))
((g . ( -> alpha2 alpha5)) (x . alpha2) (f . (-> alpha2 alpha4)))
                       |- ((f x) (g x)) : alpha6

? (afficheTypage '((f x)(f y))) --> ((f x) (f y))  non typable
```

Structure du système d'inférence de type

`(afficheTypage lambdaTerme)`
Affiche le type principal d'un lambda terme sous forme d'un jugement
$typeEnv \vdash lambdaTerme : type$.

`(initialise-type-env lambda-terme)`
Construit un environnement de type initial pour les variables libres du lambda terme.

`(lambda:Libres lambda-terme)`
Liste des variables libres d'un lambda terme.

`(typage type-env lambdaTerme)`
Infère la liste (σ , τ) telle que le typage du lambda terme soit
$\sigma \, \Gamma_O \vdash lambdaTerme : \tau$.

`(LireEnv var type-env)`
Rend le type de la variable dans l'environnement de type.

`(typage-abst type-env abstr)`
Infère le typage d'une abstraction.

`bind-let*`
Macro du chapitre 9 pour réaliser des liaisons séquentielles destructurantes.

`(genvar)`
Pour créer des variables de type préfixées par `alpha`.

`(valeur-subst subst var)`
Valeur d'une substitution sur une variable.

`(varAbstrac abstr)`

La variable d'une abstraction.

`(corpsAbstrac abstr)`
Le corps d'une abstraction.

`(typage-app type-env app)`
Infère le typage d'une application.

`(operator app)`
La partie opérateur d'une application.

`(operand app)`
La partie opérande d'une application.

`(unifier terme1 terme1 var? cte?)`
Calcule l'unificateur principal de deux termes ou rend `#f` (cf. chapitre 18 §3).

`(compose-subst subst1 subst2 var? cte?)`
Calcule la a-liste représentant la composée de deux substitutions (cf. chapitre 17 §3)

`(var-type? x)`
Pour distinguer les variables de types.

`(cte-type? x)`
Pour distinguer les constantes de types.

20.9 Isomorphisme de Curry Howard

Les règles pour le typage des lambda termes (page 615) doivent rappeler au lecteur celles pour le connecteur logique $\Rightarrow$ en déduction naturelle :

$$\frac{\Gamma\,,\,\varphi \vdash \psi}{\Gamma \vdash \varphi \Rightarrow \psi} \qquad \frac{\Gamma \vdash \varphi \Rightarrow \psi \qquad \Gamma \vdash \varphi}{\Gamma \vdash \psi}$$

et l'axiome

$$\frac{\psi \in \Gamma}{\Gamma \vdash \psi}$$

En effet, ces règles sont exactement celles du typage quand on oublie le lambda terme et que l'on assimile les propositions à des types en identifiant l'implication logique $\Rightarrow$ avec le symbole $\rightarrow$ du type fonctionnel. Cette remarque est le point de départ d'une importante relation entre la déduction naturelle et le lambda calcul typé.

Inférence de type et déduction naturelle

Pour préciser cet aspect, essayons de donner une *représentation d'une preuve* dans cette logique (dite minimale) de l'implication. De façon plus précise, on va compléter les règles précédentes par l'indication de la preuve de chaque formule.

Il faut pouvoir désigner les règles utilisées pour combiner des preuves et nommer les hypothèses employées.

Pour nommer les hypothèses, on les fait précéder d'un identificateur. On remplace une suite d'hypothèses $\{\varphi_1, \ldots, \varphi_n\}$ par $\{x_1 : \varphi_1, \ldots, x_n : \varphi_n\}$ où les x_i sont des variables distinctes désignant chacune la preuve de la formule qui la suit.

De façon générale, la notation $P : \varphi$ signifie que P représente une preuve de la formule φ.

On va définir le jugement $\Gamma \vdash P : \varphi$, il se lit : sous les hypothèses Γ on a une preuve P de la formule φ. Pour prouver un tel jugement, on décore les règles précédentes avec la méthode utilisée pour passer des hypothèses à la conclusion.

• *La règle d'élimination de* $\Rightarrow$

On combine une preuve P_1 de $\Gamma \vdash \varphi \Rightarrow \psi$ avec une preuve P_2 de $\Gamma \vdash \varphi$ pour obtenir une preuve de $\Gamma \vdash \psi$; on note $(P_1\ P_2)$ cette combinaison, d'où la règle :

$$\frac{\Gamma \vdash P_1 : \varphi \Rightarrow \psi \qquad \Gamma \vdash P_2 : \varphi}{\Gamma \vdash (P_1\ P_2) : \psi}$$

• *La règle d'introduction de* $\Rightarrow$

On part d'une preuve P de $\Gamma , \varphi \vdash \psi$ et on élimine l'hypothèse φ pour obtenir une preuve de $\Gamma \vdash \varphi \Rightarrow \psi$. Appelons x la preuve de l'hypothèse φ, on a donc $\Gamma , x : \varphi \vdash P : \psi$. Alors la preuve de $\varphi \Rightarrow \psi$, obtenue par élimination de φ, sera notée $\lambda x . P$. D'où la règle :

$$\frac{\Gamma , x : \varphi \vdash P : \psi}{\Gamma \vdash \lambda x . P : \varphi \Rightarrow \psi}$$

On voit donc que les lambda termes servent à coder des preuves. L'identité des règles ci-dessus et des règles de typage[4] montre le résultat suivant appelé isomorphisme de Curry-Howard.

Théorème 7 Le lambda terme M représente une preuve de la formule φ si et seulement si il est typable de type φ.

Par exemple, si l'on étiquette la preuve :

[4] En identifiant $\Rightarrow$ avec $\rightarrow$ et les variables propositionnelles avec les variables de type.

$$\frac{\dfrac{\varphi\,,\,\psi \vdash \varphi}{\varphi \vdash \psi \Rightarrow \varphi}}{\vdash \varphi \Rightarrow (\psi \Rightarrow \varphi)}$$

de $\varphi \Rightarrow (\psi \Rightarrow \varphi)$. On obtient :

$$\frac{\dfrac{x : \varphi\,,\, y : \psi \vdash x : \varphi}{x : \varphi \vdash \lambda y . x : \psi \Rightarrow \varphi}}{\vdash \lambda x . \lambda y . x : \varphi \Rightarrow (\psi \Rightarrow \varphi)}$$

On constate que $\varphi \rightarrow (\psi \rightarrow \varphi)$ est bien le type du lambda terme $\lambda x . \lambda y . x$.

Cet isomorphisme a de nombreuses applications, en voici deux.

Elimination des coupures

Supposons que nous ayons une preuve de la forme suivante :

$$\frac{\dfrac{\Gamma\,,\,\varphi \vdash \psi}{\Gamma \vdash \varphi \Rightarrow \psi} \qquad \Gamma \vdash \varphi}{\Gamma \vdash \psi}$$

Alors, on peut prouver plus directement $\Gamma \vdash \psi$: il suffit de substituer la preuve de $\Gamma \vdash \varphi$ à chaque utilisation de l'hypothèse φ dans la preuve de $\Gamma\,,\,\varphi \vdash \psi$. Cette opération de réduction sur les preuves s'appelle une *coupure*. Quelle est son expression avec les lambda termes qui codent les preuves ? Décorons l'arbre de preuve précédent :

$$\dfrac{\dfrac{\Gamma\,,\,x:\varphi \vdash P:\psi}{\Gamma \vdash \lambda x.P:\varphi \Rightarrow \psi} \qquad \Gamma \vdash N:\varphi}{\Gamma \vdash ((\lambda x.P)\,N):\psi}$$

La preuve réduite consiste à remplacer les occurrences libres de x dans P par N, c'est le terme $P[x \leftarrow N]$. La coupure en logique correspond donc à la bêta réduction en lambda calcul. Comme les lambda termes typés sont fortement normalisables, il en découle que l'on ne peut pas faire indéfiniment des coupures dans une preuve et qu'il y a une notion de preuve irréductible.

Programmer c'est prouver !

Le codage de la preuve de $\varphi \Rightarrow \psi$ par l'abstraction $\lambda\,x\,.\,P$ à partir des preuves x et P de φ et ψ conduit[5] à donner une interprétation constructive (ou intuitionniste) au connecteur $\Rightarrow$: il transforme une preuve de φ en une preuve de ψ. Cette interprétation constructive s'étend aux connecteurs $\wedge$ et $\vee$ dans le cadre de la déduction naturelle intuitionniste. De plus, la relation entre type et formule d'une part, et lambda terme typé et preuve d'autre part, peut s'étendre entre certaines logiques d'ordre supérieur et certains systèmes de types très généraux.
Si l'on assimile les lambda termes à des programmes (comme on l'a fait avec un fragment de Scheme), alors les types s'interprètent comme une *spécification*[6]. L'intérêt de chercher à étendre cet isomorphisme à des systèmes de type de plus en plus expressifs est justifié par le paradigme suivant : c'est en prouvant la spécification que l'on trouve un programme qui la satisfait. Cette approche prometteuse de la programmation est actuellement du domaine de la recherche.

20.10 De la lecture

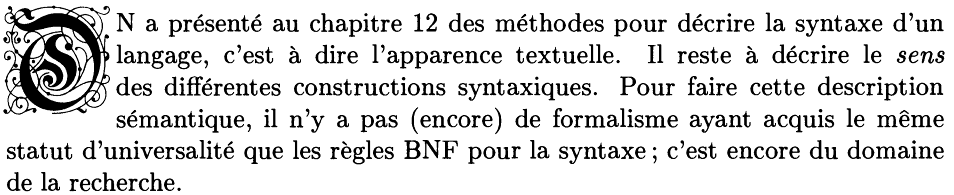
On trouvera une présentation encyclopédique du lambda calcul dans le livre de Barendregt [Bar84].
La lecture du livre de Lalement [Lal90] est suffisante pour avoir les preuves des théorèmes indiqués dans ce chapitre.
On peut approfondir l'inférence de type dans divers livres sur ML, [WL93, CM95].
Pour les résultats théoriques récents sur le lambda calcul typé et ses liens avec la programmation, on dispose de l'ouvrage de Krivine [Kri90].

[5] Historiquement les choses se sont passées dans l'ordre inverse.

[6] Par exemple, si un système de type permet de spécifier qu'un objet est non seulement une liste mais qu'elle est triée, alors le typage peut être considéré comme une spécification.

Chapitre 21

Sémantique des langages de programmation



ON a présenté au chapitre 12 des méthodes pour décrire la syntaxe d'un langage, c'est à dire l'apparence textuelle. Il reste à décrire le *sens* des différentes constructions syntaxiques. Pour faire cette description sémantique, il n'y a pas (encore) de formalisme ayant acquis le même statut d'universalité que les règles BNF pour la syntaxe ; c'est encore du domaine de la recherche.
La sémantique est l'une des principales interactions entre logique et informatique et illustre bien la célèbre prophétie de J. McCarthy : «*It is reasonable to hope that the relationship between computation and mathematical logic will be as fruitful in the next century that between analysis and physic in the last.*»

On dispose en général du manuel de l'utilisateur faisant une description du langage en langue naturelle. On a parfois en plus une présentation dans l'un des nombreux formalismes proposés pour décrire la sémantique : un interprète dans un langage de référence, la sémantique naturelle, la sémantique dénotationnelle, la sémantique opérationnelle, la méthode des assertions, ... On va présenter des exemples en utilisant les deux premières méthodes.

On distingue traditionnellement la *sémantique statique* et la *sémantique dynamique*. La première concerne les propriétés que l'on peut vérifier directement sur la structure d'un programme, c'est-à-dire essentiellement le bon typage des expressions. On ne traitera pas ce point qui a déjà été abordé partiellement à l'occasion du lambda calcul typé. La sémantique dynamique concerne l'exécution des programmes, c'est cet aspect que l'on considère dans ce chapitre sous le nom de sémantique.

Comme la sémantique n'a que faire des détails textuels d'écriture d'un programme, on décrit la sémantique à partir d'une syntaxe abstraite du langage. On considère les différentes constructions d'un langage non comme des chaînes, mais comme des termes pour un certain système de termes. On a vu que les listes fournissent

une représentation commode des termes. C'est en ce sens que l'on peut dire qu'en Lisp on programme directement en syntaxe abstraite. C'est l'un des avantages de la notation parenthésée que de mettre en évidence la structure profonde d'un programme.

La formalisation des aspects sémantiques n'a pas seulement pour objet de donner une description non ambiguë d'un langage, elle met aussi en évidence les différents choix qui s'offrent aux créateurs de langages. De plus, les méthodes formelles donnent les moyens pour prouver des propriétés des langages ou pour engendrer automatiquement de nouveaux outils.

La description en Scheme de divers dialectes Lisp est un exercice classique et instructif de description sémantique. Cette méthode a ses limites mais c'est un excellent point de départ avant d'aborder des techniques plus formelles. On décrit d'abord un interprète pour la sémantique des premiers Lisp puis, on finit avec un interprète par continuation de Scheme pour traiter le cas de la primitive `call/cc`.

On présente ensuite le formalisme de la sémantique naturelle. Pour faciliter les descriptions formelles, on sépare les aspects fonctionnels des aspects impératifs. On spécifie deux mini langages fonctionnels, le premier utilise l'appel par valeur et l'autre l'appel par nom. Pour donner une description formelle des aspects impératifs, on spécifie un langage du genre mini Pascal. Afin de ne pas noyer l'essentiel sous les détails, on négligera généralement le traitement des cas d'erreur.

21.1 Interprète pour un MiniLisp en Scheme

Comme les premiers Lisp utilisaient la liaison dynamique, on va implanter en Scheme un interprète pour un petit langage ayant une syntaxe identique à celle de Scheme mais dont la sémantique sera celle des premiers Lisp ; on le baptise MiniLisp.

Syntaxe de MiniLisp

Voici la forme des expressions admissibles en MiniLisp :

```
e ::=  nb | #t | #f | ident | (op2 e1 e2) | (op1 e)| (f e1...en)
      | (if e0 e1 e2) | (let ((x1 e1)...(xk ek)) s1...sN) | (quote e)
      | (lambda (x1...xk) s1...sN) | (begin s1 .. sN)
      | (define ident e) | (set!  ident e)
```

où `s1...sN` désigne une suite finie d'expressions.
On se contente des opérateurs binaires et unaires les plus usuels :

```
op2 ::=     +  | - | * | /  | = | <  | equal?  |  cons
op1 ::=   car  | cdr  | null? .
```

Le sens ou valeur de chaque expression va être décrit par une fonction `evaluer` écrite en Scheme. Mais attention, le sens d'une expression MiniLisp n'est pas nécessairement le même que celui de l'expression identique du langage Scheme.

Gestion de l'environnement

On sait que la valeur d'une variable dépend d'un contexte qui peut être modifié par certaines expressions. C'est le rôle de l'environnement de donner pour chaque variable sa valeur. Au début l'environnement est réduit à l'environnement initial qui décrit la valeur des identificateurs prédéfinis ; il sera ensuite modifié temporairement ou durablement lors de l'évaluation de certaines expressions comme `let`, `set!`, `define`, ...

On représente un bloc de liaisons par une a-liste et donc un environnement par une liste de a-listes.
Commençons par définir l'environnement prédéfini. On se donne la liste des *noms* de nos fonctions de base :

```
(define  *LnomsFctsPredefinies*
  '(+ - * /  = <  equal?  null?   cons car cdr))
```

Pour la *sémantique* de chacune de ces fonctions prédéfinies, on utilise celle de la fonction Scheme correspondante ; on les place dans une liste analogue

```
(define  *LfctsPredefinies*
  (list + - * / = < equal? null? cons car cdr))
```

D'où la fonction `initEnv` qui construit l'environnement initial :

```
(define  (initEnv)
  (EtendreEnv *LnomsFctsPredefinies*  *LfctsPredefinies* '()))
```

La fonction `EtendreEnv` ajoute en tête d'un environnement une nouvelle a-liste construite à partir d'une liste d'identificateurs et d'une liste de valeurs :

```
(define (EtendreEnv Lident Lval env)
  (cons (map cons Lident Lval) env))
```

Pour consulter la valeur d'une variable dans un environnement, on emploie la fonction :

```
(define (ValeurIdent ident env)
  (let ((ident.val (liaison ident env)))
    (if ident.val
      (cdr ident.val)
      (*erreur* "identificateur inconnu : " ident ))))
```

La fonction `liaison` associe à un identificateur la première paire `(ident . valeur)` trouvée dans un environnement, ou `#f` s'il est absent.

```
(define (liaison  ident env)
  (if (null? env)
    #f
    (or (assq ident (car env))
        (liaison ident (cdr env)))))
```

Les formes `set!` et `define` réalisent une modification durable de l'environnement. On utilise une modification physique pour modéliser cet effet. De façon plus précise, la fonction `ModifierEnv!` modifie physiquement la valeur associée à un identificateur donné ou ajoute en tête cette liaison si elle n'existait pas :

```
(define (ModifierEnv! ident nlle-valeur  env)
  (let ((ident.val (liaison ident env)))
    (if ident.val
      (set-cdr! ident.val nlle-valeur )
      (set-car! env (cons (cons ident  nlle-valeur)(car env))))))
```

La fonction evaluer

La fonction `evaluer` calcule la valeur d'une expression MiniLisp dans un environnement donné. Ce calcul consiste en un aiguillage selon les différentes catégories d'expressions :

- constante,
- identificateur,
- liste :
  - appel d'une forme spéciale : `if, let, quote,` ...
  - application d'une fonction prédéfinie ou utilisateur,
  - sinon on déclenche une erreur.

```
(define  (Evaluer e env)
  (cond
    ((constante? e) e)
    ((variable? e)(ValeurIdent e env))
    ((formeSpeciale? e)(EvaluerFormeSpeciale e env))
    ((application? e)
     (Appliquer (Evaluer (car e) env)(EvaluerListe (cdr e) env) env))
    (else   (*erreur*   "expression inconnue : " e))))
```

On ne considère que trois espèces de constantes :

```
(define  (constante? e)
  (or (number? e)
      (eq? '#t e)
      (eq? '#f e)))
```

Une variable est représentée par un symbole Scheme :

```
(define  (variable? e)
  (symbol? e))
```

Pour distinguer le cas d'une forme spéciale de l'appel d'une fonction ordinaire, on utilise le prédicat :

```
(define  (formeSpeciale? e)
  (and (pair? e)(memq (car e) '(quote if lambda let begin define set!))))
```

L'appel de fonction est une liste non vide qui n'est pas une forme spéciale :

```
(define (application? e)
  (and (pair? e)(not (formeSpeciale? e))))
```

Comme on utilise la stratégie de l'appel par valeur, l'appel d'une fonction consiste à appliquer la fonction sur ses arguments préalablement évalués. Pour évaluer une liste d'arguments, on utilise la fonction

```
(define  (evaluerListe Lexp env)
  (map  (lambda (e) (Evaluer e env))
        Lexp))
```

Evaluation des formes spéciales

Chaque forme spéciale nécessite un traitement particulier à cause de la variété des modes d'évaluation des arguments. L'aiguillage est réalisé par la fonction

```
(define  (EvaluerFormeSpeciale  e env)
  (case  (car e)
    ((quote)                        ;; e = (quote exp)
     (EvaluerQuotation  (cadr e)))
    ((if)                           ;; e = (if e0 e1 e2)
     (EvaluerIf (cadr e) (caddr e) (cadddr e) env))
    ((lambda)                       ;; e = (lambda (x1...xj) s1...sN)
     (EvaluerLambda  e env))
    ((let)                          ;; e = (let liaisons s1...sN)
     (EvaluerLet (cadr e) (cddr e) env))
    ((begin)                        ;; e = (begin s1...sN)
     (EvaluerSequence (cdr e) env))
    ((define)                       ;; e = (define  ident exp)
     (Evaluerdefine  (cadr e)(caddr e) env))
    ((set!)                         ;; e = (set! ident exp)
     (EvaluerSet! (cadr e)(caddr e) env))
    (else  (*erreur* "forme speciale inconnue : " e))))
```

• *quote*
La valeur de la quotation est simplement l'expression qui est quotée :

```
(define (EvaluerQuotation e)
  e)
```

• *if*
L'évaluation d'un `if` se fait en évaluant l'expression *test* et selon le résultat, on évalue l'expression `expSi` ou l'expression `expSinon` :

```
(define  (EvaluerIf test expSi expSinon  env)
  (if  (Evaluer test env)
    (Evaluer expSi env)
    (Evaluer expSinon env)))
```

• *lambda*

La valeur d'une lambda expression c'est elle-même. Pour distinguer les fonctions définies par l'utilisateur des fonctions prédéfinies, on rend en valeur une liste contenant la lambda expression et commençant par le symbole `fctUtilisateur`.

```
(define (EvaluerLambda  lambdaExp env)
  (list 'fctUtilisateur lambdaExp))
```

• *begin*

L'évaluation d'une suite (non vide) d'expressions n'a d'intérêt qu'en présence d'effets de bord car on retourne la valeur de la dernière expression :

```
(define  (EvaluerSequence  Lexp env)
  (let ((valeur (evaluer (car Lexp) env)))
    (if (null? (cdr Lexp))
      valeur
      (EvaluerSequence (cdr Lexp) env))))
```

• *let*

L'évaluation d'un `let` consiste à évaluer le corps du `let` dans l'environnement étendu par les liaisons locales du `let` :

```
(define  (EvaluerLet liaisons Lcorps env)
  (let ((LvarLocales (map  car liaisons))
        (Lexp        (map  cadr liaisons )))
    (evaluerSequence Lcorps
                     (etendreEnv LvarLocales
                                 (evaluerListe Lexp env) env))))
```

• *define et set!*

Les deux dernières formes sont à la base des effets de bord ; elles sont implantées ici par des modifications physiques de l'environnement.

L'évaluation de `define` consiste à modifier physiquement l'environnement pour lui ajouter une nouvelle liaison ou bien modifier une liaison existante.

```
(define  (EvaluerDefine ident exp  env)
  (ModifierEnv! ident (evaluer exp env) env)
  'indefini)
```

L'évaluation de `set!` est identique à celle de `define` excepté que l'on ne peut modifier qu'une liaison existante.

```
(define  (EvaluerSet! ident exp env)
  (if (liaison ident env)
    (EvaluerDefine ident exp  env)
    (*erreur*  "identificateur non defini : " ident)))
```

Application de fonctions

La fonction `appliquer` réalise l'application d'une fonction prédéfinie ou d'une fonction utilisateur sur les valeurs des arguments.

```
(define  (Appliquer fct Larg  env)
  (if (fctUtilisateur? fct)      ;; (fctUtilisateur lambdaExp)
    (AppliquerLambdaExp  (cadr fct) larg env)
    (AppliquerFctPredefinie fct Larg)))
```

Dans le cas des fonctions prédéfinies, on utilise la sémantique de la fonction Scheme de même nom.

```
(define  (AppliquerFctPredefinie fct Larg)
  (apply fct Larg))
```

L'application d'une fonction utilisateur est plus délicate : on évalue le corps de sa lambda expression dans l'environnement courant étendu par les liaisons entre les paramètres formels de la lambda et les valeurs des arguments correspondants. On verra plus loin les défauts de cette méthode d'évaluation dite *dynamique.*

```
(define  (AppliquerLambdaExp LambdaExp larg env)
  (let ((Lpar (cadr LambdaExp))
        (Lcorps (cddr LambdaExp)))
    (EvaluerSequence  Lcorps (etendreEnv Lpar Larg env))))
```

On distingue les fonctions utilisateurs des fonctions prédéfinies avec le prédicat :

```
(define  (fctUtilisateur? e)
  (and (pair? e)(eq? 'fctUtilisateur (car e))))
```

Boucle d'interaction pour MiniLisp

Pour faciliter l'utilisation de MiniLisp, on définit une boucle d'interaction en empruntant à Scheme les fonctions d'affichage et de lecture :

```
 ... --> lire --> evaluer --> afficher -->...
```

La fonction `make-top-level` construit un top level à partir d'une fonction d'évaluation ; on quitte le top level en entrant `quit` :

```
(define  (make-top-level evaluer)
  (let ((env0 (initEnv)))
    (letrec ((top-level (lambda ()
                          (display "?? ")
                          (let ((lu (read)))
                            (if (eq?   'quit  lu)
                              'au-revoir
                              (begin
                                (display-alln "== " (Evaluer lu env0))
                                (newline)
                                (top-level)))))))
      top-level)))


(define MiniLisp (make-top-level evaluer))
```

Le top level de MiniLisp est actif après appel de :

```
? (MiniLisp)
```

L'invite devient ?? et le résultat d'une évaluation est précédé de ==.

Tests pour MiniLisp

On réalise quelques tests pour mettre en évidence certaines spécificités sémantiques de MiniLisp.

Test 1 : quand une fonction utilise dans son corps une variable qui n'est pas un paramètre formel, c'est la valeur de cette variable au *moment de l'appel* qui est utilisée pour évaluer le corps (ici la valeur `a = 0`). C'est la liaison *dynamique* par opposition à la liaison *statique*, où c'est la valeur au *moment de la définition* de la fonction qui est utilisée.

```
?? (let ((a 10))
     (let ((f (lambda (x) (+ a x))))
       (let ((a 0))
         (f 5)))) ;; c'est la valeur a = 0 qui est utilisée
== 5
```

Le **test 2** montre que l'on peut définir une fonction récursive. En effet, au moment de l'appel de `fac` l'identificateur `fac` est lié à une lambda expression ce qui permet au mécanisme de la récursion d'opérer sans précaution particulière. C'est l'un des avantages de la liaison dynamique.

```
?? (define fac
     (lambda (n)
       (if (= 0 n)
         1
         (* n (fac (- n 1))))))
== indefini

?? (fac 5)
== 120
```

Le **test 3** montre, pour la même raison, que le mécanisme de la récursivité mutuelle fonctionne.

```
;; f(n) = g(n-1) + 1    et    g(n) = f(n-1)

?? (let ((f (lambda (x)(if (= 0 x) 0 (+ 1 (g (- x 1))))))
         (g (lambda (x)(if (= 0 x) 0 ( f (- x 1))))))
     (f 5))
== 3
```

Le **test 4** montre que l'on peut redéfinir des fonctions primitives.
Tout d'abord, on sauvegarde l'ancienne définition :

```
?? (define  old*  *)
```

Ensuite, on donne la nouvelle définition de la multiplication. Quand le premier opérande est nul, on rend 0 sans s'occuper de l'autre opérande. La nouvelle définition se réfère à `old*` et non à `*` pour éviter une récursivité parasite.

```
?? (define *  (lambda (x y)(if (= 0 x) 0 (old* x y))))

?? (*  0  'a)
== 0
```

Dans le **test 5**, le `let` a pour valeur 2 alors que pour `y=10` et `z=1` on attendait la valeur 11 ! C'est le phénomène dit du «funarg descendant». Quand on passe une fonction en paramètre d'une autre fonction, il peut y avoir capture de variable comme on l'a vu en lambda calcul.

```
?? (let ((application (lambda (f y)(f y)))
         (y 10))
      (application (lambda (z)(+ z y)) 1))
== 2
```

Mais le **test 5bis** montre que si l'on renomme `y` en `x` dans `(lambda (f y)(f y)))`, le problème disparaît :

```
?? (let ((application (lambda (f x)( f x)))
         (y 10))
     (application (lambda (z)(+ z y)) 1))
== 11
```

Le **test 6** met en évidence le phénomène dit du «funarg ascendant». Quand on évalue une lambda expression, on la retourne telle quelle. Cela peut provoquer la perte d'une liaison d'où le message d'erreur pour `x` : on a perdu le fait que `x=2`.

```
?? (let ((add (lambda (x)(lambda (y)(+ x y)))))
      (let ((add2 (add 2)))
        (add2 4)))
ERREUR -- identificateur inconnu : x
```

Les Lisp récents comme Scheme et Common Lisp utilisent la liaison statique qui évite les inconvénients illustrés dans les tests 5 et 6.

21.2 Interprète pour MiniScheme

Il est instructif d'étudier les modifications à apporter à MiniLisp pour remplacer la liaison statique par la liaison dynamique. On appelle MiniScheme le langage correspondant. Ces modifications entraînent quelques changements dans le code de l'interprète. On ne réécrit que les fonctions nouvelles ou modifiées, les modifications sont signalées par des *** en commentaire.

Le point essentiel réside dans l'évaluation d'une lambda expression, on sait qu'en Scheme sa valeur est une *fermeture*, c'est-à-dire un couple constitué de la lambda expression et de l'environnement de définition. En effet, on doit sauvegarder les

valeurs des variables au moment de la définition pour pouvoir les utiliser au moment de l'appel. On représente une fermeture par une liste commençant par le symbole `fermeture` : `(fermeture LambdaExp env)`, d'où la fonction :

```
(define (evaluerLambda lambdaExp  env)
  (list 'fermeture lambdaExp  env)) ; ***
```

On n'a plus besoin de passer l'environnement en paramètre dans l'application d'une fonction puisqu'il sera récupéré dans la fermeture de la fonction.

```
(define  (Evaluer e env)
  (cond
    ((constante? e) e)
    ((variable? e)(ValeurIdent e env))
    ((formeSpeciale? e)(EvaluerFormeSpeciale e env))
    ((application? e)
     (Appliquer (Evaluer (car e) env)(EvaluerListe (cdr e) env) )) ;***
    (else   (*erreur*  "expression inconnue : " e))))
```

Dans la fonction `appliquer`, on remplace `appliquerLambdaExp` par `appliquerFermeture` :

```
(define  (Appliquer fct Larg)
  (if (fctUtilisateur? fct)
    (AppliquerFermeture  (cadr fct) (caddr fct) larg ) ; ***
    (AppliquerFctPredefinie fct Larg)))
```

La valeur d'une lambda expression étant une fermeture, on doit modifier le prédicat qui sert à reconnaître les fonctions utilisateurs :

```
(define  (fctUtilisateur?  e)
  (and (pair? e)(eq? 'fermeture (car e)))) ; ***
```

La fonction `appliquerFermeture` procède comme `appliquerLambdaExp` mais avec une différence essentielle : on utilise l'environnement de définition stocké dans la fermeture au lieu de l'environnement d'appel.

```
(define  (AppliquerFermeture  lambdaExp env0 larg)
 ;;***   remplace appliquerLambdaExp
  (let ((Lpar (cadr lambdaexp))
        (Lexp (cddr lambdaexp)))
    (EvaluerSequence  Lexp (etendreEnv lpar larg env0))))
```

L'autre modification importante concerne la définition des fonctions récursives. On a vu en Scheme que la liaison statique ne permet pas d'utiliser le `let` pour définir des fonctions récursives, d'où l'introduction de la forme spéciale :

`(letrec ((x1 e1)...(xk ek)) s1 ..sN).`

L'ajout de `letrec` parmi les formes spéciales entraîne une légère modification des fonctions suivantes :

```
(define  (formeSpeciale? e)
  (and (pair?   e)
       (memq (car e) '(quote if lambda let begin define set! letrec))));***

(define  (evaluerFormeSpeciale  e env)
  (case  (car e)
    ((quote)  (EvaluerQuotation  (cadr e)))
    ((if)     (EvaluerIf (cadr e) (caddr e) (cadddr e) env))
    ((lambda) (EvaluerLambda  e env))
    ((let)    (EvaluerLet (cadr e) (cddr e) env))
    ((begin)  (EvaluerSequence (cdr e) env))
    ((define) (Evaluerdefine  (cadr e)(caddr e) env))
    ((set!)   (EvaluerSet! (cadr e)(caddr e) env))
    ((letrec) (EvaluerLetRec (cadr e) (cddr e) env)) ;  ***
    (else     (*erreur*  "forme speciale inconnue : " e))))
```

Le dernier aspect à préciser est l'évaluation de la forme `letrec`. C'est l'objet de la nouvelle fonction `evaluerLetrec` :

```
(define (EvaluerLetrec liaisons Lcorps env) ; ***
  (let ((LvarLocales (map  car liaisons))
        (Lexp        (map  cadr liaisons )))
    (evaluerSequence Lcorps
                     (EtendreEnv-rec LvarLocales Lexp env))))
```

Le point essentiel est la création, par la fonction `EtendreEnv-rec`, de l'environnement d'évaluation. Précisons le problème avec l'exemple de la définition de factorielle par un `letrec` :

```
(letrec ((fac (lambda (n)
                (if (= 0 n)
                  1
                  (* n (fac (- n 1)))))))
  (fac 5))
```

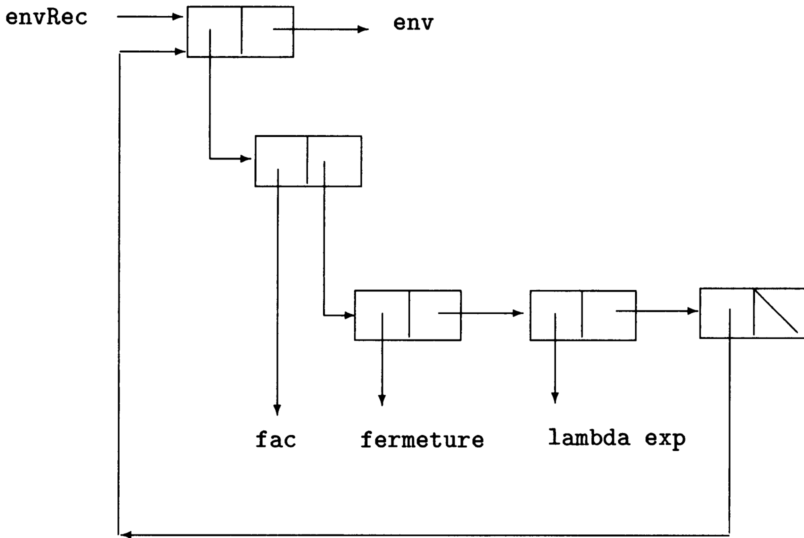
La valeur de la lambda expression est une fermeture :

`(fermeture (lambda (n)(if ...)  env)`

où l'environnement courant `env` n'a pas de raison de contenir déjà la fonction `fac` que l'on est en train de définir. Par conséquent, si l'on utilise cet environnement, à l'appel de `(fac 5)` on aura un message d'erreur car la valeur de l'identificateur `fac` ne sera pas trouvée ou bien on utilisera une valeur erronée. Il faut utiliser un environnement qui contienne la fonction que l'on en est train de définir. Cette espèce de cercle vicieux montre que cet environnement, que l'on baptise `envRec`, doit vérifier une équation de point fixe :

```
envRec =  (cons '(fac . (fermeture (lambda (n) corps)  envRec)  env)
```

Si l'on dessine les cellules représentant cette structure, on obtient une boucle dans les pointeurs.



C'est l'objet de la fonction `EtendreEnv-rec` de fabriquer cette circularité dans l'environnement courant augmenté par les liaisons du `letrec`. On réalise cette boucle de pointeurs par modification physique des liaisons ajoutées à l'environnement par le `letrec` :

```
(define (EtendreEnv-rec Lident Lexp env) ; *** nouvelle fonction
  (let ((envRec (etendreEnv Lident Lexp env)))
    (for-each (lambda (liaison exp)
                (set-cdr! liaison (evaluer exp envRec)))
              (car envRec) Lexp)
    envRec))
```

Tests pour MiniScheme

Pour tester cet interprète, on définit une boucle d'interaction par

```
(define MiniScheme (make-top-level evaluer))
```

On reprend les mêmes tests afin de faire la comparaison avec MiniLisp.
Le **test 1** montre que l'on utilise bien la valeur d'une variable au moment de la définition d'une fonction.

```
?? (let ((a 10))
     (let ((f (lambda (x) (+ a x))))
       (let ((a 0))
         (f 5)))) ;; c'est la valeur a = 10 qui est utilisée
== 15
```

Cependant le **test 1-bis** montre qu'une modification par `set!` de la valeur de cette variable au moment de l'appel est prise en compte.

```
?? (let ((a 10))
     (let ((f (lambda (x) (+ a x))))
       (set! a 0)
       (f 5)))
== 5
```

Le **test 2** montre que l'on peut toujours définir une fonction récursive par `define`

```
?? (define fac
     (lambda(n)
       (if (= 0 n)
         1
         (* n (fac (- n 1))))))
== indefini

?? (fac 5)
== 120
```

Le **test 4** montre l'utilisation du `letrec` pour les définitions récursives locales:

```
?? (letrec ((f (lambda (x)(if (= 0 x) 0 (+ 1 (g (- x 1))))))
            (g (lambda (x)(if (= 0 x) 0 ( f (- x 1))))))
     (f 5))
== 3
```

Le **test 5** montre que l'on n'a plus le phénomène de funarg descendant avec la liaison statique.

```
?? (let ((app (lambda (f y)( f y)))
         (y 10))
     (app (lambda (z)(+ z y)) 1))
== 11
```

Le **test 6** montre que l'on n'a plus le phénomène de funarg ascendant avec la liaison statique.

```
?? (let ((add (lambda (x)(lambda (y)(+ x y)))))
     (let ((add2 (add 2)))
       (add2 4)))
== 6
```

Exercice 1 *Ajouter la possibilité de définir des macros dans MiniScheme.*

21.3 Interprète Scheme par continuation

Il manque à cette description de Scheme un aspect important de ce langage: la capture de la continuation courante par la fonction prédéfinie `call/cc`[1].
L'ajout de `call/cc` va nous obliger à opérer une modification profonde de notre interprète. Pour avoir accès à la continuation du calcul d'une expression, on utilise la

[1] On peut sauter cette section en première lecture

méthode de programmation par passage de continuation introduite au chapitre 8. Autrement dit, on réécrit tout l'interprète MiniScheme dans le style CPS. Pour cela, on remplace chaque fonction par sa version par continuation. Par exemple, `evaluer` devient `cont-evaluer` de sorte que :

```
(cont-evaluer  exp  env  k)  = (k (evaluer  exp  env))
```

où la continuation `k` est une fonction qui associe une réponse à une valeur.

La continuation à utiliser dans une fonction `f` est désignée par `k-f`, par exemple la continuation utilisée par la fonction `cont-top-level` sera `k-top-level`. La programmation par continuation est plus naturelle avec le style descendant. On commence donc par la boucle d'interaction pour écrire notre nouvel interprète que l'on baptise MiniScheme-CPS.

Boucle d'interaction pour MiniScheme-CPS

```
(define (MiniScheme-cps)
  (cont-top-level (initEnv)  (initCont)))
```

Au départ, la continuation est l'application identité :

```
(define (initCont)
  (lambda (v) v))
```

La version par continuation de la boucle `top-level` peut s'écrire :

```
(define (cont-top-level env k)
  (display  "?? " )
  (cont-evaluer (read) env (k-top-level env  k)))
```

C'est la continuation `k-top-level` qui est chargée de rappeler `cont-top-level` après avoir affiché == suivi de la valeur calculée :

```
(define (k-top-level env  k)
  (lambda (valeur)
    (if (eq?  'quit valeur)
      'au-revoir
      (begin (display-alln "== " valeur)
             (newline)
             (cont-top-level  env  k)))))
```

La fonction `cont-evaluer`

La fonction d'évaluation conserve la même structure :

```
(define (cont-evaluer e env k)
  (cond
    ((constante? e) (k e))
    ((variable? e)(cont-valeurIdent e env k))
    ((formeSpeciale? e)(cont-evaluerFormeSpeciale e env k))
    ((application? e)(cont-evaluerListe e env (k-application k)))
    (else (cont-erreur "expression inconnue " e env k))))
```

On ne change pas les prédicats `constante?`, `identificateur?`, `formeSpeciale?` et `application?` .
L'évaluation d'une liste est typique de la transformation en style CPS. La fonction `cont-evaluerListe` évalue le premier élément avec `cont-evaluer` et charge sa continuation `k-evaluerListe` d'évaluer les autres et de passer le résultat à la fonction `k` :

```
(define  (cont-evaluerListe Lexp env  k)
  (if (null? Lexp)
    (k '())
    (cont-evaluer  (car Lexp) env  (k-evaluerListe (cdr Lexp) env k))))

(define (k-evaluerListe Lexp env k)
  (lambda (v)
    (if (null? Lexp)
      (k (list v) )
      (cont-evaluerListe Lexp env  (lambda (Lval)(k (cons v Lval)))))))
```

Dans `cont-evaluer`, la fonction `cont-evaluerListe` prend en paramètre la continuation `(k-application k)`. Cette continuation a pour mission d'appliquer la tête de la liste rendue par `cont-evaluerListe` à son reste.

```
(define (k-application k)
  (lambda (Lv)(cont-Appliquer (car Lv) (cdr Lv) k)))
```

Le style CPS conduit donc à préciser *l'ordre* des évaluations des arguments d'une fonction. Cela peut être un défaut quand, comme dans Scheme, on ne souhaite pas préciser ce point. Un moyen, assez complexe, pour l'éviter est de demander que la sémantique soit invariante par permutation de l'ordre des évaluations des arguments des fonctions, on ne développera pas ce point de vue.
En contrôlant la suite des calculs, la programmation par continuation a en revanche l'avantage de permettre de s'échapper d'une expression en cas d'erreur tout en revenant au top level de l'interprète.

```
(define  (cont-erreur  message e env  k)
  (display-alln "erreur: "  message  " " e)
  (cont-top-level env  k))
```

Gestion de l'environnement

Le seul changement dans l'environnement initial est l'adjonction de la fonction `call/cc`, sa valeur sera définie plus loin.

```
(define  *LnomsFctsPredefinies*
  '(+ - * /  = <  equal?  null?  cons car cdr  call/cc))

(define  *LfctsPredefinies*
  (list + - * / = < equal? null? cons car cdr  'call/cc))

(define  (initEnv)
  (EtendreEnv *LnomsFctsPredefinies*  *LfctsPredefinies* '()))
```

La gestion de l'environnement par continuation est une traduction immédiate des fonctions en style CPS.

```
(define (cont-ValeurIdent ident env k)
  (cont-liaison ident env (k-valeurIdent ident env k)))

(define (cont-liaison  ident env k)
  (if (null? env)
    (k #f)
    (let ((ident.val (assq ident (car env))))
      (if ident.val
        (k ident.val)
        (cont-liaison  ident (cdr env) k)))))

(define (k-valeurIdent  ident env k)
  (lambda (pair)
    (if pair
      (k (cdr pair))
      (cont-erreur "identificateur inconnu " ident env k))))


(define (cont-ModifierEnv! ident val1 env k)
  (cont-liaison ident env
              (lambda (ident.val)
                (if ident.val
                  (k (begin (set-cdr! ident.val val1) 'indefini))
                  (k (begin (set-car! env (cons (cons ident val1)(car env)
                             'indefini)))))))
```

Les fonctions pour étendre l'environnement sont pratiquement inchangées :

```
(define (EtendreEnv Lident Lval env)
  (cons (map cons Lident Lval) env))

(define (EtendreEnv-rec Lpar Lexp env)
  (let ((envRec (EtendreEnv Lpar Lexp env)))
    (for-each (lambda (liaison exp)
              (set-cdr! liaison (cont-evaluer exp envRec (lambda (x) x))))
               (car envRec)  Lexp)
    envRec))
```

Mécanisme du `call/cc` et appel de fonction

L'ajout de `call/cc` à MiniScheme oblige à introduire des modifications dans le traitement de l'appel de fonction.

• Une des façons d'expliquer le fonctionnement de la *fonction* `call/cc` est de détailler l'évaluation de `(cont-evaluer (call/cc e) env k)`. Quand l'évaluateur reconnaît l'appel de `call/cc`, il y a calcul de la valeur $\overline{e}$ de son argument `e` dans l'environnement `env`. Puis, et c'est le point-clé, cet appel est transformé en l'appel de $\overline{e}$ sur la réification de `k` : `(`$\overline{e}$ `(reifier k))`.

L'opération de réification consiste à associer à la valeur `k` du langage d'implantation (Scheme) une valeur `(reifier k)` du langage implanté (MiniScheme-CPS). Revenons à la valeur de `e`, comme `e` doit être une lambda expression à un paramètre, sa valeur est une fermeture de la forme `(fermeture (lambda (k1) corps) env)`. On est donc ramené à l'application d'une fermeture, sa valeur est la valeur du `corps` dans l'environnement `env` étendu par la liaison entre `k1` et `(reifier k)`. Si l'évaluation du corps ne provoque pas d'appel à `k1`, il n'y a rien de spécial. En revanche, si l'on rencontre un appel du type `(k1 e1)`, on doit calculer la valeur de `((reifier k) e1)`. Par définition, cette valeur est donnée par (k $\overline{e1}$), on a donc abandonné la continuation courante pour utiliser la continuation capturée.

En résumé, voici les points à considérer dans l'implantation de l'application :
- il faut ajouter le cas de l'appel de `call/cc` quand on applique une fonction prédéfinie,
- on choisit de représenter la réification d'une continuation `k` par la liste `(continuation k)` et il faut reconnaître l'application d'une telle valeur à une expression.

La version CPS de `appliquer` consiste encore à faire l'aiguillage entre les divers types de fonctions, mais il faut ajouter le cas des continuations capturées par `call/cc`.

```
(define  (cont-appliquer fct Larg  k)
  (cond
    ((fctUtilisateur? fct)
     (cont-AppliquerFermeture  (cadr fct) (caddr fct) Larg k) )
    ((continuation? fct)
     (cont-AppliquerContinuation (cadr fct) (car larg)))
    (else
     (cont-AppliquerFctPredefinie fct Larg k))))
```

• Pour les *fonctions utilisateurs*, il n'y a pas de réel changement.

```
(define  (fctUtilisateur?  e)
  (and (pair? e)(eq? 'fermeture (car e))))

(define (cont-AppliquerFermeture  lambdaExp env0 Larg k)
  (let ((Lpar (cadr lambdaExp))
        (Lexp (cddr lambdaExp)))
    (cont-evaluerSequence  Lexp (etendreEnv Lpar Larg env0) k)))
```

On reconnaît une réification de continuation par le prédicat :

```
(define (continuation? e)
  (and (pair? e)(eq? 'continuation (car e))))
```

L'application d'une telle réification est donnée par :

```
(define (cont-AppliquerContinuation k val)
  (k val))
```

• Pour l'application des *fonctions prédéfinies*, il suffit d'ajouter le cas de la fonction `call/cc` :

```
(define (cont-AppliquerFctPredefinie fct Larg k)
  (if (eq? 'call/cc fct)
    (cont-Appliquer  (car Larg) (list (reifier k)) k)
    (k (apply fct Larg))))
```

On choisit de représenter la réification d'une continuation par une liste commençant par le symbole `continuation` :

```
(define (reifier k)
  (list 'continuation  k))
```

Evaluation par continuation des formes spéciales

On réalise le traitement de chaque forme spéciale par une fonction particulière.

```
(define  (cont-evaluerFormeSpeciale  e env k)
  (case  (car e)
    ((quote)   (cont-evaluerQuotation   (cadr e) k))
    ((if)      (cont-evaluerIf (cadr e) (caddr e) (cadddr e) env k))
    ((lambda)  (cont-evaluerLambda      e                    env k))
    ((let)     (cont-evaluerLet         (cadr e) (cddr  e)   env k))
    ((begin)   (cont-evaluerSequence    (cdr  e)             env k))
    ((define)  (cont-evaluerdefine      (cadr e) (caddr e)   env k))
    ((set!)    (cont-evaluerSet!        (cadr e) (caddr e)   env k))
    ((letrec)  (cont-evaluerLetRec      (cadr e) (cddr  e)   env k))
    (else      (cont-erreur "forme speciale inconnue : " e   env k))))
```

L'évaluation de la citation consiste à appliquer la continuation `k` au résultat usuel :

```
(define (cont-evaluerQuotation exp-quotee k)
  (k exp-quotee))
```

L'évaluation du `if` consiste à évaluer le `test` puis à déléguer à la continuation le choix d'évaluer `expSi` ou `expSinon` selon la valeur du `test` :

```
(define  (cont-evaluerIf test expSi expSinon  env k)
  (cont-evaluer test env (k-evaluerIf expSi expSinon env k)))

(define (k-evaluerIf expSi expSinon env k)
  (lambda (v-test )
    (if  v-test
      (cont-evaluer  expSi env  k)
      (cont-evaluer  expSinon env  k))))
```

L'évaluation d'une lambda consiste à appliquer la continuation `k` au résultat usuel.

```
(define (cont-evaluerLambda  lambdaExp env k)
  (k (list 'fermeture lambdaExp env)))
```

L'évaluation du `let` commence par évaluer les expressions des variables locales puis la continuation consiste à évaluer le corps dans l'environnement étendu :

```
(define (cont-evaluerLet liaisons Lcorps env  k)
  (let ((LvarLocales (map  car liaisons))
        (Lexp        (map  cadr liaisons )))
    (cont-evaluerListe Lexp env (k-evaluerLet LvarLocales  Lcorps env  k))))

(define (k-EvaluerLet LvarLocales  Lcorps env  k)
  (lambda (Lval)
    (cont-EvaluerSequence Lcorps (etendreEnv LvarLocales  Lval env) k)))
```

L'évaluation d'une séquence consiste à évaluer la première puis à évaluer les autres – s'il y en a d'autres – en rendant la valeur de la continuation `k` sur la dernière :

```
(define (cont-evaluerSequence Lexp env k)
  (if (null?  Lexp)
    (k 'indefini)
    (cont-evaluer  (car Lexp) env  (k-evaluerSequence (cdr Lexp) env k))))

(define (k-evaluerSequence Lexp env k)
  (lambda (v)
    (if (null? Lexp)
      (k v)
      (cont-evaluerSequence  Lexp  env  k))))
```

L'évaluation d'une définition consiste à évaluer l'expression puis à effectuer la modification de l'environnement correspondant :

```
(define (cont-evaluerDefine ident exp env k)
  (cont-evaluer exp env (k-evaluerDefine ident env k)))

(define (k-evaluerDefine ident env k)
  (lambda (val)(cont-ModifierEnv!  ident  val  env k)))
```

L'évaluation de `set!` est analogue mais on exige que la liaison existe déjà :

```
(define (cont-evaluerSet! ident exp env  k)
  (cont-liaison  ident env  (k-evaluerSet! ident exp env k)))

(define (k-evaluerSet! ident exp env k) ; la continuation de set!
  (lambda (liaison)
    (if liaison
      (cont-evaluer exp env (k-evaluerDefine ident env k))
      (cont-erreur "identificateur non  defini " ident  env k))))
```

Enfin, l'évaluation de `letrec` est basée sur la fonction `EtendreEnv-rec` de construction de l'environnement récursif :

```
(define  (cont-evaluerLetrec  liaisons Lcorps env k)
  (let ((LvarLocales (map  car liaisons))
        (Lexp        (map  cadr liaisons)))
    (cont-EvaluerSequence Lcorps (EtendreEnv-rec LvarLocales Lexp env) k)))
```

Testons l'utilisation de `call/cc` dans MiniScheme-CPS

Dans le premier test, on vérifie bien que la continuation `k` est représentée par la fonction `(lambda (x)(* 4 (+ x 2))` :

```
?? (* 4 (+ (call/cc (lambda (k)(k (* 5 8))))
           2))
== 168
```

Le deuxième test montre la capture d'une continuation dans une variable globale `*stop*` :

```
?? (define *stop* '?)

?? (call/cc (lambda (k)(set! *stop* k)))

?? (/ (* 8 (*stop* 5)) 0)
== 5
```

Enfin, on utilise la continuation `*stop*` pour arrêter le calcul du produit des éléments d'une liste quand on rencontre `0` :

```
?? (define produitL
     (lambda (L)
       (if (null? L)
         1
         (if (= 0 (car L))
           (*stop* 0)
           (* (car L)(produitL (cdr L)))))))

?? (produitL '(1 2 0 a b))
== 0
```

21.4 Sémantique naturelle

Ces descriptions de sémantique par des programmes Scheme ne sont pas entièrement satisfaisantes. En effet, l'utilisation de modifications physiques de l'environnement pour exprimer la sémantique des expressions `set!`, `define` et `letrec` n'est pas toujours limpide. De plus, ce type de description sous-entend une connaissance a priori du langage Scheme. On aimerait donner une description plus mathématique. Pour cela, on va introduire le formalisme dit de la *sémantique naturelle*.

Exécution = preuve

C'est une sémantique où l'exécution d'un programme est vue comme la *preuve* d'une formule logique. Pour faire cette preuve, on se donne un système formel qui décrit la façon de prouver chaque construction syntaxique du langage concerné. Ce système formel rappellera le système de la déduction naturelle d'où le nom donné à cette approche de la sémantique. On va reprendre dans cet esprit différents aspects qui peuvent être présents dans un langage de programmation : les aspects

fonctionnels, les aspects impératifs, les déclarations, ... On illustrera chaque aspect à l'aide d'un langage jouet.

L'idée de base consiste à décrire la sémantique d'une construction d'un langage au moyen d'un jugement logique. Par exemple, on spécifiera l'évaluation des expressions d'un langage par un jugement de la forme $env \vdash e : v$. Le jugement sera vrai si dans l'environnement env, l'expression e a pour valeur v. La description sémantique de l'évaluateur consiste à se donner des règles de déduction pour prouver de tels jugements.
On a déjà utilisé, sans le dire, cette méthode de description sémantique. En effet, le typage des termes du lambda calcul typé a été défini avec un tel système de règles.

Pour donner une première idée de la méthode, décrivons l'évaluation d'expressions booléennes.

Exemple des expressions booléennes

On suppose qu'il y a deux constantes prédéfinies $vrai$ et $faux$, des variables (distinctes des symboles $vrai$ et $faux$) et le connecteur binaire ET :

$$exp \quad ::= \quad vrai \mid faux \mid var \mid (ET \quad exp \quad exp)$$

La présence de variables entraîne l'utilisation d'un environnement. Ici environnement est une fonction qui associe une valeur booléenne à une variable. Un environnement env, concerne toujours qu'un nombre fini de variables, aussi peut-on l'écrire sous la forme d'une suite $env = \{x_1 : b_1, \ldots, x_k : b_k\}$ signifiant que la variable x_i a pour valeur le booléen $b_i = env(x_i)$.

L'évaluation d'une constante ou d'une variable est immédiate et correspond aux axiomes de notre sémantique :

$$\frac{}{env \vdash vrai : vrai} \qquad \frac{}{env \vdash faux : faux} \qquad \frac{}{env \vdash x : env(x)}$$

L'évaluation du ET est définie par la règle triviale demandant d'évaluer les arguments puis d'en prendre le $\wedge$ logique

$$\frac{env \vdash e_1 : b_1 \qquad env \vdash e_2 : b_2}{env \vdash (ET\, e_1\, e_2) : b_1 \wedge b_2}$$

Il s'agit ici du ET à la Pascal, mais comment modifier cette règle du ET pour avoir un *ET séquentiel* comme en Scheme ?
Il faut deux règles, l'une dans le cas où le premier argument a une valeur fausse, car alors on ne doit pas évaluer le second argument :

$$\frac{env \vdash e_1 : faux}{env \vdash (ET\, e_1\, e_2) : faux}$$

L'autre cas nécessite l'évaluation des deux arguments et on rend la valeur du deuxième :

$$\frac{env \vdash e_1 : vrai \qquad env \vdash e_2 : b_2}{env \vdash (ET\, e_1\, e_2) : b_2}$$

Calculons, avec cette sémantique, la valeur de $(ET\ vrai\ (ET\ x\ vrai)$ dans un environnement $\{x : faux\}$. Le ET externe oblige à employer la deuxième règle et le ET interne la première :

$$\frac{\dfrac{}{\{x : faux\} \vdash vrai : vrai} \qquad \dfrac{\{x : faux\} \vdash x : faux}{\{x : faux\} \vdash (ET\, x\, vrai) : faux}}{\{x : faux\} \vdash (ET\ vrai\ (ET\ x\ vrai)) : b}$$

On trouve une preuve dans le cas où on prend pour b la valeur $faux = vrai \wedge faux$.

Exercice 2 *Ajouter le connecteur OU.*

Terminons cette introduction à la sémantique naturelle par quelques remarques.

- Comme indiqué au début du chapitre, la sémantique utilise de préférence la syntaxe abstraite des constructions du langage ; ici on a utilisé une forme préfixe pour représenter les expressions booléennes.

- La preuve est guidée par la syntaxe de l'expression à évaluer. Mais il se peut que plusieurs règles puissent pouvoir s'appliquer à un moment, mais il n'est pas obligé qu'elles conduisent toutes à une preuve (voir dans cet exemple, le cas des règles du ET).

- Considérons le calcul de la valeur de l'expression $(ET\ vrai\ y)$ dans l'environnement $\{x : vrai\}$. On ne trouvera pas de preuve pour le jugement

  $\{x : vrai\} \vdash (ET\ vrai\ y)$ car il n'y pas de règle permettant de prouver un jugement de la forme $\{x : vrai\} \vdash y : b$. Cela correspond au cas d'erreur d'une variable non définie, mais ici on n'a pas besoin de préciser explicitement les cas d'erreur car ils seront non prouvables.

21.5 Sémantique naturelle de MiniML

On commence par spécifier les aspects purement fonctionnels : évaluation d'expressions, définition et application de fonctions, ... Pour illustrer ces aspects, on considère un mini langage fonctionnel qui possède une syntaxe abstraite voisine de celle de MiniScheme où l'on a supprimé tous les aspects impératifs.

Syntaxe abstraite

On désigne par *vrai* et *faux* les deux booléens et on admet seulement ces deux booléens comme valeur d'un test. Les autres données sont les nombres et les listes.

Pour changer, on abandonne un aspect fondamental de Lisp : on fait la distinction, au niveau des notations, entre un *objet liste et un appel de fonction.* On ne considère que des listes *plates*, une liste s'écrira sous la forme `(:: v1 ... vn)`. Par raison d'homogénéité la liste vide est notée `(::)`. Le symbole `::`, qui préfixe toute liste permet de faire la distinction entre un appel de fonction et une liste (on convient qu'une fonction ne peut être désignée par `::`). Une conséquence importante de cette distinction est que l'on *n'a plus besoin de quoter* une liste pour désigner sa valeur littérale, une liste est toujours une constante.

Appelons MiniML ce petit langage car on peut le considérer comme un sous-ensemble du langage ML.

Voici la syntaxe abstraite[2] de MiniML :

```
exp ::= nombre | vrai | faux |ident | (op2 e1 e2) | (op1 e)| (:: v1...vn)
      | (lambda  (x1...xk) e)  | (if e0 e1 e2)  | (f e1...eN)
      | (let ((x1 e1)...(xn en)) e) | (letrec ((f1 e1)...(fk ek)) e)

op2  ::=  + | - | * | / | egal? | < | cons
op1  ::=  null? | tete | reste
```

Il s'agit de spécifier l'évaluation d'une expression de MiniML. Le jugement de base est de forme : $env \vdash exp : v$.
Au départ, l'environnement initial *env*0 contient les liaisons entre les noms d'opérateurs prédéfinis et les fonctions associées. Pour calculer la valeur v d'une expression *exp*, il faut prouver le jugement

$$env0 \vdash exp : v$$

Règles pour MiniML et portée lexicale

• On commence par donner les règles pour les expressions qui s'évaluent immédiatement, ce seront des axiomes de notre système formel :

$$\frac{}{env \vdash nb : nb} \qquad \frac{}{env \vdash vrai : vrai} \qquad \frac{}{env \vdash faux : faux}$$

Les listes sont des constantes :

$$\frac{}{env \vdash (::\ v_1 \ldots v_n) : (::\ v_1 \ldots v_n)}$$

• La valeur d'un identificateur s'obtient en consultant l'environnement :

[2] Pour ne pas trop aloudir les notations, on prendra souvent des libertés avec la syntaxe abstraite. Par exemple, il faudrait remplacer `nombre` et `ident` par des expressions non ambiguës comme `(constante nombre)`, `(variable ident)`.

$$\frac{}{env \vdash x : env(x)}$$

• Les opérateurs arithmétiques évaluent leurs arguments. Contentons-nous de donner la sémantique de l'addition.

$$\frac{env \vdash e_1 : v_1 \qquad env \vdash e_2 : v_2}{env \vdash (+\, e_1\, e_2) : v_1 + v_2}$$

• Les fonctions sur les listes évaluent leurs arguments, on appelle encore `cons` le constructeur mais l'accès au premier élément ou au reste d'une liste est respectivement noté `tete` et `reste` :

$$\frac{env \vdash e : v_1 \qquad env \vdash L : (::\ v_2 \ldots v_n)}{env \vdash (cons\ e\ L) : (::\ v_1 \ldots v_n)}$$

$$\frac{env \vdash e : (::\ v_1 \ldots v_n)}{env \vdash (tete\ e) : v_1} \qquad \frac{env \vdash e : (::\ v_1\, v_2 \ldots v_n)}{env \vdash (reste\ e) : (::\ v_2 \ldots v_n)}$$

On n'a pas besoin de préciser les cas d'erreur pour l'emploi des fonctions `tete`, `reste` et `cons`. Il suffit de donner les règles dans les cas admissibles, les autres cas ne seront pas prouvables.

$$\frac{env \vdash e : (::)}{env \vdash (null?\ e) : vrai} \qquad \frac{env \vdash e : (::\ v_1\, v_2 \ldots v_n)}{env \vdash (null?\ e) : faux}$$

• Il y a deux règles pour le `if` selon que la valeur du test est vraie ou fausse :

$$\frac{env \vdash e_1 : vrai \qquad env \vdash e_2 : v_2}{env \vdash (if\ e_1\, e_2\, e_3) : v_2} \qquad \frac{env \vdash e_1 : faux \qquad env \vdash e_3 : v_3}{env \vdash (if\ e_1\, e_2\, e_3) : v_3}$$

Ces deux règles impliquent que les seules valeurs acceptables pour le test d'un `if` sont `vrai` et `faux`.

• On décide que ce langage MiniML utilise la portée lexicale, la valeur d'une lambda expression est une fermeture :

$$\frac{}{env \vdash (lambda\ Lparametre\ e) : (fermeture \quad (lambda\ Lparametre\ e) \quad env)}$$

• L'appel de fonction utilisateur revient à évaluer le corps de cette fonction dans l'environnement augmenté des liaisons entre paramètres formels et valeurs des arguments.

L'adjonction des liaisons $\{x_1 : v_1, \ldots, x_n : v_n\}$ à un environnement *env* est un environnement noté $env_1 = env\{x_1 : v_1, \ldots, x_n : v_n\}$. Il est défini[3] par la fonction :

$$env_1(x) = env(x) \text{ si } x \text{ est distinct des } x_j \text{ et si } x = x_i, \text{ on rend } v_i$$

[3]Pour être tout à fait complet, les opérations annexes comme la gestion de l'environnement devraient aussi faire l'objet d'une description en sémantique naturelle.

Par manque de place horizontale, on écrira en colonne les hypothèses de certaines règles :

$$\frac{\begin{array}{l} env \vdash f : (fermeture\ (lambda\ (x_1,\ldots,x_n)\ corps) \quad env_1) \\ env \vdash e_1 : v_1 \quad \ldots \quad env \vdash e_n : v_n \\ env_1\,\{x_1 : v_1,\ldots,x_n : v_n\} \vdash corps : w \end{array}}{env \vdash (f\, e_1,\ldots,{}_e n) : w}$$

• Comme il se doit, la règle pour le `let` est analogue :

$$\frac{env \vdash e_1 : v_1 \quad \ldots \quad env \vdash e_n : v_n \qquad env\,\{x_1 : v_1,\ldots,x_n : v_n\} \vdash corps : w}{env \vdash (let\ ((x_1\ e_1)\ldots(x_n\ e_n))\ corps) : w}$$

A titre d'exemple, voici l'arbre de preuve correspondant au calcul de la valeur de :

```
(let ((x 1)(y 2))(+ x y))
```

On part de l'expression à prouver et l'on essaye de remonter à des axiomes, ici la preuve réussit quand on instancie v avec la valeur 3 :

$$\frac{\dfrac{}{env_0 \vdash 1 : 1} \quad \dfrac{}{env_0 \vdash 2 : 2} \quad \dfrac{\dfrac{}{env_0\{x : 1, y : 2\} \vdash x : 1} \quad \dfrac{}{env_0\{x : 1, y : 2\} \vdash y : 2}}{env_0\{x : 1, y : 2\} \vdash (+\, x\, y) : 1+2}}{env_0 \vdash (let\ ((x\ 1)(y\ 2)) : v}$$

C'est un exemple où la preuve est complètement déterminée par la syntaxe de l'expression à évaluer. Dans notre sémantique, `if` est une expression disposant de deux règles, si l'utilisation de l'une échoue on devra essayer l'autre, mais on ne pourra pas utiliser les deux. En effet, on peut prouver par récurrence sur la hauteur d'une preuve que toute expression MiniML a au plus une seule valeur.

Exercice 3 *Donner l'arbre de preuve correspondant au calcul de l'expression :*

```
(let ((a 10))
  (let ((f (lambda (x) (+ x a))))
    (let ((a 0))
      (f 0))))
```

et vérifier que la sémantique décrit bien la liaison statique.

• La règle du `letrec` doit décrire les hypothèses pour prouver le jugement

$$env \vdash (letrec\ ((f_1\ e_1)\ldots(f_k\ e_k))\ corps) : w$$

On sait que l'on doit évaluer le corps dans un environnement *envRec* de la forme :

$$envRec = env\,\{f_1\ :\ v_1, \ldots, f_k\ :\ v_k\}$$

où les v_i doivent être les valeurs des e_i dans cet environnement :

$$env\,\{f_1\ :\ v_1, \ldots, f_k\ :\ v_k\}\ \vdash\ e_i\ :\ v_i$$

Autrement dit, l'environnement *envRec* doit vérifier :

$$\begin{array}{ccc} envRec(f_i) = v_i \quad , & envRec\ \vdash\ e_i\ :\ v_i & \text{pour } i = 1 \ldots k \\ \text{et} \quad envRec(x) = env(x) & \text{si} & x \neq f_i \end{array}$$

D'où la règle pour le `letrec` :

$$\frac{\begin{array}{l} env\,\{f_1\ :\ v_1, \ldots, f_k\ :\ v_k\}\ \vdash\ e_1\ :\ v_1 \\ \ldots \\ env\,\{f_1\ :\ v_1, \ldots, f_k\ :\ v_k\}\ \vdash\ e_k\ :\ v_k \\ env\,\{f_1\ :\ v_1, \ldots, f_k\ :\ v_k\}\ \vdash\ corps\ :\ w \end{array}}{env\ \vdash\ (letrec\ ((f_1\ e_1) \ldots (f_k\ e_k))\ corps)\ :\ w}$$

Quelques points forts de la sémantique naturelle

Preuve d'équivalence

Un avantage du formalisme logique de la sémantique naturelle est de faciliter les preuves sur les langages. Par exemple, on peut donner une *preuve* de l'équivalence entre le `let` et l'application d'une lambda expression. De façon précise, on a l'équivalence :

$$env\ \vdash\ (let\,((x_1\ e_1) \ldots (x_n\ e_n))\ corps)\ :\ w$$

si et seulement si

$$env\ \vdash\ ((lambda(x_1 \ldots x_n)\ corps)\ e_1 \ldots e_n)\ :\ w$$

En effet, l'application de la règle pour le `let` et celle pour l'appel de fonction nous conduisent toutes les deux à prouver les mêmes hypothèses :

$$env\ \vdash\ e_1\ :\ v_1 \quad \ldots \quad env\ \vdash\ e_n\ :\ v_n \qquad env\,\{x_1\ :\ v_1, \ldots, x_n\ :\ v_n\}\ \vdash\ corps\ :\ w$$

Modularité : exemple de la portée dynamique

Un autre avantage est la modularité de la présentation par règles d'inférence. Si l'on décide de spécifier le même langage en *portée dynamique*, il n'y a que *deux* règles à changer :

- la règle d'évaluation d'une lambda expression :

$$\frac{}{env\ \vdash\ (lambda\ Lparametre\ e)\ :\ (fctUtilisateur\quad (lambda\ Lparametre\ e))}$$

- la règle de l'appel d'une fonction utilisateur :

$$\frac{\begin{array}{l} env \vdash f : (fctUtilisateur \quad (lambda\,(x_1,\ldots,x_n)\;corps)) \\ env \vdash e_1 : v_1 \quad \ldots \quad env \vdash e_n : v_n \\ env_1\,\{x_1 : v_1,\ldots,x_n : v_n\} \vdash corps : w \end{array}}{env \vdash (f\,e_1,\ldots,e_n) : w}$$

Exercice 4 *Vérifier sur l'exemple de factorielle qu'en sémantique dynamique le* `letrec` *est inutile car la définition locale d'une fonction récursive se fait directement par un* `let`. *On détaillera la preuve de :*

$$env_0 \vdash (let\;((f\;(lambda(n)(if\;(egal?\;0\;n)\;1\;(*\;(f\;(-\,n\;1))))))\;(f\;1))\;:\;1$$

Spécification exécutable

Cette description sémantique en termes de règles de déduction peut être automatiquement traduite dans un langage de programmation logique. A chaque règle sémantique $\frac{H_1 \ldots H_k}{C}$, on associe la clause $C \leftarrow H_1 \wedge \ldots \wedge H_k$ selon le même principe que l'implantation du calcul des séquents en Prolog donnée à la fin du chapitre 18.

Pour l'implanter en Scheme, il faudrait écrire un système de preuve analogue au calcul des séquents du premier ordre du chapitre 19. On va se contenter d'écrire un interprète pour le langage MiniML défini par cette sémantique naturelle. C'est un peu moins satisfaisant, car ce n'est pas toujours une traduction directe des règles logiques et on perd la définition des cas d'erreurs comme étant des cas sans preuve.

21.6 Interprète pour MiniML en Scheme

MiniML est un langage proche de MiniScheme dont aurait supprimé les effets de bord aussi procède-t-on par modifications dans le code de l'évaluateur MiniScheme. On donne ce code en entier car il servira de point de départ pour le langage du paragraphe suivant.

Pour simplifier, on ne considère pas les cas d'erreur.

Les principales différences avec MiniScheme concernent le traitement des opérations sur les listes et la représentation de l'environnement.

Environnement et fonction Scheme

Pour affirmer notre approche purement fonctionnelle, on ne représente plus un environnement par une a-liste mais par une *fonction Scheme.* La consultation de l'environnement consiste à appliquer la fonction qui le représente.

```
(define (ValeurIdent ident env)
  (env ident))
```

L'environnement initial est une fonction construite par la fonction `initEnv`, on tient compte des nouveaux noms pour les booléens et du nouveau traitement des listes.

```
(define  (initEnv)
  (lambda (ident)
    (case ident
      ((+) +)
      ((-) -)
      ((*) *)
      ((/) /)
      ((egal?) (lambda (v1 v2) (if (equal? v1 v2) 'vrai 'faux))
      ((<)     (lambda (v1 v2) (if (< v1 v2) 'true 'false)))
      ((tete)  (lambda (L) (cadr L)))
      ((reste) (lambda (L) (cons ':: (cddr  L))))
      ((cons)  (lambda (v1 v2)(cons  '::  (cons v1 (cdr v2)))))
      ((null?) (lambda (x)(if (and (pair? x)(eq? ':: (car x))(null? (cdr x))
                              'vrai 'faux))) )))
```

L'opération d'extension d'un environnement *env* par des liaisons $\{x_1 : v_1, \ldots, x_n : v_n\}$ utilise la fonction `etendreEnv`. Elle prend en paramètre la liste des identificateurs x_i et la liste des valeurs correspondantes v_i.

```
(define (etendreEnv Lident Lvaleur env)
  (lambda (x)
    (if (member x Lident)
      (elementAssocie x Lident Lvaleur)
      (env x))))
```

La fonction auxiliaire `elementAssocie` permet d'associer, à un élément de la liste `Lident`, la valeur de l'élément correspondant de la liste `Lvaleur` :

```
(define (ElementAssocie id Lident Lvaleur)
  (if  (eq? id (car Lident))
    (car Lvaleur)
    (elementAssocie id (cdr Lident) (cdr Lvaleur))))
```

L'environnement `envRec` introduit dans la règle du `letrec` doit satisfaire aux conditions de point fixe :

```
(envRec   fi) = (evaluer   ei  envRec)
(envRec   x)  = (env x)   si  x n'est pas un des  fi
```

Ce qui justifie sa construction à l'aide du `letrec` de Scheme :

```
(define (EtendreEnv-rec  Lident Lexp env)
  (letrec ((envRec (lambda (x)
                     (if (member x Lident)
                       (evaluer  (ElementAssocie x Lident Lexp) envRec)
                       (env x)))))
    envRec))
```

La fonction d'évaluation

La fonction `evaluer` a la même apparence que celle utilisée pour MiniScheme :

```
(define  (Evaluer e env)
  (cond
    ((constante? e) e)
    ((identificateur? e)(ValeurIdent e env))
    ((formeSpeciale? e)(EvaluerFormeSpeciale e env))
    ((application? e)
     (Appliquer (Evaluer (car e) env)(EvaluerListe (cdr e) env)))
    ))
```

Les constantes comportent les nombres, les booléens `vrai` et `faux` et les listes.

```
(define  (constante? e)
  (or (number? e)
      (eq?  'vrai  e)
      (eq?  'faux  e)
      (and (pair? e)(eq? ':: (car e)))))

(define  (identificateur? e)
  (symbol? e))

(define  (formeSpeciale? e)
  (and (pair?    e)
       (memq (car e) '(if lambda let  letrec))))

(define application? pair?)

(define  (evaluerListe Lexp env)
  (map  (lambda (e) (Evaluer e env)) Lexp))
```

L'application d'une fonction et le traitement des formes spéciales utilisent les mêmes[4] fonctions que MiniScheme. On les redonne pour avoir un code complet :

```
(define  (Appliquer fct Larg  )
  (if (fctUtilisateur? fct)
    (AppliquerFermeture  (cadr fct) (caddr fct) larg )
    (AppliquerFctPredefinie fct Larg)))

(define  (fctUtilisateur?  e)
  (and (pair? e)(eq? 'fermeture (car e))))

(define  (AppliquerFermeture  lambdaExp env0 Larg )
  (let ((Lpar (cadr lambdaexp))
        (corps (caddr lambdaexp)))
    (Evaluer  corps (etendreEnv Lpar Larg env0))))

(define  (AppliquerFctPredefinie fct Larg)
  (apply fct Larg))
```

[4] Avec une petite modification dans `EvaluerIf` pour tenir compte des noms des booléens.

```
(define  (EvaluerFormeSpeciale  e env)
  (case  (car e)
    ((if)  (EvaluerIf (cadr e) (caddr e) (cadddr e) env))
    ((lambda) (EvaluerLambda  e env))
    ((let)    (EvaluerLet (cadr e) (caddr e) env))
    ((letrec)  (EvaluerLetRec (cadr e) (caddr e) env)) ))

(define  (EvaluerIf test expSi expSinon  env)
  (if  (eq? 'vrai (Evaluer test env))
    (Evaluer expSi env)
    (Evaluer expSinon env)))

(define (EvaluerLambda  lambdaExp env)
  (list 'fermeture lambdaExp env))

(define  (EvaluerLet liaisons corps env)
  (let ((LvarLocales (map  car liaisons))
        (Lexp        (map  cadr liaisons )))
    (evaluer corps (etendreEnv LvarLocales (evaluerListe Lexp env) env))))

(define (EvaluerLetrec liaisons corps env)
  (let ((LvarLocales (map  car liaisons))
        (Lexp        (map  cadr liaisons )))
    (evaluer corps (EtendreEnv-rec LvarLocales Lexp env))))
```

Quelques tests pour MiniML

On définit le top level MiniML par :

```
? (define MiniML (make-top-level evaluer))
```

Après appel de MiniML, on teste l'utilisation des listes

```
?? (cons 1 (cons 2 (cons 3 (::))))
== (:: 1 2 3)

?? (:: 4 6)
(:: 4 6)

?? (tete (reste (:: 1 2 3)
== 2

?? (reste (reste (cons 1 (:: 2))))
== (::)
```

Remarque 1 Comme annoncé, l'absence de confusion entre une liste et l'appel de fonction nous dispense d'introduire le mécanisme de la quotation.

Définissons par exemple une fonction `appartient?` qui teste l'appartenance d'un nombre à une liste de nombres :

```
?? (letrec ((appartient?
             (lambda (x L)
               (if (null? L)
                 faux
                 (if (egal? x (tete L))
                   vrai
                   (appartient? x (reste L)))))))
     (appartient?  3  (:: 1 2 3 4)))
== vrai
```

Terminons avec notre exemple habituel de fonctions récursives mutuelles :

```
?? (letrec ((f (lambda (x)(if (egal? 0 x) 0 (+ 1 (g (- x 1))))))
           (g (lambda (x)(if (egal? 0 x) 0 (f (- x 1))))))
     (f 5))
== 3
```

21.7 Un langage fonctionnel paresseux : MiniLML

On conserve la syntaxe abstraite de MiniML, mais on remplace la sémantique de l'*appel par valeur* par celle dite de l'*appel par nom*. On baptise miniLML cette variante de MiniML, le L pour le mot anglais *lazy* (voir la section sur l'appel paresseux). On a rencontré ce type d'appel dans le cadre de la réduction d'expression du lambda calcul. C'était une stratégie de réduction qui avait l'avantage de toujours aboutir à la forme irréductible quand elle existe.

Appel par nom

Désignons par `f` la fonction constante `(lambda (x) 1)` et considérons les deux expressions suivantes :

```
e1 = (f  (/ 1  0))

e2 = (f  (fac 10000))
```

Pour tous les langages que nous avons décrits, le calcul de `e1` donnera une erreur et le calcul de `e2` finira par donner 1 (du moins si les ressources de la machine sont suffisantes). Avec l'appel par nom, on n'évalue pas l'argument mais le corps de la fonction après y avoir substitué l'argument à la place du paramètre formel. Cette substitution correspond donc à la bêta réduction. Dans notre exemple, le corps de `f` est la constante 1, aussi la valeur de `e1` et celle de `e2` sont immédiatement 1.

Dans ces exemples, artificiels, l'avantage revient à l'appel par nom. Mais dans d'autres situations — plus courantes — où l'on doit évaluer plusieurs fois un argument, l'appel par valeur sera plus efficace. C'est par exemple le cas de l'évaluation de :
`(g (fac 10000))` où `g` est la fonction `(lambda (x)(* x (cos x) (sin x)))`.
L'appel par nom calculera trois fois `(fac 10000)` et l'appel par valeur une seule fois.

On pourrait définir l'appel par nom par une règle inspirée de la bêta conversion :

$$\frac{\begin{array}{l} env \vdash f : (fermeture\ (lambda\ (x_1 \ldots x_n)\ corps) \quad env_1) \\ env_1 \vdash corps\,[x_1 \leftarrow e_1 \ldots x_n \leftarrow e_n] : w \end{array}}{env \vdash (f\, e_1 \ldots e_n) : w}$$

Mais on se rappelle que la substitution $corps\,[x_1 \leftarrow e_1 \ldots x_n \leftarrow e_n]$ des e_j à la place des x_j dans le *corps* doit se faire *sans capture.* Cette condition impose de faire des renommages assez peu efficaces. Une autre méthode consiste à conserver dans l'environnement les expressions *non évaluées* des arguments e_i. Pour cela, on s'inspire de la technique des fermetures : on encapsule les arguments d'un appel de fonction dans une structure appelée *glaçon.*

Glaçons

L'opération de gel d'une expression dans un environnement sera représentée par un nouveau jugement noté $env \vdash_{gel} exp : exp{-}gelee$. Elle est spécifiée par l'axiome :

$$\frac{}{env \vdash_{gel} exp : (gelee\ exp\ env)}$$

L'introduction de valeurs gelées oblige à ajouter une règle pour les dégeler, on évalue l'expression associée au glaçon dans l'environnement de création du glaçon :

$$\frac{env_1 \vdash exp : v}{env \vdash (gelee\ exp\ env_1) : v}$$

Pour réaliser l'appel par nom dans $(f\ e_1\ \ldots\ e_n)$, on associe à chaque argument e_j un *glaçon* g_j. Quand on aura effectivement besoin de la valeur d'un argument, on procédera au *dégel* du glaçon, c'est-à-dire à l'évaluation de son expression dans l'environnement conservé.
Décrivons ce mécanisme en sémantique naturelle par une nouvelle règle pour l'appel de fonction utilisateur :

$$\frac{\begin{array}{l} env \vdash f : (fermeture\ (lambda\ (x_1 \ldots x_n)\ corps) \quad env_1) \\ env \vdash_{gel} e_1 : g_1 \quad \ldots \quad env \vdash_{gel} e_n : g_n \\ env_1\,\{x_1 : g_1, \ldots, x_n : g_n\} \vdash corps : w \end{array}}{env \vdash (f\, e_1 \ldots e_n) : w}$$

Comme le `let` correspond à l'application d'une lambda expression, on doit aussi geler les expressions e_i pour évaluer le corps du `let` :

$$\frac{env \vdash_{gel} e_1 : g_1 \ \ldots \ env \vdash_{gel} e_n : g_n \quad env\,\{x_1 : g_1, \ldots, x_n : g_n\} \vdash corps : w}{env \vdash (let\ ((x_1\ e_1) \ldots (x_n\ e_n))\ corps) : w}$$

Pour l'évaluation de `(letrec ((f1 e1)...(fk ek)) corps)`, on gèle les arguments e_i dans l'environnement récursif *envRec.* Il doit donc satisfaire à l'égalité :

$$envRec = env\,\{f_1 : (gelee\ e_1\ envRec), \ldots, f_k : (gelee\ e_k\ envRec)\}$$

Ce que l'on traduit par la règle :

$$\frac{\begin{array}{l} env\{f_1 : g_1, \ldots, f_k : g_k\} \vdash_{gel} e_1 : g_1 \\ \ldots \\ env\{f_1 : g_1, \ldots, f_k : g_k\} \vdash_{gel} e_k : g_k \\ env\{f_1 : g_1, \ldots, f_k : g_k\} \vdash corps : w \end{array}}{env \vdash (letrec\ ((f_1\ e_1) \ldots (f_k\ e_k))\ corps) : w}$$

Fonction cons paresseuse

Pour les fonctions prédéfinies, il n'y pas de changement car on souhaite, en général, conserver pour ces fonctions une stratégie d'appel par valeur. Mais il y a l'exception importante des fonctions sur les listes. En effet, on a vu, à l'occasion des flots, l'intérêt d'avoir une fonction `cons` qui n'évalue pas son deuxième argument. Pour changer, on se place ici dans le cas où elle n'évalue *aucun* de ses arguments mais les gèle, on la baptise `Lcons`.

$$\frac{env \vdash_{gel} e : g_1 \qquad env \vdash_{gel} L : g_2}{env \vdash (Lcons\ e\ L) : (::\ g_1\ g_2)}$$

Une liste devra toujours être construite par `Lcons` pour avoir ses éléments gelés, on ne peut plus se la donner explicitement. Ceci nécessite de modifier la sémantique des fonctions `tete` et `reste`, elles doivent dégeler les éléments correspondant à la tête ou au reste de la liste.

$$\frac{env_1 \vdash e_1 : v}{env \vdash (tete\ (::\ (gelee\ e_1\ env_1)\ g_2) : v} \qquad \frac{env_2 \vdash e_2 : L}{env \vdash (reste\ (::\ g_1\ (gelee\ e_2\ env_2)) : L}$$

Implantation en Scheme du langage MiniLML

L'implantation du langage MiniLML est voisine de celle de MiniML, aussi n'indique-t-on que les fonctions modifiées ou ajoutées.

On doit ajouter dans l'évaluateur les valeurs de type glaçon et distinguer les cas d'appel par nom de ceux d'appel par valeur :

```
(define  (evaluer e env)
  (cond
    ((constante? e) e)
    ((identificateur? e)(ValeurIdent e env))
    ((gelee? e)(Degeler  e))                                ;***
    ((formeSpeciale? e)(EvaluerFormeSpeciale e env))
    ((application? e)
     (let ((fct (Evaluer (car e) env)))
       (if (or (fctUtilisateur? fct) (eq? 'Lcons (car e)))
         (Appliquer  fct (GelerListe   (cdr e) env))   ; ***
         (Appliquer  fct (EvaluerListe (cdr e) env)))))  ))
```

Les glaçons sont traités par les quatre nouvelles fonctions suivantes :
On reconnaît un glaçon avec le prédicat :

```
(define (gelee? e)
  (and (pair? e)(eq? 'gelee (car e))))
```

La construction d'un glaçon consiste à capturer l'expression et l'environnement dans une liste[5] :

```
(define (geler e env)
  (list 'gelee e env))

(define (gelerListe Lexp env)
  (map (lambda (e)(geler e env)) Lexp))
```

Pour dégeler un glaçon, on évalue l'expression dans l'environnement qui l'accompagne, cela peut conduire à une suite d'évaluations si la valeur est encore un glaçon.

```
(define  (degeler e)
  (if (gelee? e)
    (evaluer  (cadr e) (caddr e))
    e))
```

Quand on demande la valeur d'un identificateur, on le dégèle si c'est un glaçon :

```
(define (ValeurIdent ident env)
  (let ((valeur (env ident)))
    (if (gelee? valeur)
      (degeler valeur)     ; ***
      valeur)))
```

L'égalité :

```
envRec = env {f1 :  (gelee e1 envRec) ,..., fk :  (gelee ek envRec)}
```

montre que l'environnement récursif doit être construit avec les expressions `ej` gelées :

```
(define (etendreEnv-rec   Lident Lexp env)
  (letrec ((envRec
             (lambda (ident)
               (if (memq ident Lident)
                 (geler  (elementAssocie ident Lident Lexp) envRec)    ;***
                 (env ident)))))
    envRec))
```

Enfin, il faut modifier la sémantique des fonctions prédéfinies `tete` et `reste` :

[5] On pourrait éviter de geler à nouveau un glaçon en le testant.

```
(define  (initEnv)
  (lambda (ident)
    (case ident
      ((+) +)
      ((-) -)
      ((*) *)
      ((/) /)
      ((egal?) (lambda (v1 v2) (if (equal? v1 v2) 'vrai 'faux)))
      ((<)     (lambda (v1 v2) (if (< v1 v2)  'vrai 'faux)))
      ((tete)  (lambda (L)                    ; ***
                 (let ((v (cadr L)))
                   (degeler  v))))
      ((reste) (lambda (L)                   ; ***
                 (let ((v (caddr L)))
                   (degeler  v))))
      ((Lcons) (lambda (g1 g2)(list ':: g1 g2)))     ; ***
      ((null?) (lambda (x)(if (and (pair? x)(eq? ':: (car x))(null? (cdr x))
                             'vrai 'faux))) )))
```

Testons cet interprète MiniLML

• L'appel par nom permet parfois de se passer de l'évaluation des arguments d'une fonction ; c'est heureusement le cas dans les deux exemples suivants :

```
?? ((lambda (x) 1) (/ 4 0))
== 1

?? (letrec ((fac (lambda (n)
                   (if (egal? 0 n)
                     1
                     (* n (fac (- n 1)))))))
     ((lambda (x) 1) (fac 10000)))
== 1 ;; immédiatement
```

• Le remplacement de cons par Lcons ne nécessite aucune modification dans les fonctions de manipulation de liste. La fonction appartient? s'écrit encore :

```
?? (letrec ((appartient? (lambda (x L)
                           (if (null? L)
                             faux
                             (if (egal? x (tete L))
                               vrai
                               (appartient? x (reste L)))))))
     (appartient? 5  (Lcons 1 (Lcons 2 (Lcons 3 (::))))))
== faux
```

• Mais la fonction Lcons permet de manipuler des listes *infinies*. On retrouve une structure analogue aux flots du chapitre 10. L'expression suivante définit simultanément la liste des entiers plus grands que n et une fonction qui additionne les k premiers éléments d'une liste infinie :

```
?? (letrec ((Lentiers-pg-n (lambda (n)(Lcons n (Lentiers-pg-n (+ 1 n)))))
            (somme (lambda (i Lnb)
                     (if (egal? 1 i)
                       (tete Lnb)
                       (+ (tete Lnb)
                          ( somme (- i 1) (reste Lnb)))))))
     (somme 5 (Lentiers-pg-n 0)))
== 10
```

Evaluation paresseuse

On peut optimiser cette implantation de l'appel par nom. On a signalé au début qu'un des défauts de l'appel par nom est qu'il y a évaluation de chaque occurrence d'un même argument. Une idée, pour éviter une telle réévaluation, est de stocker le résultat de la *première* évaluation. C'est le principe de l'*évaluation paresseuse* : il n'y a évaluation qu'en cas de nécessité et on ne refait pas ce qui a déjà été fait. la sémantique est la même que l'appel par nom mais les arguments d'une fonction conserveront la mémoire du fait d'avoir déjà été évalués.

Pour réaliser cette optimisation, on reprend la technique utilisée pour définir `force` et `delay` au chapitre 10. On ajoute dans chaque glaçon un booléen qui indique si le glaçon a déjà été ou non dégelé. Lors de la première évaluation d'un glaçon on changera le booléen en `#t` et l'on remplacera l'expression gelée par sa valeur. Ainsi, à la prochaine demande d'évaluation, on saura qu'il suffit de lire la valeur conservée dans le glaçon. Pour avoir une évaluation paresseuse, il suffit donc de modifier que les deux fonctions `geler` et `degeler`.
Quand une expression est gelée, son glaçon contient un indicateur de dégel qui est initialisé à `#f` :

```
(define (geler e env)
  (list 'gelee  e env #f))
```

Le *premier* dégel consiste changer l'indicateur de dégel en `#t` et à remplacer l'expression par sa valeur. Les dégels suivants consistent à rendre directement sa valeur.

```
(define  (Degeler e)
  (if (gelee? e)
    (let ((degelee? (cadddr e)))
      (if degelee?
        (cadr e)
        (let ((valeur (evaluer (cadr e) (caddr e))))
          (set-car! (cdddr e) #t)
          (set-car! (cdr e) valeur)
          valeur)))
   e))
```

On constate qu'avec cette implantation, l'expression suivante est évaluée plus rapidement qu'avec l'appel par nom :

```
?? (letrec ((fac (lambda (n)
                   (if (egal? 0 n)
                     1
                     (* n (fac (- n 1)))))))
     ((lambda (x)(* x x)) (fac 6)))

== 518400
```

21.8 Sémantique naturelle d'un langage impératif

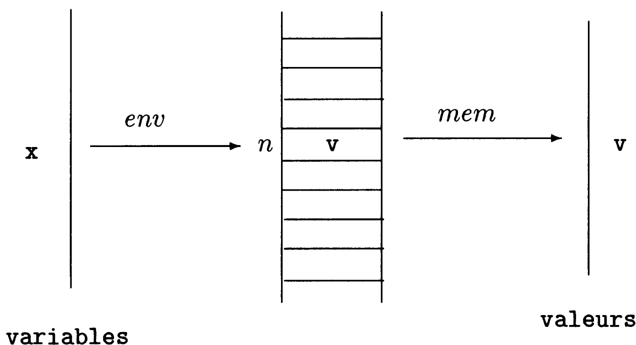
Au §2 on a décrit des aspects impératifs comme `set!` au moyen de modifications physiques de l'environnement, on va voir qu'on peut en donner une description *purement fonctionnelle* en introduisant le concept de *mémoire*.

Mémoire et environnement

Dans un langage impératif (Pascal, C, ...), l'instruction de base est l'affectation. Une affectation `x := exp` a pour effet de modifier le contenu de la case mémoire où est rangée la valeur de la variable `x` pour y placer la valeur de `exp`. Une variable désigne un emplacement dans la mémoire et la valeur d'une variable est le contenu de cet emplacement en mémoire. Aussi, l'association entre une variable et sa valeur se fait au moyen de deux fonctions l'environnement et la mémoire :

- l'environnement est une fonction, notée `env`, qui associe à une variable son adresse ;
- la mémoire est une fonction, notée `mem`, qui associe à une adresse une valeur.

On a vu au chapitre 5 §2 que ce modèle s'applique également aux variables de Scheme.
Pour simplifier, on représente les adresses par des entiers, d'où le schéma :



La valeur v d'une variable x est donnée par la composition des fonctions *mem* et *env* : $v = mem\,(env(x))$. Pour modifier la valeur d'une variable on modifie le contenu de sa case mémoire, mais on ne change pas son adresse. La déclaration

d'une variable a pour objet de lui attribuer la première adresse libre disponible. Aussi, une mémoire est non seulement une application des adresses dans les valeurs, mais doit permettre également d'indiquer la prochaine adresse libre.

Syntaxe abstraite du langage Blaise

Pour illustrer les principaux aspects impératifs, on va décrire un mini langage impératif dans la lignée de Pascal. On baptise Blaise ce langage, il ne manipulera qu'un seul type : les nombres.
L'aspect essentiel n'est plus les expressions mais les *instructions*. Un programme Blaise est constitué d'une suite de déclarations suivie d'une suite d'instructions. On peut déclarer des constantes numériques, des variables, des fonctions, des procédures. Pour simplifier, on ne permet pas de déclaration à l'intérieur d'une fonction ou d'une procédure. Pour simplifier également, les fonctions auront *un seul* paramètre et suivront l'appel par valeur. Les procédures auront aussi *un seul* paramètre mais suivront l'*appel par variable* qui sera décrit plus loin.
Il n'y a pas de notion de boucle de top level, aussi les communications avec l'utilisateur se font par les instructions habituelles d'entrée/sortie : lire une variable, écrire la valeur d'une expression.

Pour décrire la syntaxe abstraite de Blaise, on doit distinguer quatre espèces de termes : les programmes, les déclarations, les instructions et les expressions.

```
Programme     ::= (programme  Ldeclarations  Lintr)

Ldeclarations ::= ({declaration})

declaration   ::= (constante  ident   nombre)  |  (variable  ident)
                | (fonction   nom-fct  ident  exp)
                | (procedure  nom-proc ident  Linstr)

instruction   ::= (affectation var  exp) | (si exp  Linstr Linstr)
                |(ecrire  exp) | (lire  var)
                | (tantque  exp  Linstr)  | (nom-proc var⁶)

Linstr         ::=  (debut {instruction})

exp           ::=  nombre | ident | (op2  exp  exp) | (if exp exp exp)
                | (nom-fct exp)
```

Pour distinguer *l'expression* conditionnelle de *l'instruction* conditionnelle, la première est notée `if` et la seconde `si`. On convient de représenter les booléens vrai et faux par les constantes 1 et 0.

Sémantique naturelle de Blaise

On doit donc définir la sémantique de ces quatre sortes de termes. La *sémantique d'une expression* nécessite de connaître l'environnement et la mémoire pour cal-

[6] L'appel par variable d'une procédure nécessite de lui passer en argument une variable.

culer la valeur des variables. Comme l'évaluation d'une expression Blaise n'induit pas d'effet de bord, elle est décrite par le jugement :

$$env\,,\, mem \vdash exp : valeur$$

Les instructions ont seulement pour effet de modifier la mémoire, aussi la *sémantique d'une instruction* est décrite par un jugement de la forme :

$$env\,,\, mem \vdash instr : mem'$$

où mem' représente la mémoire après exécution de l'instruction $instr$.

Les déclarations construisent l'environnement et modifient aussi la mémoire en changeant la valeur de la prochaine adresse libre. La *sémantque des déclarations* est décrite par un jugement de la forme :

$$env\,,\, mem \vdash decl : env' \times mem'$$

où le couple $env' \times mem'$ représente les nouvelles valeurs de l'environnement et de la mémoire après la déclaration.

La *sémantique d'un programme* est représentée par l'état mémoire après son exécution. Elle est décrite par un jugement de la forme :

$$\vdash prog : mem$$

Donnons la description de ces quatre catégories de jugement. Par abus de notation, ou plus précisément par surchage du symbole $\vdash$, on a utilisé le même signe $\vdash$ pour ces quatre types de jugement.

Sémantique des expressions : $env\,,\, mem \vdash exp : valeur$

La valeur d'un entier, d'une constante ou d'une variable est donnée par des axiomes.

• *Nombre*

$$\frac{}{env\,,\, mem \vdash nb : nb}$$

• *Constante*

$$\frac{}{env\,,\, mem \vdash (cte\ n) : n}$$

• *Variable*

$$\frac{}{env\,,\, mem \vdash var : mem\,(env\,(var))}$$

• *Opérateurs arithmétiques* `+, -, *` ... : Pour abréger, on considère que des opérateurs binaires. Voici, par exemple, le cas de l'addition :

$$\frac{env\,,\, mem \vdash e_1 : v_1 \qquad env\,,\, mem \vdash e_2 : v_2}{env\,,\, mem \vdash (+\ e_1\ e_2) : v_1 + v_2}$$

• *Opérateurs relationnels* : =, < ... : on n'explicite que le cas de < :

$$\frac{env\,,\,mem\,\vdash\,e_1\,:\,v_1 \qquad env\,,\,mem\,\vdash\,e_2\,:\,v_2}{env\,,\,mem\,\vdash\,(<\,e_1\,e_2)\,:\,(if\,(<\,v_1\,v_2)\,1\,0)}$$

• *Expression if* : on évalue e_1 ou e_2 selon la valeur 1 ou 0 du test e_0 :

$$\frac{env\,,\,mem\,\vdash\,e_0\,:\,1 \qquad env\,,\,mem\,\vdash\,e_1\,:\,v_1}{env\,,\,mem\,\vdash\,(if\;e_0\,e_1\,e_2)\,:\,v_1}$$

$$\frac{env\,,\,mem\,\vdash\,e_0\,:\,0 \qquad env\,,\,mem\,\vdash\,e_2\,:\,v_2}{env\,,\,mem\,\vdash\,(if\;e_0\,e_1\,e_2)\,:\,v_2}$$

• *Appel de fonction par valeur* : soit v la valeur de son argument, on lui applique la fonction *val-fonc = env (nom-fct)* associée, au moment de la déclaration, au nom de la fonction (voir page 668) :

$$\frac{env\,,\,mem\,\vdash\,exp\,:\,v}{env\,,\,mem\,\vdash\,(\textit{nom-fct exp})\,:\,((env(\textit{nom-fct}))(v\,,\,mem)}$$

Sémantique des instructions : $env\,,\,mem\,\vdash\,instr\,:\,mem'$

• *Affectation* : c'est l'instruction de base, elle modifie le contenu de la case mémoire à l'adresse de la variable concernée.

$$\frac{env\,,\,mem\,\vdash\,exp\,:\,v}{env\,,\,mem\,\vdash\,(affectation\;var\;exp)\,:\,mem\,\{env(var)\,:\,v\}}$$

On a utilisé la notation $mem\,\{adr\,:\,v\}$ pour représenter la même fonction mémoire que mem à cela près que dorénavant le contenu à l'adresse adr est v.

• *Lecture d'une variable* : elle se fait par l'intermédiaire d'une fonction système *READ*, on remplace le contenu de la case mémoire par la valeur lue.

$$\frac{\vdash\,READ\,:\,v}{env\,,\,mem\,\vdash\,(lire\;var)\,:\,mem\,\{env(var)\,:\,v\}}$$

• *Ecriture* d'une expression : elle provoque l'affichage de la valeur de l'expression par l'intermédiaire d'une fonction système *WRITE*.

$$\frac{env\,,\,mem\,\vdash\,exp\,:\,v \qquad \vdash\,WRITE(v)}{env\,,\,mem\,\vdash\,(ecrire\;exp)\,:\,mem}$$

• *Instruction si* : on exécute l'*Linstr*1 ou l'*Linstr*2 selon la valeur du *test* :

$$\frac{env\,,\,mem\,\vdash\,test\,:\,1 \qquad env\,,\,mem\,\vdash\,Linstr_1\,:\,mem_1}{env\,,\,mem\,\vdash\,(si\;test\;Linstr_1\;Linstr_2)\,:\,mem_1}$$

$$\frac{env\,,\,mem\,\vdash\,test\,:\,0 \qquad env\,,\,mem\,\vdash\,Linstr_2\,:\,mem_2}{env\,,\,mem\,\vdash\,(si\;test\;Linstr_1\;Linstr_2)\,:\,mem_2}$$

• *Instruction tantque* : il y a deux règles.

Dans le cas où le *test* a pour valeur 0, la mémoire est inchangée :

$$\frac{env\,,\, mem \vdash test\ :\ 0}{env\,,\, mem \vdash (tantque\ test\ Linstr)\ :\ mem}$$

Quand le *test* a pour valeur 1, on exécute le corps puis l'on recommence l'exécution du *tantque* dans le nouvel état mémoire :

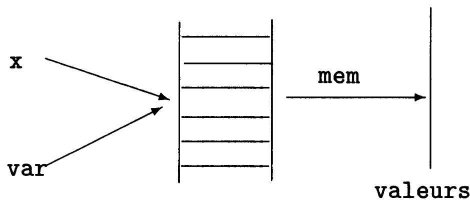
$$\frac{\begin{array}{l} env\,,\, mem \vdash test\ :\ 1 \\ env\,,\, mem \vdash Linstr\ :\ mem_1 \\ env\,,\, mem_1 \vdash (tantque\ test\ Linstr)\ :\ mem_2 \end{array}}{env\,,\, mem \vdash (tantque\ test\ Linstr)\ :\ mem_2}$$

• *Séquence d'instructions* : on calcule les états mémoires successifs :

$$\frac{env\,,\, mem \vdash instr_1\ :\ mem_1 \quad \ldots \quad env\,,\, mem_{k-1} \vdash instr_k\ :\ mem_k}{env\,,\, mem \vdash (debut\ instr_1 \ldots instr_k)\ :\ mem_k}$$

• *Appel de procédure avec passage de paramètre par variable.*

Si l'on a déclaré une procédure $p(x)$, l'appel par variable ($p\ var$) consiste à associer au paramètre formel x l'adresse ($env\ var$) de la variable *var* puis à exécuter la fonction *proc-val* définie au moment de la déclaration de la procédure (voir page 668). La fonction *proc-val* prend en paramètre l'adresse de la variable *var* et retourne le nouvel état mémoire après exécution du corps de la procédure. Si, pendant l'exécution du corps, on est amené à effectuer des affectations sur le paramètre x, ces affectations modifieront donc de façon permanente la case mémoire de *var*.



Comme la valeur de *val-proc* est donnée par *env(nom-proc)*, on a l'axiome :

$$\frac{}{env\,,\, mem \vdash (nom\text{-}proc\ var)\ :\ (env(nom\text{-}proc))(env(var)\ ,\ mem)}$$

Sémantique des déclarations : $env\,,\, mem \vdash decl\ :\ env' \times mem'$

• *Déclaration de constante* : on ajoute dans l'environnement la liaison entre le nom *ident* de la constante et l'expression (*cte nb*) qui désigne une valeur constante.

$$\frac{}{env\,,\, mem \vdash (constante\ ident\ nb)\ :\ env\{ident\ :\ (cte\ nb)\} \times mem}$$

• *Déclaration de variable* : on lie la variable de nom *ident* avec la première adresse libre de la mémoire puis on incrémente la valeur de cette première adresse libre.

$$\frac{}{env\,,\,mem \vdash (variable\ ident)\ :\ env\{ident\ :\ mem(adrLibre)\}\ \times\ mem\{incr\ adrLibre\}}$$

où $mem\{incr\ adrLibre\}$ signifie que l'on a incrémenté de 1 l'adresse de la première case mémoire libre ; c'est cela qui oblige à rendre aussi la mémoire en résultat.

• *Déclaration de fonction* : Blaise est un langage à liaison statique, on pourrait donc représenter les fonctions par des fermetures. Mais, pour insister sur le fait qu'une fois déclarée, une fonction Blaise ne peut plus être modifiée, on associe dès la déclaration une valeur fonctionnelle à une fonction. Cette valeur fonctionnelle prend en paramètre la valeur de l'argument d'appel ainsi que l'état mémoire à l'appel, elle est notée *val-fonc*. C'est donc une application

$$val\text{-}fonc : Nombre\ \times\ Memoire\ \rightarrow\ Nombre$$

Quand on évalue le corps, noté *exp*, de la fonction en liant le paramètre formel x avec une valeur arbitraire v (via la prochaine adresse libre), on doit trouver la même chose que l'application de sa valeur fonctionnelle *val-fonc* sur v et sur l'état mem_1 de la mémoire à l'appel. De plus, pour permettre les définitions récursives, on a ajouté dans *env* la liaison entre la fonction et sa valeur *val-fonc* :

$$\frac{\begin{array}{c} env\,\{nom\text{-}fct\ :\ val\text{-}fonc\ ,\ x : mem_1(adrLibre)\}\quad ,\quad mem_1\{adrLibre\ :\ v\} \\ \vdash\ exp\ :\ val\text{-}fonc(v\,,\,mem_1) \end{array}}{env\,,\,mem\ \vdash\ (fonction\ nom\text{-}fct\ x\ exp)\ :\ env\,\{nom\text{-}fct\ :\ val\text{-}fonc\}\ \times\ mem}$$

• *Déclaration de procédure.*

La déclaration de procédure suit le même principe que celui utilisé pour les fonctions. La déclaration d'une procédure a pour effet d'associer au nom de cette procédure une valeur procédurale. Au lieu d'utiliser une fermeture on représente cette valeur procédurale par une fonction notée *val-proc*. C'est une application

$$val\text{-}proc : Adresse\ \times\ Memoire\ \rightarrow\ Memoire.$$

qui à toute adresse *adr* et à tout état mem_1 de la mémoire à l'appel, calcule le nouvel état mémoire après exécution du corps, noté *Linstr*, de la procédure.

$$\frac{env\{nom\text{-}proc\ :\ val\text{-}proc\ ,\ var\ :\ adr\}\quad ,\quad mem_1 \vdash Linstr\ :\ val\text{-}proc(adr\,,\,mem_1)}{env\,,\,mem\ \vdash\ (procedure\ nom\text{-}proc\ var\ Linstr)\ :\ env\{nom\text{-}proc\ :\ val\text{-}proc\}\ \times\ mem}$$

• *Liste de déclarations* : on traite successivement chaque déclaration :

$$\frac{\begin{array}{l} env\,,\,mem\ \vdash\ decl_1\ :\ env_1\ \times\ mem_1 \\ env_1\,,\,mem_1\ \vdash\ (decl_2 \ldots decl_n) : env_2\ \times\ mem_2 \end{array}}{env\,,\,mem\ \vdash\ (decl_1 \ldots decl_n)\ :\ env_2\ \times\ mem_2}$$

Sémantique des programmes $\vdash$ *prog* : *mem*

On part d'un environnement initial *env*0 vierge de toute liaison et d'une mémoire initiale *mem*0 dont aucune adresse n'est utilisée (et où $adrLibre = 1$) et l'on exécute en séquence les déclarations puis les instructions du programme. Le résultat est représenté par l'état final de la mémoire.

$$\frac{env_0\,,\,mem_0 \vdash Ldeclarations : env_1 \times mem_1 \qquad env_1\,,\,mem_1 \vdash Linstr : mem_2}{\vdash (programme\ Ldeclarations\quad Linstr) : mem_2}$$

21.9 Notion de sémantique dénotationnelle

On donne quelques indications sur l'approche dénotationnelle de la sémantique d'un langage. Contrairement à l'approche logique prônée en sémantique naturelle, la sémantique dénotationnelle est une approche purement fonctionnelle. Toutes les constructions sont décrites par des fonctions mathématiques. On doit donc commencer par décrire les domaines des objets manipulés. Comme il est toujours instructif de préciser les domaines des objets sémantiques, faisons-le dans le cas de Blaise.

Le domaine **EV** des valeurs d'expressions (*Expression Values* en anglais), comme toutes nos expressions sont à valeurs dans les nombres, on a :

EV = `NOMBRE`

Le domaine **DV** des valeurs nommables (*Denotable Values* en anglais), c'est le domaine des objets que l'on peut désigner par un identificateur : on peut nommer une constante, une adresse, une fonction, une procédure :

DV = `NOMBRE` +[7] `ADRESSE + FONC + PROC`

Une adresse est un entier ≥ 0 (on fait comme si l'on avait une mémoire infinie).

ADRESSE = **N**

Un environnement est une application des identificateurs dans les objets nommables, c'est un élément du domaine :

ENV = `IDENT` $\rightarrow$ `DV + ERREUR`[8]

Le langage Blaise ne comporte que les nombres comme structure de données aussi le contenu des cases mémoires (*Storable Values* en anglais) est :

SV = `NOMBRE`

Une mémoire est élément du domaine :

MEMOIRE = `ADRESSE + {adr-libre}` $\rightarrow$ `SV + ADRESSE`

[7] Le signe + entre des domaines signifie la réunion disjointe.

[8] Par définition, ERREUR représente le domaine des valeurs en cas d'erreur.

où le domaine réduit à l'élément `adr-libre` permet de connaître la valeur de la prochaine adresse libre.

Le domaine des valeurs fonctionnelles peut s'exprimer par :

FONC = `EV x MEMOIRE` $\rightarrow$ `EV + ERREUR`

car il n'y a pas d'effet de bord dans les expressions de Blaise.

Le domaine des valeurs procédurales peut s'exprimer par :

PROC = `ADRESSE x MEMOIRE` $\rightarrow$ `MEMOIRE + ERREUR`

On peut regarder l'ensemble de ces définitions comme un système d'équations récursives entre des domaines. Dans le cas présent, l'existence d'une solution est évidente, mais dans des cas plus complexes, on doit faire usage de la méthode du point fixe (voir chapitre 20 §5) pour étudier l'existence d'une solution. Il faut alors définir plus précisément la notion de domaine et les opérations que l'on peut y effectuer.

A chacun des quatre jugements sémantiques on associe une fonction sémantique dont on va préciser les domaines de définition et d'arrivée dans le cas du langage Blaise.

• La sémantique des expressions est définie par la fonction :

SemExp : `EXP x ENV x MEMOIRE` $\rightarrow$ `EV + ERREUR`

• La sémantique des instructions[9] est définie par la fonction :

SemInstr : `INSTR x ENV x MEMOIRE` $\rightarrow$ `MEMOIRE + ERREUR`

• La sémantique des déclarations est définie par la fonction :

SemDecl : `DECL x ENV x MEMOIRE` $\rightarrow$ `ENV x MEMOIRE + ERREUR`

• La sémantique d'un programme est définie par la fonction :

SemProg : `PROG` $\rightarrow$ `MEMOIRE + ERREUR`

Ce type de description sémantique est appelé le *style direct*. Pour décrire le contrôle dans l'évaluation des expressions ou l'exécution des instructions, il est commode d'utiliser aussi des continuations, on obtient le *style par continuation*. Le style par continuation en sémantique dénotationnelle est à rapprocher de l'interprète MiniScheme-CPS. On trouvera une description complète de Scheme dans ce style dans le rapport R^4RS [CRa91].

[9] En fait, la mémoire ne suffit pas à décrire les entrées/sorties, il faudrait ajouter une notion de fichier d'entrée et de fichier de sortie pour être complet.

21.10 Interprète pour Blaise

L'implantation en Scheme d'un interprète pour Blaise consiste à programmer les quatre fonctions :

```
SemExp   :  EXP x ENV x MEMOIRE   →   EV

SemInstr : INSTR x  ENV x MEMOIRE  →   MEMOIRE

SemDecl  : DECL x  ENV x MEMOIRE   →   ENV x MEMOIRE

SemProg  : PROG      →   MEMOIRE
```

Comme pour les deux précédents interprètes, on ne considère pas les cas d'erreur. On commence par donner les fonctions de manipulation de l'environnement et de la mémoire.

Environnement et mémoire

```
ENV =  IDENT  →    ADRESSE  +  NOMBRE  +  FONC  +  PROC + ERREUR
```

Pour distinguer une constante d'une adresse, la constante de valeur `x` est représentée par la liste `(cte x)` alors qu'une adresse est un entier. Les valeurs de type `FONC` et `PROC` sont respectivement de la forme `(fct fct-Scheme)` et `(proc fct-Scheme)`.

```
(define (ValeurIdent id env mem)
  (let ((valeur (env id)))
    (if (adresse? valeur)
      (mem valeur)
      (cadr valeur))))

(define adresse? integer?)
```

L'environnement initial ne contient pas de liaison :

```
(define env0
  (lambda (x) 'indefini))
```

Pour ajouter à un environnement une liaison entre un identificateur et une valeur, on utilise :

```
(define (etendreEnv ident Dval env)
  (lambda (x)
    (if (eq? ident x)
      Dval
      (env x))))
```

```
MEMOIRE = ADRESSE + {adr-libre}  →  NOMBRE  +  ADRESSE
```

Pour avoir la valeur de la prochaine adresse libre d'une mémoire `mem`, on l'applique sur le symbole `adrLibre` ; au début la première adresse libre est 0.

```
(define mem0
  (lambda (x)
    (if (eq? 'adrLibre  x)
      0
      'indefini)))
```

Pour ajouter une valeur dans une mémoire, on place cette valeur à l'adresse libre et on incrémente cette adresse :

```
(define (etendreMem val mem )
  (let ((adr-libre (mem 'adrLibre)))
    (lambda (x)
      (cond ((eq? 'adrLibre x)(+ 1 adr-libre))
            ((= adr-libre x) val)
            (else (mem x))))))
```

On remplace le contenu d'une mémoire à une adresse donnée par une valeur donnée :

```
(define (modifierMem  adr val mem)
  (lambda (x)
    (if (eq? x adr)
      val
      (mem x))))
```

Sémantique des expressions

On définit la fonction SemExp de domaines :

```
  SemExp : EXP x ENV x MEMOIRE  →   NOMBRE.
```

La sémantique des expressions est voisine des sémantiques précédentes, avec cependant un changement important : une fermeture n'est plus représentée par un couple : lambda expression et environnement mais par une fonction Scheme. Aussi l'appel de fonction se réduit-il à l'exécution de la fonction Scheme associée à l'identificateur au moment de la déclaration. Pour simplifier, les fonctions prédéfinies sont toutes binaires.

```
(define (SemExp e env mem)
  (cond ((number? e)  e)
        ((identificateur? e)(ValeurIdent e env mem))
        ((ifExpression? e)                ;; (if test e1 e2)
         (SemExpIf (cadr e) (caddr e) (cadddr e) env mem))
        ((app-fct-predefinie? e)          ;; (op2 e1 e2)
         (app-fct-predefinie  (car e)
                              (SemExp (cadr e) env mem)
                              (SemExp (caddr e) env mem)))
        (else                             ;; (nom-fct exp)
         (SemAppFct-utilisateur (car e)
                                (SemExp (cadr e) env mem)
                                 mem))))
```

```
(define identificateur? symbol?)

(define (ifExpression? e)
  (and (pair? e)(eq? 'if (car e))))
```

La sémantique de l'expression `if` tient compte de la représentation des booléens par 0 et 1 :

```
(define (SemExpIf test e1 e2 r m)
  (let ((valeur-test (SemExp test r m)))
    (if (= 1 valeur-test)
      (SemExp e1 r m)
      (SemExp e2 r m))))


(define (app-fct-predefinie? e)
  (and (pair? e)(member (car e) '(+ - * / = < ))))

(define (app-fct-predefinie op v1 v2 )
    (case op
      ((+) (+ v1 v2))
      ((-) (- v1 v2))
      ((*) (* v1 v2))
      ((/) (/ v1 v2))
      ((=) (if (= v1 v2) 1 0))
      ((<) (if (< v1 v2) 1 0)) ))

(define (SemAppFct-utilisateur nom-fct  valeur-argt env  mem)
  ((ValeurIdent nom-fct env mem) valeur-argt  mem))
```

Sémantique des instructions

On définit la fonction `SemInstr` de domaines :

```
SemInstr : INSTRUCTION x  ENV x MEMOIRE  →  MEMOIRE
```

Cette fonction sous-traite à des fonctions spécialisées le traitement de chaque espèce d'instruction :

```
(define (SemInstr instr env mem)
  (case (car instr)
    ((affectation)                          ;; (affectation x e)
     (SemAffectation (cadr instr)(caddr instr) env mem))
    ((ecrire)                               ;; (ecrire exp)
     (SemEcrire (cadr instr) env mem))
    ((lire)                                 ;; (lire var)
     (SemLire (cadr instr) env mem))
    ((si)                                   ;; (si test instr-si instr-sinon)
     (SemSi (cadr instr)(caddr instr)(cadddr instr) env mem))
    ((tantque) ;; (tantque test corps)
```

```
     (SemTantque (cadr instr)(caddr instr) env mem))
    ((debut)                               ;; (debut Instr1... )
     (SemSequence(cdr instr) env mem))
    (else                                  ;; (nom-proc  var)
     (SemAppelProc (car instr)(cadr instr) env mem))))
```

Comme les instructions sont essentiellement basées sur l'affectation, un point essentiel réside dans la sémantique de l'affectation. Elle a pour effet de modifier la mémoire sans changer l'environnement.

```
(define (semAffectation var exp env mem)
  (let ((valeur (SemExp exp env mem)))
    (modifierMem (env var) valeur mem)))
```

L'écriture réalise un effet de bord mais ne change pas l'état de la mémoire :

```
(define  (semEcrire  exp env mem)
  (let ((valeur (SemExp exp env mem)))
    (display-alln "--> " valeur)
    mem))
```

La lecture est voisine d'une affectation car elle affecte la variable lue :

```
(define  (semLire  var env mem)
  (display ">> ")
  (modifierMem (env var) (read) mem))
```

La sémantique de l'instruction `si` tient compte de notre représentation des booléens :

```
(define  (SemSi test instr-si instr-sinon env mem)
  (let ((valeur-test (SemExp test env mem)))
    (if (= 1 valeur-test)
      (SemInstr instr-si  env mem)
      (SemInstr instr-sinon  env mem))))
```

On exécute le corps d'une boucle tant que son test a pour valeur 1 :

```
(define  (SemTantQue test corps env mem)
  (let ((valeur-test (SemExp test env mem)))
    (if (= 1 valeur-test)
      (SemTantQue test corps env (SemInstr corps env mem))
      mem)))
```

La sémantique d'une suite d'instructions consiste à effectuer la suite des changements d'état mémoire :

```
(define (SemSequence Linstrs env mem)
  (if (null? Linstrs)
    mem
    (SemSequence (cdr Linstrs) env (SemInstr (car Linstrs) env mem))))
```

Un autre point nouveau est l'appel de procédure. Au moment de la déclaration, on a associé avec le nom de la procédure une fonction Scheme qu'il suffira ensuite d'appeler. Le passage de paramètre *par variable* signifie que l'on utilise comme adresse l'adresse (env var) de la variable passée en argument :

```
(define (SemAppelProc nom-proc var env mem)
  ((ValeurIdent nom-proc env mem) (env var) mem))
```

Sémantique des déclarations

Avec cette présentation de la sémantique, l'essentiel du travail est réalisé au moment des déclarations. La fonction semDecl rend une liste constituée d'un environnement et d'une mémoire :

```
SemDecl : DECLARATION x  ENV x MEMOIRE  →  (ENV  MEMOIRE)
```

Une déclaration de constante par (constante nom-cte nb) ajoute à l'environnement une liaison
(nom-cte . (cte nb)). Le terme (cte nb) permet de distinguer ce type de valeur d'une adresse.

Une déclaration de variable par (variable nom-var) ajoute à l'environnement une liaison (nom-var . son-adresse). Son adresse est obtenue en utilisant la première adresse libre de la mémoire, ce qui a donc aussi pour effet d'incrémenter l'adresse de la prochaine case mémoire libre.

```
(define (SemDecl decl env mem)
  (case (car decl)
    ((constante)
     (list (etendreEnv (cadr decl) (list 'cte (caddr decl)) env)
           mem))
    ((variable)
     (let ((Nlle-adr (mem 'adrLibre)))
       (list (EtendreEnv (cadr decl) Nlle-adr env)
             (EtendreMem 'undefini mem))))
    ((fonction)                        ;; (fonction nom-fct parametre exp)
     (SemDecl-fct  (cadr decl)(caddr decl) (cadddr decl) env mem))
    ((procedure)                       ;; (procedure nom-proc var Linstr)
     (SemDecl-proc (cadr decl)(caddr decl) (cadddr decl) env mem))))
```

Une déclaration de fonction ajoute à l'environnement une liaison
{nom-fonction : (fct val-fct)}. La valeur fonctionnelle val-fct est une fonction Scheme SV x MEMOIRE → RV qui utilise la valeur de l'environnement au moment de la déclaration. Pour permettre les fonctions récursives, l'évaluation du corps de la fonction se fait dans un environnement qui contient déjà la liaison que l'on est en train de définir et augmenté de la liaison entre le paramètre formel et la valeur de l'argument.

```
(define (SemDecl-fct nom-fct parametre corps env mem)
  (letrec ((val-fct
            (lambda (val mem1)
```

```
              (SemExp corps
                     (let ((Nlle-adr (mem1 'adrLibre)))
                       (etendreEnv parametre Nlle-adr
                                  (etendreEnv nom-fct
                                             (list 'fct val-fct) env)))
                     (etendreMem val mem1) ))))
    (list (etendreEnv nom-fct (list 'fct val-fct) env) mem)))
```

Une déclaration de procédure ajoute à l'environnement une liaison `{nom-procedure : (proc val-proc)}`. La valeur procédurale `val-proc` est une fonction Scheme `ADRESSE x MEMOIRE` $\rightarrow$ `MEMOIRE`, dont le calcul est analogue à celui de `val-fct`.

```
(define (SemDecl-proc nom-proc var corps env mem)
  (letrec ((val-proc
            (lambda (adr mem1)
              (SemInstr corps
                        (etendreEnv var  adr
                                   (etendreEnv nom-proc
                                              (list 'proc val-proc) env))
                        mem1))))
    (list (etendreEnv nom-proc (list 'proc val-proc) env) mem)))
```

Sémantique d'un programme

La sémantique d'un programme consiste à renvoyer l'état mémoire final après exécution des déclarations puis du corps du programme. On accède aux parties d'un programme par les fonctions :

```
(define Ldecl-programme  cadr)
(define corps-programme  caddr)
```

On traite la liste des déclarations avec la fonction `SemLdecl`. Comme la sémantique d'une déclaration rend une liste de deux valeurs, on utilise la macro `bind` pour en récupérer le contenu.

```
(define (SemLdecl Ldecl env mem)
  (if (null? Ldecl)
    (list env mem)
    (bind  (env1  mem1) (SemDecl (car Ldecl) env mem)
      (SemLdecl (cdr Ldecl) env1 mem1))))
```

Ensuite, on calcule la sémantique du corps en utilisant l'environnement et la mémoire résultant des déclarations. D'où la fonction `SemProg` : `PROG` $\rightarrow$ `MEMOIRE` :

```
(define (SemProg pg)
  (bind (env1 mem1) (SemLdecl (cadr pg) env0 mem0)
    (SemInstr (caddr pg)  env1 mem1)))
```

Exécution de quelques programmes en Blaise

Voici des exemples qui illustrent certains aspects de ce langage.

- Calcul de factorielle par une boucle **tantque** :

```
? (SemProg  '(programme  ; r = factorielle  n
            ((variable r)
             (variable n))
            (debut
              (affectation r 1)(lire n)
              (tantque (< 0 n)
                       (debut (affectation r (* r n))
                              (affectation n (- n 1))))
              (ecrire r))))
 >> 5
--> 120
```

- Calcul de factorielle par une fonction récursive :

```
? (SemProg  '(programme
            ((fonction f x  (if (= 0 x) 1 (* x  (f (- x 1))))))
            (debut
              (ecrire (f 5)))))
--> 120
```

- Appel d'une procédure par variable :

```
? (SemProg  '(programme
            ((variable n)
             (constante c 2)
             (procedure p x (affectation x  c)))
            (debut
              (affectation n 0)
              (ecrire n)
              (p n)
              (ecrire n))))
--> 0
--> 2
```

- Procédure récursive ; elle affiche les entiers en décroissant depuis 3 :

```
? (SemProg  '(programme
            ((variable v)
             (constante n 3)
             (procedure p x (debut (ecrire x)
                                   (affectation x  (- x 1))
                                   (si (= 0 x)
                                       (ecrire x)
                                       (p x )))))
            (debut
              (affectation v 3)
              (p v)
              )))
```

```
--> 3
--> 2
--> 1
--> 0
```

Exercice 5 *1. Blaise est comme le langage C, tout est au même niveau car il n'y a pas de structure de bloc. Le modifier pour admettre aussi des déclarations à l'intérieur des fonctions et des procédures.*

2. En s'inspirant de Blaise, donner une version purement fonctionnelle de l'interprète de MiniScheme en ajoutant une variable mémoire pour décrire la sémantique de `define` *et* `set!`*.*

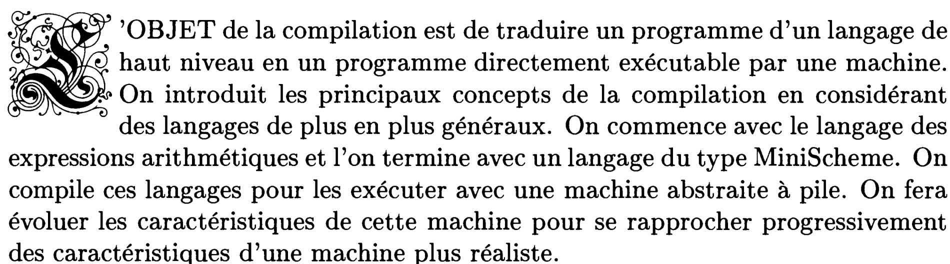
21.11 De la lecture

On trouvera une grande variété de descriptions de langages par des interprètes programmés en :
- Pascal dans le livre de Kamin [Kam90],
- Hope dans celui de Field et Harrison [FH88],
- Scheme dans l'ouvrage de Friedman, Wand and Haynes[FWH92].

Une étude approfondie des sémantiques de divers dialectes Lisp est présentée dans le livre de Queinnec [Que94].
On s'est contenté d'effleurer la sémantique dénotationnelle car elle est traitée dans de nombreux ouvrages, par exemple [Sto77, Sch86]. On peut aussi consulter l'ouvrage de Gordon [Gor79] qui en donne une présentation un peu moins mathématique.
Les bases de la sémantique, dite naturelle, ont été fournies par G. Plotkin dans un cours au Danemark [Plo81]. Ce point de vue a été développé systématiquement au sein de l'environnement Centaur par l'équipe de G. Kahn. Le livre de Nielsson et Nielsson [NN93] est l'un des rares à présenter la sémantique naturelle.

Chapitre 22

Introduction à la compilation



L'OBJET de la compilation est de traduire un programme d'un langage de haut niveau en un programme directement exécutable par une machine. On introduit les principaux concepts de la compilation en considérant des langages de plus en plus généraux. On commence avec le langage des expressions arithmétiques et l'on termine avec un langage du type MiniScheme. On compile ces langages pour les exécuter avec une machine abstraite à pile. On fera évoluer les caractéristiques de cette machine pour se rapprocher progressivement des caractéristiques d'une machine plus réaliste.

On introduit progressivement les notions de zone d'activation, lien dynamique, compteur ordinal, adresse de retour et lien statique. On donne une implantation des listes dans la zone de mémoire appelée tas et l'on présente des méthodes de récupération des doublets inutilisés ou GC. Le problème de la sauvegarde des informations quand on exécute un appel de fonction nous amène à discuter des mérites respectifs de l'allocation mémoire en pile ou dans le tas.

22.1 Calculette numérique et machine SC

En guise de préliminaire, on commence par le cas des expressions arithmétiques traitées au chapitre 12 §6. On a vu que la forme postfixée d'une expression arithmétique s'interprète immédiatement comme le code à exécuter sur une machine à pile. Reprenons rapidement ce point. On appelle `calculette` le langage des expressions suivantes[1] :

```
calculette →  nb | (op2 e1 e2)
op2        →   + | - | * | =
```

[1]On part de la forme préfixe, car on a vu au chapitre 13 comment passer de la forme infixe à la forme préfixe au moyen de l'analyse syntaxique.

La machine SC

On considère une *machine à pile* qui accepte les instructions suivantes :
`(LOADCTE n)` ; on empile le nombre n.

`(ADD)` ; on dépile la valeur `v2` du sommet de pile ainsi que la valeur `v1` du sous-sommet et l'on empile la somme `v1 + v2`.

On procède de façon analogue pour les autres opérateurs binaires `-, *, =` :
`(SUB)` ; on empile `v1 - v2`.

`(MUL)` ; on empile `v1 * v2`.

`(CMP)` ; on empile 1 si `v1 = v2` et 0 sinon.
Pour ne pas introduire des valeurs booléennes, on a représenté le vrai par 1 et le faux par 0.

Par souci d'uniformité, on a représenté une instruction avec ou sans argument par une liste. On appelle `C` le *code* formé de la *liste* des instructions à exécuter.
Pour spécifier le fonctionnement de la machine, on décrit l'état de la pile `S` et du code `C` avant et après l'exécution de la première instruction du code. On n'utilise pas le formalisme des règles de déduction car, dans le cas présent, il n'y a que des axiomes.

On utilise la notation pointée `v.S` pour exprimer que la pile a pour sommet la valeur `v` et pour reste `S`. De même, on met en évidence la première instruction `instr` à exécuter du code par `instr.C`. Avec ces notations, voici les règles de transition de la machine pour chaque type d'instruction :

PILE	CODE	après exécution	PILE	CODE
S	(LOADCTE n).C	→	n.S	C
v2.v1.S	(ADD).C	→	(v1+v2).S	C
v2.v1.S	(SUB).C	→	(v1-v2).S	C
v2.v1.S	(MUL).C	→	(v1*v2).S	C
v2.v1.S	(CMP).C	→	(if (= v1 v2) 1 0).S	C

On appelle SC cette machine, on y ajoute la pseudo-instruction `(STOP)` pour faire cesser l'exécution du code et afficher la valeur présente en sommet de pile.

```
v.S           (STOP).C     →     (display-alln "==> " v)
```

La fonction `executer-codeSC` simule en Scheme le fonctionnement de cette machine. On reprend le principe de la machine décrite au chapitre 12 §6. C'est une boucle qui exécute chaque instruction jusqu'à l'instruction `(STOP)`.
La fonction `Mnemo` donne le mnémonique d'une instruction et la fonction `argt-instr` rend son éventuel argument. On utilise n'importe quelle implantation des piles mutables donnée au chapitre 7 §2.

```
(define (executer-codeSC code)
  (letrec ((exec-aux              ;; la boucle d'exécution du code
            (lambda (S C)
              (let   ((instr (car C)))
                (case (mnemo instr)
                  ((LOADCTE)
```

```
                    (exec-aux (pile!:empiler (argt-instr instr) S) (cdr C)))
                   ((ADD) (let* ((v2 (pile!:depiler S))
                                 (v1 (pile!:depiler S)))
                            (exec-aux (pile!:empiler (+ v1 v2) S) (cdr C))))
                   ((SUB) (let* ((v2 (pile!:depiler S))
                                 (v1 (pile!:depiler S)))
                            (exec-aux (pile!:empiler (- v1 v2) S) (cdr C))))
                   ((MUL) (let* ((v2 (pile!:depiler S))
                                 (v1 (pile!:depiler S)))
                            (exec-aux (pile!:empiler (* v1 v2) S) (cdr C))))
                   ((CMP) (let* ((v2 (pile!:depiler S))
                                 (v1 (pile!:depiler S)))
                            (exec-aux (pile!:empiler (if (= v1 v2) 1 0) S)
                                      (cdr C))))
                   ((STOP) (display-alln "==> " (pile!:sommet S)))))))
    (exec-aux (pile!:vide)  code)))
```

On accède au nom et à l'argument d'une instruction par les fonctions :

```
(define Mnemo car)

(define (argt-instr instr)
  (if (not (null? (cdr instr)))(cadr instr)))
```

Compilateur pour le langage calculette

Pour utiliser cette machine, il reste à générer le code associé à une expression arithmétique. Il s'agit essentiellement de traduire une expression préfixe en une expression postfixe. La sémantique naturelle de cette traduction est spécifiée par le jugement : $\vdash exp : code$.

Voici les règles, on utilise le symbole • pour désigner la fonction cons en infixe :

$$\frac{}{\vdash nb \, : \, ((LOADCTE \; nb))}$$

$$\frac{\vdash e_1 : C_1 \quad \vdash e_2 : C_2}{\vdash (op2 \; e_1 \; e_2) : C_1 \bullet (C_2 \bullet ((code\text{-}op \; op2)))}$$

Cette traduction est réalisée par la fonction compile, elle accumule dans la variable code-suivant le code généré. Noter que le premier argument d'un opérateur binaire est compilé en premier, sa valeur sera donc située sous celle du deuxième argument au moment de l'exécution du code.

```
(define (compile   exp code-suivant)
  (if (number? exp)
    (cons `(LOADCTE ,exp) code-suivant)
    (compile (opd1 exp)
             (compile (opd2 exp)
                      (cons (code-op (op exp))
                            code-suivant)))))
```

La fonction compiler se contente d'ajouter l'instruction (STOP) à la fin du code.

```
(define (compiler exp)
  (compile  exp  '((STOP))))

? (compiler '(- (* (+ 10 97) 72) 13))
((loadcte 10) (loadcte 97) (add) (loadcte 72) (mul) (loadcte 13) (sub) (stop
```

Pour accéder aux composantes d'une expression arithmétique, on a utilisé les fonctions :

```
(define op    car)
(define opd1  cadr)
(define opd2 caddr)
```

Le code machine associé à un opérateur arithmétique est donné par la fonction `code-op`

```
(define (code-op op)
  (case op
    ((+) '(ADD))
    ((-) '(SUB))
    ((*) '(MUL))
    ((=) '(CMP))
    (else (*erreur* "operateur inconnu"))))
```

Pour calculer, avec la machine SC, la valeur d'une expression préfixée, on utilise la fonction :

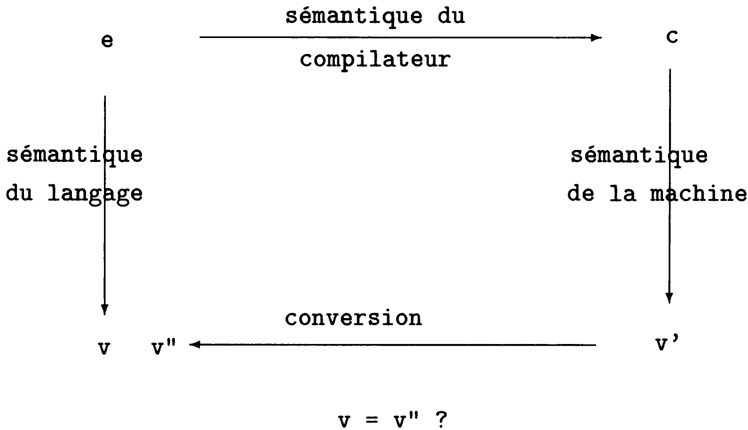
```
(define (calculette exp)
  (executer-codeSC (compiler exp)))

? (calculette  '(- (* (+ 10 97) 72) 13))
==> 7691
```

Preuve de compilateur

Comment s'assurer que notre compilateur est correct ? Il faut d'abord donner un sens précis à cette question. Il s'agit de prouver la cohérence d'au moins trois sémantiques :

- la sémantique de la traduction ou compilation de l'expression `e` en un code `C`,
- la sémantique du langage considéré pour calculer la valeur `v` de l'expression `e`,
- la sémantique de la machine qui exécute le code `C` et fournit une valeur `v'`,
- et on doit éventuellement faire une conversion de la valeur `v'` rendue par la machine en une valeur v" du type des valeurs du langage.

On schématise cette situation sous forme d'un diagramme :



La cohérence revient à prouver la commutativité du diagramme : le chemin de `e` à `v` fournit-il le même résultat que le chemin `e` → `c` → `v'` → `v"` ?
Ceci permet de prouver la cohérence des spécifications ; il reste ensuite à s'assurer que l'implantation du compilateur est correcte ce qui met en jeu la sémantique du langage d'implantation ...
Dans le cas d'un vrai langage, la preuve d'un compilateur est donc une entreprise de longue haleine ce qui explique qu'il n'y en a pas beaucoup d'exemples. Notons que le formalisme de la sémantique naturelle est bien adapté à ce type de preuve car il fournit les logiques qui permettent de raisonner sur chaque flèche du diagramme.

Dans le cas du langage `calculette` c'est très facile. On prouve cette cohérence par récurrence sur la structure des expressions à compiler. Voici le principe :

- si l'expression est une constante, c'est une vérification immédiate,
- si l'expression est de la forme `(op2 e1 e2)`, on raisonne par cas selon la nature de l'opérateur `op2`. Dans chaque cas on utilise l'hypothèse de récurrence pour traiter les sous-expressions `e1` et `e2`.

22.2 Calculette à mémoire et machine SEC

On ajoute à notre calculette la faculté de mémoriser un résultat dans une *variable* et la possibilité de modifier la valeur d'une variable. L'introduction d'effet de bord, nous conduit à inclure aussi la séquence d'expressions. De façon plus précise, on considère les expressions suivantes

```
calculette-memoire →   calculette  | ident | (set! x e)
                      | (let ((x1 e1) ...(xk ek) e) | (begin e1 ...en)
```

Compilation de variables et adressage

La principale nouveauté est la présence de variables. D'où la question : comment compiler une variable? Pour cela, on se réfère au principe de base de la compilation pour une machine à pile : *il faut que l'exécution du code associé à une expression ait pour effet de placer la valeur de l'expression en sommet de pile.*

Aussi, introduit-on une nouvelle instruction `LOADVAR` qui place en sommet de pile la valeur d'une variable. Mais comment connaître, au moment de l'exécution, la valeur d'une variable? On pourrait faire comme pour l'évaluateur MiniScheme : conserver dans une liste les blocs de liaisons entre chaque variable et sa valeur. Mais, à chaque utilisation d'une variable, on devra chercher dans cet environnement la valeur associée. En fait, on va voir que l'on peut faire cette recherche *une fois pour toutes* et remplacer la variable par l'indication de l'endroit où est rangé sa valeur. C'est l'un des avantages de la compilation sur l'interprétation.

Par exemple, considérons l'expression :

```
(let ((x 5)(y 7))
  (let ((u  (+ x y) (v (- x y))
    (- (* u x)(* v y)))
```

Au moment du calcul de `(- (* u x)(* v y)))` par l'interprète MiniScheme, l'environnement est de la forme :

`( ((u .  12)(v .  -2)) ((x .  5)(y .  7)) ...)`

Pour calculer la valeur de cette expression avec notre machine, on ne garde dans un *environnement d'exécution* `E` que les listes des valeurs ;

`( (12 -2) (5 7))`

et, à chaque variable, on associe une *adresse.* Cette adresse indique la position de sa valeur dans `E`. L'adresse est un couple d'entiers `p , i` ; le premier indique le numéro `p` (pour profondeur) de la sous-liste qui contient la variable et l'autre entier indique l'indice de la variable dans cette sous liste. On choisit de représenter ce couple par un vecteur `#(p i)`. Dans notre exemple, les variables `u, v, x y` ont pour adresses respectives les vecteurs :

`#(0 .  0), #(0 .  1), #(1 .  0), #(1 .  1).`

La liaison lexicale utilisée par Scheme permet de connaître l'adresse d'une variable d'après sa position dans le texte de l'expression, on peut donc calculer les adresses au moment de la compilation. Pour cela, on utilise un *environnement de compilation* noté `env-c`. Dans cet exemple, l'environnement de compilation sera de la forme `((u v) (x y))` au moment de la compilation du corps
`(- (* u x)(* v y)))`.

La description en sémantique naturelle de la compilation des expressions `calculette-memoire` est décrite par des jugements de la forme :

$$\textit{env-c} \vdash \textit{exp} : \textit{code}$$

où `env-c` est l'environnement de compilation utilisé pour calculer les adresses des variables. Ainsi, la compilation d'une variable `x` est décrite par la règle (en fait un axiome) :

$$\frac{}{\textit{env-c} \vdash \textit{var} : ((\textit{LOADVAR}(\textit{adresse var env-c})))}$$

où la fonction `adresse` donne l'adresse de `x` calculée dans l'environnement de compilation. Voici l'expression en Scheme de cette fonction. La boucle `adresse-aux` calcule le numéro `p` du bloc et l'indice `i` de la variable dans le bloc.

```
(define (adresse var env-c)
  (letrec ((adresse-aux
            (lambda (Lvar env-c  p i)
              (cond ((null? env-c)(*erreur*   "var indéfinie " var))
                          ((null? Lvar)
                     (adresse-aux (car env-c) (cdr env-c) (+ 1 p) 0))
                    ((eq? var (car Lvar))(vector p i))
                    (else (adresse-aux (cdr Lvar) env-c p (+ 1 i)))))))
    (adresse-aux (car env-c) (cdr env-c) 0 0)))

? (adresse 's '((x y) (u v)(r s w))) -> #(2  1)
```

Compilation de la forme `let`

On sait que l'évaluation de `(let ((x1 e1) ...(xk ek) e)` consiste à évaluer le corps `e` dans l'environnement étendu par les liaisons entre les variables `xi` et les valeurs des expressions `ei`. Le calcul de la valeur par une machine à pile est analogue : on exécute le code du corps dans un environnement d'exécution étendu par les valeurs des `ei`. Pour étendre cet environnement, on calcule la liste des valeurs des `ei` et on l'ajoute en tête de `E`. D'où la règle :

$$\frac{\textit{env-c} \vdash e_1 : C_1 \ \ldots\ \textit{env-c} \vdash e_k : C_k \qquad (x_1, \ldots, x_k).\textit{env-c} \vdash e : C}{\textit{env-c} \vdash (\textit{let}\ ((x_1\ e_1)\ldots(x_k\ e_k)\ e) : (\textit{NIL})\bullet C_k\bullet(\textit{ARG})\ldots\ldots C_1\bullet(\textit{ARG})\bullet(\textit{ADD–ENV})\bullet C\bullet((\textit{DEL–ENV}))}$$

La suite d'instructions `(NIL).Ck.(ARG) ... C1.(ARG).(ADD-ENV)` a pour effet d'ajouter la liste `(v1...vk)` des valeurs des `ei` en tête de l'environnement d'exécution. Détaillons ce point :

- l'instruction `(NIL)` sert à initialiser le calcul de `(v1...vk)` en empilant la liste vide ;
- l'instruction `(ARG)` réalise un `cons` du sous-sommet et du sommet de la pile ;
- l'exécution du code `(NIL).Ck.(ARG) ... C1.((ARG))` a donc pour effet de placer la liste `(v1...vk)` en sommet de pile.
- l'instruction `(ADD-ENV)` ajoute à l'environnement d'exécution la valeur du sommet de pile.

Après exécution du corps du `let`, il faut supprimer de l'environnement la liste `(v1...vk)`, c'est l'objet de l'instruction `(DEL-ENV)`.

Compilation de la forme set!

L'exécution du code de (set! var e) doit avoir pour effet de modifier l'environnement d'exécution à l'adresse de la variable var et d'y placer la valeur de e. Pour placer la valeur du sommet de pile dans l'environnement d'exécution, à une adresse donnée, on introduit l'instruction (STORE adr). La compilation de (set! x e) est maintenant immédiate :

$$\frac{\textit{env-c} \vdash e : C}{\textit{env-c} \vdash (\textit{set! var } e) : C \bullet ((\textit{STORE}\,(\textit{adresse var env-c})))}$$

Compilation d'une séquence

L'évaluation de (begin e1...en) consiste à évaluer chaque ei et à ne retenir que la valeur de la dernière expression. On doit donc exécuter en séquence le code de chaque expression en oubliant les valeurs de e1...en-1, pour cela on introduit l'instruction (POP) qui dépile la pile S. D'où la règle :

$$\frac{\textit{env-c} \vdash e_1 : C_1 \ldots \textit{env-c} \vdash e_n : C_n}{\textit{env-c} \vdash (\textit{begin } e_1 \ldots e_n) : C_1 \bullet (\textit{POP}) \bullet \ldots C_{n-1} \bullet (\textit{POP}) \bullet C_n}$$

Compilateur pour le langage de la calculette à mémoire

L'implantation du compilateur pour notre calculette à mémoire est la traduction directe de ces règles. Dans toute la suite on signale par *** en commentaire les principales modifications apportées à une machine ou un compilateur.

```
(define (compile  env-c  exp code-suivant)
  (cond ((number? exp)
         (cons `(LOADCTE ,exp) code-suivant))
        ((symbol? exp)
         (cons `(LOADVAR ,(adresse exp env-c)) code-suivant))
        (else
         (case (car exp)
           ((set!)              ;;*** (set! var exp)
            (compile-set! env-c  exp code-suivant))

           ((begin)             ;;*** (begin e1 ...en)
            (compile-begin env-c (cdr exp) code-suivant))

           ((let)               ;;*** (let ((x1 e1)..) corps)
            (compile-let env-c exp code-suivant))

           ((+ - * =)
            (compile env-c (opd1 exp)
                     (compile env-c (opd2 exp)
                              (cons (code-op (op exp))
                                    code-suivant)))))))
```

La compilation d'un set! suit exactement la règle correspondante :

```
(define (compile-set! env-c set!-exp code-suivant)
  (let ((var (cadr  set!-exp))
        (exp (caddr set!-exp)))
    (compile env-c exp
             (cons `(STORE ,(adresse var env-c))
                   code-suivant))))
```

On compile le corps du `let` dans l'environnement étendu par les variables locales :

```
(define (compile-let env-c let-exp code-suivant)
  (let ((Lvar (map car (cadr let-exp)))
        (Lexp (map cadr (cadr let-exp)))
        (corps (caddr let-exp)))
    (compile-Larg env-c Lexp
                  (cons '(ADD-ENV)
                        (compile (cons Lvar env-c) corps
                                 (cons '(DEL-ENV)
                                       code-suivant))))))
```

La fonction `compile-Larg` génère le code représenté par :
`(NIL).Ck.(ARG). ...  .C1.(ARG).code-suivant`

```
(define (compile-Larg env-c Lexp code-suivant)
  (letrec ((comp-Larg-aux
            (lambda (Lexp code-suivant)
              (if (null? Lexp)
                code-suivant
                (comp-Larg-aux (cdr Lexp)
                               (compile env-c (car Lexp)
                                        (cons '(ARG)
                                              code-suivant)))))))
    (cons '(NIL) (comp-Larg-aux Lexp code-suivant))))
```

La fonction `compile-begin` engendre le code représenté par
`C1.(POP) ...  Cn-1.(POP).Cn.code-suivant`

```
(define (compile-begin env-c Lexp code-suivant)
  (compile env-c (car Lexp)
           (if (null? (cdr Lexp))
             code-suivant
             (cons '(POP) (compile-begin env-c (cdr Lexp) code-suivant)))))
```

Enfin, la fonction `compiler` ajoute l'instruction `(STOP)` à la fin du code de l'expression

```
(define (compiler exp)
  (compile '()  exp  '((STOP))))
```

Voici deux exemples de compilation où l'on voit l'utilisation de l'adressage :

```
? (compiler '(let ((x 5)(y 7))
               (let ((u  (+ x y)) (v (- x y)))
                 (- (* u x)(* v y)))))
```

```
((nil) (loadcte 7) (arg) (loadcte 5) (arg) (add-env)
       (nil) (loadvar #(0 0)) (loadvar #(0 1)) (sub) (arg)
             (loadvar #(0 0)) (loadvar #(0 1)) (add) (arg) (add-env)
             (loadvar #(0 0)) (loadvar #(1 0)) (mul)
             (loadvar #(0 1)) (loadvar #(1 1)) (mul) (sub)
       (del-env)
 (del-env)
(stop))

? (compiler '(let ((a 1))
               (begin (let ((b 15))
                        (set! a  (+ b 5)))
                      a)))

((nil) (loadcte 1) (arg) (add-env)
       (nil) (loadcte 15) (arg) (add-env)
             (loadvar #(0 0)) (loadcte 5) (add) (store #(1 0))
       (del-env)
       (pop) (loadvar #(0 0))
 (del-env)
(stop))
```

La machine SEC

On appelle SEC la machine obtenue en ajoutant à la machine SC un environnement d'exécution E. On conserve les règles de la machine SC en ajoutant la composante E inchangée. Voici les règles ajoutées pour les nouvelles instructions :

```
S          E    (LOADVAR adr).C    --->   (lire-env adr E).S    E    C

v.S        E    (STORE adr).C      --->   (modifier-env! adr  E  v)
                                           v.S                  E'    C

S          E    (NIL).C            --->   ().S                  E    C

v2.v1.S    E    (ARG).C            --->   (cons v1 v2).S        E    C

Lv.S       E    (ADD-ENV).C        --->   S                  Lv.E    C

S       Lv.E    (DEL-ENV).C        --->   S                     E    C

v.S        E    (POP).C            --->   S                     E    C
```

La fonction `Lire-env` rend la valeur située à l'adresse `adr` dans `E` :

```
? (lire-env '#(2  1) '((3 6)(-7 9)(4 11 8))) -> 11
```

L'appel `(modifier-env!  adr E v)` réalise un effet de bord sur l'environnement `E` : on remplace par `v` la valeur située à l'adresse `adr`.

L'exécution d'une liste d'instructions par la machine SEC est réalisée par la fonction `executer-codeSEC`. On a ajouté la composante `E` à la machine SC et décrit les nouvelles instructions.

```
(define (executer-codeSEC code)
  (let (          ;; *** gestion de l'environnement d'exécution
       (Lire-env
         (lambda (adr E)
           (let ((profondeur (vector-ref adr 0))
                 (indice     (vector-ref adr 1)))
             (list-ref (list-ref E profondeur) indice))))

       (Modifier-env!
         (lambda (adr E v)
           (let ((profondeur (vector-ref adr 0))
                 (indice     (vector-ref adr 1)))
             (set-car! (list-tail (list-ref E profondeur) indice) v)))) )

    (letrec ((exec-aux
              (lambda (S E C)
                (let ((instr (car C)))
                  (case (mnemo instr)
                    ((LOADCTE)
                     (exec-aux (pile!:empiler (argt-instr instr) S)
                               E (cdr C)))
                    ((ADD)
                     (let* ((v2 (pile!:depiler S))
                            (v1 (pile!:depiler S)))
                       (exec-aux (pile!:empiler (+ v1 v2) S) E (cdr C))))
                    ((SUB)
                     (let* ((v2 (pile!:depiler S))
                            (v1 (pile!:depiler S)))
                       (exec-aux (pile!:empiler (- v1 v2) S) E (cdr C))))
                    ((MUL)
                     (let* ((v2 (pile!:depiler S))
                            (v1 (pile!:depiler S)))
                       (exec-aux (pile!:empiler (* v1 v2) S) E (cdr C))))
                    ((CMP)
                     (let* ((v2 (pile!:depiler S))
                            (v1 (pile!:depiler S)))
                       (exec-aux (pile!:empiler (if (= v1 v2)1 0) S)
                                 E (cdr C))))
                    ((STOP) (display-alln "==> " (pile!:sommet S)))

                    ;;*** nouvelles instructions
                    ((LOADVAR)
                     (exec-aux (pile!:empiler
                                 (lire-env (argt-instr instr) E) S)
                               E (cdr C)))
                    ((STORE)
                     (modifier-env!  (argt-instr instr) E (pile!:sommet S))
                     (exec-aux S E (cdr C)))
```

```
                    ((NIL)
                     (exec-aux (pile!:empiler '() S) E (cdr C)))
                    ((ARG)
                     (let* ((v1 (pile!:depiler S))
                            (v2 (pile!:depiler S)))
                       (exec-aux (pile!:empiler (cons v1 v2) S) E (cdr C))))
                    ((ADD-ENV)
                     (let ((v (pile!:depiler S)))
                       (exec-aux S (cons v E) (cdr C))))
                    ((DEL-ENV)
                     (exec-aux S (cdr E) (cdr C)))
                    ((POP)
                     (pile!:depiler S)
                     (exec-aux S E (cdr C))) )))))
      (exec-aux (pile!:vide) '() code))))
```

Le calcul de la valeur d'une expression de type `calculette-memoire` est réalisé par la fonction :

```
(define (calculette-memoire exp)
  (executer-codeSEC (compiler exp)))

? (calculette-memoire '(let ((x 5)(y 7))
                         (let ((u  (+ x y)) (v (- x y)))
                           (- (* u x)(* v y)))))
==> 74


? (calculette-memoire '(let ((a 1))
                         (begin (let ((b 15))
                                  (set! a  (+ b 5)))
                                a)))
==> 20
```

22.3 Compilateur MiniScheme et machine SECD

Il manque à notre langage un point essentiel : les fonctions. On les obtient en ajoutant les lambda expressions et l'application d'une fonction à des arguments. Pour pouvoir définir des fonctions récursives, on ajoute également la conditionnelle `if`. On appelle MiniScheme notre nouveau langage d'expression bien qu'il ne dispose pas (encore) des listes.

```
MiniScheme  →    calculette-memoire  | (if test e1 e2)
               | (lambda (x1 ...xn) e) | (f e1 ... en)
```

Compilation d'une lambda expression

On sait que l'évaluation d'une lambda expression donne une fermeture, constituée d'une partie fonction et d'une partie environnement. Le code d'une lambda expression va provoquer l'empilement d'un couple formé du code du corps de la lambda et de l'environnement `E`.

Le code du corps de la lambda doit être généré dans l'environnement étendu par la liste des paramètres de la lambda. On ajoute l'instruction (RTS) à la suite du code du corps pour signaler la fin de l'exécution de ce corps.
Le code `C1 = C • ((RTS))` est une *sous-liste* du code complet pour permettre de le distinguer comme un tout.

$$\frac{(x_1 \ldots x_n).\textit{env-c} \vdash \textit{corps} : C}{\textit{env-c} \vdash (\textit{lambda}\,(x_1 \ldots x_n)\,\textit{corps}) : (LOADFCT) \bullet (C \bullet ((RTS)))}$$

La nouvelle instruction (LOADFCT) a pour but de provoquer l'empilement d'une paire constituée par le code C1 et par l'environnement d'exécution E:

```
S     E     (LOADFCT).C1.C     --->     (C1 . E).S     E     C
```

Compilation d'un appel de fonction

C'est l'aspect le plus important de cette nouvelle extension du compilateur. Considérons l'expression :

```
(let ((x 10))
  (+ 7 (* ((lambda (x)(+ x 1) 4)  x))))
```

Elle comporte la sous-expression `((lambda (x)(+ x 1) 4)` qui est un appel de fonction. Pour exécuter cet appel, on doit évaluer le corps `(+ x 1)` dans un environnement où `x=4` puis revenir au calcul du produit dans lequel `x` vaut `10`. Le temps de calculer la valeur de l'appel, on doit donc sauvegarder l'environnement courant (ici la liaison `x=10`), la suite du code à effectuer après l'appel (ici la multiplication par `x` puis l'ajout de 7) et le contenu courant de la pile qui servira à faire ces opérations (ici la valeur 7). Pour effectuer ces sauvegardes, on introduit une pile supplémentaire D (pour *Dump*).

La stratégie d'évaluation par valeur demande de procéder à l'évaluation des arguments avant de faire l'appel de fonction. On fera en sorte qu'au moment de l'appel, les valeurs des arguments soient réunies dans une liste située sur la pile. Bien entendu, cela rappelle l'évaluation d'un `let` qui en est un cas particulier. Avant de faire l'appel, on évalue aussi l'expression en position fonctionnelle ce qui aura pour effet d'empiler une fermeture sur la pile.

$$\frac{\textit{env-c} \vdash e_1 : C1 \quad \ldots \quad \textit{env-c} \vdash e_n : Cn \qquad \textit{env-c} \vdash f : C_0}{\textit{env-c} \vdash (f\, e_1 \ldots e_n) : (NIL) \bullet C_n \bullet (ARG) \ldots . C_1 \bullet (ARG) \bullet C_0 \bullet ((JSR))}$$

Comme pour le `let`, la suite d'instructions `(NIL) • Cn • (ARG)...  C1 • (ARG)` sert à empiler la liste des valeurs des arguments. Ensuite, l'instruction `Co` a pour effet d'empiler la fermeture associée à la fonction. Par conséquent, au moment de l'appel, la pile est de la forme `(C1 .  Eo).Lv.S` où `C1` est le code du corps de la lambda, `Eo` l'environnement de définition et `Lv` la liste des valeurs des arguments. L'instruction `(JSR)` (pour *Jump to Subroutine*) sert à lancer l'exécution du code

du corps de la lambda après avoir sauvegardé dans D l'état de la machine et remplacé l'environnement par celui de la fermeture augmenté de la liste des valeurs des arguments :

```
(C1 . Eo).Lv.S    E    (JSR).C    D           --->  ()    Lv.Eo   C1   (S E C).D
```

L'exécution du code C1 de la fonction se termine par l'instruction (RTS) — pour *Return from Subroutine* —, elle restaure l'état de la machine sauvegardé dans D et conserve en sommet de pile la valeur résultant de l'appel de fonction :

```
v.S        E      (RTS).C    (S1 E1 C1).D ---> v.S1     E1     C1          D
```

Compilation de la forme if

L'évaluation de (if test e1 e2) consiste à évaluer le test et, selon qu'il vaut 1 ou 0, on évalue la forme e1 ou e2.
On doit donc compiler le test et, selon la valeur générée par l'exécution de ce code, on exécute le code de e1 ou celui de e2. Le branchement entre ces deux codes est réalisé par l'instruction (BRANCH) :

```
1.S  E  (BRANCH).(CodeSI).(CodeSinon).C   D   --->  S  E   CodeSi.C        D
0.S  E  (BRANCH).(CodeSI).(CodeSinon).C   D   --->  S  E   CodeSinon.C     D
```

D'où la règle pour compiler le if, on utilise le symbole @ pour désigner la concaténation des listes d'instructions :

$$\frac{\textit{env-c} \vdash \textit{test} \,:\, C_0 \quad \textit{env-c} \vdash e_1 \,:\, C_1 \quad \textit{env-c} \vdash e_2 \,:\, C_2}{\textit{env-c} \vdash (\textit{if test}\; e_1; e_2) \,:\, C_0 \,@\, ((\textit{BRANCH}))\,@\,(C1)\,@\,(C2)}$$

Compilateur pour MiniScheme

On ajoute au compilateur précédent le traitement des formes if, des lambda expressions et des appels de fonctions.

• La compilation d'un if.
On doit engendrer un code de la forme :
Co @ ((BRANCH)) @ (C1) @ (C2))

```
(define (compile-if env-c if-exp code-suivant)
  (let ((exp-test  (cadr   if-exp))
        (exp-si    (caddr  if-exp))
        (exp-sinon (cadddr if-exp)))
    (let ((code-si    (compile env-c exp-si    code-suivant))
          (code-sinon (compile env-c exp-sinon code-suivant)))
      (compile env-c exp-test (list '(BRANCH) code-si code-sinon)))))
```

• La compilation d'une lambda expression.
Elle consiste à engendrer un code de la forme :
(LOADFCT).(list C.((RTS)))
où C est le code du corps dans l'environnement étendu par la liste des paramètres formels.

```
(define (compile-lambda env-c lambda-exp code-suivant)
  (let ((Lvar  (cadr  lambda-exp))
        (corps (caddr lambda-exp)))
    (cons '(LOADFCT)
          (cons (compile (cons Lvar env-c) corps '((RTS)))
                code-suivant))))
```

• Compilation des appels de fonctions.
On donne une forme plus générale à la compilation des fonctions prédéfinies pour faciliter l'ajout ultérieur de nouvelles fonctions non nécessairement binaires :

```
(define (fct-predefinie? fct)
  (and (symbol? fct)
       (memq fct '(+ - * =))))

(define (compile-app-predefinie env-c Larg code-suivant)
  (if (null? Larg)
    code-suivant
    (compile env-c (car Larg)
             (compile-app-predefinie env-c (cdr Larg) code-suivant))))
```

Pour compiler un appel de fonction utilisateur, on compile les arguments comme pour un `let` et on ajoute le code de la fonction suivi de l'instruction d'appel `(JSR)` :

```
(define (compile-fct-utilisateur env-c Larg fct code-suivant)
  (compile-Larg env-c Larg (compile env-c fct (cons '(JSR)
                                                     code-suivant))))
```

• Compilation du `let`.
Maintenant que nous disposons des lambda expressions, il n'est plus indispensable de faire un traitement spécial pour le `let`. On le traite comme une macro, c'est-à-dire que l'on compile la lambda expression équivalente. De façon plus précise, on remplace la compilation de `(let ((x1 e1)...(xk ek) corps)` par celle de `((lambda (x1...xk) corps) e1...ek)`.

```
(define (compile-let env-c let-exp code-suivant)
  (let ((Lvar  (map car  (cadr let-exp)))
        (Lexp  (map cadr (cadr let-exp)))
        (corps (caddr let-exp)))
    (compile  env-c `((lambda ,Lvar ,corps) ,@Lexp) code-suivant)))
```

• Compilation du `letrec`.
En procédant comme pour le `let`, on peut ajouter la forme `letrec` à notre langage : on compile la forme équivalente.
On a vu au chapitre 5 §3 que la forme `(letrec ((f1 e1)...(fk ek)) corps)` peut être remplacée par l'expression :

```
(let ((f1 1)...(fk 1)) ;; le choix de 1 est arbitraire
  (let ((f1-aux e1) ... (fk-aux ek))
    (begin
      (set! f1 f1-aux)
```

```
      ...
      (set! fk fk-aux)
      corps)))
```

Comme les variables `fi-aux` doivent être nouvelles, on utilise la fonction non standard `gensym` pour les créer.
Cette transformation est réalisée par la fonction `elimine-letrec`:

```
(define (elimine-letrec letrec-exp)
  (let ((Lvar  (map car  (cadr letrec-exp)))
        (Lexp  (map cadr (cadr letrec-exp)))
        (corps (caddr letrec-exp)))
    (let ((Lvar-aux (map (lambda (x)(gensym)) Lvar)))
      `(let ,(map (lambda (var)(list var 1)) Lvar)
         (let ,(map (lambda (var-aux exp)(list var-aux exp)) Lvar-aux Lexp)
           (begin
            ,@(map (lambda (var var-aux) `(set! ,var ,var-aux))
                   Lvar Lvar-aux)
            ,corps))))))
```

Par exemple

```
? (elimine-letrec '(letrec ((f (lambda (x)
                                  (if (= x 0) 1 (* x (f (- x 1)))))))
                      (f 3)))
(let ((f 1))
  (let ((g25 (lambda (x) (if (= x 0) 1 (* x (f (- x 1)))))))
    (begin (set! f g25)
           (f 3))))
```

D'où la compilation du `letrec`:

```
(define (compile-letrec env-c  letrec-exp code-suivant)
  (compile env-c (elimine-letrec letrec-exp) code-suivant))
```

• Le compilateur réalise un aiguillage vers la méthode de compilation à utiliser:

```
(define (compile env-c exp code-suivant)
  (cond ((number? exp)
         (cons `(LOADCTE ,exp) code-suivant))
        ((symbol? exp)
         (cons `(LOADVAR ,(adresse exp env-c)) code-suivant))
        (else
         (case (car exp)
           ((set!)
            (compile-set! env-c  exp code-suivant))

           ((begin)
            (compile-begin env-c (cdr exp) code-suivant))
```

```
((let)
 (compile-let env-c exp code-suivant))

((if)                        ;; *** (if test exp-si exp-sinon)
 (compile-if env-c  exp code-suivant))

((lambda)                     ;; ***   (lambda (x1...xn) corps)
 (compile-lambda env-c  exp code-suivant))

((letrec)                    ;; *** (letrec ((f1 e1)...(fk ek)) corps)
 (compile-letrec env-c exp code-suivant))

(else
 (let ((Larg (cdr exp))
       (fct  (car exp)))
   (if (fct-predefinie? fct)      ;; ****(op2 e1 e2)
     (compile-app-predefinie env-c Larg (cons (code-op fct)
                                              code-suivant))
     (compile-fct-utilisateur env-c Larg fct ;; ***(f e1 ...en)
                              code-suivant))))
))))
```

Quelques tests

```
? (compiler '((lambda (x y)(+ x y)) 4 5))
((nil) (loadcte 5) (arg) (loadcte 4) (arg)
      (loadfct) ((loadvar #(0 0)) (loadvar #(0 1)) (add)
 (rts)) (jsr)
(stop))

? (compiler '(letrec ((f (lambda (x)(if (= x 0) 1 (* x (f (- x 1))))))
                       (f 3)))
((nil) (loadcte 1) (arg)(loadfct)
 ((nil) (loadfct) ((loadvar #(0 0)) (loadcte 0) (cmp)
  (branch) ((loadcte 1) (rts))
  (loadvar #(1 0))(jsr) (mul) (rts)))(arg) (loadfct)
  ((loadvar #(0 0)) (store #(1 0)) (pop) (nil)
  (loadcte 3) (arg) (loadvar #(1 0)) (jsr) (rts))
 (jsr) (rts))
```

La machine SECD

Notre suite de machines avait pour objectif d'aboutir à la machine SECD, inventée par J. Landin vers 1965, pour compiler les langages fonctionnels. On passe de la machine SEC à la machine SECD en ajoutant les trois instructions LOADFCT, JSR et RTS et la pile supplémentaire D. En revanche, le traitement du `let` comme une macro nous permet de supprimer les instructions ADD-ENV, DEL-ENV.

Pour observer le fonctionnement de la machine SECD on ajoute une possibilité d'affichage des registres S, E, C, D quand le paramètre `trace?` vaut `#t`.

```
(define (executer-codeSECD code trace?)
```

```
(let ((Lire-env
        (lambda (adr E)
          (let ((profondeur (vector-ref adr 0))
                (indice     (vector-ref adr 1)))
            (list-ref (list-ref E profondeur) indice))))

      (Modifier-env!
        (lambda (adr E v)
          (let ((profondeur (vector-ref adr 0))
                (indice     (vector-ref adr 1)))
            (set-car! (list-tail (list-ref E profondeur) indice) v)))) )

  (letrec ((exec-aux
             (lambda (S E C D)
               (let   ((instr (car C)))
                 (if trace?               ;;*** trace facultative
                   (display-alln "S :" S   #\newline
                                 "instr: " instr "  E:"E   "  D:"D
                                  #\newline))
                 (case (mnemo instr)
                   ((LOADCTE)
                    (exec-aux (pile!:empiler (argt-instr instr) S)
                              E (cdr C) D))
                   ((ADD)
                    (let* ((v2 (pile!:depiler S))
                           (v1 (pile!:depiler S)))
                      (exec-aux (pile!:empiler (+ v1 v2) S) E (cdr C) D)))
                   ((SUB)
                    (let* ((v2 (pile!:depiler S))
                           (v1 (pile!:depiler S)))
                      (exec-aux (pile!:empiler (- v1 v2) S) E (cdr C) D)))
                   ((MUL)
                    (let* ((v2 (pile!:depiler S))
                           (v1 (pile!:depiler S)))
                      (exec-aux (pile!:empiler (* v1 v2) S) E (cdr C) D)))
                   ((CMP)
                    (let* ((v2 (pile!:depiler S))
                           (v1 (pile!:depiler S)))
                      (exec-aux (pile!:empiler (if (= v1 v2) 1 0) S)
                                E (cdr C) D)))
                   ((STOP) (display-alln "==> " (pile!:sommet S)))
                   ((LOADVAR)
                    (exec-aux (pile!:empiler (lire-env (argt-instr instr) E)
                                             S) E (cdr C) D))
                   ((STORE)
                    (modifier-env!  (argt-instr instr) E (pile!:sommet S))
                    (exec-aux S E (cdr C) D))
                   ((NIL)
                    (exec-aux (pile!:empiler '() S) E (cdr C) D))
                   ((ARG)
                    (let* ((v1 (pile!:depiler S))
```

```
                              (v2 (pile!:depiler S)))
                         (exec-aux (pile!:empiler (cons v1 v2) S)
                                   E (cdr C) D)))
                      ((POP)
                       (pile!:depiler S)
                       (exec-aux S  E (cdr C) D))

                      ;;*** nouvelles instructions
                      ((LOADFCT)
                       (let ((codeCorps (cadr C))
                             (codeSuite (cddr C)))
                         (exec-aux (pile!:empiler (cons codeCorps E) S)
                                   E codeSuite D)))
                      ((BRANCH)
                       (let ((v (pile!:depiler S)))
                         (if (= 1 v)
                           (exec-aux S E   (cadr C)   D)
                           (exec-aux S E   (caddr C)  D))))
                      ((JSR)
                       (let* ((fermeture   (pile!:depiler S))
                              (codeCorps (car fermeture))
                              (E0        (cdr fermeture))
                              (Larg      (pile!:depiler S)))
                         (exec-aux (pile!:vide) (cons Larg E0) codeCorps
                                   (pile!:empiler (list S E (cdr C)) D))))

                      ((RTS)
                       (let ((SEC (pile!:depiler D)))
                         (let ((S1 (car   SEC))
                               (E1 (cadr  SEC))
                               (C1 (caddr SEC))
                               (v  (pile!:depiler S)))
                           (exec-aux (pile!:empiler v S1) E1 C1 D)))) )))))

      (exec-aux (pile!:vide) '()  code  (pile!:vide)))))
```

Le calcul d'une expression MiniScheme est réalisé par la fonction :

```
(define (MiniScheme  exp trace?)
  (executer-codeSECD (compiler exp) trace?))
```

Quelques tests

```
? (MiniScheme  '(let ((a 1))
                  (let ((f (lambda (x)(+ x a))))
                    (f  5))) #f)
==> 6

? (MiniScheme  '(let ((f (lambda (x) x)))
                 (let ((g (lambda () 1)))
                   (f (g)))) #f)
==> 1
```

```
? (MiniScheme   '(let ((f 1))
                   (begin (let ((a 15))
                            (set! f  (lambda (x)(+ a x)))
                          (f 7))) #f)
==> 22

? (MiniScheme   '(letrec ((fac (lambda (x)
                                  (if (= x 0) 1 (* x (fac (- x 1))))))
                  (fac 5)) #f)
==> 120
```

22.4 Appel terminal et optimisation JSR-RTS

Certaines séquences d'instructions sont fréquentes car elles correspondent à des structures courantes de programmes. On peut parfois les remplacer par une autre séquence qui nécessitera moins de calcul par la machine. L'optimisation de code consiste à déceler de telles situations et à faire le remplacement. Un cas simple concerne l'*appel terminal* d'une fonction.

On dit qu'un appel de fonction est terminal dans le corps d'une fonction si cet appel est la dernière expression à évaluer :
- dans la fonction `(lambda (x) (+ (g x) 4))` l'appel `(g x)` n'est pas terminal car on doit ensuite effectuer l'addition,
- l'appel de g est terminal dans la fonction `(lambda (x) (g (+ x 4)))`.

Un appel terminal peut se dispenser de la sauvegarde dans `D` de l'état de la machine car il n'y aura rien qui viendra ensuite écraser cet état.
Comparons les parties en italique des codes correspondants :

```
? (compiler '(let ((f (lambda (y) (+ 7 y))))
               (f (* 2 6))))
((nil) (loadfct) ((loadcte 7)(loadvar #(0 0)) (add) (rts)) (arg)(loadfct)
      ((nil) (loadcte 2) (loadcte 6) (mul) (arg)(loadvar #(0 0))
      (jsr) (rts))
      (jsr) (stop))

? (compiler '(let ((f (lambda (y) (+ 7 y))))
               (* 2 (f 6))))
((nil) (loadfct) ((loadcte 7) (loadvar #(0 0)) (add) (rts)) (arg) (loadfct)
      ((loadcte 2) (nil) (loadcte 6) (arg) (loadvar #(0 0))
      (jsr) (mul) (rts))
      (jsr) (stop))
```

Dans le premier code, l'appel de `jsr` est directement suivi par l'instruction `rts`, alors que dans le deuxième la multiplication `mul` s'intercale entre `jsr` et `rts`. On détecte la présence d'un appel terminal par la succession `(jsr) (rts)`. Si l'on remarque une telle situation à la compilation, on peut remplacer ces deux

instructions par la nouvelle instruction (jsr-term) qui se comporte comme (jsr) mais n'effectue pas de sauvegarde dans D.
La structure de notre compilateur nous permet de déceler facilement la présence d'une telle séquence. Il suffit de vérifier, au moment où l'on génère l'instruction (jsr), que l'instruction suivante est (rts). Si c'est le cas, on les remplace par (jsr-term).

Pour ajouter cette optimisation à notre compilateur, il suffit de modifier la fin de la fonction compile-fct-utilisateur :

```
(define (compile-fct-utilisateur env-c Larg fct code-suivant)
  (compile-Larg env-c Larg
              (compile env-c fct
                       (if (rts? code-suivant)   ; ***
                         (cons '(JSR-TERM) (cdr code-suivant))
                         (cons '(JSR) code-suivant)))))
```

et de définir le test :

```
(define (rts? code)
  (eq? (caar code) 'RTS))
```

Après cette modification, le code du premier exemple devient :

```
? (compiler '(let ((f (lambda (y) (+ 7 y))))
              (f (* 2 6))))
((nil) (loadfct) ((loadcte 7) (loadvar #(0 0)) (add) (rts)) (arg) (loadfct)
      ((nil) (loadcte 2) (loadcte 6) (mul) (arg) (loadvar #(0 0))
      (jsr-term))
      (jsr) (stop))
```

Pour exécuter ce code, il faut ajouter à la machine SECD le cas de l'instruction (JSR-TERM) :

```
((JSR-TERM)
 (let* ((fermeture   (pile!:depiler S))
        (codeCorps (car fermeture))
        (E0        (cdr fermeture))
        (Larg      (pile!:depiler S)))
   (exec-aux (pile!:vide) (cons Larg E0) codeCorps  D)))
```

Avec cette nouvelle machine, on trouve :

```
(MiniScheme '(let ((f (lambda (y) (+ 7 y))))
              (f (* 2 6))) #f)
==> 19
```

Il y a d'autres optimisations possibles, comme l'appel récursif terminal, essentiel en Scheme, mais on renvoie aux ouvrages sur la compilation pour l'étude de ces aspects.
La machine SECD permet de compiler notre langage MiniScheme, mais c'est une machine virtuelle assez éloignée de nos ordinateurs.

22.5 Machine avec zone d'activation

On conserve le langage MiniScheme mais on va modifier la SECD pour se rapprocher d'une machine plus réaliste.

Pile d'exécution

Un premier pas consiste à fusionner le rôle des piles S et D en une seule pile. Cette pile sera située dans la mémoire de la machine, aussi on la représente à l'aide d'un *vecteur*. On note PILE ce vecteur de taille **tailleMax** et on ajoute une variable locale SP qui indique l'indice du sommet courant. L'empilement d'une valeur v consiste à incrémenter SP de 1 et à placer la valeur dans la composante d'indice SP, le dépilement consiste à décrémenter SP de 1. Bien entendu, il faut s'assurer, avant d'empiler, que l'on n'a pas atteint la limite du vecteur et, pour dépiler, que l'on a SP > 0. D'où les fonctions locales suivantes pour manipuler cette pile.

```
(empiler!
  (lambda (v)
    (set! SP (+  SP 1))
    (if (< SP tailleMax)
      (vector-set! PILE SP v)
      (*erreur* "pile pleine"))))

(depiler!
  (lambda ()
    (if (< SP 0)
      (*erreur* "pile vide")
      (let ((v (sommet)))
        (set! SP (- SP 1))
        v))))

(sommet
  (lambda ()(if (< -1 SP)
              (vector-ref PILE SP)
              (*erreur* "pile vide"))))

(incr!
  (lambda (n)(let ((i (+ SP n)))
               (if (and (<= -1 i)(< i tailleMax))
                 (set! SP i)
                 (*erreur* "debordement de pile")))))
```

La mémoire de la machine se confond avec la pile, c'est donc un vecteur que l'on lit/écrit à une adresse donnée avec les fonctions :

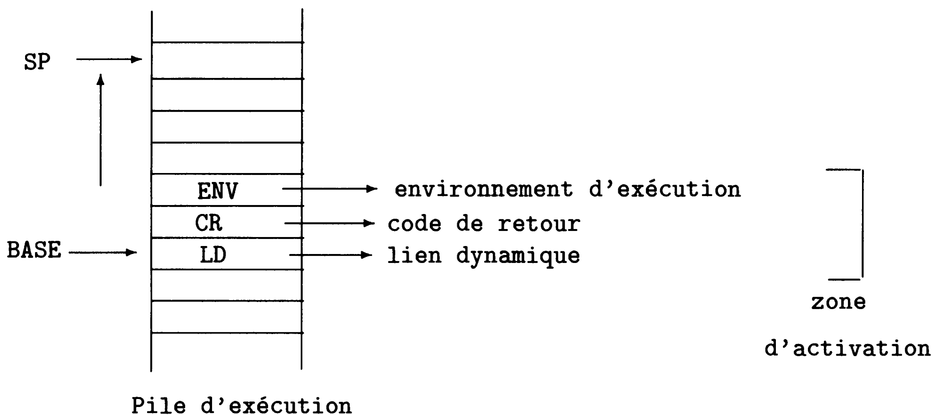
```
(define (lire-memoire adr memoire)
  (vector-ref memoire adr))

(define (ecrire-memoire! adr memoire valeur)
  (vector-set! memoire adr valeur))
```

Zone d'activation

On remplace l'opération de sauvegarde dans la pile `D` par l'empilement d'informations sur la PILE. Ces informations sont rangées dans une zone dite *zone d'activation*. Cette zone est constituée de trois informations :
- l'indice de la première case de la zone d'activation *précédente* ; on l'appelle le *lien dynamique*. L'indice de la première case de la zone d'activation *courante* est stocké dans une variable appelée `BASE` ;
- un pointeur sur la zone de code qu'il faudra exécuter après l'appel : *code de retour* ;
- un pointeur sur l'environnement d'exécution.

La nouvelle fonction pour exécuter le code prend comme paramètre supplémentaire la taille maximum de la pile. Comme il n'y a plus qu'une seule pile, on définit des fonctions locales `empiler!`, `depiler!`, `sommet` et `incr!` qui agissent sur la variable implicite `PILE`. La fonction `incr!` sert à déplacer d'un nombre `n` le sommet de la pile.

A chaque appel de fonction on empile dans la pile une nouvelle zone d'activation. L'ordre de ses trois composantes est assez arbitraire : on a choisi de placer d'abord l'indice de la base de l'appelant, puis le code à exécuter après cet appel et enfin l'environnement d'exécution. Voici un schéma de l'aspect de la pile :



Avec cette structure de la zone d'activation, l'adresse de l'environnement d'exécution est donnée par l'entier `BASE + 2`. D'où l'introduction de la fonction locale

```
(adr-env
  (lambda ()(+ BASE 2)))
```

Pour accéder à cet environnement, on définit la fonction locale : `env` :

```
(env
  (lambda ()(lire-memoire (adr-env) PILE)))
```

Modification du compilateur

Pour la compilation d'un appel de fonction, on ajoute l'instruction BLOC. Elle a simplement pour objet de réserver la place dans la pile pour la zone qui sera empilée à l'appel. Aussi, la seule modification du compilateur concerne la fonction **compile-fct-utilisateur**:

```
(define (compile-fct-utilisateur env-c Larg fct code-suivant)
  (cons '(BLOC) (compile-Larg env-c Larg          ;; ***
                              (compile env-c fct (cons '(JSR)
                                                       code-suivant)))))
```

Modifications de la machine

Les modifications de la fonction **exec-aux** sont assez immédiates. Pour simplifier on ne considère que la version sans optimisation de code. La fonction **exec-aux** ne possède plus que le paramètre C car :
- la variable S est remplacée par la variable PILE,
- l'environnement E à utiliser est stocké dans la zone d'activation courante,
- la pile D est remplacée par la sauvegarde dans la zone d'activation.

Les deux principales modifications de la machine concernent la gestion de la zone d'activation par les instructions JSR et RTS.

L'instruction JSR sert à l'installer dans la pile d'une nouvelle zone d'activation. Soit dans l'ordre :
- on empile la valeur de l'ancienne base,
- à ce moment le pointeur SP a pour valeur la nouvelle base, on la sauve dans la variable BASE
- puis on empile le code qui suit l'appel, dit code de retour,
- puis on empile l'environnement d'exécution à utiliser pendant l'appel, il est obtenu en ajoutant à l'environnement sauvé dans la fermeture la liste des valeurs des arguments,
- enfin, on exécute le code de la fonction appelée.

L'instruction RTS termine l'appel et est en quelque sorte l'inverse de JSR. Elle restaure l'état avant l'appel :
- on sauve la valeur v de l'appel qui se trouve en sommet de pile,
- on récupère le code de retour,
- on place dans BASE la valeur de la BASE de l'appelant,
- après ces dépilements, on replace la valeur v au sommet de la pile,
- on exécute le code de retour.

```
(define (executer-code  code tailleMax trace?)
  (let ((PILE (make-vector tailleMax ))
        (SP -1)
        (BASE 0)
        (lire-env
          (lambda (adr E)
            (let ((profondeur (vector-ref adr 0))
                  (indice     (vector-ref adr 1)))
```

```
             (list-ref (list-ref E profondeur) indice))))

      (modifier-env!
        (lambda (adr E v)
          (let ((profondeur (vector-ref adr 0))
                (indice     (vector-ref adr 1)))
            (set-car! (list-tail (list-ref E profondeur) indice) v)))) )

  (letrec (                         ;;*** gestion de l'envt d'exécution
           (adr-env
             (lambda ()(+ BASE 2)))

           (env
             (lambda ()(lire-memoire (adr-env) PILE)))

                                   ;;*** gestion de la pile d'exécution
           (empiler!
             (lambda (v)
               (set! SP (+  SP 1))
               (if (< SP tailleMax)
                 (vector-set! PILE SP v)
                 (*erreur* "pile pleine"))))
           (depiler!
             (lambda ()
               (if (< SP 0)
                 (*erreur*  "pile vide")
                 (let ((v (sommet)))
                   (set! SP (- SP 1))
                   v))))
           (sommet
             (lambda ()(if (< -1 SP)                    ;;***
                          (vector-ref PILE SP)
                          (*erreur*  "pile vide"))))
           (incr!
             (lambda (n)(let ((i (+ SP n)))             ;;***
                           (if (and (<= -1 i)(< i tailleMax))
                             (set! SP i)
                             (*erreur*  "debordement de pile")))))

           ;; la boucle d'exécution des instructions
           (exec-aux
             (lambda (C)
               (let   ((instr (car C)))
                 (if trace?
                   (display-alln  (affiche-pile PILE SP)  "  SP:" SP
                                #\newline
                                   "instr: " instr   "   BASE:" BASE
                                #\newline ))
                 (case (mnemo instr)
                   ((LOADCTE)
                    (empiler! (argt-instr instr))
```

```
 (exec-aux  (cdr C)))
((ADD)
 (let* ((v2 (depiler!))
        (v1 (depiler!)))
   (empiler! (+ v1 v2))
   (exec-aux (cdr C))))
((SUB)
 (let* ((v2 (depiler!))
        (v1 (depiler!)))
   (empiler! (- v1 v2))
   (exec-aux (cdr C))))
((MUL)
 (let* ((v2 (depiler!))
        (v1 (depiler!)))
   (empiler! (* v1 v2))
   (exec-aux (cdr C))))
((CMP)
 (let* ((v2 (depiler!))
        (v1 (depiler!)))
   (empiler!(if (= v1 v2) 1 0))
   (exec-aux (cdr C))))
((STOP) (display-alln "==> " (sommet)))
((LOADVAR)
 (empiler! (lire-env (argt-instr instr) (env)))
 (exec-aux  (cdr C)))
((STORE)
 (modifier-env!  (argt-instr instr)  (env) (sommet))
 (exec-aux  (cdr C)))
((NIL)
 (empiler! '())
 (exec-aux  (cdr C)))
((BLOC)      ; *** on réserve la place du nouveau bloc
 (incr! 3)
 (exec-aux  (cdr C)))
((ARG)
 (let* ((v1 (depiler!))
        (v2 ( depiler!)))
   (empiler! (cons v1 v2))
   (exec-aux (cdr C))))
((POP)
 (depiler!)
 (exec-aux  (cdr C)))
((LOADFCT)
 (let ((codeCorps (cadr C))
       (codeSuite (cddr C))
       (E0 (env)))
   (empiler! (cons codeCorps E0))
   (exec-aux  codeSuite)))
((BRANCH)
 (let ((v (depiler!)))
   (if (= 1 v)
```

```
                    (exec-aux  (cadr  C))
                    (exec-aux  (caddr C)))))
               ((JSR)                     ;***
                (let ((fermeture (depiler!)))
                  (let ((codeCorps (car fermeture))
                        (E0        (cdr fermeture))
                        (Larg      (depiler!)))
                    (incr! -3)    ;on se place sous la base du bloc
                    (empiler! BASE);on installe le lien dynamique
                    (set!     BASE SP)   ;on actualise la BASE
                    (empiler! (cdr C))   ;on sauve le codeRetour
                    (empiler! (cons Larg E0));on installe l'envt
                    (exec-aux codeCorps))));exécution du code de fct
               ((RTS)
                (let ((v (depiler!)))  ;v = la valeur de l'appel
                  (depiler!)           ;on oublie l'envt d'exécution
                  (set! C   (depiler!));récupère le code de retour
                  (set! BASE (depiler!));restaure l'ancienne BASE
                  (empiler!  v)     ;replace la valeur v au sommet
                  (exec-aux  C))) )))));exécute le code de la suite
    (exec-aux code))))
```

Pour afficher la partie active de la pile, on définit la fonction `affiche-pile` :

```
(define (affiche-pile pile SP)
  (display "PILE:")
  (if (< -1 SP)
    (begin (display "(")
           (do ((i 0 (+ 1 i)))
               ((< SP i))
             (display (vector-ref pile i))(display "  "))
           (display ")"))
    'PileVide))
```

Tests

Pour tester notre machine avec des petits exemples, on se donne une taille pour la pile d'exécution :

```
(define *tailleMax* 30)
```

On calcule la valeur d'une expression avec la fonction :

```
(define (MiniScheme  exp trace?)
  (executer-code  (compiler exp) *TailleMax* trace?))

? (MiniScheme  '(let ((a 1)(b 4))
                  (let ((c 2))
                    ((lambda (x)(begin (set! b 8)(+ a x)))
                     ((lambda (x)(+ b x)) 9))))  #f)
==> 14

? (MiniScheme  '(let ((f 1))
```

```
                 (begin (let ((a 15))
                          (set! f  (lambda (x)(+ x a)))
                        (f 7))) #f)
==> 22

? (MiniScheme '(let ((a 10)
                     (f (lambda (x) (* 2 x))))
                 (let ((g (lambda (y)(- y 1))))
                   (let ((h (lambda (z)(+ z a))))
                     (h (f (g 4)))))) #f)
==> 16

? (MiniScheme  '(letrec ((fac (lambda (x)
                                (if (= x 0)
                                  1
                                  (* x (fac (- x 1)))))))
                 (fac 4))) #f)
==> 24
```

Variables globales

On n'a pas introduit la notion de variable globale dans MiniScheme car son ajout alourdirait la présentation sans apporter d'idées nouvelles. Esquissons néanmoins une méthode pour traiter les variables globales introduites par la forme `define`.

Au moment de la *compilation*, on utilise un environnement supplémentaire constitué de la liste des variables introduites par un `define`. Aussi doit-on modifier la fonction donnant l'adresse d'une variable :
- si l'on trouve une variable dans l'environnement `env-c`, il n'y a rien de changé
- si l'on ne la trouve pas, on cherche dans l'environnement global et on rend l'indice de sa position ou un message d'erreur si elle n'y est pas.

Au moment de *l'exécution* du code, on ajoute un environnement d'exécution qui contient les valeurs des variables globales. On rencontre deux types d'adresse : les adresses à deux entiers pour les variables locales et à un entier pour les globales. Cette distinction est utilisée par les instructions `LOADVAR` et `STORE`. C'est l'environnement d'exécution qui est consulté ou modifié quand on exécute une instruction `LOADVAR` ou `STORE` avec une adresse réduite à un entier.

22.6 Assemblage et machine avec compteur ordinal

Dans une vraie machine, le code est rangé en mémoire, il est donc naturel de le modéliser par un *vecteur* d'instructions au lieu d'une liste d'instructions. La transformation du code d'une liste en un vecteur se fait en deux temps :
- dans un premier temps, on transforme le code en une liste plate en introduisant la notion d'étiquette symbolique ;

- dans un deuxième temps, on procède à l'*assemblage* : on remplace la liste par un vecteur et les étiquettes symboliques par des adresses en mémoire.

Génération de code avec étiquettes symboliques

Pour aplatir le code on doit considérer les deux occasions où l'on introduisait des sous-listes dans le code : la compilation des formes `lambda` et `if`.

```
? (compiler '((lambda (x) (+ x 1)) 4))
((bloc) (nil) (loadcte 4) (arg) (loadfct)
       ((loadvar #(0 0)) (loadcte 1) (add) (rts))
       (jsr)
 (stop))

? (compiler '(if (= 0 1) 3 4)))
((loadcte 0) (loadcte 1) (comp)
 (branch)((loadcte 3) (stop))
         ((loadcte 4) (stop)))
```

Ces sous-listes permettent de sauter d'un seul coup toute une suite d'instructions. On va obtenir le même effet en introduisant des instructions de saut et de label :

`(LABEL etiquette)` Cette instruction sert à désigner une place dans le code à l'aide d'une étiquette symbolique mais n'a pas d'effet sur la machine.

`(JUMP etiquette)` Cette instruction a pour effet de modifier l'enchaînement des instructions. On doit continuer à exécuter le code qui se trouve après le `LABEL` de même étiquette. C'est le fameux `GOTO` si décrié dans les langages de haut niveau.

On aura aussi besoin d'une instruction de saut conditionnel :

`(JUMPZ etiquette)` cette instruction se comporte comme `(JUMP etiquette)` si la valeur du sommet de pile est `0` et on peut l'ignorer sinon.

Avec ces instructions de saut, on va linéariser le code des deux formes précédentes.

Une expression `if` se compile en la suite d'instructions suivantes :

```
(if test exp-si exp-sinon)

--->  (
       contenu du code-test
       (JUMPZ  etiq-sinon)
       contenu du code-exp-si
       (JUMP etiq-suite)
       (LABEL etiq-sinon)
       contenu du code-exp-sinon
       (LABEL etiq-suite)
      )
```

Si le test a pour valeur 0, on saute directement à l'exécution du code de **exp-sinon** et dans le cas contraire, on exécute le code de **exp-si** puis on saute à la suite.

Une lambda expression se compile en la suite d'instructions suivantes :

```
(lambda Lpar corps)

---> (
      (LOADFCT)
      (JUMP etiq-suite)
      contenu du code-corps
      (RTS)
      (LABEL etiq-suite)
     )
```

Après chargement du code de la fonction, l'instruction (JUMP etiq-suite) sert à sauter le code du corps pour passer à l'instruction qui suit.

Voici les nouvelles expressions pour le code des deux exemples précédents :

```
? (compiler '(if (= 0 1) 3 4))
((loadcte 0) (loadcte 1) (cmp)  ; code du test
 (jumpz etiq1)
 (loadcte 3)                     ; code exp-si
 (jump etiq2)
 (label etiq1)
 (loadcte 4)                     ; code exp-sinon
 (label etiq2)
 (stop))

? (compiler '((lambda (x) (+ x 1)) 4))
((bloc) (nil) (loadcte 4) (arg)
 (loadfct)
 (jump etiq3)
 (loadvar #(0 0)) (loadcte 1)(add)  ; code du corps
 (rts)
 (label etiq3)
 (jsr)
 (stop))
```

Pour engendrer ce nouveau code, il suffit de modifier les fonctions de compilation des formes **lambda** et **if**. La génération des noms des étiquettes est faite avec le générateur de symboles **creer-genvar**.

```
(define  gen-etiquette  (creer-genVar "etiq"))

(define (compile-lambda env-c lambda-exp code-suivant)
  (let ((Lvar  (cadr  lambda-exp))
        (corps (caddr lambda-exp))
        (etiq-suite (gen-etiquette)))
    (cons '(LOADFCT)
          (cons `(JUMP ,etiq-suite)
```

```
                (compile (cons Lvar env-c) corps
                         (cons '(RTS)
                               (cons `(LABEL ,etiq-suite)
                                     code-suivant))))))

(define (compile-if env-c if-exp code-suivant)
 (let ((exp-test   (cadr   if-exp))
       (exp-si     (caddr  if-exp))
       (exp-sinon  (cadddr if-exp))
       (etiq-sinon (gen-etiquette))
       (etiq-suite (gen-etiquette)))
   (compile env-c  exp-test
          (cons  `(JUMPZ ,etiq-sinon)
                 (compile env-c exp-si
                          (cons `(JUMP  ,etiq-suite)
                                (cons `(LABEL ,etiq-sinon)
                                      (compile env-c exp-sinon
                                               (cons `(LABEL ,etiq-suite)
                                                     code-suivant)))))))))
```

Remarque 1 Ce type d'instruction de saut est très commode pour compiler des boucles. Par exemple, la boucle `(while test corps)` se compile en :

```
((LABEL etiq-while)
 contenu du code-test
 (JUMPZ etiq-suite)
 contenu du code-corps
 (JUMP etiq-while)
 (LABEL etiq-suite))
```

Assemblage

La deuxième étape consiste à recopier la liste d'instructions dans un vecteur en supprimant les instructions LABEL et en remplaçant les étiquettes symboliques par les adresses dans le vecteur. Cette transformation s'appelle l'*assemblage* de la liste d'instructions. Pour une vraie machine, l'assemblage consiste également à remplacer tous les mnémoniques par des codes binaires.

L'assemblage se fait en deux passes. La première passe parcourt la liste d'instructions et note le numéro i de l'instruction qui suit chaque LABEL, sans compter les instructions LABEL. On construit ainsi une a-liste `Letiq.indice` qui associe à chaque étiquette sa position dans le futur vecteur de code et on construit également la liste Lcode des instructions sans les LABEL. A la fin de la première passe, l'indice i est égal à la longueur du code.

Après avoir recopié toutes les instructions dans un vecteur, on effectue la seconde passe. Elle remplace chaque étiquette symbolique d'un saut par l'adresse correspondante dans le code, cette adresse est trouvée dans la a-liste `Letiq.indice`.

```
(define (assembler Linstr)
  (letrec ((Passe1
            (lambda (Linstr i Letiq.indice Lcode)
              (if (null? Linstr)
                (Passe2 i Letiq.indice  (list->vector (reverse Lcode)))
                (let ((instr (car Linstr)))
                  (if (eq? 'LABEL (Mnemo instr))
                    (Passe1 (cdr Linstr) i
                            (acons (argt-instr instr) i Letiq.indice) Lcode
                    (Passe1 (cdr Linstr) (+ i 1)
                            Letiq.indice (cons instr Lcode)))))))

           (Passe2
            (lambda (Long-code Letiq.indice code-vect)
              (do ((i 0 (+ i 1)))
                  ((= Long-code i) code-vect)
                (let ((instr  (vector-ref code-vect i)))
                  (if (memq  (Mnemo instr) '(JUMP JUMPZ))
                    (set-car! (cdr instr)
                              (cdr (assq (argt-instr instr) Letiq.indice))
                                   )))))))
    (Passe1 Linstr 0 '() '())))

? (assembler (compiler '(if (= 0 1) 3 4)))
#((loadcte 0) (loadcte 1) (cmp)
  (jumpz 6) (loadcte 3)
  (jump 7) (loadcte 4) (stop))

? (assembler (compiler '((lambda (x) (+ x 1)) 4))
(#((bloc) (nil) (loadcte 4) (arg)(loadfct)
   (jump 8) (loadvar #(0 0)) (loadcte 1) (add) (rts)
   (jsr)
   (stop))
```

Machine avec compteur ordinal

Pour exécuter ce code assemblé, on doit légèrement modifier notre machine. On supprime l'instruction `BRANCH` et on ajoute à la place les instructions `JUMP` et `JUMPZ`. De plus, la gestion du code est différente, on introduit une variable `PC` (*Program Counter*), appelée *compteur ordinal*, qui contient l'adresse de la prochaine instruction à effectuer.

Remarque 2 Dans une vraie machine, les valeurs des variables numériques comme `SP`, `BASE`, `PC` et le codage de l'instruction courante `instr` sont situés dans des *registres*, de façon à minimiser les opérations de lecture en mémoire.

La fonction `executer-code` utilise la nouvelle fonction locale `charger-instr` (*fetch* en anglais) pour lire l'instruction à exécuter et incrémenter de 1 le compteur ordinal. On peut donc se dispenser de paramètre dans la fonction locale `exec-aux`.

Les principales modifications concernent l'introduction des instructions JUMP et JUMPZ. Par ailleurs, on remplace dans la fermeture d'une lambda le code du corps par l'adresse de ce code. D'après la forme de la compilation d'une lambda expression, on sait que le code du corps se trouve deux cases plus loin que celui de l'instruction LOADFCT. Au moment de l'appel, on ne sauve plus le code de retour mais l'adresse de ce code appelée *adresse de retour*. Voici les parties modifiées dans la machine :

```
(define (executer-code  code tailleMax trace?)
  (let ((PILE (make-vector tailleMax))
        ...
        (PC 0)            ;*** compteur ordinal
         ...
    (letrec (
             ...
             (charger-instr (lambda () ;*** actualise le compteur ordinal
                                 (set! PC (+ PC 1))
                                 (vector-ref code (- PC 1))))
             ;; la boucle d'exécution des instructions
             (exec-aux
               (lambda ()
                 (let   ((instr (charger-instr)))
                   (if trace?
                     (display-alln (affiche-pile PILE SP)  "  SP:" SP
                                   #\newline
                                   "PC :" PC "instr: "instr " BASE: " BASE
                                   #\newline))
                   (case (Mnemo instr)
                     ((LOADCTE) (empiler! (argt-instr instr))
                                (exec-aux))
                     ((ADD)
                      (let* ((v2 (depiler!))
                             (v1 (depiler!)))
                        (empiler! (+ v1 v2))
                        (exec-aux)))
                     ...

                     ((LOADFCT)
                      (let ((adr-codeCorps (+ PC 1))       ;; ***
                            (E0 (env)))
                        (empiler! (cons adr-codeCorps E0))
                        (exec-aux)))
                     ((JUMP)                               ;; ***
                      (set! PC (argt-instr instr))
                      (exec-aux))
                     ((JUMPZ)                              ;; ***
                      (let ((v (depiler!)))
                        (if (zero? v)
                          (set! PC (argt-instr instr)))
                        (exec-aux)))
                     ((JSR)
```

```
                        (let ((fermeture (depiler!)))
                          (let ((PC-codeCorps (car fermeture))
                                (E0           (cdr fermeture))
                                (Larg         (depiler!)))
                            (incr! -3)       ;on se place sous la base du bloc
                            (empiler! BASE) ;installe le lien dynamique
                            (set!     BASE SP); actualise la BASE
                            (empiler! PC)      ;***  sauve l'adr de retour
                            (empiler! (cons Larg E0)); installe l'envt
                            (set! PC  PC-codeCorps);*** début du code du corps
                            (exec-aux)))       ;on exécute le code de la fct
                       ((RTS)
                        (let ((v (depiler!)))   ;v = la valeur de l'appel
                          (depiler!)            ;on oublie l'envt d'exécution
                          (set! PC    (depiler!));***récupère l'adr de retour
                          (set! BASE (depiler!)) ;restaure l'ancienne BASE
                          (empiler!  v)       ; replace la valeur v au sommet
                          (exec-aux)))  ))))); exécute le code de la suite

      (exec-aux)))
```

Pour calculer une expression avec cette machine, on doit auparavant assembler le code produit par le compilateur :

```
(define (MiniScheme  exp trace?)
  (executer-code  (assembler (compiler exp)) *Taille* trace?))
```

On teste avec le calcul de 4! :

```
? (MiniScheme '(letrec ((fac (lambda (x)
                                (if (= x 0) 1 (* x (fac (- x 1)))))))
                (fac 4)))) #f)
==> 24
```

22.7 Représentation des doublets et machine Lisp

Notre MiniScheme présente une importante lacune : pas de gestion des listes. On détaille dans ce paragraphe les modifications à apporter pour y remédier.

Modifications du compilateur

On ajoute à notre langage les expressions relatives aux listes :

```
() | (cons  e1 e2) | (car  e)| (cdr e)| (null? e)| (pair? e)| (eq? e1 e2)
```

Pour tenir compte des nouveaux opérateurs prédéfinis : `cons, car, cdr, eq?, null?, pair?`, on doit modifier les deux fonctions suivantes du compilateur.

```
(define (fct-predefinie? fct)
  (and (symbol? fct)
       (memq fct '(+ - * = car cdr cons null? eq? pair?)))) ; ***
```

```
(define (code-op op)
  (case op
    ((+)      '(ADD))
    ((-)      '(SUB))
    ((*)      '(MUL))
    ((=)      '(CMP))
    ((cons)   '(_CONS))    ;*** et suivantes
    ((car)    '(_CAR))
    ((cdr)    '(_CDR))
    ((null?)  '(_NULL?))
    ((eq?)    '(_EQ?))
    ((pair?)  '(_PAIR?))
    (else (*erreur* "fonction inconnue"))))
```

Il faut aussi prévoir parmi les constantes le cas de la liste vide (on n'a pas introduit l'opération de citation dans notre langage). On convient de représenter la liste () par le symbole vide. Cela conduit à modifier la compilation d'une constante :

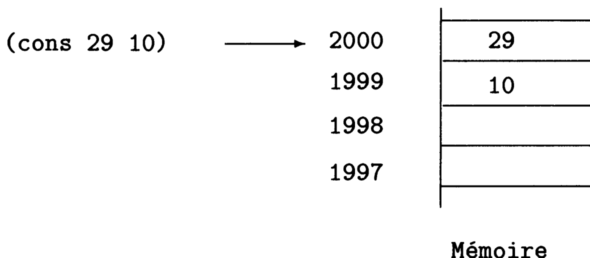
```
(define (compile env-c exp code-suivant)
  (cond ((number? exp)
         (cons `(LOADCTE ,exp) code-suivant))
        ((null? exp)                    ;;*** cas de la liste vide
         (cons '(LOADCTE vide) code-suivant))
        ...
```

La suite du code du compilateur est inchangée.

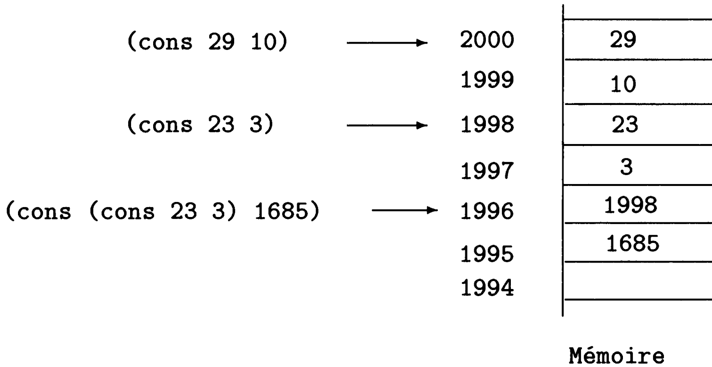
Représentation des doublets

Actuellement, les seules valeurs possibles pour une expression sont les nombres et les fermetures. Pour y ajouter les listes, le point important est la représentation en machine des doublets (voir aussi le chapitre 2 §10).

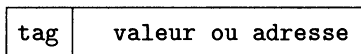
On convient de représenter un doublet par deux cases mémoires consécutives, la première pour le car et la seconde pour le cdr (il y a d'autres représentations possibles). On choisit d'allouer les adresses des doublets en descendant, l'adresse du cdr sera celle du car -1. La valeur d'un doublet est par définition l'adresse de son car. Par exemple, si le doublet (cons 29 10) est rangé à partir de l'adresse mémoire 2000, on a le schéma :



Considérons maintenant la paire `p = (cons (cons 23 3) 1685)` : son `car` est la paire `(cons 23 3)` dont la valeur est rangée à l'adresse 1998. Par conséquent, on range dans la première case attribuée à `p` (disons 1996) la valeur 1998 et dans la seconde la valeur 1685.

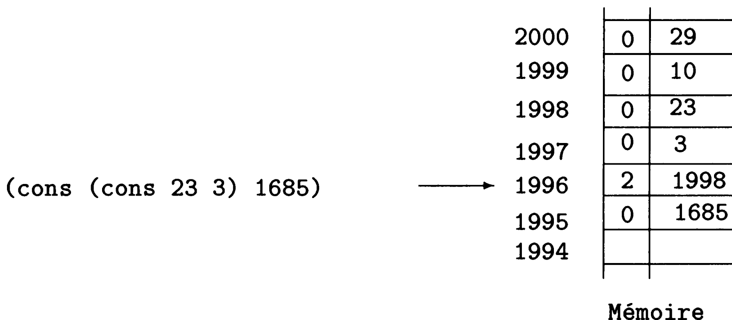


Mais les nombres 1998 et 1685 n'ont pas du tout la même signification : l'un est une adresse mémoire, l'autre un entier. Sous cette forme, on pourrait penser que la paire rangée à l'adresse 1996 est `(cons 1998 1685)`, on doit donc avoir un moyen pour distinguer la nature des valeurs rangées en mémoire. Une méthode consiste à utiliser une marque ou *tag*, c'est-à-dire que l'on réserve quelques bits dans chaque case mémoire pour coder le type du contenu. Par exemple, une valeur entière (petite) aura le tag 0, une procédure le tag 1, une paire le tag 2, la liste vide le tag 3, un caractère le tag 4 ...



Représentation d'une valeur en mémoire

Avec ces conventions, notre schéma devient :



Il est clair maintenant que la valeur [2 1996] est celle d'un doublet d'expression `(cons (cons 23 3) 1685)`.

Valeurs avec tag

Pour la clarté de l'implantation, on choisit des tag symboliques : nb pour les «petits» entiers (c.-à-d. ceux qui tiennent dans une case mémoire), fermeture pour les valeurs fonctionnelles, null pour la liste vide, cons pour les valeurs doublets. Une valeur avec tag sera codée par un vecteur de la forme #(tag valeur).
Par exemple :
#(nb 333) représente le nombre 333,
#(cons 2004) représente un doublet dont le car est à l'adresse 2004
(et donc le cdr est à l'adresse 2003),
#(null -1) représente la liste vide, on affecte arbitrairement la valeur -1
à son champ adresse car il ne sera pas utilisé.

C'est l'utilisation systématique de valeurs avec tag qui permet de dire que Scheme est un langage dont les *valeurs* sont typées contrairement aux variables qui elles n'ont pas de type.

D'où les fonctions de manipulation des valeurs avec tag :

```
(define (creer-valeur-tag  tag valeur)
  (vector tag valeur))

(define (son-tag  vt)
  (vector-ref vt 0))

(define (son-adresse  vt)
  (vector-ref vt 1))
```

On appelle son-adresse l'accès au champ valeur car dans la plupart des cas cette valeur est une adresse.

Cette modification de la représentation des valeurs dans la pile conduit à changer les fonctions arithmétiques. Pour faire une opération il faut accéder à la valeur puis associer au résultat une valeur avec tag. Par exemple, dans le cas de ADD, il faut écrire :

```
((ADD)
 (let* ((v2 (son-adresse (depiler!)))
        (v1 (son-adresse (depiler!))))
   (empiler! (creer-valeur-tag 'nb (+ v1 v2)))
   (exec-aux)))
```

Implantation des doublets : le tas

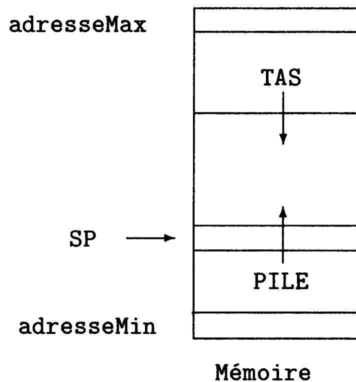
Pour rendre l'implantation des fonctions car, cdr, cons indépendante du choix de la représentation, on définit une couche de fonctions pour manipuler les doublets.
On reconnaît une valeur doublet par son tag :

```
(define (doublet? vt)
  (eq? 'cons (son-type vt)))
```

Quand une valeur avec tag représente un doublet, le car et le cdr sont donnés par :

```
(define (le-car vt memoire)
  (lire-memoire (son-adresse vt) memoire))

(define (le-cdr vt memoire)
  (lire-memoire (- (son-adresse vt) 1) memoire))
```

Chaque appel de la fonction binaire cons provoque la création d'un nouveau doublet. Pour fixer les idées, on se place dans le cas de figure où la zone mémoire utilisée pour ranger les doublets est la partie haute non occupée par la pile. D'une façon générale, on appelle *tas* (*heap* en anglais) la zone de la mémoire où l'on range les doublets.



Le premier doublet est alloué à l'adresse adresseMax, puis les suivants dessous jusqu'à ce que l'on rencontre la zone utilisée par la PILE. Quand cela se produit, on essaye de faire le ménage dans le tas pour récupérer de la place (voir §8). De même, quand on augmente la pile, il faut s'assurer que l'on ne rencontre pas le tas.

On utilise une variable globale *adr-doublet-libre* qui contient l'adresse du dernier doublet créé. Pour en créer un nouveau, on appelle la fonction prochaine-adresse-libre. Cette fonction cherche une nouvelle adresse pour ranger un doublet sans téléscoper la PILE. On écrit à cette adresse la valeur du car puis dessous celle du cdr. Ceci est réalisé par la fonction creer-doublet, elle rend l'adresse du doublet avec le tag cons.

```
(define (creer-doublet son-car son-cdr memoire SP) ; ***
  (let ((adresse-doublet (nouvelle-adresse-libre SP)))
    (ecrire-memoire!       adresse-doublet memoire son-car)
    (ecrire-memoire! (- adresse-doublet 1) memoire son-cdr)
    (creer-valeur-tag  'cons  adresse-doublet)))
```

La fonction nouvelle-adresse-libre dépend de la méthode de récupération des doublets inutilisés (voir §8). Elle peut être de la forme suivante :

```
(define (nouvelle-adresse-libre SP) ;***
  (let ((adr (- *adr-doublet-libre*  2)))
    (cond ((< SP (- adr 1))
           (set! *adr-doublet-libre* adr)
           adr)
          (else (*erreur* "memoire pleine ")))))
```

On initialise la variable `*adr-doublet-libre*` par :

```
(define *adr-doublet-libre* (+ *adresseMax* 2))
```

Il est alors immédiat d'ajouter à notre machine des instructions `_CONS`, `_CAR`, `_CDR`.
L'exécution de `_CONS` consiste à dépiler ses arguments qui fournissent le `car` et le `cdr` du nouveau doublet qui sera empilé :

```
((_CONS)
 (let* ((son-cdr (depiler!))
        (son-car (depiler!)))
   (empiler! (creer-doublet son-car son-cdr PILE SP))
   (exec-aux)))
```

Avant d'exécuter `_CAR`, on s'assure que le sommet de pile est bien un doublet, si oui, il suffit d'empiler la valeur de son `car` :

```
((_CAR)
 (let ((vt (depiler!)))
   (if (doublet? vt)
     (begin (empiler! (le-car vt PILE))
            (exec-aux))
     (*erreur* "ce n'est pas un doublet "))))
```

On procède de façon analogue pour les autres fonctions sur les listes : `_CDR`, `_NULL?`, `_PAIR?`.

Machine Lisp

Mais les modifications de notre machine ne se bornent pas à l'ajout des fonctions sur les listes. On doit y remplacer *l'utilisation des listes Scheme par leurs représentations dans notre machine.* Il s'agit principalement de la gestion de l'environnement qui sera une liste située dans le tas de la machine.
Pour lire l'environnement ou le modifier,, il faut remplacer les fonctions Scheme `list-ref`, `list-tail`, `set-car!` par leurs équivalents `_LIST-REF`, `_LIST-TAIL`, `_SET-CAR!` dans la machine. Ce seront des fonctions locales dont voici les expressions immédiates :

```
(_LIST-REF
  (lambda (doublet i)
    (if (zero? i)
      (le-car doublet PILE)
      (LIST-REF (le-cdr doublet PILE) (- i 1)))))
```

```
(_LIST-TAIL
  (lambda (doublet i)
    (if (zero? i)
      doublet
      (LIST-TAIL (le-cdr doublet PILE) (- i 1)))))

(_SET-CAR!
  (lambda (doublet vt)
    (ecrire-memoire! (son-adresse doublet) PILE vt)))
```

La manipulation de l'environnement est toujours assurée par les fonctions suivantes, mais on a substitué les fonctions machines aux fonctions Scheme :

```
(lire-env
  (lambda (adr E)
    (let ((profondeur (vector-ref adr 0))
          (indice     (vector-ref adr 1)))
      (_LIST-REF  (_LIST-REF  E profondeur) indice))))

(Modifier-env!
  (lambda (adr E vt)
    (let ((profondeur (vector-ref adr 0))
          (indice     (vector-ref adr 1)))
      (_SET-CAR! (_LIST-TAIL (_LIST-REF E profondeur) indice) vt))))
```

On doit aussi modifier les instructions NIL et ARG pour les adapter aux listes machines.

```
((NIL)
 (empiler! (creer-valeur-tag  'null -1))
 (exec-aux))

((ARG)
 (let* ((vt1 (depiler!))
        (vt2 (depiler!)))
   (empiler! (creer-doublet vt1 vt2 PILE SP)))
 (exec-aux))
```

Une fermeture n'est plus une paire Scheme mais un couple constitué de deux valeurs avec tag : l'adresse de son code et l'adresse de son environnement. Ce qui oblige à remplacer les fonctions `car` , `cons`, `cdr` par leurs analogues machines : `le-car`, `le-cdr`, `creer-doublet` dans les instructions `LOAD-FCT`, `JSR`, `RTS`.

Les modifications de la machine étant assez nombreuses, on donne intégralement le code de la machine Lisp :

```
(define (executer-code  code adresseMax trace?)
  (let ((PILE (make-vector (+ 1 adresseMax)))
        (SP -1)
        (BASE 0)
```

```
    (PC 0))

  (letrec ( ;;*************** gestion de la pile d'exécution
          (empiler!
            (lambda (v)
              (set! SP (+  SP 1))
              (if (< SP *adr-doublet-libre*)
                (vector-set! PILE SP v)
                (*erreur* "memoire pleine"))))
          (depiler!
            (lambda ()
              (if (< SP 0)
                (*erreur*  "pile vide")
                (let ((v (sommet)))
                  (set! SP (- SP 1))
                  v))))
          (sommet
            (lambda ()
              (if (< -1 SP)
                (vector-ref PILE SP)
                (*erreur*  "pile vide"))))
          (incr!
            (lambda (n)(let ((i (+ SP n)))
                          (if (and (<= -1 i)(< i *adr-doublet-libre*))
                            (set! SP i)
                            (*erreur*  "mémoire pleine")))))

          ;;*************** gestion de l'environnement d'exécution

          (_LIST-REF
            (lambda (doublet i)
              (if (zero? i)
                (le-car doublet PILE)
                (LIST-REF (le-cdr doublet PILE) (- i 1)))))

          (_LIST-TAIL
            (lambda (doublet i)
              (if (zero? i)
                doublet
                (LIST-TAIL (le-cdr doublet PILE) (- i 1)))))

          (_SET-CAR!
            (lambda (doublet vt)
              (ecrire-memoire! (son-adresse doublet) PILE vt)))

          (lire-env
            (lambda (adr E)
              (let ((profondeur (vector-ref adr 0))
                    (indice     (vector-ref adr 1)))
                (LIST-REF  (LIST-REF  E profondeur) indice))))
```

```
(Modifier-env!
  (lambda (adr E vt)
    (let ((profondeur (vector-ref adr 0))
          (indice     (vector-ref adr 1)))
      (_SET-CAR! (_LIST-TAIL (_LIST-REF E profondeur) indice)
                 vt))))

(adr-env
  (lambda () (+ BASE 2)))

(env
  (lambda ()(lire-memoire  (adr-env) PILE)))

;; le compteur ordinal PC
(charger-instr
  (lambda ()
    (let ((instr (vector-ref code  PC)))
      (set! PC (+ PC 1))
      instr)))

;; la boucle d'exécution des instructions
(exec-aux
  (lambda ()
    (let ((instr (charger-instr)))
      (if trace?    ;;
        (display-alln  (affiche-pile PILE SP)  "  SP:" SP
                       #\newline
                       "PC :"PC" instr: "instr " BASE:" BASE
                       #\newline))
      (case (Mnemo instr)
        ((LOADCTE) (empiler! (let ((valeur (argt-instr instr)))
                               (if (eq? 'vide valeur)
                                 (creer-valeur-tag 'null -1)
                                 (creer-valeur-type 'nb
                                                    valeur))))
                   (exec-aux))
        ((ADD)
         (let* ((v2 (son-adresse (depiler!)))
                (v1 (son-adresse (depiler!))))
           (empiler! (creer-valeur-type 'nb (+ v1 v2)))
           (exec-aux)))
        ((SUB)
         (let* ((v2 (son-adresse (depiler!)))
                (v1 (son-adresse (depiler!))))
           (empiler! (creer-valeur-type 'nb (- v1 v2)))
           (exec-aux)))
        ((MUL)
         (let* ((v2 (son-adresse (depiler!)))
                (v1 (son-adresse (depiler!))))
           (empiler! (creer-valeur-type 'nb (* v1 v2)))
           (exec-aux)))
```

```
((CMP)
 (let* ((v2 (son-adresse (depiler!)))
        (v1 (son-adresse (depiler!))))
   (empiler! (creer-valeur-type 'nb
                                (if (= v1 v2) 1 0)))
   (exec-aux)))
((_CONS)
 (let* ((son-cdr (depiler!))
        (son-car (depiler!)))
   (empiler! (creer-doublet son-car son-cdr PILE SP))
   (exec-aux)))
((_CAR)
 (let ((vt (depiler!)))
   (if (doublet? vt)
     (begin (empiler! (le-car vt PILE))
            (exec-aux))
     (*erreur* "ce n'est pas un doublet "))))
       ((_CDR)
        (let ((vt (depiler!)))
   (if (doublet? vt)
     (begin (empiler! (le-cdr vt PILE))
            (exec-aux))
     (*erreur* "ce n'est pas un doublet "))))
((_NULL?)
 (let ((vt  (depiler!)))
   (empiler!
     (creer-valeur-type 'nb
                        (if (eq? 'null (son-tag vt))
                            1 0))))
 (exec-aux))
((_PAIR)
 (let ((vt  (depiler!)))
   (empiler!
     (creer-valeur-type 'nb  (if (doublet? vt) 1 0))))
 (exec-aux))
((_EQ?)
 (let* ((vt1 (depiler!))
        (vt2 (depiler!)))
   (empiler!
     (creer-valeur-type 'nb
                        (if (and
                              (eq? (son-tag vt1)
                                   (son-tag vt2))
                              (eq? (son-adresse vt1)
                                   (son-adresse vt2)))
                          1
                          0))))
 (exec-aux))
((STOP) (display-alln  "==> " (sommet)))
((LOADVAR)
 (empiler! (lire-env (argt-instr instr) (env)))
```

```
 (exec-aux))
((STORE)
 (modifier-env!  (argt-instr instr) (env) (sommet))
 (exec-aux))
((NIL)
 (empiler! (creer-valeur-type  'null 0))
 (exec-aux))
((BLOC)
 (incr! 3)
 (exec-aux))
((ARG)
 (let* ((vt1 (depiler!))
        (vt2 (depiler!)))
   (empiler! (creer-doublet vt1 vt2 PILE SP)))
 (exec-aux))
((POP)
 (depiler!)
 (exec-aux))
((LOADFCT)
 (let ((adr-codeCorps (+ PC 1))
       (E0 (env)))
   (let ((une-fermeture
           (creer-doublet
             (creer-valeur-type 'nb adr-codeCorps)
             E0 PILE SP)))
     (empiler! (creer-valeur-type
                 'fermeture
                 (son-adresse une-fermeture))))
   (exec-aux)))
((JUMP)
 (set! PC (argt-instr instr))
 (exec-aux))
((JUMPZ)
 (let ((vt (depiler!)))
   (if (zero? (son-adresse vt))
     (set! PC (argt-instr instr)))
   (exec-aux)))
((JSR)
 (let ((fermeture (depiler!)))
   (let ((PC-codeCorps (son-adresse
                         (le-car fermeture PILE)))
         (E0 (le-cdr fermeture PILE))
         (Larg (depiler!)))
     (incr! -3)
     (empiler! BASE)
     (set!     BASE SP)
     (empiler! PC)
     (empiler! (creer-doublet Larg E0 PILE SP))
     (set! PC  PC-codeCorps)))
 (exec-aux))
((RTS)
```

```
                    (let ((v (depiler!)))
                      (depiler!)
                      (set! PC   (depiler!))
                      (set! BASE (depiler!))
                      (empiler!  v)
                      (exec-aux))  )))))
      (set! *adr-doublet-libre* (+ adresseMax 2))
      (exec-aux))))
```

Quelques tests

```
? (MiniScheme  '(car (cdr (cons 1 (cons 2 ())))) #f)
==> 2

? (MiniScheme  '(eq?  (cons 1 (cons 2 ()))
                                   (cons 1 (cons 2 ()))) #f)
==> 0

? (MiniScheme  '(letrec ((dernier (lambda (L)
                                    (if (null? (cdr L))
                                      (car L)
                                      (dernier (cdr L))))))
                 (dernier (cons 1 (cons 2 (cons 3 ()))))))) #f)
==> 3

? (MiniScheme '(cons (cons 23 3) 1685) #f)
==>  #(doublet 97)

 ? (MiniScheme '() #t)
==> vide
```

Un imprimeur pour Scheme

Les deux derniers tests soulèvent un problème : on n'a pas adapté l'imprimeur à l'affichage des valeurs calculées par notre machine.
La valeur `#(doublet 97)` de la paire `(cons (cons 23 3) 1685)` n'est pas ce qu'attendait l'utilisateur, il aimerait voir `((23 . 3) . 1685)`. De même, on souhaite que l'affichage de la liste vide soit `()`. Pour cela, on ajoute la fonction `valeur-tag-str` qui calcule la chaîne d'affichage d'une valeur avec tag. Elle prend en paramètre la mémoire car l'affichage d'un doublet oblige à lire le contenu mémoire associé au car et au cdr.

```
(define (valeur-tag-str vt memoire)
  (case (son-tag vt)
    ((nb) (number->string (son-adresse vt)))
    ((null) "()")
    ((fermeture)  "#(procedure)")
    ((cons)
     (let ((son-car (le-car vt memoire))
           (son-cdr (le-cdr vt memoire)))
       (string-append
```

```
          "(" (valeur-tag-str son-car memoire) " . "
              (valeur-tag-str son-cdr memoire) ")") ))
    (else (*erreur* " valeur de type inconnu "))))
```

Pour avoir un affichage satisfaisant du résultat d'un calcul, il reste à remplacer dans la machine la ligne :

```
((STOP) (display-alln  "==> " (sommet) ))
```

par

```
((STOP) (display-alln  "==> " (valeur-tag-str (sommet) PILE)))
```

Maintenant, on a :

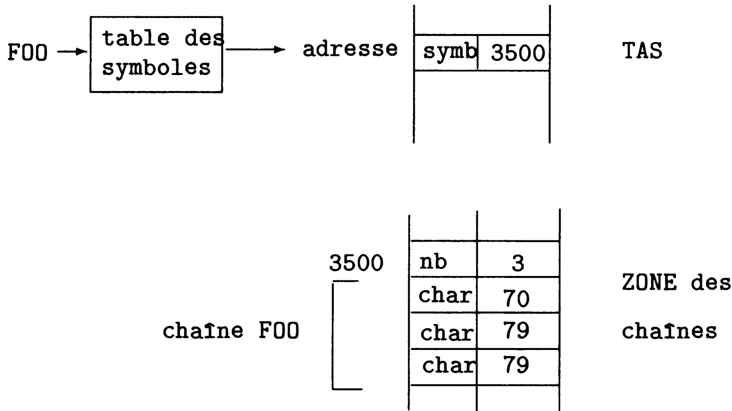
```
? (MiniScheme  '(cons (cons 23 3) 1685) #f)
==>  ((23 . 3) . 1685)

? (MiniScheme  '(lambda (x) (+ x 1)) #f)
==> #(procedure)

? (MiniScheme  '() #f)
==> ()
```

Le lecteur pourra raffiner cet imprimeur en effectuant un affichage sans paire pointée quand le doublet est en fait une vraie liste.

Les valeurs symboles

Pour ne pas alourdir plus la machine Lisp, on n'a pas introduit l'opération de citation ni les valeurs symboles. Voici néanmoins le principe d'une représentation possible. Une valeur symbole est introduite par évaluation d'une expression de la forme `'symbole`. Le compilateur lui associe une instruction `(LOADCTE symbole)`. Pour exécuter cette instruction, la machine Lisp dispose d'une table des symboles déjà rencontrés.[2] C'est une table qui associe à chaque symbole une adresse dans le tas. Quand on rencontre un symbole, on consulte cette table pour savoir si l'on a déjà vu ce symbole, si oui, on obtient son adresse, sinon, on doit créer (on dit *interner*) une nouvelle entrée pour ce symbole. Il faut trouver une adresse dans le tas pour y ranger la représentation du symbole.
Un symbole est représenté avec un tag spécifique aux symboles et une valeur qui est l'adresse du début de la zone où sera stockée la chaîne des caractères qui composent son nom.
Une chaîne est, en général, représentée par une suite contiguë de cases mémoires ; la première case codant le nombre des caractères et les autres contenant les codes des caractères successifs. On résume tout ceci par un petit schéma :

[2] On utilise généralement la technique de l'adressage dispersé vue au chapitre 7 §6 pour définir cette table.



22.8 Récupération des doublets inutilisés : GC

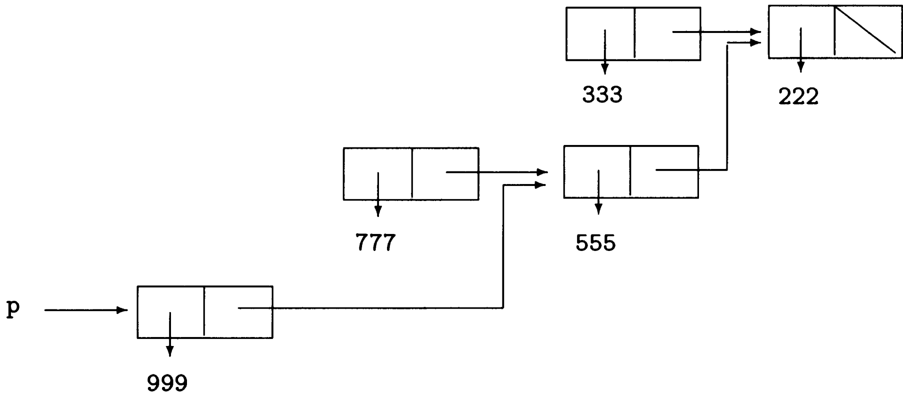
La représentation des listes en mémoire est basée sur la fonction `nouvelle--adresse-libre` qui à chaque appel doit trouver dans le tas deux cases mémoires consécutives pour y ranger un nouveau doublet. Il est clair que l'on ne peut pas utiliser longtemps la fonction `cons` sans saturer l'espace mémoire disponible. Heureusement, de nombreuses cases redeviennent inutilisées au cours de l'exécution d'un programme. D'où l'idée d'un système qui ferait *automatiquement* le ménage de temps en temps.

Remarque 3 Dans un langage comme Pascal, la fonction `new` pour créer un pointeur, joue un rôle analogue à `creer-doublet` mais c'est au programmeur de récupérer ensuite la place avec l'instruction `dispose`. Bien entendu, l'absence de structure de liste en Pascal rend la question moins aiguë.

Pour illustrer ce point, considérons l'évolution d'une pile représentée par une liste. On initialise la pile `p` avec la liste vide :

```
(let ((p  ()))
  (begin
    (set! p (cons 222 p))
    (set! p (cons 333 p))
    (set! p (cdr p))
    (set! p (cons 555 p))
    (set! p (cons 777 p))
    (set! p (cdr p))
    (set! p (cons 999 p))
    p))
```

Après exécution, on a la situation suivante pour les doublets :



On constate que deux doublets ne sont plus utilisés. La zone mémoire correspondante pourra donc être réutilisée. On va étudier des méthodes qui récupèrent ce type de doublets inutilisés pour les remettre à disposition.
Une méthode pour effectuer le recyclage des doublets s'appelle un ramasse miettes (ou *Garbage Collector* en anglais, en abrégé GC), ou encore un *glaneur de cellules* ce qui est plus poétique et possède les mêmes initiales que le terme anglais. Par la suite on dira simplement un *GC*.
Dès la naissance de Lisp on a dû résoudre ce problème, aussi existe-il de nombreux algorithmes de GC. On va décrire les deux méthodes les plus connues : la méthode marquer puis balayer (*mark and sweep*) et la méthode par recopie (*stop and copy*).

Méthode de GC par marquage et balayage

Le principe en est simple et se décompose en deux temps :
- dans un premier temps, on marque tous les doublets utilisés. Un doublet est marqué comme utilisé s'il est pointé par une des cases de la partie utile de la pile ou si, récursivement, il est pointé par un des doublets utilisés,
- dans un deuxième temps, on balaye toute la mémoire en chaînant dans une liste toutes les cases non marquées et en démarquant celles qui le sont. Cette liste sera la nouvelle liste de doublets disponibles.

Pour marquer un doublet on utilise un *bit de marquage*, pour notre implantation on ajoute une troisième composante à une valeur avec tag. Cette valeur sera le symbole `m` pour marqué et `f` sinon (pour *free*). Pour manipuler ces marques, on introduit les fonctions :

```
(define (marquer! adr  memoire)
  (vector-set! (lire-memoire adr memoire) 2 'm))

(define (demarquer! adr memoire)
  (vector-set! (lire-memoire adr memoire) 2 'f))

 (define (non-marque?  adr memoire)
   (not (eq? (vector-ref  (lire-memoire adr memoire) 2) 'm)))
```

```
(define (adr-doublet? adr memoire)
  (doublet? (lire-memoire adr memoire)))

(define (adr-car adr-doublet memoire)
  (son-adresse (lire-memoire adr-doublet memoire)))

(define (adr-cdr adr-doublet memoire)
  (- (adr-car adr-doublet memoire) 1))
```

La fonction de marquage part de l'adresse d'un doublet utilisé. Elle emploie une fonction locale qui marque les adresses atteintes directement ou par l'intermédiaire du car ou du cdr d'un doublet marqué.

```
(define (marquage adr-doublet memoire)
  (letrec ((marquage-aux
             (lambda (adr)
               (if (non-marque? adr  memoire)
                 (begin (marquer! adr memoire)
                        (if (adr-doublet? adr memoire)
                          (begin (marquage-aux (adr-car adr memoire))
                                 (marquage-aux (adr-cdr adr memoire)))))))))
    (marquage-aux adr-doublet)
    (marquage-aux (- adr-doublet 1))))
```

Figurons dans un tas imaginaire les doublets qui résultent de l'évaluation de l'expression `let` ci-dessus. On numérote les dix cases mémoire de 0 à 9. On indique par une flèche la case du doublet utilisé p :

	9	nb	222	f
	8	null	-1	f
	7	nb	333	f
	6	cons	9	f
	5	nb	555	f
	4	cons	9	f
	3	nb	777	f
	2	cons	5	f
p ⟶	1	nb	999	f
	0	cons	5	f

Avant le marquage

9	nb	222	m
8	null	-1	m
7	nb	333	f
6	cons	9	f
5	nb	555	m
4	cons	9	m
3	nb	777	f
2	cons	5	f
1	nb	999	m
0	cons	5	m

Après le marquage

Le balayage doit construire une liste des doublets non marqués et rendre l'adresse de cette liste notée adr-libre. On parcourt le tas en partant de l'adresse la plus basse. Quand on trouve une case non marquée, on place dans son cdr l'adresse libre précédente, on actualise adr-libre avec l'adresse du dernier doublet et l'on recommence tant que l'on ne dépasse pas l'adresse maximum. Quand on rencontre une case marquée, on la démarque (ainsi que la case de dessous) en vue du prochain marquage.

```
(define (balayage adr-basse adr-haute memoire)
  (letrec ((balayage-aux
             (lambda (adr-libre adr tag)
               (cond ((<= adr-haute adr) adr-libre)
                     ((non-marque? adr memoire)
                      (marquer! adr memoire)
                      (ecrire-memoire!  (- adr 1) memoire
                                        (vector  tag  adr-libre 'f))
                      (balayage-aux adr (+  2 adr) 'cons))
                     (else (demarquer! adr memoire)
                           (demarquer! (- adr 1) memoire)
                           (balayage-aux adr-libre (+ 2 adr) tag))))))
    (let ((adr-libre (balayage-aux -1 adr-basse 'null)))
      (if (= -1 adr-libre)
        (*erreur* "memoire pleine")
        adr-libre))))
```

Dans notre exemple, après le balayage on a adr-libre = 7 et on a récupéré deux doublets !

	9	nb	222	f
	8	null	-1	f
adr-libre ⟶	7	nb	333	f
	6	cons	3	f
	5	nb	555	f
	4	cons	9	f
	3	nb	777	f
	2	null	-1	f
p ⟶	1	nb	999	f
	0	cons	5	f

Après le balayage

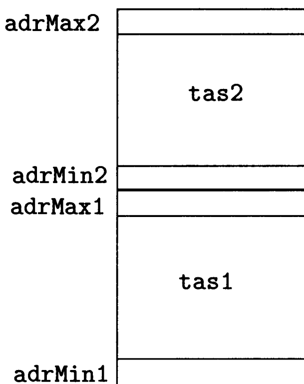
Pour coupler cette méthode de GC avec la machine Lisp, le lecteur devra procéder à quelques ajustements :

- pour marquer tous les doublets utilisés, on applique la méthode de marquage en partant de chaque doublet pointé par une case de la pile active ;

- on doit modifier la fonction nouvelle-adresse-libred ; elle consiste à rendre les adresses des éléments de la liste des doublets libres ou, s'il n'y en a plus, à relancer un GC.

Méthode de GC par recopiage

Une autre méthode de GC consiste à partager la zone mémoire consacrée au tas en deux parties égales : tas1 et tas2. On appelle adrMin1, adrMax1, adrMin2, adrMax2 les adresses minimales et maximales de ces deux zones de tas.



Partition du tas en deux zones

Chaque zone est utilisée à tour de rôle pour servir de tas. Quand une zone est pleine, on recopie les doublets *utilisés* dans l'autre. La difficulté est dans cette recopie car il faut conserver la structure des listes utilisées tout en actualisant les adresses. On recopie les doublets dans la nouvelle zone en les empilant, ce qui a l'avantage de compacter cette zone.

Détaillons le mécanisme de la recopie, dans le nouveau tas, d'un doublet donné par son adresse. Ce tas est géré comme une pile avec un pointeur SP sur le sommet courant.

Voici les actions à effectuer pour transférer un doublet dans le nouveau tas :
- on recopie un doublet dans la nouvelle zone en empilant son cdr puis son car, sa nouvelle adresse est donc SP+2 ,
- on laisse dans l'ancienne case de son car un avis pour indiquer sa nouvelle adresse aux doublets qui pointent sur lui. Pour cela, on place à son ancienne adresse la valeur de sa nouvelle adresse avec un nouveau tag fs pour indiquer que le doublet a déménagé et qu'il faut faire suivre à l'adresse indiquée,
- si son car est un doublet, on procède de même à partir de l'adresse de ce car et l'on remplace l'adresse de son car par la nouvelle adresse trouvée,
- même chose si son cdr est un doublet,
- quand un doublet est déjà dans la nouvelle zone, il n'y a rien à faire.

Pour implanter la copie de doublet, on appelle adrMin et adrMax l'adresse basse et l'adresse haute de la nouvelle zone de tas. On la gère comme une pile avec au

départ un pointeur de pile SP de valeur adrMin-1. La fonction copier-doublet empile dans la nouvelle zone le doublet d'adresse donnée (sauf s'il est déjà dans la nouvelle zone) et rend en résultat sa nouvelle adresse.

Le travail d'empilement et d'actualisation des adresses est réalisé par la fonction locale copier-aux.

```
(define (copier-doublet adr-doublet memoire adrMin adrMax)
  (let ((SP (- adrMin 1))
        (nlle-zone? (lambda (adr)(and (<= adrMin adr)(<= adr adrMax))))
        (faire-suivre? (lambda (vt)(eq? 'fs (son-tag vt)))))
    (letrec (
             (empiler!
               (lambda (v)
                 (set! SP (+ SP 1))(ecrire-memoire! SP memoire v)))

             (copier-aux
               (lambda (adr)
                 (let ((nlle-adr (+ 2 SP))
                       (vt1 (lire-memoire adr memoire))
                       (vt2 (lire-memoire (- adr 1) memoire)))
                   (cond
                     ((nlle-zone? adr) adr)
                     ((faire-suivre? vt1)(son-adresse vt1))
                     (else
                       (empiler! vt2)
                       (empiler! vt1)
                       (ecrire-memoire! adr  memoire
                                        (creer-valeur-tag  'fs nlle-adr))
                       (if (doublet? vt1)
                         (let ((nlle-adr-car
                                 (copier-aux (son-adresse vt1))))
                           (ecrire-memoire!
                             nlle-adr  memoire
                             (creer-valeur-tag  'cons  nlle-adr-car))))
                       (if (doublet? vt2)
                         (let ((nlle-adr-cdr
                                 (copier-aux (son-adresse vt2))))
                           (ecrire-memoire!
                             (- nlle-adr 1)  memoire
                             (creer-valeur-tag  'cons nlle-adr-cdr) )))
                       nlle-adr))))))
      (copier-aux adr-doublet))))
```

On reprend notre exemple pour illustrer cette méthode. On suppose que le tas1 actuellement utilisé est constitué des adresses de 0 à 9 et que le tas2 est constitué des adresses 10 à 19. On part du doublet p d'adresse 1 et l'on recopie dans le tas2. La nouvelle adresse du doublet p est 11.

	19		
	18		
	17		
	16		
	15		
	14		
	13		
	12		
	11		
	10		
	9	nb	222
	8	null	-1
	7	nb	333
	6	cons	9
	5	nb	555
	4	cons	9
	3	nb	777
	2	cons	5
p →	1	nb	999
	0	cons	5

	19		
	18		
	17		
	16		
tas2	15	nb	222
	14	null	-1
	13	nb	555
	12	cons	15
p →	11	nb	999
	10	cons	13
	9	fs	15
	8	null	-1
	7	nb	333
	6	cons	9
tas1	5	fs	13
	4	cons	9
	3	nb	777
	2	cons	5
	1	fs	11
	0	cons	5

`Après recopie en partant`

`de l'adresse 1`

Pour coupler ce GC avec notre machine Lisp, il faut changer la fonction `nouvelle-adresse-libre`. Quand il n'y a plus de place, elle doit lancer un GC et ensuite allouer les doublets dans la zone libre du nouveau tas (sauf s'il est déjà plein !). Pour procéder au GC, on doit permuter les rôles des tas1 et tas2 puis, pour chaque adresse de doublet situé dans la zone active de la pile d'exécution, on remplace la partie adresse par l'adresse donnée par la fonction `copier-doublet`.

Comparaison de ces deux méthodes

La méthode par recopie ne parcourt que la zone mémoire utilisée contrairement à l'opération de balayage qui doit parcourir toute la mémoire. De plus, la recopie des doublets a l'avantage de compacter la mémoire, ensuite on dispose d'une zone libre contiguë contrairement à la méthode de balayage qui construit une liste des doublets libres. Enfin, la recopie ne nécessite pas de tag de marquage.

En revanche, la méthode par recopie ne peut utiliser que la moitié de la zone mémoire attribuée au tas. Chaque méthode possède de nombreux raffinements que l'on trouvera dans la littérature.
Notons que le GC se fait au moment où l'on manque de mémoire, il faut donc faire en sorte que l'exécution de l'algorithme de GC ne consomme pas trop de mémoire. Tout ceci explique pourquoi la réalisation d'un GC est souvent assez complexe.

22.9 Machine avec lien statique

Allocation dans la pile/allocation dans le tas

Cet aperçu sur la représentation de listes en machine nous montre que l'utilisation des listes pour l'environnement se fait au détriment de la place mémoire disponible. Quand on a fini d'exécuter un appel de fonction, il y a dépilement automatique de la zone d'activation, mais l'environnement d'exécution qui a été crée est une liste située dans le tas et cette liste n'est pas automatiquement détruite (et elle ne doit pas l'être en général).

On peut se demander s'il serait possible d'*utiliser aussi la pile* pour faire le stockage de l'*environnement d'exécution*. La réponse est complexe car elle dépend des possibilités du langage. Avec Scheme, une stratégie d'allocation entièrement dans la pile est impossible. Le problème provient des valeurs fonctionnelles. Considérons le cas de l'expression suivante :

```
(let ((f 1))
  (begin (let ((a 15))
            (set! f  (lambda (x)(+ x a)))
         (f 7)))
```

ou plutôt sa forme équivalente sans `let` :

```
((lambda (f)
  (begin ((lambda (a)
             (set! f  (lambda (x)(+ x a)))
           15)
         (f 7)))
  1)
```

La valeur `15` de l'argument `a` est capturée dans la fermeture de f et est utilisée *après* avoir terminé l'appel de la lambda qui définissait la valeur de `a`. Une stratégie d'allocation purement en pile aurait perdu la valeur de `a` au moment de l'appel `(f 7)`. En effet, quand l'appel `((lambda (a)...) 15)` est terminé on dépile la zone utilisée pour stocker la valeur de l'argument `a` et cette zone est ensuite écrasée par l'appel `(f 7)`.

De façon plus générale, la durée de vie illimitée des liaisons en Scheme n'est pas compatible avec une allocation d'espace mémoire en pile. On peut essayer d'utiliser au maximum la pile en analysant plus finement la structure des expressions à compiler. C'est tout l'art de la compilation Scheme. Il y a un compromis à trouver entre complexité du compilateur et allocation mémoire efficace. Mais la

notion d'efficacité dépend aussi du type des programmes à compiler, selon que l'on fait ou non un usage massif des fermetures (voire des continuations), une méthode d'allocation peut être favorable ou non.

Mais, si l'on impose des restrictions sur le langage, on peut utiliser une stratégie d'allocation totalement basée sur la pile. C'est le cas de la plupart des langages non fonctionnels.

Un langage avec portée limitée

On revient à un langage *sans* les listes et on n'autorise que des expressions pour lesquelles on n'a pas besoin que la durée de vie d'une liaison dépasse la durée de l'exécution de l'expression qui l'a créée. On appelle PseudoScheme cette version bridée de MiniScheme.

On reprend la machine du §6 et on va la modifier pour allouer *uniquement dans la pile* l'ensemble des informations utiles pour effectuer un appel de fonction. Il s'agit donc de modifier notre notion de zone d'activation.

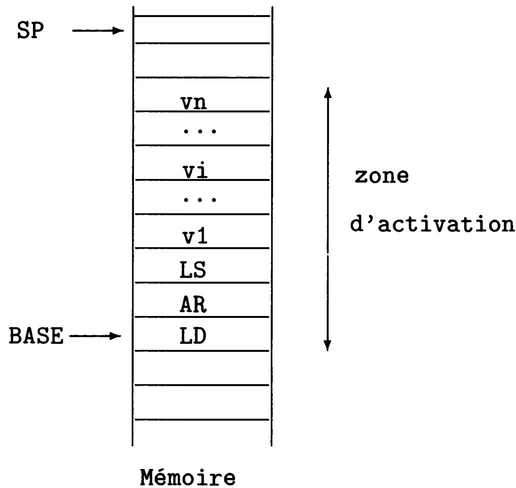
Au lieu de placer dans une liste les valeurs des arguments d'une fonction, on empile ces valeurs sur la zone d'activation de l'appel. Pour accéder à ces valeurs, il suffit de connaître l'adresse de la base du bloc car le ième argument sera à l'adresse `BASE + 3 + i`. Si l'on doit accéder à une variable d'adresse `#(p i)`, il faut commencer par «remonter» de `p` zones d'exécution. Pour remonter, il faut connaître l'adresse de la base du bloc englobant, elle s'appelle le *lien statique*.

On sauve le lien statique dans la fermeture avec l'adresse du code de la fonction. Au moment de l'appel, on place cette valeur dans la deuxième case située au-dessus de la base. On y accède par la fonction `lien-statique`

```
(define (lien-statique memoire BASE)
  (lire-memoire  (+ BASE 2) memoire))
```

La zone d'activation est donc constituée de trois adresses et des valeurs des arguments :

- le lien dynamique LD = adresse de la base de l'appelant,
- l'adresse du code de retour : AR,
- le lien statique LS = adresse de la base du bloc englobant,
- la valeur du premier argument, puis celle du second ...



Si une variable a pour adresse `#(p i)` à la compilation, le numéro de sa case mémoire est calculée par la fonction `adresse-memoire`. Elle remonte de `p` zones et rend la valeur située à l'adresse `(+ BASE i 3)` :

```
(define (adresse-memoire adr memoire BASE) ; ***
  (let ((profondeur (vector-ref adr 0))
        (indice     (vector-ref adr 1)))
    (letrec ((adr-aux (lambda (p BASE)
                        (if (zero? p)
                          (+ BASE indice 3)
                          (let ((BASE1 (lien-statique memoire BASE))
                            (adr-aux (- p 1) BASE1))))))
      (adr-aux profondeur BASE))))
```

Donc la valeur d'une variable d'adresse `#(p i)` est donnée par la fonction locale :

```
(lire-env (lambda (adr memoire BASE)  ;***
            (lire-memoire (adresse-memoire  adr memoire BASE) memoire)))
```

Si l'on doit modifier cette valeur (à l'occasion d'un `set!`), on utilise la fonction locale :

```
(Modifier-env! (lambda (adr  memoire BASE v) ; ***
                 (ecrire-memoire! (adresse-memoire  adr memoire BASE)
                                  memoire v)))
```

Comme on empile les arguments, on a besoin de connaître le *nombre* des arguments pour exécuter certaines instructions comme JSR et RTS.

```
((JSR)
 (let ((nb-argt (argt-instr instr)) ;***
       (fermeture (depiler!)))
```

```
   (let ((PC-codeCorps (vector-ref fermeture 0))
         (LS (vector-ref fermeture 1)))
     (incr! (- (+ 3 nb-argt)))  ;on se place sous la base du bloc
     (empiler! BASE)            ;on installe le lien dynamique
     (set!     BASE SP)         ;on actualise la BASE
     (empiler! PC)              ;on sauve l'adr de retour ***
     (empiler! LS)              ;on installe le lien statique
     (set! PC  PC-codeCorps)    ;début du code du corps ***
     (incr!  nb-argt)           ;on se place au sommet du bloc ***
     (exec-aux)))               ;on exécute le code de la fct

((RTS)
 (let ((nb-argt (argt-instr instr));***
       (v (depiler!)))            ;v = la valeur de l'appel
   (incr! (- (+ 1 nb-argt)))      ;on oublie les argts et le LS
   (set! PC    (depiler!))        ;on récupère le code de retour ***
   (set! BASE (depiler!))         ;on restaure l'ancienne BASE
   (empiler!  v)                  ;on replace la valeur v au sommet
   (exec-aux))  )))))             ;on exécute le code de la suite
```

On doit donc modifier la compilation des lambda et des appels de fonctions utilisateur pour ajouter le nombre d'arguments en paramètre de JRS et RTS.

```
(define (compile-lambda env-c lambda-exp code-suivant)
  (let ((Lvar  (cadr  lambda-exp))
        (corps (caddr lambda-exp))
        (etiq-suite (gen-etiquette)))
    (cons '(LOADFCT)
          (cons `(JUMP ,etiq-suite)
                (compile (cons Lvar env-c) corps
                         (cons `(RTS ,(length Lvar))  ;***
                               (cons `(LABEL ,etiq-suite)
                                     code-suivant))))))))

(define (compile-fct-utilisateur env-c Larg fct code-suivant)
  (let ((nb-argt (length Larg)))
    (cons '(BLOC)
          (compile-Larg env-c Larg
                        (compile env-c fct (cons `(JSR ,nb-argt) ;***
                                                 code-suivant))))))
```

Maintenant la compilation des arguments d'une fonction ne doit plus construire la liste de leurs valeurs mais simplement empiler chacune de ces valeurs :

```
(define (compile-Larg env-c Lexp code-suivant) ;***
  (letrec ((comp-Larg-aux
            (lambda (Lexp code-suivant)
              (if (null? Lexp)
                code-suivant
                (comp-Larg-aux (cdr Lexp)
                               (compile env-c (car Lexp)
```

```
                                        code-suivant)))))
    (comp-Larg-aux Lexp code-suivant)))
```

On n'a donc plus besoin de l'instruction machine `NIL`.
Le reste de la machine du §6 est inchangé.

On essaye cette machine

```
? (PseudoScheme '(let ((x 4)(y 5))
                   (let ((u x)(v y))
                     (+ u v)))  #f)
==> 9

? (PseudoScheme '(let ((f 1))
                   (let ((a 15))
                     (begin (set! f  (lambda (x)(+ x a)))
                            (f 7)))) #f)
==> 22

? (PseudoScheme '(let ((a 15))
                   (let ((f  (lambda (x)(+ x a))))
                     (f 7))) #t)
==> 22

? (PseudoScheme '(letrec ((fac (lambda (x)
                                 (if (= x 0)
                                   1
                                   (* x (fac (- x 1)))))))
                    (fac 3)) #f)
==> 6

? (PseudoScheme '(let ((f 1))
                   (begin (let ((a 15))
                            (set! f  (lambda (x)(+ x a))))
                          (f 7))) #t)
==> 14
```

Mais la dernière valeur est erronée! On doit trouver 22. L'argument 7 a écrasé la valeur 15 de `a`. C'est une expression non admissible pour PseudoScheme car la durée de vie de la liaison `(a . 15)` dépasse le temps d'exécution du bloc `(let ((a 15)) ...)`.

Il y aurait encore beaucoup d'aspects à préciser pour se rapprocher d'une machine réelle. Citons par exemple, l'utilisation des registres. Pour limiter les accès mémoires, les calculs ne sont pas faits dans la pile mais dans des registres. Comme il n'y a qu'un petit nombre de registres disponibles, cela pose le délicat problème de l'allocation des registres.
Par ailleurs, il y a des liens entre la compilation d'un programme et sa mise sous forme CPS. En effet, la mise sous forme CPS rapproche un programme d'une forme itérative et peut donc servir de point de départ pour l'écriture du code associé, voire les références.

22.10 De la lecture

La machine SECD est présentée dans de nombreux ouvrages, citons [Bur75, Hen80, Kog91]. On consultera [FH88] pour une preuve de la compilation de la machine SECD.
On trouvera des compilateurs pour (mini)Scheme dans [ASS89, Nor92, FWH92].
La compilation par transformation en style CPS est traitée dans [App92, FWH92].
Pour favoriser la portabilité, on peut aussi utiliser le langage C comme langage cible : la compilation de Scheme vers C est détaillée dans [Que94].
Le problème de l'allocation des registres est abordé dans [ASS89, ASU90].
Pour les principales méthodes de GC on renvoie par exemple à [Kog91].
Il y a beaucoup de livres sur les méthodes de compilation, mais en général ils ne concernent que le cas des langages impératifs, citons [Wir76, FL88, ASU90, Kam90].

Annexe A

Codes ASCII

déci	hex	abrév	clavier	nom anglais
0	00	NUL	^@	null
1	01	SOH	^A	start of header
2	02	STX	^B	start of text
3	03	ETX	^C	end of text
4	04	EOT	^D	end of tape
5	05	ENQ	^E	enquiry
6	06	ACK	^F	acknowledge
7	07	BEL	^G	bell
8	08	BS	^H	backspace
9	09	HT	^I	tab
10	0A	LF	^J	line feed
11	0B	VT	^K	vertical tab
12	0C	FF	^L	forme feed
13	0D	CR	^M	carriage return
14	0E	SO	^N	shift out
15	0F	CI	^O	shift in
16	10	DLE	^P	data link escape
17	11	DC1	^Q	device control1
18	12	DC2	^R	device control2
19	13	DC3	^S	device control3
20	14	DC4	^T	device control4
21	15	NAK	^U	negative acknoledge
22	16	SYN	^V	synchronous idle
23	17	ETB	^W	end transmission block
24	18	CAN	^X	cancel
25	19	EM	^Y	end medium
26	1A	SUB	^Z	substitute
27	1B	ESC	^[	escape
28	1C	FS	^\	field separator
29	1D	GS	^]	ground separator
30	1E	RS	^^	record separator

31	1F	US	^_	unit separator
32	20		SP	space
33	21		!	exclamation mark
34	22		"	quotation marks
35	23		#	sharp
36	24		$	dollar
37	25		%	percent
38	26		&	ampersand
39	27		'	quote
40	28		(	left parenthesis
41	29		)	right parenthesis
42	2A		*	asterisk ou star
43	2B		+	plus
44	2C		,	comma
45	2D		-	minus
46	2E		.	period
47	2F		/	slash
48	30		0	digit zero
...			...	digit ...
57	39		9	digit nine
58	3A		:	colon
59	3B		;	semicolon
60	3C		<	less than
61	3D		=	equal
62	3E		>	greater than
63	3F		?	question mark
64	40		@	commercial at
65	41		A	upper case A
...			...	upper case ...
90	5A		Z	upper case Z
91	5B		[	left bracket
92	5C		\	back slash
93	5D		]	right bracket
94	5E		^	circumflex accent
95	5F		_	underline
96	60		`	backquote
97	61		a	lower case a
...			...	lower case ...
122	7A		z	lower case z
123	7B		{	left curly brace
124	7C		|	vertical bar
125	7D		}	right curly brace
126	7E		~	tilde
127	7F	DEL		delete

Annexe B

Interprètes et compilateurs Scheme

Il existe de très nombreuses réalisations de Scheme, la plupart sont disponibles librement et d'excellente qualité.
Comme il s'agit d'un domaine en perpétuelle évolution, on conseille au lecteur de consulter sur Internet la FAQ Scheme (*Frequently Asked Questions*) qui décrit en particulier l'ensemble des versions de Scheme disponibles. Actuellement, le principal serveur est situé à l'université d'Indiana aux USA (voir référence [Col]), il collecte la plupart des informations sur Scheme.

Implantations disponibles sur Internet

A titre indicatif, voici par ordre alphabétique quelques versions de Scheme que l'on peut récupérer (début 1996) par ftp anonyme sur Internet.

BIGLOO : permet de compiler Scheme vers C, pour systèmes Unix.

ftp.inria.fr:/INRIA/Projects/icsla/Implementations

Elk : interprète et outils d'extensions, pour systèmes Unix.

http://www.informatik.uni-bremen.de/~net/elk

Gambit : interprète et compilateur, pour systèmes Unix et Macintosh.

ftp.iro.umontreal.ca:/pub/parallele/gambit

MIT Scheme : environnement complet de programmation pour Scheme, pour systèmes Unix et Windows.

swiss-ftp.ai.mit.edu:/archive/scheme-7.3

PCS/Geneva pour DOS.

cui.unige.ch:/PUBLIC/pcs/

SCM : interprète et librairies dont la SLIB, portable sur les principaux systèmes.

swiss-ftp.ai.mit.edu:/archive/scm

SIOD : interprète compact en C, portable sur les principaux systèmes.
ftp.cs.indiana.edu:/pub/scheme-repository/imp/siod-v3.0-shar

STk : interprète couplé avec la boîte à outils Tk, pour systèmes Unix
(une version pour Windows est en préparation).
kaolin.unice.fr:/pub

Implantations commerciales de Scheme

Citons aussi deux versions commerciales :

Pour systèmes Unix : Chez Scheme ftp.cs.indiana.edu:/pub/scheme/chezscheme

Pour Windows ou Macintosh : EdScheme 71020.1774@compuserve.com

Structure actuelle de l'archive Scheme sur Internet (voir [Col])

Cette archive est composée des répertoires suivants :

code Code Scheme pour la recherche, l'éducation, le plaisir, ...

doc

- **lit** Extraits de divers livres
- **misc** Documentations diverses (par exemple la FAQ)
- **prop** Propositions pour la prochaine version de la norme
- **pubs** Articles sur des sujets liés à Scheme
- **standards** Documents écrits par des auteurs de la norme RNRS

ext Extensions diverses

imp Implantations disponibles sur le réseau

instruct Matériaux complémentaires pour l'éducation

new Dernières contributions

promo Pour la promotion ou la démonstration de produits

utl Utilitaires pour Scheme

Il y a aussi un newsgroup très actif consacré à Scheme : `comp.lang.scheme`.

Articles de recherche

Pour ne pas entrer dans les délicats problèmes d'attribution de paternité, on s'est limité à ne citer dans la bibliographie que des livres ou bien des articles de synthèse. Cependant, si le lecteur souhaite s'informer sur l'activité de recherche dans le domaine de la programmation avec des langages fonctionnels comme Scheme, nous lui conseillons de consulter en premier la revue LISP AND SYMBOLIC COMPUTATION qui est publiée depuis 1988.

Bibliographie

[App92] A. W. Appel. *Compiling with continuations.* Cambridge University Press, 1992. Compilation par conversion en style CPS, étude du cas du langage SML.

[ASS89] H. Abelson, G. J. Sussman, et J. Sussman. *Structure et interprétation des programmes informatiques.* InterEditions, 1989. Souvent considéré comme l'un des meilleurs livres d'initiation à la programmation. Utilise le langage Scheme mais n'est pas un livre sur Scheme.

[ASU90] A. Aho, R. Sethi, et J. Ullman. *Compilateurs. Principes, techniques et outils.* InterEditions, 1990. Un classique sur la compilation.

[AU93] A. Aho et J. Ullman. *Concepts fondamentaux de l'informatique.* Dunod, 1993. Une introduction très pédagogique aux principaux outils formels utilisés en informatique.

[Bar84] H. P. Barendregt. *The Lambda Calculus, its Syntax et Semantics.* North Holland, 1984. Pour tout savoir sur le lambda calcul, ou presque.

[BM79] R. S. Boyer et J. S. Moore. *A Computational logic.* Academic Press, 1979. Description d'un système pour la preuve automatique de programmes Lisp.

[Boi88] P. Boizumault. *Prolog, l'implantation.* Masson, 1988. Description en Lisp d'un interprète et d'un compilateur pour Prolog.

[Bur75] W. H. Burge. *Recursive Programming Techniques.* Addison Wesley, 1975. Un livre précurseur, écrit en 1975 à partir des travaux de Landin, il est toujours actuel.

[CLR95] T. H. Cormen, C. E. Leiserson, et R. L. Rivest. *Introduction à l'algorithmique.* Dunod, 1995. Présentation détaillée d'un large éventail d'algorithmes.

[CM95] G. Cousineau et M. Mauny. *Approche fonctionnelle de la programmation.* Ediscience international, 1995. Une présentation agréable et rigoureuse de la programmation fonctionnelle, basée sur le langage Caml.

[Col] Collectif. *Scheme repository.* Vaste source d'information sur Scheme (implantations, programmes, articles, ...) disponible par ftp. Adresse actuelle : ftp.cs.indiana.edu répertoire /pub/scheme-repository. Image miroir en France à l'INRIA ftp.inria.fr répertoire /lang/Scheme.

[Com84] D. Comer. *Operating System Design. The Xinu Approach.* Prentice Hall, 1984. Une remarquable illustration du slogan : implanter pour mieux comprendre.

[CRa91] W. Clinger, J. Rees, et all. *Revised4 Report on the Algorithmic Language Scheme*, 1991. La norme du langage Scheme en une cinquantaine de pages.

[DJ90] N. Derchowitz et J-P. Jouanneaud. Rewrite system. Dans J. van Leeuwen, editor, *Handbook of theoretical Computer Science.* Elsevier Science Publishers, 1990. Une bonne synthèse sur la réécriture et les théories associées.

[DJL88] P. Deransart, M. Jourdan, et B. Lorho. *Attribute Grammars : Definitions, Systems et Bibliography.* Lectures Notes in Computer Sciences No 323. Springer Verlag, 1988. Un des rares ouvrages de synthèse sur ce sujet.

[Dyb87] R. K. Dybvig. *The SCHEME Programming Language.* Prentice Hall, 1987. Une présentation concise de Scheme avec des exemples intéressants.

[FF91] D. P. Friedman et M. Felleisen. *Le petit Lispien.* Masson, 1991. Initiation plaisante à la programmation en Scheme.

[FGS90] C. Froidevaux, M-C. Gaudel, et M. Soria. *Types de données et algorithmes.* Ediscience international, 1990. Une présentation claire et rigoureuse des algorithmes classiques.

[FH88] A. J. Field et P. G. Harrison. *Functional Programming.* Addison Wesley, 1988. Présentation très lucide des principaux concepts de la programmation fonctionnelle avec des exemples basés sur le langage Hope.

[FL88] C. N. Fisher et R. J. LeBlanc. *Crafting a Compiler.* Benjamin Cummings, 1988. Présentation détaillée des principes de la compilation couplée avec la réalisation d'un compilateur pour un langage du type mini Ada.

[FvDFH95] J. D. Foley, A. van Dam, S. K. Feiner, et J. F. Hugues. *Introduction à l'infographie.* Addison Wesley, 1995. Un classique dans le domaine de l'infographie.

[FWH92] D. P. Friedman, M. Wand, et C. T. Haynes. *Essentials of Programming Languages.* MIT Press, 1992. Une étude approfondie de diverses catégories de langages par l'écriture d'interprètes en Scheme.

[Gal] E. Gallésio. *STk.* Une implantion de Scheme couplée à la *toolbox TK* de J. Ousterhout. Propose aussi une extension objet dans le style de CLOS. Disponible sur Internet (voir la référence [Col]).

[Gal86] J. H. Gallier. *Logic for Computer Science.* Harper et Row, 1986. Une présentation de la logique sous l'angle du calcul des séquents.

[GKPvC85] F. Giannesini, H. Kanoui, R. Pasero, et M. van Caneghem. *Prolog.* InterEditions, 1985. La programmation avec Prolog II présentée par des chercheurs de l'équipe de A. Colmeraeur.

[Gol94] D. E. Golberg. *Algorithmes génétiques - Exploration, optimisation et apprentissage automatique.* Addison Wesley, 1994. Une présentation très agréable à lire de cette nouvelle approche de l'optimisation.

[Gor79] M.J.C. Gordon. *The Denotational Description of Programming Languages An Introduction.* Springer Verlag, 1979. Un exposé très abordable de la sémantique dénotationnelle.

[Gra94] P. Graham. *On Lisp : Advanced Techniques for Common Lisp.* Prentice Hall, 1994. Excellent rapport volume/quantité d'information. Plusieurs chapitres remarquables sur l'écriture des macros.

[Hen80] P. Henderson. *Functional Programming Application et Implementation.* Prentice Hall, 1980. Toujours aussi enrichissant à lire, comme le Burge : ne vieillit pas.

[Hof85] D. Hofstaster. *Gödel, Escher, Bach.* InterEditions, 1985. Variations brillantes sur des thèmes de logique où l'on peut voir la récursivité comme un dénominateur commun de ces trois créateurs.

[HP94] J. Hennessy et D. Pattersson. *Organisation et conception des ordinateurs.* Dunod, 1994. Présentation très complète de l'architecture des ordinateurs et particulièrement des architectures RISC.

[HS78] E. Horowitz et S. Sahni. *Fundamentals of Computer Algorithms.* Computer Science Press, 1978. Riche source d'information et nombreux exercices.

[HW94] B. Harvey et M. Wright. *Simply Scheme. Introducing Computer Science.* MIT Press, 1994. Une introduction à Scheme qui met l'accent sur la programmation d'ordre supérieur et sur le traitement des chaînes.

[Jaf] A. Jaffer. *Scm.* Une implantion de Scheme complétée par une importante librairie d'utilitaires : la SLIB. Il y a aussi un système de calcul formel Jacal. Disponible sur Internet (voir la référence [Col]).

[JGS93] N. D. Jones, C. K. Gomard, et P. Sestoft. *Partial Evaluation et Automatic Program Generation.* Prentice Hall, 1993. Où l'on découvre un lien entre l'interprétation et la compilation.

[Kam90] S. N. Kamin. *Programming Languages An Interpreted-Based Approach.* Addison Wesley, 1990. Une étude de divers types de langages par l'écriture d'interprètes en Pascal.

[KdRB92] G. Kicsales, J. des Rivières, et D. G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1992. Pour comprendre le fonctionnement d'un protocole méta-objet à la CLOS.

[Kle62] S. C. Kleene. *Introduction to Metamathematics.* North Holland, 1962. Reste, malgré son âge, l'une des meilleures références en logique.

[Knu73] D. E. Knuth. *The Art of Computer Programming* (trois tomes publiés). Addison Wesley, 1973. On ne présente plus ce traité.

[Knu84] D. E. Knuth. *The TEX Book.* Addison Wesley, 1984. La description du célèbre système de formatage de texte.

[Kog91] P. M. Kogge. *The Architecture of Symbolic Computers.* McGraw-Hill, 1991. Une mine de renseignements sur l'implantation des langages de la famille Lisp et Prolog.

[Kri90] J-L. Krivine. *Lambda-calcul, types et modèles.* Masson, 1990. Une présentation rigoureuse des développements récents en lambda calcul.

[Lal90] R. Lalement. *Logique, réduction, résolution.* Masson, 1990. Un excellent livre sur les aspects logiques de l'informatique avec des exemples en ML ou en Prolog.

[Lau85] J-L. Laurière. *Intelligence artificielle, résolution de problèmes par l'homme et la machine.* Eyrolles, 1985. Exposé très documenté d'un domaine où les espérances l'emportent parfois sur la raison.

[MAE+62] J. McCarthy, W. Abraham, D. Edwards, T. P. Hart, et M. Levin. *Lisp 1.5 Programmers's Manual.* MIT Press, 1962. Le livre bleu, un document historique.

[Mar89] M. Margenstern. *Langage Pascal et logique du premier ordre* (deux tomes). Masson, 1989. Une présentation très approfondie d'outils logiques en relation avec le langage Pascal. Il est dommage que la lecture de ce livre soit souvent gênée par une typographie trop compacte.

[ML95] V. M. Manis et J. J. Little. *Schematics of Computation.* Prentice Hall, 1995. Une introduction extrêmement pédagogique à la programmation en Scheme. Comporte beaucoup d'exemples intéressants et des notes culturelles.

[MNC+89] G. Masini, A. Napoli, D. Colnet, D. Léonard, et K. Trombe. *Les langages à objets.* InterEditions, 1989. Une présentation synthétique des différents styles de programmation par objets. Intéressante implantation en Lisp d'un noyau du système ObjLisp de P. Cointe.

[MW93] Z. Manna et R. Waldinger. *The Deductive Foundation of Computer Programming.* Addison Wesley, 1993. Une présentation axiomatique des principales structures de données.

[Nil92] N. J. Nilsson. *Principles of Artificial Intelligence.* Springer Verlag, 1992. Présentation agréable du sujet.

[NN93] H. R. Nielson et F. Nielson. *Semantics with Applications : A Formal Introduction.* John Wiley, 1993. Un des rares ouvrages exposant la sémantique naturelle.

[Nor92] P. Norvig. *Paradigm of Artificial Intelligence Programming: Case Studies in Common Lisp.* Morgan Kaufmann, 1992. Ouvrage très riche. Comporte en particulier de beaux chapitres sur la programmation des jeux et le traitement des langages naturels.

[Pau91] L. Paulson. *ML For the Working Programmer.* Cambridge University Press, 1991. Une présentation détaillée du dialecte SML de ML. Avec des applications à la logique.

[PJS92] H. O. Peitgen, H. Jürgens, et D. Saupe. *Chaos et Fractals New Frontiers of Science.* Springer Verlag, 1992. Une mine d'information sur ces questions. Une réussite de l'édition scientifique. Les quelques petits programmes en BASIC ne sont pas au niveau du reste de l'ouvrage.

[Plo81] G. D. Plotkin. *A structural approach to operational semantics.* Technical report, Computer Science Dpt Aarhus Univ Denmark, 1981. Pose les principes de la sémantique naturelle. Mériterait d'avoir une diffusion moins confidentielle.

[Que90] C. Queinnec. *Le filtrage: une application de (et pour) Lisp.* InterEditions, 1990. Une suite de variations sur le filtrage de listes en Lisp.

[Que94] C. Queinnec. *Les langages Lisp.* InterEditions, 1994. Une présentation très approfondie de l'interprétation et de la compilation des dialectes Lisp. Tout le code est en Scheme. C'est une suite naturelle de notre livre.

[SB80] J. Stoer et R. Burlisch. *Introduction to Numerical Analysis.* Springer Verlag, 1980. Livre de référence sur les aspects numériques. Donne une description des algorithmes en pseudo Pascal.

[Sch86] D. A. Schmidt. *Denotational Semantics, a Methodology for language Development.* Allyn et Bacon, 1986. Présentation très lisible de la sémantique dénotationnelle, illustrée à l'aide d'une variété de petits langages.

[SF90] G. Springer et D. P. Friedman. *Scheme et the Art of Programming.* MIT Press, 1990. Une présentation assez complète de la programmation avec Scheme. Dan Friedman est l'un des grands évangélistes de Scheme.

[SG93] G. L. Steele et R. P. Gabriel. Evolution of lisp. *ACM Sigplan Notices*, 23(3) : 1–80, 1993. Une présentation de l'histoire de Lisp par des auteurs qui l'ont vécue.

[SJ93] E. Saint-James. *La programmation applicative.* Hermés, 1993. Une présentation originale et profonde du sujet. Malheureusement, l'utilisation d'un dialecte non standard de Lisp complique la lecture des exemples.

[SS93] L. Sterling et E. Shapiro. *L'art de Prolog.* Masson, 1993. Une présentation très riche du langage Prolog.

[Ste90] G. L. Steele. *Common LISP The Language Second edition.* Digital Press 1990, 1990. La bible de Common Lisp.

[Sto77] J. E. Stoy. *Denotational Semantics : The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977. Un classique sur les travaux de l'école anglaise de sémantique des années 70.

[Sza70] M. E. Szabo. *The Collected Papers of Gerhard Gentzen.* Elsevier North Holland, 1970. Une édition des travaux fondamentaux de Gentzen en théorie de la preuve. Cela reste très lisible.

[Tan77] A. Tanenbaum. *Architecture de l'ordinateur.* InterEditions, 1977. Présentation très pédagogique du fonctionnement et de la structure des ordinateurs.

[Voy87] R. Voyer. *Moteurs de systèmes experts.* Eyrolles, 1987. Un des rares livres sur les systèmes experts où l'on explique le code Lisp de plusieurs systèmes.

[WH90] S. A. Ward et R. H. Halstead. *Computation Structures.* MIT Press, 1990. Une présentation très détaillée des composants d'un microprocesseur.

[Wir76] N. Wirth. *Algorithms + Data Structures = Programs.* Prentice Hall, 1976. Un grand classique pour les programmeurs Pascal, mais mérite aussi d'être lu par les autres. La première édition se termine par un chapitre remarquable sur la compilation.

[WL93] P. Weis et X. Leroy. *Le langage Caml.* InterEditions, 1993. Une présentation riche et lucide du dialecte Caml de ML.

[Wol88] S. Wolfram. *Mathematica, A System for Doing Mathematics by Computer.* Addison Wesley, 1988. Description d'un système de calcul formel très utilisé dans l'enseignement. Dispose d'un puissant langage fonctionnel basé sur la réécriture conditionnelle.

Index

abort, 255
erreur, 256
#f, 11
#t, 11

A
a-listes, 48
accesseur, 51, 186
Ackerman, 92
acons, 50
adressage dispersé, 202
adresse, 53, 397
adresse de retour, 711
alpha conversion, 594, 614
analyse lexicale, 415
and, 14, 81, 283
antilogie, 481
appel par nom, 657
appel par valeur, 612, 657
appel par variable, 667
appel terminal, 88, 698
append, 30, 36
append-map, 80
append2!, 110
apply, 79
approximations successives, 73
arête, 225
arbre, 204
arbre binaire, 207
arobasque, 58
assemblage, 709
assoc, 49
assq, 48
assv, 49
atome, 31
attribut, 433
attribut hérité, 453
attribut synthétisé, 453
automate cellulaire, 393
automate fini, 384, 414
axiome, 496

B
bêta réduction, 594
base de connaissance, 538
BDD, 493
begin, 284
Bezout, 163, 241
bind, 288, 298
bloc de liaison, 126
booléen, 11
booleen?, 12
boucle d'interaction, 3
bus, 400

C
c...r, 30
calcul des séquents, 503, 558
calcul formel, 58, 533
calculable, 402
call-with-current-continuation, 249
call-with-input-file, 306
call-with-output-file, 306
call/cc, 249, 639
car, 29
caractère, 113
case, 32
cdr, 29
chaîne, 6, 113
champ d'un quantificateur, 565
char->integer, 113
chaînage arrière, 538
citation, 25
classe, 332
clause, 485
clause de Horn, 537
close, 566

close-input-port, 306
close-output-port, 306
code de retour, 701
code, 680
combinateur, 594
commentaire, 5
compiler, 694
complétude, 502
compte est bon, 37
compteur, 139
compteur ordinal, 397, 710
cond, 13, 297
connecteur logique, 476
cons, 29
constructeur, 51, 181
continuation, 235, 248
contrainte, 144
coroutine, 257
correction, 41
creer-genVar, 141
curryfication, 72

D
décidable, 403
déduction naturelle, 497, 568, 621
déduction sémantique, 482
délégation, 335
démonstration, 495
define, 7, 9, 66
define-syntax, 291
defmethod, 346, 361
delay, 311
diagramme de boîtes, 55
display, 22
display-all , 84
display-alln, 84
do, 105
dolist, 276
domaine, 522, 610
dotimes, 276
doublet, 46, 713
durée de vie, 126

E
echanger, 117
échappement, 242, 251
else, 33
ensemble, 86
environnement, 8, 128, 663
environnement d'exécution, 684
environnement de compilation, 684
eof-object?, 306
Eratosthène, 316
êta réduction, 596
eval, 46
every, 81
évaluation, 45
évaluation paresseuse, 662
évaluation partielle, 492
expression, 125
expression littérale, 28
expression régulière, 406

F
f-car, 314
f-cdr, 314
f-cons, 313
factorielle, 19, 236
fermeture, 69, 131
Fibonacci, 20, 92, 146, 242
file, 195
filtrage, 289, 514
filtre, 72, 525
fluid-let, 287
for-each, 79
force, 311
format, 116
formatage, 439
forme clausale, 485, 533
forme spéciale, 45
formule atomique, 563
formule du premier ordre, 563
formule propositionnelle, 477
fractal, 340
funarg, 603, 635, 640
fusion, 310

G
générateur, 247, 254
GC, 724
gensym, 143, 281
glaçon, 656
glaneur de cellules, 726
grammaire attribuée, 433

grammaire formelle, 419
graphe, 228

H
Hanoi, 23
héritage, 333, 346
heuristique, 98

I
identificateurs, 6
if, 12
incr!, 273, 293
indéfinie, 46
indentation, 5, 323
inférence de type, 616
infixe, 193
input-port?, 306
instance, 346
instruction, 664
integer->char, 113
integer?, 12
interner, 724
intersection, 87
iota, 35
irréductible, 530

J
jeu de la vie, 393
jugement, 495
jump, 707

L
lambda calcul, 591
lambda expression, 65, 83
lambda substitution, 595
lambda terme, 591
lambda terme typé, 614
langage , 386
langage régulier, 387
last-pair, 110
length, 31
let, 15, 67, 285
let*, 16, 285
letrec, 132, 286
lexicale, 124
liaison, 8
lien dynamique, 701
lien statique, 733
Lindenmayer, 343
list, 30
list->vector, 118
list-it, 82
list?, 31
liste, 27
liste circulaire, 111
liste plate, 31
liste vide, 28
littéral, 485
logique égalitaire, 586
logique intuitionniste, 502
Logo, 337

M
machine à pile, 397, 680
machine de Mealy , 385
machine de Moore, 385
machine de Turing, 390
macro, 270
macrocaractère, 271
macro-expansion, 270
make-class, 346
make-instance, 360
make-vector, 118
map, 77
map-vector, 119
member, 32
memq, 32
memv, 32
menu, 22, 252
mémo fonction, 143
mémoire, 53, 128, 661
méthode, 347
microprocesseur, 400
modificateur, 185
mot, 386
mot-clé, 8, 27
moteur d'inférence, 538
moteur de propagation, 151
mutable, 186
mutation, 136

N
newline, 22
non terminal, 419
normalisable, 600

not, 12
null?, 31
number?, 12

O
objet, 346
occurrence, 565
occurrence liée, 565
occurrence libre, 565
open-input-file, 305
open-output-file, 305
or, 14, 81, 283
ordre bien fondé, 531
ou exclusif, 488
output-port?, 306

P
pair?, 31
Peano, 527, 554, 588
peek-char, 307
permuter, 280
permuter!, 294
phénomène de capture, 567
pile, 187
pile mutable, 274
point fixe, 608
polynôme, 58, 119
pop!, 278
portée, 68, 124
portée dynamique, 68, 126, 652
portée statique, 68, 126
postfixe, 193
précondition, 257
prédicat, 563
préfixe, 386
procedure?, 12
producteur/consommateur, 260
profondeur, 31
programmation fonctionnelle, 65
programmation impérative, 101
programmation logique, 555
programmation modulaire, 134
programmation par objets, 331
programmation récursive, 34
Prolog, 553
promesse, 311
prototype, 334
push!, 274

Q
quantificateur existentiel, 562
quantificateur universel, 562
quasiquote, 57
quicksort, 221
quote, 26

R
RAM, 397
random, 142
read, 21, 306
read-char, 307
recherche, 220
récursive, 18
redex, 594
réduction, 599
réduction normale, 600
réduction par valeur, 601
réécriture, 291, 530
réification, 250
remove, 36
représentation interne, 51
retour-arrière, 94
reunion, 86
reverse, 35

S
S-expression, 7
satisfaisable, 481
SchemO, 346
self, 337
sémantique dénotationnelle, 669
sémantique dynamique, 647
sémantique naturelle, 646
sémantique statique, 645
send, 347
send-next, 348
séparateur, 115
séquent, 573
session, 4
set!, 101
set-car!, 107
set-cdr!, 107
signature, 186
some, 81
sous-liste, 31

sous-mot, 386
spécification, 43, 626
string-length, 113
string-ref, 113
string-set!, 113
string?, 12
sublis, 49
substitution, 522, 567
substitution admissible, 569
suffixe, 386
symbole, 27
syntaxe abstraite, 431
système expert, 538
système formel, 495

T
table, 199
table de vérité, 479
tag, 714
taquin, 98
tautologie, 481
terme, 518
terminaison, 42
terminal, 419
test d'occurrence, 547
théorème, 497
top level, 3
trace, 18, 300
tri, 221
type abstrait, 186

U
unificateur, 545
unless, 275
untrace, 19, 301

V
valeurs multiples, 245
valuation, 480
variable, 8, 27, 53, 125
variable libre, 68, 566
vector, 118
vector->list, 118
vector-length, 118
vector-ref, 117
vector-set!, 117
visibilité, 125
von Koch, 341

W
when, 275
while, 104, 282
write, 22, 308
write-char, 308

Z
zero?, 12
zone d'activation, 701

PROGRAMMER AVEC SCHEME

Le langage de manipulations symboliques Scheme fait partie de la famille Lisp. Il a été choisi depuis 1980 par les enseignants du MIT et le nombre de ses adeptes n'a fait qu'augmenter. Il fait maintenant son entrée dans les premiers et deuxièmes cycles des universités françaises. Ce simple fait en dit long sur les qualités de Scheme comme support à l'enseignement de la programmation et de ses fondements

Cet ouvrage de référence est conçu pour être utilisable avec profit aussi bien par un lecteur débutant que par un lecteur plus expérimenté. La richesse de son contenu permet de le recommander tout au long d'un cursus en informatique, du DEUG au DEA. C'est non seulement un livre sur la programmation fonctionnelle avec Scheme, mais plus généralement un excellent ouvrage d'accompagnement pour des cours sur l'intelligence artificielle, la logique, la sémantique, la compilation,... Les exemples utilisés ne sont pas de simples exemples d'école. Ils sont pertinents, significatifs et parfois même amusants.

Le livre est structuré en deux parties :

- La première partie introduit progressivement le lecteur aux joies de la programmation avec Scheme (ou Lisp). Y sont expliqués les différents styles de programmation : fonctionnel, symbolique, par continuations, par objets, impératif, par macros, par flots,... Les méthodes récursives y jouent un rôle central, aussi l'auteur s'est efforcé d'en dissiper les mystères en les présentant sous une grande variété de points de vue. Chaque chapitre comporte des compléments sur des sujets variés : fractals, jeu donjon et dragon, contraintes et CAO, extension objet de Scheme, simulation et coroutines, visualisation en 3D, calcul formel, vie artificielle,... Ils ont aussi pour but d'exercer le lecteur à l'écriture de programmes de taille plus importante. De nombreux exemples et exercices permettent au lecteur de progresser à son rythme.
- La seconde partie explique la théorie et l'implantation des concepts de base en programmation : machines et automates finis, analyse lexicale et syntaxique, formatage de texte ou de formules, systèmes experts et Prolog, systèmes déductifs, lambda calcul, sémantique, interprétation, compilation,... La possibilité de programmer puis d'expérimenter une théorie est l'une des meilleures façons de l'assimiler, d'où le slogan de l'auteur : « Implanter pour mieux comprendre. »

L'AUTEUR

JACQUES CHAZARAIN est professeur à l'université de Nice Sophia Antipolis. Il y enseigne la programmation fonctionnelle (Scheme, Lisp, ML) depuis une douzaine d'années. Il est membre du laboratoire CNRS I3S à Sophia Antipolis où il est responsable de l'équipe Janus qui développe en particulier des environnements de programmation autour du langage Scheme.

INTERNATIONAL
THOMSON
PUBLISHING
FRANCE