# Scheme : un langage applicatif pour l'enseignement de l'informatique en milieu aride

Laurent Arditi[1] & Stéphane Ducasse[2]

*1: Stanford University,*
*Computer System Laboratory,*
*Gates Building 3A,*
*Stanford, CA 94305*
arditi@sprout.stanford.edu
*2: Université de Bern, IAM-SCG,*
*Neubruckstrasse 10,*
*CH-3012 Bern*
ducasse@iam.unibe.ch

Dans cet article, nous présentons une expérience d'enseignement de l'informatique «en milieu aride» : volume horaire faible, faible coefficient, étudiants peu motivés. Dans ce contexte défavorable, nous avons choisi de privilégier un concept, la récursivité dans le cadre de la programmation fonctionnelle, plutôt qu'un langage. Ce choix nous a conduit à considérer le langage Scheme pour sa simplicité. Nous décrivons ici notre cours, les problèmes que nous avons rencontrés et les solutions que nous avons choisies.

## 1. Introduction

Plus que le choix d'un langage informatique, se pose le problème des concepts à enseigner. Le choix de ces concepts dépend principalement du contexte dans lequel le cours intervient. Ceci est particulièrement vrai lorsque le volume horaire est faible et les étudiants[1] peu motivés du fait de coefficients faibles. Dans un tel «milieu aride», nous avons choisi d'enseigner la programmation récursive et fonctionnelle. Cet objectif et la rigueur du volume horaire nous ont amené à considérer le langage Scheme.

[1] Dans la suite, le terme «étudiants» est général : il désigne aussi bien les *étudiantes* que les *étudiants*.

## 1.1. Contexte des enseignements

Ce cours est destiné à des étudiants de DEUG (deux premières années d'université). Les filières concernées sont scientifiques : dans la filière nommée «SM PE», la prédominance est donnée aux sciences physiques et aux mathématiques ; la filière «MASS» est centrée sur les mathématiques et les mathématiques appliquées aux sciences sociales. Toutes filières cumulées le nombre total d'étudiants est d'environ 300 étudiants.

Ce cours est réalisé sur un semestre, c'est à dire 12 semaines. Le volume horaire hebdomadaire est de 2 heures (une heure de cours et une heure de travaux pratiques) ou 1 heure 30 (de cours/travaux pratiques). Il faut noter que le volume horaire total est particulièrement faible : 18 ou 24 heures, alors qu'un cours d'informatique en DEUG est généralement réalisé sur une cinquantaine d'heures. Notre cours intervient après un cours général sur Pascal incluant parfois des notions sur les pointeurs et les fichiers.

D'autre part, l'importance de l'informatique, et plus particulièrement du cours de Scheme, est très faible comparée aux autres matières : la note de Scheme ne représente qu'un dixième de la moyenne générale de l'unité de valeur (UV). Et les étudiants suivent au moins trois UVs.

Pour ces deux raisons, nous pouvons dire que l'audience de ce cours n'est *a priori* que très peu motivée.

## 1.2. Objectifs

Trop souvent, les enseignements réalisés dans un contexte défavorable sont perçus comme peu enthousiasmants. Nous pensons bien au contraire qu'il est motivant de trouver les moyens de faire un cours ayant une certaine portée malgré de telles contraintes. Nous décrivons cette expérience dans cet article.

Notre expérience de l'enseignement de l'informatique avec Pascal [13] nous a montré la grande confusion que font les étudiants entre les procédures, les fonctions, les effets de bords,... Ceci est dû à la trop grande liberté procurée par ce langage et son manque de cohérence. On peut ainsi citer la possibilité de définir des fonctions qui ne rendent pas de résultat, des fonctions dont certains paramètres sont modifiés, de modifier des variables globales depuis une fonction, de déclarer des variables de types const pour les initialiser avec des valeurs par défaut ... Les étudiants ne savent généralement pas quand utiliser une procédure ou une fonction, ils confondent le fait de rendre un résultat et de réaliser un affichage. De plus, l'utilisation de structures de données flexibles (comme les listes) demandent une bonne compréhension des pointeurs et de l'allocation de la mémoire. Ceci parasite et éloigne

l'enseignement de son but initial : la perception des concepts généraux de la programmation.

Quelques tentatives ont été menées pour définir des sous-ensembles de Pascal plus «propres» [12, 13] : suppression des variables globales, limitation des paramètres de sortie,... Mais cette manière de faire n'est pas complètement satisfaisante. On s'aperçoit rapidement que la programmation applicative et l'utilisation de la récursivité à la place de l'itération sont des alternatives intéressantes. Nous avons donc choisi d'enseigner la programmation récursive dans un langage applicatif.

Nous attendons de ce cours que les étudiants soient à même de définir des fonctions récursives correctes même si elles ne sont pas triviales, qu'ils comprennent et appliquent des concepts généraux de la programmation : abstraction des données et modularité, optimisation de la complexité, arbres binaires,...

**Plan de l'article.** Nous présentons dans la partie suivante les particularité du langage Scheme et les raisons qui nous ont poussés à l'utiliser. La partie 3 donne le plan de notre cours. Nous discutons ensuite de nos choix quant au programme traité dans la partie 4. La partie 5 expose les résultats obtenus. Nous concluons ensuite en donnant des perspectives possibles.

# 2. Scheme, un langage adapté

Le choix d'un langage informatique adapté à l'enseignement de l'informatique a toujours été une question difficile. De plus en plus de langages applicatifs comme Lisp, ML, Scheme et Gofer [15] sont utilisés pour l'enseignement en premier cycle. Cette évolution a débuté aux États-Unis mais concerne aussi la France depuis quelques années [4, 5, 12]. Ainsi des ouvrages en français sont maintenant disponibles [1, 10, 8, 14, 6].

## 2.1. Pourquoi Scheme ? : simplicité et uniformité

Lorsque nous avons choisi le langage que nous allions utiliser, nous nous sommes posés la contrainte suivante : ne pas modifier ou particulariser un langage existant juste pour les besoins d'un enseignement afin que les étudiants travaillent avec un véritable langage ayant un certain nombre d'ouvrages accessibles à la bibliothèque universitaire ou dans le commerce [2].

[2] Les étudiants ont souvent très peur d'apprendre un langage qui n'est utilisé que dans le cadre d'un cours. Ils posent souvent des questions quant à l'utilisation du

Bien que des langages comme ML ou Gofer offrent un typage fort (voir en 4.7), nous les avons écartés car ils possèdent une syntaxe complexe et non-uniforme et nécessitent au minimum l'introduction du typage.

Scheme [7] nous ait apparu comme le langage satisfaisant les contraintes très fortes de temps que nous avons. La principale raison réside dans sa simplicité, sa cohérence et la disponibilité de nombreux interprètes. La syntaxe de Scheme, préfixée et parenthésée, est très simple et uniforme. Des conventions judicieuses comme l'utilisation du «`?`» pour nommer les prédicats et la quasi absence de typage permettent d'aborder l'ensemble des constructions très rapidement et clairement. Cette uniformité est accentuée par le fait que les fonctions sont représentées et manipulées comme les autres types de données.

### 2.2. Environnement de programmation

L'environnement de programmation est un élément de grande importance dans la réceptivité des étudiants vis-à-vis d'un enseignement [4]. Les interprètes de Scheme permettent une approche plus souple de la programmation. En effet, les fonctions peuvent être testées immédiatement sans devoir écrire un programme principal comme en Pascal (lecture des données par `read`, affichage des résultats par `write`).

De nos jours, de bons environnements existent comme EdScheme [17], MacGambit [9] ou STk [11]. EdScheme est le plus abouti pour l'enseignement. Il propose l'indentation automatique, la différenciation visuelle des éléments du langage (fonctions définies par l'utilisateur, fonctions prédéfinies, mot-clefs,...), le repérage des parenthèses correspondantes et le concept de «boîtes» encadrant les expressions. Ces points sont des aides non négligeables pour l'étudiant.

## 3. Organisation du cours

Nous présentons ici le contenu et l'organisation du cours de Scheme sans faire de remarque particulière quant à son accueil par les étudiants.

### 3.1. Plan du cours

De part la forte contrainte de temps imposée, nous avons choisi de ne présenter qu'un sous-ensemble restreint du langage Scheme. Mais nous avons par contre essayé de présenter en détail les

langage à l'extérieur de l'université. Le fait qu'ils trouvent des ouvrages relatifs au langage les rassurent.

concepts fondamentaux de la programmation, plus précisément, de la programmation fonctionnelle et de la récursivité. L'enseignement est composé de 5 cours logiques[3] que nous exposons ici.

**1. Présentation du langage.** Nous présentons tout d'abord dans cette partie la syntaxe de Scheme et les différents types de données.

Nous insistons sur les principes de l'évaluation et de la définition des variables. Pour ce faire, nous mentionnons la notion d'environnement comme étant une table à deux entrées dans laquelle il est possible de rajouter des éléments, par `define`, mais dans laquelle il est impossible de modifier des éléments déjà existants. La définition des fonctions est introduite en faisant l'analogie avec les mathématiques.

Nous donnons ensuite les fonctions prédéfinies les plus utilisées (`+`, `equal?`, `symbol?`,...) et les formes spéciales (`if`, `cond`, `let`).

La dernière partie de ce cours présente les listes, leurs constructeurs (`nil` et `cons`), leurs accesseurs (`car`, `cdr`) et les fonctions relatives (`null?`, `append`,...). Nous insistons ici sur le fait que les accesseurs ne doivent s'utiliser qu'avec des listes non vides. Ceci facilite la compréhension ultérieure de la récurrence structurelle sur les listes[4].

A la suite de ce cours, aucun nouvel élément du langage Scheme ne sera introduit. Les exercices relatifs à ce cours sont nombreux. Ils demandent une parfaite compréhension de l'évaluation. Nous demandons ensuite de définir quelques fonctions non récursives.

**2. La récursivité.** Ce cours est certainement la clé de voûte de l'enseignement. Nous introduisons la récursivité par l'écriture d'une suite récurrente pour montrer aux étudiants que cette notion n'est absolument pas nouvelle pour eux. Nous montrons que le principe d'écriture d'une suite récurrente (cas de base, cas général) se retrouve dans une démonstration par récurrence.

Nous passons ensuite dans le langage informatique Scheme pour écrire concrètement une fonction qui calcule les termes d'une suite récurrente vue auparavant sous forme mathématique. La transition d'une notion mathématique assez simple vers la programmation récursive est ainsi réalisée en douceur.

La récursivité sur les listes est alors exposée. Certains étudiants ont alors un blocage classique : ils ne comprennent pas l'hypothèse de récurrence. Un retour sur une suite mathématique récurrente montre

[3] Un cours logique ne correspond pas à une semaine. Sa durée varie en fonction de son contenu et de sa compréhension par les étudiants.

[4] Le fait que `(car '())` et `(cdr '())` provoquent une erreur, selon la norme R4RS [7], est de ce point de vue très appréciable.

que cette hypothèse est bien acceptée en mathématiques et qu'elle doit l'être de la même façon en informatique. Une fois l'écriture des fonctions récursives simples acquise, grâce à beaucoup d'exercices, nous montrons très rapidement la récursivité terminale (voir en 4.1).

Nous abordons ensuite des fonctions récursives plus complexes : qui traitent récursivement les sous-listes, qui rendent plusieurs résultats (voir 4.5).

**3. L'abstraction de données.** La notion d'abstraction est présentée au travers de l'exemple classique des ensembles. Nous insistons sur le fait que les fonctions de manipulation des objets abstraits (les ensembles) ne doivent qu'utiliser les primitives mises à disposition et en aucun cas présupposer de l'implantation concrète (les listes par exemple). Certains étudiants ont des difficultés à utiliser ce principe mais nous préférons ne pas prendre trop de temps ici car cette notion est revue dans le cours sur les arbres binaires.

**4. La programmation d'ordre supérieur.** Nous donnons plus loin les raisons qui ne nous permettent pas de présenter la programmation d'ordre supérieur sous ses deux aspects : la possibilité de définir des fonctions dont des arguments sont des fonctions, et celle de définir des fonctions dont les résultats sont des fonctions. Nous ne traitons que la première possibilité.

Pour cela, nous reprenons des fonctions récursives simples, par exemple la fonction qui calcule la somme des nombres contenus dans une liste ; puis celle qui calcule le produit,.... Nous proposons alors de réaliser une abstraction de la fonction élémentaire réalisée et de l'élément neutre. Nous donnons d'autres exemples du même ordre dont une fonction simplifiée de `map`.

Dans les exercices, nous proposons d'écrire de nombreuses fonctions dont un argument est un prédicat pour filtrer une liste, compter des candidats,... Ici, un point très important est l'utilisation de ces fonctions. Nous demandons aux étudiants de savoir réutiliser ces abstractions pour créer de nouvelles fonctions (comme des opérations de sélection sur les listes). Ce point pose souvent des difficultés (voir 4.4).

**5. Les arbres binaires.** Ce cours permet d'une part de présenter une structure de données importante en informatique et dont le principe d'induction structurelle est nouveau ; et d'une autre part d'illustrer l'abstraction de données puisque ces arbres sont représentés par des listes.

Nous commençons par présenter les primitives de manipulation des

arbres, mais sans demander aux étudiants de les implanter. Nous exposons ensuite le principe de récurrence et l'illustrons par des fonctions de parcours.

**6. Une petite application.** Pour les groupes les plus avancés, nous proposons une application plus conséquente : une implémentation des arbres d'Huffman. Il est important de montrer aux étudiants (même dans un cours à horaire réduit) que la programmation en Scheme ne se limite pas à écrire de petites fonctions. Cette application a pour mérite de proposer un problème qui recoupe de manière transversale le cours : les listes, les arbres et l'abstraction des données doivent être utilisés.

### 3.2. Matériel

Les étudiants reçoivent un support de cours de 60 pages. Il contient de façon condensée la majorité des concepts vus lors des cours magistraux, de nombreux exercices et leurs corrections. Ces exercices (plus d'une centaine) sont souvent simples : chacun demande généralement l'écriture d'une fonction dépassant rarement 8 lignes. [2] est une version nettement étendue de ce support de cours.

Les séances de travaux pratiques se font sur des compatibles PC avec l'interprète EdScheme.

## 4. Réflexions et choix

Le cours que nous venons de présenter est le résultat de nos réflexions et de notre expérience. Nous présentons les points sur lesquels il nous semble important de s'arrêter et nous justifions nos choix qui sont souvent guidés par les contraintes de temps.

### 4.1. Des concepts plutôt qu'un langage

Nous aurions pu présenter rapidement l'ensemble de Scheme en traitant en plus les chaînes, les vecteurs, les fichiers, les macros...Notre choix a été de présenter des concepts plutôt qu'un langage. Scheme n'est pas le principal objectif de ce cours, nous ne le percevons que comme un outil. De plus, nous avons préféré donné de bonnes bases simples aux étudiants plutôt que vouloir couvrir trop de sujets. C'est pourquoi la récursivité et l'abstraction des données sont les seuls fils conducteurs du cours.

**Programmation fonctionnelle pure.** Scheme n'est pas un langage fonctionnel pur. Il permet de réaliser des effets de bord grâce aux primitives `set!`, `set-car!` et `set-cdr!`. L'utilisation de ces primitives et de la séquentialité, avec `begin`, fait qu'il est possible de programmer en Scheme dans un style impératif. Cette possibilité doit-elle être enseignée lors d'une introduction à la programmation avec Scheme ? Nous pensons que non pour les raisons suivantes :

- Le volume horaire dont nous disposons est tout juste suffisant pour montrer les principaux aspects fonctionnels de Scheme. Il faudrait en éliminer une partie pour avoir la possibilité de traiter les aspects impératifs.
- Les étudiants ont une expérience de la programmation impérative en Pascal. Nous n'indiquons seulement le fait que Scheme permet aussi de programmer comme il est habituel de faire en Pascal.
- Le modèle purement fonctionnel a une sémantique claire. Il est de ce fait élégant et permet de réaliser des preuves formelles. Ceci n'est pas le cas si l'on introduit les modifications physiques.

Notre cours est donc purement fonctionnel. Nous n'introduisons aucune primitive de modification physique, ni `begin`, ni les procédures d'entrée/sortie (`display`, `read`). Cette solution est radicale mais elle possède de nombreux avantages : elle permet d'éviter la confusion entre effet de bord et résultat rendu. Les étudiants ne se posent plus la question – comme ils le faisaient en Pascal – de savoir si il faut utiliser une procédure avec des paramètres de sorties ou si il faut une fonction.

Dans le même état d'esprit, nous ne présentons pas aux étudiants les constructions de Scheme qui supposent une séquentialité. Ainsi, notre présentation de `lambda` montre qu'il est absurde d'écrire une fonction comme celle qui suit :

```
(lambda (x)
  (+ x 10)
  (* x 3))
```

Quelques étudiants font au début des erreurs de ce type. Cela montre qu'ils n'ont pas réellement assimilé la notion de «fonctionnalité» et c'est une bonne occasion de revenir sur ce point primordial. Il nous aurait été facile de modifier l'interprète pour empêcher ces erreurs. Mais d'une part, nous voulons que les étudiants programment avec un vrai langage (voir 2.1) et d'autre part, il est préférable qu'elles soient commises : nous pouvons ainsi détecter des défauts de compréhension.

`cond` ne fait pas apparaître explicitement que plusieurs expressions peuvent être évaluées pour une même clause. La définition normale de `cond` est :

```
(cond  (test-1 expr-1-1 expr-1-2 ... expr-1-m)
       (test-2 expr-2-1 expr-2-2 ... expr-2-m)
               ...
       (test-n expr-n-1 expr-n-2 ... expr-n-m)
       (else expr-o))
```

mais nous notre présentation est la suivante :

```
(cond  (test-1 expr-1)
       (test-2 expr-2)
               ...
       (test-n expr-n)
       (else expr-o))
```

La justification de notre utilisation de `cond` est contestable, nous pourrions utiliser exclusivement `if`. Nous justifions son introduction par le fait que la plupart des ouvrages[5], même de bas niveau, l'introduisent et que nous voulons que nos étudiants puissent se servir de ces ouvrages. D'autre part, `cond` permet d'écrire des fonctions plus compactes. Il faut noter que cette façon de procéder n'a posé aucun problème.

**Itératif/Récursif.** Nous montrons à nos étudiants que tout en restant dans un cadre purement fonctionnel, il est tout de même possible de programmer dans un style itératif : en définissant des fonctions à «récursivité terminale» où un ou plusieurs arguments représentent des accumulateurs. Par exemple, la fonction factorielle `fact` peut être définie comme suit :

```
(define fact
  (lambda (n)
     (letrec ((faide (lambda (n acc)
                          (if (= n 0)
                              acc
                              (faide (- n 1) (* acc n)))))))
         (faide n 1)))
```

Ce style de programmation est plus efficace qu'un style purement récursif (dit à «récursivité profonde»). Mais les raisons ne peuvent pas être facilement et rapidement données aux étudiants. Nous préférons donc seulement montrer que ce style de programmation existe mais sans insister pour les raisons suivantes :

- la transformation entre une fonction récursive profonde et une récursive terminale n'est pas toujours simple,

[5] [10] introduit exclusivement `cond`.

- les étudiants mélangent souvent les deux[6]. Rare sont ceux qui maîtrisent les deux façons de penser. Pour avoir corrigé des examens de nos collègues utilisant en abondance ce type de récursivité, nous nous sommes aperçus que les étudiants n'avaient pas compris l'essence de la récursivité terminale : l'absence «d'empilement de calculs» mais s'étaient arrêtés à l'aspect syntaxique : la présence d'un accumulateur.

- Nous avons remarqué que des étudiants habitués à utiliser la récursivité terminale avaient énormément de mal à traiter des exercices sur les arbres.

## 4.2. Mathématiques et informatique

Nous abordons maintenant une question très importante à nos yeux. Dans quelle mesure faut-il utiliser les mathématiques pour l'illustration des concepts informatiques ?

Nous aurions pu comme le fait J.P. Roy [16] utiliser abondamment les mathématiques comme base d'explication et d'exercice pour introduire la récursivité. Nous aurions pu justifié cela aisément au vu du contexte mathématiques dans lequel baignent les étudiants. Bien au contraire, notre choix a été tout autre. Les mathématiques nous servent exclusivement à leur faire comprendre qu'il existe un continuum de pensée entre les mathématiques et l'informatique et à présenter certaines notions comme les lambda-expressions et les premières fonctions récursives. Voici les raisons qui justifient notre choix :

- Nos étudiants ont des cours de mathématiques à très forts coefficients et très gros volumes horaires. Il s'agissait donc pour nous de nous positionner différemment, de proposer une matière ludique et nouvelle. D'autre part, les étudiants ont déjà des cours d'analyses numériques en Pascal qui par contre pourraient très bien utiliser Scheme comme outil.

- De manière plus profonde, l'informatique étant une science différente des mathématiques, un premier cours doit faire ressentir cette distinction profonde. L'utilisation des listes et des arbres est fondamentale de ce point de vue.

- De plus, nombre d'exercices à base mathématique utilisent des paramètres fonctionnels. Contrairement à l'approche choisie par

[6] Nous leur disons qu'ils sont libres de choisir la façon dont ils veulent programmer leurs fonctions et de ne pas confondre ces deux façons de penser.

[16], nous préférons ne pas mélanger, dans un premier temps, la récursivité et les fonctions d'ordre supérieur. Dans notre cours, les fonctions d'ordre supérieur comme `tous`, `au-moins-un` ... qui sont présentées après des fonctions rendant plusieurs arguments sont très bien traitées par les étudiants.

- Nous avons remarqué une grande réticence, de la part des étudiants, quand il s'agit de traiter des problèmes mathématiques ou de physique en informatique. Il semblerait que la conjugaison des deux ait un effet paralysant. L'écriture de fonctions même très simples leur pose des problèmes quand elles sont trop liées à un problème mathématique précis. De plus, lorsque la fonction n'est pas commune ou simple, l'exercice revient alors chercher une solution à un problème mathématique.

## 4.3. La place des listes

Sans les listes les exercices portant sur la récursivité ne sont quasiment que des problèmes mathématiques : suites, arithmétique de Peano, dichotomie,... Les précédentes remarques impliquent donc d'introduire les listes très tôt dans le cours comme support à l'introduction de la récursivité. Nous introduisons donc dès la seconde séance les listes avant même la présentation de la récursivité. Cela nous permet de revenir sur l'évaluation qui est traitée durant la première séance. De plus, l'utilisation des listes ont les avantages suivants :

- Le concept est très simple.
- Il n'est pas connoté mathématiquement ce qui permet aux étudiants de ne pas être «parasités» par l'aspect mathématique d'un problème. Il n'est pas nécessaire de comprendre le pgcd pour comprendre la récursivité.
- Finalement, les listes n'offrent pas le même support intellectuel que les suites et forcent les étudiants à faire le point sur leur compréhension de la récursivité. En effet, nombre d'étudiants admettent facilement la preuve par récurrence, écrivent très facilement leurs premières fonctions récursives basées sur des suites, mais ont des problèmes pour compter le nombre de symboles d'une liste.

## 4.4. Le cas des lambda-expressions

Lors de la première année d'enseignement de Scheme, nous avons différé l'utilisation de l'instruction `lambda`. Nous présentions alors tous les types

de données et les instructions sauf `lambda` et nous utilisions la notation MIT, en opposition à la notation Indiana, pour définir les fonctions :

```
(define fact                    (define (fact n)
   (lambda (n)                     ...
     ...                           ))
   ))
```

en notation Indiana — en notation MIT

Notre but était de rester le plus simple possible dans les premiers cours et de n'aborder la possibilité de définir des fonctions à l'aide de `lambda`, notion qui nous paraissait plus difficile à assimiler pour les étudiants, que lors du cours sur la programmation d'ordre supérieur. Ceci s'est avéré être une erreur : la définition de fonctions à l'aide de `lambda` n'ont pas été bien comprises et donc la programmation d'ordre supérieur aussi.

Lors des années suivantes, nous avons donc choisi de présenter `lambda` en même temps que toutes les autres instructions, c'est-à-dire dans le premier ou deuxième cours. [12] indique un choix similaire. En fait, les étudiants n'éprouvent pas de difficulté majeure à écrire et utiliser `lambda`. Nous introduisons cette instruction en expliquant qu'elle crée des «fonctions anonymes» par analogie aux fonctions mathématiques comme $x \rightarrow x + 2$. La définition d'une fonction n'est alors que l'association d'une fonction anonyme à une variable. Nous faisons encore une fois une analogie à l'écriture mathématique $f : x \rightarrow x + 2$. Pour cela, nous n'utilisons, du moins dans un premier temps, que la notation Indiana.

Durant les premiers cours nous n'insistons pas sur la puissance de la programmation d'ordre supérieur, ce qui aide les étudiants à comprendre les concepts de base. Cette approche est beaucoup plus satisfaisante car les étudiants ont dès le début toutes les notions de Scheme nécessaires à la poursuite du cours. Ils comprennent l'importance des fonctions anonymes car elles sont, en notation Indiana, indispensables aux définitions des fonctions. Les étudiants sont ainsi beaucoup plus habitués à les manipuler et le cours sur la programmation d'ordre supérieur est beaucoup mieux compris.

## 4.5. Complexité des fonctions

La complexité est un domaine important de l'informatique mais qu'il est difficile d'aborder lors d'une initiation. C'est pourtant un concept qu'il faut évoquer. Nous le faisons en des termes peu académiques mais simples pour forcer les étudiants à considérer la «qualité» de leurs fonctions. Nous ne parlons pas du tout de la notation $O$, des problèmes

NP-complets, difficiles,...

En pratique, nous enseignons que la qualité d'une fonction récursive se mesure au nombre de fois qu'elle se rappelle ou qu'elle calcule quelque chose d'assez complexe. Ces critères sont assez flous mais suffisants.

Par exemple, après avoir écrit la fonction factorielle, et avoir vu que les temps de calcul peuvent être importants, nous demandons d'écrire la fonction qui calcule

$$\frac{x!+1}{x!+2}+x!$$

Cet exercice montre l'utilité du `let`. Elle est généralement bien comprise.

Après la récursivité, nous abordons des problèmes assez simples mais que nous demandons de résoudre le plus efficacement possible. Un exemple classique est le calcul de la moyenne arithmétique d'une liste de nombres. La première solution proposée est la suivante :

```
(define somme
   (lambda (L)
      (if (null? L)
          0
          (+ (car L) (somme (cdr L))))))

(define moyenne (lambda (L) (/ (somme L) (length L))))
```

Les étudiants comprennent alors rapidement que cette fonction, bien qu'étant parfaitement correcte, n'est pas satisfaisante d'un point de vue théorique car «la liste est parcourue deux fois». Nous introduisons alors les fonctions qui rendent plusieurs résultats. Nous insistons sur ce point et nous demandons aux étudiants d'être capables de reconnaître les cas ou plusieurs résultats sont nécessaires. Les examens comportent toujours au moins une question sur ce sujet car les étudiants se satisfont souvent d'une fonction «qui marche» ; nous leur demandons aussi qu'elle «marche vite».

Les fonctions rendant plusieurs résultats correspondent au point le plus ardu abordé dans ce cours. Elles sont pédagogiquement très intéressantes car elles obligent l'étudiant à réfléchir sur le résultat terminal rendu et la construction d'un résultat à partir du rappel récursif.

## 4.6. Ordre supérieur simplifié

Notre cours présente les fonctions d'ordre supérieur de façon réduite : nous abordons les fonctions ayant des fonctions en arguments mais pas celles qui rendent des fonctions.

Ce choix n'est guidé que par des contraintes de temps. Il est clair que la programmation d'ordre supérieur, quand elle est complètement

traitée, permet d'introduire des concepts intéressants : les fermetures, la programmation par passage de continuation (CPS), les acteurs, les flots... Mais ceci est d'un niveau trop élevé, du moins comparativement aux autres aspects que nous traitons.

## 4.7. Absence de typage

L'absence de typage dans Scheme peut à première vue apparaître comme un avantage : l'écriture des fonctions est plus immédiate. Mais dans la pratique, il s'avère que cette absence amène des ambiguïtés et ne met pas en évidence certaines erreurs.

Par exemple, dans la norme R4RS de Scheme [7], il est indiqué que tout ce qui n'est pas `#f` est vrai. Autrement dit, les nombres, symboles,... représentent aussi la valeur *vrai*. Le type des booléens n'est donc pas complètement dissocié des autres types. Il est par exemple possible d'évaluer `(if (not 3) 'oui 'non)` ce qui rend `non`. Pour notre part, nous limitons strictement l'utilisation des fonctions booléennes à `#t` et `#f`. Nous considérons donc l'expression ci-dessus comme produisant une erreur.

Les étudiants se posent souvent des questions quant aux résultats des cas terminaux dans les fonctions récursives. C'est surtout le cas quand les fonctions rendent des résultats complexes. Par exemple, la fonction à deux résultats qui calcule la somme des éléments d'une liste ainsi que sa longueur est définie comme suit :

```
(define somme-et-long
  (lambda (L)
     (if (null? L)
         ???
         (let ((R (somme-et-long (cdr L))))
           (cons (+ (car L) (car R)) (+ 1 (cdr R)))))))
```

Ici le résultat est une paire pointée dont le `car` indique la somme et le `cdr` la longueur. De nombreux étudiants ne donnent pas la valeur terminale correcte remplaçant les `???` : ils indiquent `0` ou la liste vide[7]. Avec un langage typé, ces erreurs ne seraient pas commises.

On s'aperçoit que de très nombreuses erreurs proviennent d'une inattention vis-à-vis du typage. Pour palier à ce problème nous conseillons aux étudiants d'indiquer en commentaires les signatures des fonctions. Pour la fonction précédente, ce commentaire est `liste de nombres -> paire pointée de deux nombres`. Lorsque ce conseil est suivi, les erreurs de types sont moins fréquentes.

[7] La réponse correcte est `(cons 0 0)`.

Un système de typage, peut-être plus souple que celui de ML, est réellement souhaitable. Nous développons cette idée dans la conclusion de cet article.

# 5. Résultats et évaluation

Le cours présenté ici a évolué au cours des années. Notre expérience nous porte à croire qu'il a maintenant atteint une certaine stabilité à la vue des résultats obtenus par rapport à ceux qui étaient attendus et de l'opinion des étudiants.

## 5.1. Réceptivité

Il est tout d'abord certain que les étudiants apprécient le fait d'être rapidement capables d'écrire des programmes conséquents avec un langage très simple et réduit. Comparativement à Pascal, que les étudiants connaissent partiellement, ils préfèrent généralement Scheme.

Il est d'ailleurs important de noter que de nombreux étudiants pas du tout intéressés par Pascal, qui ont eu de mauvaises notes lors des examens de Pascal, se sont beaucoup plus investis dans Scheme et ont obtenus des résultats bien meilleurs. De notre point de vue, ceci est en partie lié à l'autonomie offerte aux étudiants comme le décrit le point suivant.

## 5.2. Support de cours et corrections

Nous pensons qu'il est important que les étudiants disposent d'une grande quantité d'exercices ainsi que de leurs corrections. Bien que certains de nos collègues nous aient fustigés car ils ne peuvent pas donner les mêmes exercices en examen, nous pensons que cette solution est très intéressante à plusieurs points de vue. Tout d'abord, elle responsabilise les étudiants et leur donne une certaine autonomie. Ils sont en général très lucides par rapport à leur attitude et savent que faire de la recopie de correction ne sert à rien. D'après les retours que nous avons, les étudiants apprécient le fait d'avoir les corrections. De plus, les étudiants concernés étant initialement peu motivés par l'informatique, ils ne peuvent (ou ne veulent) se permettre de travailler cette matière en dehors des créneaux horaires prévus. Très peu le font effectivement. Nous avons encouragé ceux qui possèdent des ordinateurs en distribuant les versions du domaine public de MacGambit [9] et de PC-Scheme[8] [18]. Nous avons remarqué

[8] Bien que celui-ci ne soit malheureusement pas à la norme R4RS.

que le fait de leur fournir les corrections, n'empêchait pas un travail productif lors des travaux dirigés. En fait, il semble que les étudiants prennent l'enseignement de Scheme comme un «jeu», complètement dissocié des matières réputées plus ardues comme les mathématiques et la physique[9]. Leur jeu consiste donc à réellement chercher à écrire les fonctions demandées. Une partie des étudiants préfère d'ailleurs pour cela travailler sans ordinateur. Ils ne se réfèrent aux corrections qu'après avoir proposé une solution. Cet état d'esprit est très appréciable pour les enseignants. Les étudiants les plus faibles, n'étant pas capables d'écrire correctement une fonction ont un éventail très large d'exemples qui leur est utile pour travailler cette matière.

## 5.3. Évaluation

Ce cours est évalué par le biais d'une interrogation qui intervient après les quatre ou cinq premières séances, puis un examen final. Nous faisons preuve d'un peu de démagogie pour motiver les étudiants (dont certains ont déjà abandonné tout espoir de comprendre l'informatique après leur expérience en Pascal) : la première interrogation est relativement facile. De plus, nous n'avons pas de scrupule à donner de très bonnes notes aux étudiants s'étant investis dans le cours car l'examen final est de difficulté supérieure à la fois par le niveau des exercices et par l'étendue des sujets : les arbres, les fonctions rendant plusieurs résultats, la programmation d'ordre supérieur sont alors au programme. L'examen final nous offre alors une vision de leur compréhension de la totalité du cours.

Voici à titre d'exemple les notes (note maximale : 20) des étudiants d'une section lors de l'interrogation intermédiaire (figure 1) et de l'examen final (figure 2). On peut voir le grand nombre de notes «satisfaisantes» à l'interrogation. Lors de l'examen, la répartition des notes n'est pas du tout la même. Elle reflète plus fidèlement le niveau des étudiants. On distingue en fait trois grand groupes.

Les étudiants du groupe le plus faible (  ) avaient des notes inférieures à la moyenne à la première interrogation et ont complètement «coulé» lors de l'examen. En fait, on constate que les étudiants de ce groupe n'avaient que superficiellement compris les concepts exposés dans les premiers cours. Ils n'ont pas pu rattraper leur retard par la suite.

Les étudiants du second groupe (  ) ont eu des notes entre 10 et 16 à l'interrogation et ont régressé lors de l'examen. Pour eux, les principes de base ont été acquis.

Le groupe le meilleur (  ) est composé d'étudiants ayant

[9]Cette remarque est à mettre en relation avec le fait que nous évitons de traiter des problèmes trop mathématiques.

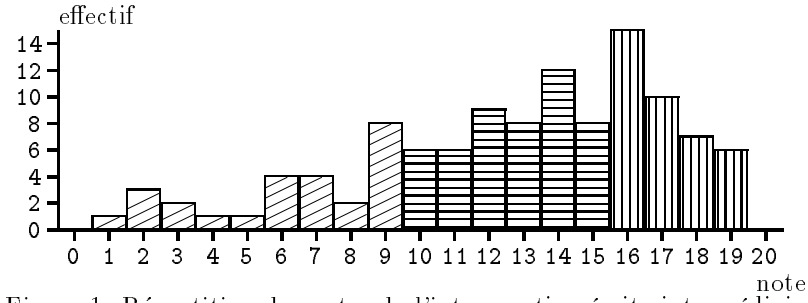


Figure 1: Répartition des notes de l'interrogation écrite intermédiaire.

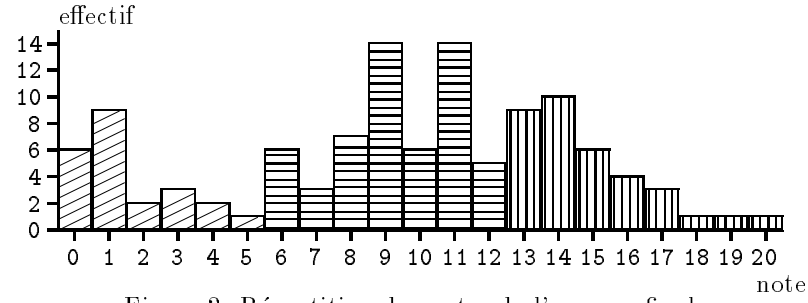


Figure 2: Répartition des notes de l'examen final.

de bonnes notes à l'interrogation mais aussi à l'examen. A titre d'information, nous travaillons en étroite collaboration avec le responsable de la licence en lui communiquant les examens et les statistiques réalisées. Nous donnons un avis favorable à l'admission en licence d'informatique pour la meilleure moitié de ce groupe.

## 5.4. Comparaisons avec Pascal

Notre expérience d'enseignement de la programmation impérative avec Pascal [13] au même type d'étudiants dans des conditions similaires[10] pendant deux ans nous permet maintenant d'établir quelques comparaisons. L'utilisation de Scheme présente plusieurs avantages.

**Rapidité d'acquisition.** Il ne faut que deux séances pour que les étudiants acquièrent l'ensemble des aspects syntaxiques et des fonctionnalités de Scheme qu'ils vont utiliser pendant le reste du cours.

**Plus de concepts abordés.** Pour un volume horaire aussi faible, la simplicité des aspects syntaxiques permet d'aller plus loin

[10] Les corrections d'exercices n'étaient pas fournies.

sur le plan des concepts présentés. Ainsi en Pascal avec le double d'heures, les étudiants de première année ne savaient que difficilement faire la distinction entre une procédure et une fonction, entre faire un affichage et rendre un résultat. Et très peu d'entre eux savaient découper logiquement un petit problème comme ranger par ordre croissant une série de nombres choisis aléatoirement (deux procédures). En comparaison le langage Scheme nous permet d'aborder les points présentés dans le cours.

**Égalité devant la nouveauté.** Scheme est un langage nouveau pour tous les étudiants. De plus, l'attention que nous portons à l'aspect fonctionnel et récursif amènent tous les étudiants, du moins au début, sur un plan d'égalité. Ceci présente le double avantage de, d'une part permettre de construire de nouvelles connaissances sur un terrain vierge dans le cas des autodidactes. D'autre part, cette égalité est motivante pour les étudiants qui se sentent *a priori* peu attirés par les ordinateurs. Nous avons d'ailleurs remarqué que les mêmes étudiantes qui étaient moyennes en Pascal, réussissent souvent mieux en Scheme et sont souvent brillantes. D'autre part, il arrive même que des étudiants brillants en première année en Pascal du fait de cours en lycée soient décontenancés au début du cours de Scheme.

Cette comparaison donne l'avantage à Scheme ; cependant, nous ne sommes pas de ceux qui pensent que Scheme répond à toutes les attentes d'un enseignement de l'informatique. Nous pensons qu'il est important que les étudiants programment aussi en DEUG avec un langage impératif de type Pascal. En effet, certains étudiants s'orientent vers des filières où des langages comme C sont utilisés. Après avoir enseignés Pascal puis Scheme aux mêmes étudiants nous voudrions expérimenter l'inverse ; c'est-à-dire enseigner Scheme aux premières années et Pascal aux secondes années mais en allant plus rapidement aux concepts importants : itération, procédures, structures, pointeurs,...

# 6. Perspectives

Il est devenu courant de dire que Scheme est un langage bien adapté à l'enseignement. Notre expérience nous a montré que c'est le cas même si l'audience n'est au premier apport que peu motivée. Pour ce faire, nous avons proposé un cours réduit mais présentant des aspects fondamentaux de la programmation, en particulier la récursivité.

**A propos du typage.** Nous pensons qu'utiliser un langage ayant un typage peut être l'évolution naturelle de ce type de cours. Ce typage devrait permettre de s'assurer de la correction des types des arguments et des résultats des fonctions. Nous envisageons deux solutions :

- Utiliser un langage, dérivé de Scheme, avec un typage plus fort. Nous pensons qu'un typage automatique à la ML mais plus souple serait le bienvenu. Nous sommes en train d'évaluer RICE Scheme qui offre un «typeur» basé sur une analyse à base d'ensembles *(set based analysis)*.
- L'autre solution est d'évaluer un cours utilisant Caml avec les mêmes restrictions que notre cours. Il faudrait introduire exclusivement la notion de type à un niveau très simple, ne pas aborder le pattern matching, ne présenter qu'une seule forme de définition de fonctions ...

**Amélioration de l'environnement.** Au niveau de l'environnement, l'éditeur d'EdScheme pourrait être amélioré en proposant une vérification automatique des formes au lieu de simplement vérifier qu'il y a autant de parenthèses ouvrantes que fermantes[11].

**Améliorations du cours.** Nous aimerions améliorer certains points du cours : renforcer l'aspect abstraction des données et accentuer l'utilisation de applications pour permettre aux étudiants d'acquérir une plus grande autonomie face au découpage d'un problème en sous-problèmes. D'autre part, si nous avions quelques heures en plus, nous souhaiterions aborder l'ordre supérieur de manière complète.

**Remerciements.** Nous tenons à remercier Yves Hervier responsable des enseignements d'informatiques de premiers cycles de l'université de Nice Sophia-Antipolis d'avoir laissé de côté ses réticences quant à l'utilisation de Scheme et d'avoir expérimenter à nos côtés cette approche. Nous remercions également les relecteurs anonymes qui nous ont permis, par leurs remarques, de mieux argumenter notre propos.

# Bibliographie

[1] H. Abelson, G. J. Sussman, and J. Sussman. *Structure et Interprétation des Programmes Informatiques*. InterEditions, 1989.

[11] En fait, de tels environnements existent [3] mais ne permettent à notre connaissance pas une facile intégration d'un interprète de Scheme ni une utilisation sur des machines compatibles PC de faibles performances.

[2] L. Arditi and S. Ducasse. *La programmation. Une approche fonctionnelle et récursive avec Scheme.* Eyrolles, Mar. 1996.

[3] P. Borras, D. Clement, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: the system. In $3^{rd}$ *Symposium on Software Development Environments*, Boston, USA, Nov. 1988.

[4] M. Briand and A. Pic. Enseigner l'informatique avec Scheme en école d'ingénieurs. *Bigre*, (73), June 1991.

[5] P. Castéran and L. Favreau. Scheme et l'enseignement de l'informatique en premier cycle : l'expérience Bordelaise. *Bigre*, (73), June 1991.

[6] J. Chazarain. *Programmer avec Scheme. De la pratique à la théorie.* International Thomson Publishing, 1996.

[7] Clinger et al. Revised$^4$ report on the algorithmic language scheme. *ACM Lisp Pointers*, 4(3), July 1991.

[8] G. Cousineau and M. Mauny. *Approche Fonctionnelle de la Programmation.* Ediscience International, 1995.

[9] M. Feeley. MacGambit. InterprŔte de Scheme pour Macintosh. `ftp://ftp.iro.montreal.ca/pub/paralle/gambit`.

[10] M. Felleisen and D. P. Friedman. *Le Petit LISPien.* Masson, 1991.

[11] E. Gallesio. *Manual of STk.* http://kaolin.unice.fr/, 1993.

[12] J. Haguel and J. Goulet. Remarques sur le cours de base de Scheme. *Bigre*, (73), June 1991.

[13] Y. Hervier. Super Pascal. MIPS, Université de Nice - Sophia Antipolis.

[14] J.-M. Hufflen. *Programmation fonctionnelle en Scheme.* Masson, 1996.

[15] M. P. Jones. *An introduction to Gofer.* University of Yale, ftp.cs.nott.ac.uk/nott-fp/languages/gofer, 1991.

[16] J.-P. Roy. Support de cours. MIPS, Université de Nice - Sophia Antipolis.

[17] Schemers Inc. EdScheme. Interprète de Scheme pour PC Windows et Macintosh. Contacter `71020.1774@compuserve.com` ou `100015.1465@compuserve.com`.

[18] Université de Genéve. PCscheme. Interprète de Scheme pour DOS. `ftp://cui.unige.ch/PUBLIC/pcs/`.