Rapport de stage :
Amélioration du parser de Pharo sur
la gestion des erreurs et des nested comments

Inria

40 Avenue Halley, 59650 Villeneuve-d'Ascq

Du 06 Avril au 28 Août

Tuteur Professionel : Guillermo Polito

Université de Lille

Cité Scientifique, 59650 Villeneuve-d'Ascq

Licence 3 Informatique parcours Info

Tuteur Universitaire : Léopold Weinberg

Remerciements

Je souhaite remercier mon tuteur professionel Guillermo Polito, Stephane Ducasse le chef scientifique de l'équipe et Pablo Tesone ingénieur scientifique pour m'avoir accompagné tout au long du stage, pour avoir pris de leur temps pour m'aider lorsque j'avais des difficultés, pour m'expliquer ce que je ne comprenais pas, pour leurs retours éclairés et bienveillants sur mon travail mais aussi pour la passion qu'ils ont partagée avec moi.

Résumé

Stage sur "Amélioration du parser de Pharo sur la gestion des erreurs et des nested comments" par Philippe LESUEUR au sein du service RmoD d'Inria.
Ce rapport a pour but de rendre compte du travail effectué pendant ce stage.
Le projet principal était de retravailler le parser de Pharo pour améliorer la récupération d'erreurs. Le second projet était de permettre de commenter et décommenter des commentaires imbriqués et faire la même chose pour les chaînes de caractères.
Pour le projet principal, j'ai donc commencé par étudier le parser et créer des tests permettant de vérifier son bon fonctionnement. Ensuite, j'ai effectué des changements de le parser pour pouvoir créer un retour en temps réel. Après quoi, j'ai étudié la recherche de réparation d'erreur, les outils de création de menu contextuel et créé des classes de réparation.
Pour le second projet, j'ai modifié non seulement les méthodes de copier coller et les "smartCharacters" mais également la recherche de noeud pour permettre les imbrications.
Pour finir, à la fin du stage, j'ai dû réaliser l'intégration de mon code dans Pharo et des productions pour rendre compte de mon étude du parser et ma création de retour en temps réel.

Internship on "Amelioration of the parser of Pharo on error handling and nested comments" by Philippe LESUEUR inside Inria's RmoD team.
The main project was to rework the parser of Pharo to improve error handling. The Second one was finding a way to nest and unnest comments and strings.
For the main project, I started with the study of the parser and the creation of test allowing to easily spot errors in the parser's inner working. Then, I changed the parser to create a real time feedback. After what, I studied the error reparation research, the context menu creation tools and created classes for code reparation.
For the second project, I, not only, modified copy, paste and the "smartCharacters" methods but also the node search to allow nesting and unnesting.
Finally, at the end of my internship, I integrated my code in Pharo and produced reports on my study of the parser and my creation of the real time feedback.

Sommaire

Introduction

J'ai réalisé mon stage au sein d'Inria car la mission qui m'a été proposée m'intéressait sur plusieurs points. La mission était de modifer le Parser de Pharo pour offrir une meilleure gestion des erreurs syntaxiques. Le Parser est l'outil qui sert à interpréter un langage informatique et, ici, il s'agit du langage Pharo. Cette mission m'a fortement intéressée, premièrement, pour sa qualité pédagogique. En effet, il s'agit de travailler sur les bases d'un langage informatique et donc de pouvoir étudier ses mécanismes. Je pense que comprendre ces derniers me permettra de mieux anticiper certains problèmes qui pourraient survenir lors de futurs projets et donc d'écrire du code plus solide. Ensuite, ce projet est intéressant car il offre l'opportunité d'avoir une réelle utilité. Pharo est un langage utilisé par de nombreuses personnes, notamment au sein d'Inria. Lors de l'écriture d'un code, nous commettons tous des erreurs et jusqu'à maintenant, les erreurs syntaxiques affichaient seulement l'emplacement de celles-ci. Les améliorations implémentées suite à mon stage seraient donc profitables pour toutes les personnes utilisant ce langage car elles permettraient d'offrir une meilleure compréhension et de proposer des réparations possibles à l'utilisateur. Pour finir, cette mission offre un défi personnel car le Parser est un outil complexe et essentiel qui permet donc de tester mes capacités et offre peu de place aux erreurs sous peine de détraquer le langage. La tâche annexe d'offrir une gestion automatique des commentaires et chaînes de caractères imbriquées qui m'a été assignée m'a également beaucoup plu car bien qu'assez facile et rapide, c'est une tâche qui permet de se reposer tout en étant productive et qui est une amélioration considérable de l'envirronnement de travail, ce qui est donc très gratifiant. Pour finir, la perspective de travailler dans un envirronnement open source m'intéressait fortement car il s'agit finalement d'un espace de coopération et d'échange qui demande de pouvoir s'adapter à tous les contributeurs du code de Pharo pour éviter les conflits et avoir un système fonctionnel.

Contexte

Inria est l'institut national de recherche en sciences et technologies du numérique. Créée en 1967, sous le nom de IRIA, Institut de recherche en informatique et automatique. Cet institut a pour but de créer un lien entre la recherche et l'industrie et de faire avancer des projets risqués et ambitieux en sciences et technologies du numérique. Aujourd'hui, sa mission est d'accélérer, par la recherche et l'innovation dans le numérique, la construction d'un leadership scientifique, technologique et industriel de la France dans la dynamique européenne. Inria dispose de 9 centres de recherche au coeur des grandes universités de recherche, soutient plus de 170 startup, comporte plus de 3500 scienfiques répartis dans plus de 200 équipes-projets.
Dans le contexte de l'évolution des logiciels, l'objectif de l'équipe RmoD est de soutenir la remodularisation des applications orienté objet. Cet objectif est géré à travers deux perspectives complémentaires : remodulariser des applications existantes et offrir des supports de langage pour des design modulaires. Pour la remodularisation, il s'agit de pouvoir proposer une partie d'un ensemble fortement couplé sans que les fonctionnalités ne s'en retrouvent trop affectées, pouvoir identifier les parties réutilisables, pouvoir modulariser une application orienté objet. Pour les supports de langage, l'objectif est de créer des logiciels extrêmement flexibles et facilement recomposables.

Inria favorise la prise de risque scientifique, notamment à travers l'interdisciplinarité et les partenariats industriels, promouvant le développement de technologies qui sont souvent au cœur de l'activité scientifique, accompagnant les démarches entrepreneuriales. Pour accélérer l'innovation avec un objectif d'impact, Inria est engagée dans la diffusion au niveau mondial de logiciels open source, l'aide au développement technologique, l'élaboration d'un dialogue stratégique avec les grands acteurs de l'industrie française et européenne et les leaders mondiaux et la mise en place d'un dispositif global et de bout en bout, avec le dispositif nommé Inria Startup Studio, pour augmenter la création de startups technologiques. Inria s'engage également à la mise en œuvre des politiques publiques à très fort enjeu, comme récemment, le développement de l'application Stop-Covid. Inria souhaite également être un moteur de l'éducation à la culture numérique en renforçant ses liens avec l'Éducation nationale sur des projets où sa valeur ajoutée est avérée.

Le problème était le suivant : RmoD utilise le langage de programmation Pharo (cf Mooc de Pharo) et lors de l'écriture du code, celui-ci est parsé pour obtenir une stylisation du texte telle que les variables écritent en bleu, les commentaires en gris et les erreurs en rouge. Ce système restait fort incomplet, il n'y avait aucune distinction entre les types d'erreur ni aucune information sur celles-ci. La précision des erreurs était très faible puisqu'à la rencontre d'une erreur, tout le code qui suivait était considéré comme erroné. Et il y avait également des problèmes à l'intérieur de l'AST qui faisaient perdre des informations et dupliquer de morceaux de code.

Le stage qui m'a été proposé été donc de retravailler le parser pour avoir une meilleur gestion des erreurs, avoir un retour en temps réel sur celles-ci et proposer un système de réparation, avec comme projet secondaire, de permettre une imbrication facile de commentaires dans des commentaires et de chaînes de caractères dans des chaînes de caractères.

Au cours du stage, j'ai réalisé une étude du parser et écrit des tests vérifiant son bon fonctionnement, fournis un systême de retour en temps réel des noeuds d'erreur à l'aide d'un outil pré-existant, effectué l'étude des cas d'erreur pour trouver des méthodes de recherche de réparation, créé des classes de réparation, étudié la création du menu contextuel pour pouvoir créer un bouton d'exécution de la réparation, implémenté l'imbrication des commentaires et des chaînes de caractères et intégré mon code dans le système de Pharo. J'ai également réalisé un papier sur les fonctionnements du parser et de la réparation d'un AST erroné, ainsi qu'une vidéo sur le fonctionnement du retour en temps réel. Ces derniers n'ont pas encore été publiés au moment de l'écriture de ce rapport.

Contribution

I – Etude et Modifications du Parser

Le parser de Pharo s'appelle RBParser et se situe dans le package AST-Core. Le but du parser est d'analyser une chaîne de caractères de code et de créer un arbre de syntaxe abstraite (AST) contenant seulement l'analyse syntaxique du code. Le parser utilise un outil appelé RBScanner pour diviser le code en tokens et utilise ces tokens pour créer les nœuds qui composent l'AST.

1. Construction de l'AST

Le parser utilise six méthodes publiques pour créer un AST (seules 4 d'entre elles sont souvent utilisées : parseExpression, parseFaultyExpression, parseMethod, parseFaultyMethod).
La différence entre parseExpression et parseMethod est décrite dans leur nom. ParseMethod utilisée dans le contexte de la création ou modification de méthode donc elle attend en entête le selecteur de la méthode et éventuellement des pragmas (annotations de méthode).
Là où, parseExpression est utilisé dans un contexte comme celui du Playground de Pharo, où le but est seulement d'exécuter le code, pas de le concerver dans une méthode. Néanmoins, l'expression est placée dans un nœud de méthode générique qui a pour sélecteur #noMethod mais n'accepte pas les pragmas. Le nœud retourné est aussi différent puisqu'il va retourner le plus petit nœud avec un chemin unique.

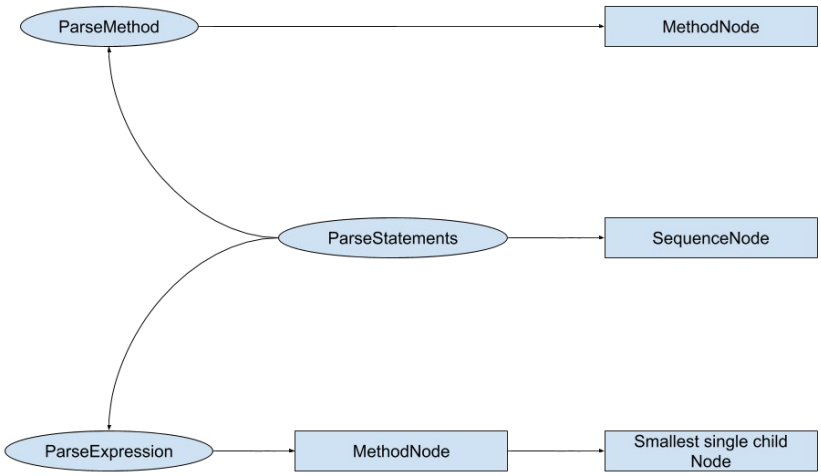


figure 1

NB: The child of a node is a subnode of this one. SequenceNode is always a single child of a MethodNode so ParseExpression never returns a MethodNode.

La différence entre parse et parseFaulty est très simple. Qand on utilise parse, lorsque l'on rencontre une erreur syntaxique, on arrête le parsing, aucun AST n'est retourné et une SyntaxErrorNotification est levée.
Avec parseFaulty, lorsque l'on rencontre une erreur syntaxique, on crée un nœud d'erreur avec la fonction parseErrorNode qui devient un nœud de l'AST (qui est retourné).

2. Scanner

1) Tokens

Comme dit précédemment, le parser utilise un scanner pour réaliser une partition primaire du code sous la forme de tokens. Les tokens sont composés de 4 éléments importants : leur identification (Literal, Identifier, Keyword, Binary, Value, EOF etc...) représentée par différentes sous-classes de RBToken, leur valeur, leur point de départ et leur point d'arrêt. J'ai donc créé des tests qui ciblent chacun un de ces éléments, pour chaque élément de chaque token.
Au niveau de l'identification, la classe abstraite RBToken possède une méthode d'identification pour chaque type qui renvoie 'false' par défaut. Chaque sous-classe surcharge alors la méthode qui lui correspond pour renvoyer 'true'. Le test est alors très simple, on appelle la méthode correspondant au token attendu. (cf Annexe - 1 - token type. p.37)

Tester la valeur d'un token est ce qu'il y a de plus important car le scanner possède extrêmement peu de cas d'erreur. Il n'attend pas nécessairement un séparateur pour finir son token. Si le caractère courant ne convient pas à la séquence en cours, le scanner revoie le token et en recommence un nouveau. Je pense que si le scanner agit ainsi, c'est pour permettre une syntaxe souple, comme ne pas exiger d'espace avant ou après une parenthèse et permettre certaines facilités. Par exemple 'halt' est une méthode permettant de mettre un break dans l'exécution d'un code. C'est un message qui peut être adressé à n'importe quel receveur car il ne se sert que de son contexte. On peut donc écrire '1halt'. Le scanner séparant cette chaîne en 2 token, on obtient alors un message halt envoyé à 1.
Pour tester que les tokens ont la bonne valeur il faut donc scanner des chaînes de caractères possédant des caractères incompatibles avec les précédents et vérifier que le token ne contient que la sous-chaîne valide. On peut vérifier également que le token suivant ne contient pas cette sous-chaîne. (cf Annexe - 1 - token value. p.37)
Tester les valeurs de début et de fin est très facile. Dans pharo un chaîne de caractères est un tableau de caractères dont le premier caractère est d'indice 1. Le début doit donc correspondre à la position du premier caractère du token, dans la chaîne et la fin à celle du dernier caractère du token. On va donc tester cela pour tous les types de tokens et dans des configurations où il y a plusieurs tokens. (cf Annexe - 1 - token start and stop. p.38)

2) Analyse du code

Le scanner est un automate initialisé avec un ReadStream qui est ensuite analysé caractère par caractère. Pour avancer dans le scan, il y a deux méthodes. Une méthode est utilisée par le parser pour récupérer le prochain token : next. L'autre méthode est utilisée par le scanner à l'intérieur des méthodes de scan pour passer au caractère suivant : step.
L'appel à step change deux variables d'instance : currentCharacter et characterType. La première est le caractère à analyser et la seconde, sa spécification dans Pharo. Par exemple, '(' est un caractère spécial, 'a' est un caractère alphabétique, '>' est un caractère binaire, etc....
Je souhaite préciser que dans Pharo l'undercore est un caractère alphabétique.
La méthode next déclenche la création d'un nouveau token. Pour chaque token, le scanner entrepose la valeur dans un tampon qui doit donc être remis à zéro au début de la méthode.

Puis, on enregistre la position actuelle dans le ReadStream pour connaître le point de départ du token. Si la spécification du caractère courant est EOF, cela signifie qu'il ne reste plus rien à scanner et on retourne donc un EOF token.
C'est, ici, l'un des premiers changements que j'ai effectué car c'était la classe abstraite RBToken qui était utilisée. Pour des raisons de lisibilité, je l'ai remplacé par un RBEOFToken. Ce qui a eu un impact sur la méthode atEnd du parser qui comparait la classe du token courant à celle du token abstrait. Ce qui a été remplacé par l'ajout d'une méthode isEOF. (cf Annexe - 2 – EOF token. p.38)
Pour finir, la méthode scan le token, retire les séparateurs et ajoute les commentaires au token.

3) Scanner un token

La méthode utilisée pour créer un token est scanToken. L'image ci-dessous représente tous les chemins permettant de créer un token. Quand la spécification du caractère suivant ne permet pas de continuer dans la branche courante, le token est retourné.

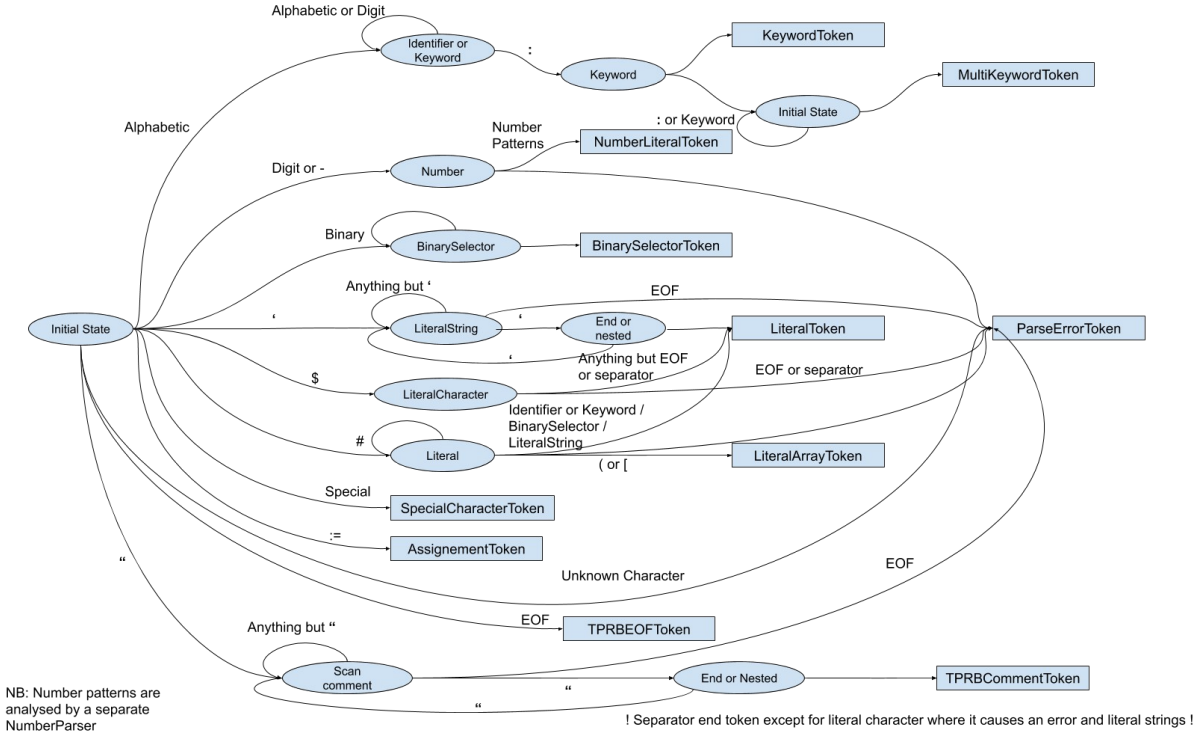


figure 2

4) Retirer les séparateurs

La méthode utilisée s'appelle stripSeparators et considère que les commentaires font partie des séparateurs. Elle fonctionne en avançant dans le ReadStream tand que le caractère courant est un séparateur. Lorsque ce n'est plus un séparateur, si le caractère est une guillemet, le commentaire est scanné et une fois le commentaire terminé, on recommence à retirer les séparateurs. Il s'agit également d'un changement que j'ai réalisé. Auparavant, le commentaire n'était pas scanné mais retiré et seul un intervalle était récupéré allant du début à la fin du token (guillemets non comprises). J'ai modifié cela pour avoir un RBCommentToken ayant pour valeur tout le contenu du commentaire. Le fonctionnement de scanComment est décrit ci-dessus (il s'agit du même fonctionnement que pour une chaîne de caractère, cf Annexe – 3 – scanComment, p.39). Ce commentaire est ensuite stocké dans le token précédent.

3. Parser

Nous avons, maintenant, vu comment sont créés les tokens et le parser fait la même chose que le scanner avec des token pour créer des nœuds au lieu de caractères pour créer des tokens. Toutefois, là où le scanner scanne de façon linéaire, le parser non.

1) Analyse du code

Comme le scanner, la méthode utilisée pour aller au token suivant est step. La première chose que cette méthode fait est d'extraire les commentaires du token actuel pour construire un nœud RBComment qui est ensuite stocké dans une collection (dans une variable d'instance) pour être rattaché à un nœud par la suite.
Après la méthode regarde s'il y a déjà un token de construit (contenu dans la variable d'instance nextToken) pour être placé en token actuel. Mais par défaut, le token suivant n'est pas encore créé et donc on en génère un nouveau en faisant appel à la fonction next du scanner.

2) Nodes

RBMethodNode : il s'agit du nœud le plus haut de l'AST, chaque nœud de l'arbre peut le

retrouver en utilisant la méthode « methodNode ».
Variables utiles : body → le reste de l'AST, selector → le nom de la méthode, arguments → le nom des variables fournies à la méthode, pragmas → les pragmas.
Méthode utile : doSemanticAnalysis → cette méthode fait l'analyse sémantique de tout l'AST.

RBSequenceNode : il s'agit du nœud contenant une liste de déclarations. Il est utilisé par les nœuds de méthode et block.
Variables utiles : statements → la collection de nœuds, temporaries → les variables temporaires associées au nœud duquel il fait partie, leftBar → début et rightBar fin de l'interval du code source contenant les variables temporaires.

RBComment : il s'agit du nœud de commentaire, pouvant être contenu dans n'importe quel autre nœud dans la variable d'instance comments.
Variable utile : contents → le contenu du commentaire

RBReturnNode : il s'agit du nœud indiquant le retour de la valeur qu'il contient. A l'exécution, ce nœud signifie la fin de la méthode.
Variable utile : value → le nœud de valeur dont le résultat sera retourné.

RBValueNode : il s'agit du nœud abstrait superclasse des tous les nœuds de valeur. Tous les nœuds qui vont suivre sont ses sous-classes.
Variable utile : parentheses → permet de dire si le nœud est parenthésé ou non et de donner le position

RBAssignementNode : il s'agit d'un nœud d'affection d'une valeur à une variable. La valeur peut être n'importe quel nœud de valeur.
Variables utiles : variable → la variable à laquelle on affecte la valeur, value → la valeur

RBArrayNode : il s'agit d'un tableau contenant des déclarations. Les valeurs conservées dans ce tableau sont donc les résultats de ces déclarations.
Variable utile : statements → la collection des déclarations composant le tableau

RBBlockNode : il s'agit d'un petit nœud de méthode à l'intérieur de la méthode. Il renvoie soit la valeur contenu dans un nœud de retour (terminant la méthode) ou la valeur de sa dernière déclaration. Elle peut contenir ses propres temporaires mais pas de pragmas.
Variables utiles : body → le contenu de la mini-méthode

RBMessageNode : il s'agit d'un nœud composé d'un receveur et d'un sélecteur. Le receveur peut être n'importe quel nœud de valeur sauf un nœud d'erreur et le sélecteur est le message envoyé au receveur. Ce message peut également être accompagné d'un argument s'il s'agit d'un sélecteur binaire et voir plusieurs s'il s'agit d'un sélecteur mot clé avec plusieurs mots clé.
Variables utiles : receiver → le receveur, selector → le sélecteur, keywordPositions → la collection des positions du début des mots clé ou du sélecteur, arguments → la collection des arguments.

RBCascadeNode : il s'agit d'un raccourci pour les messages. Il permet de créer des nœuds de messages avec le même receveur.
Variable utile : messages → la collection des nœuds de message de la cascade

RBLiteralNode : il s'agit de la superclasse abstraite de conteneurs de valeurs.

RBLiteralArrayNode : il s'agit d'un tableau de valeurs littérales.
Variable utile : contents → la collection de valeurs, isByteArray → booléen indiquant si les valeurs contenues sont des octets

RBLiteralValueNode : il s'agit d'une valeur littérale.
Variables utiles : value → la valeur, sourceText → le texte original de la valeur

RBVariableNode : il s'agit du nœud contenant le nom d'une variable. (superclasse de nœuds spécialisés qui sont attribués après analyse sémantique)
Variable utile : name → le nom de la variable

RBParseErrorNode : il s'agit du nœud générique d'erreur. Il contient le code du nœud d'erreur et le message d'erreur.

RBEnglobingErrorNode : il s'agit de la sous-classe de RBParseErrorNode et la superclasse des nœuds d'ouverture ou fermeture manquante.
Variable utile : content → les nœuds parsés
3) Parsing

Nous avons vu pour la création de l'AST que parseExpression comme parseMethod utilisent parseStatements pour créer le contenu de le nœud de méthode. Ce que fait cette méthode est : parser les déclarations et les conserve dans un nœud de séquence. Ce ne sont cependant les seuls à utiliser cette méthode car les blocks on un contenu identique à celui d'une expression.

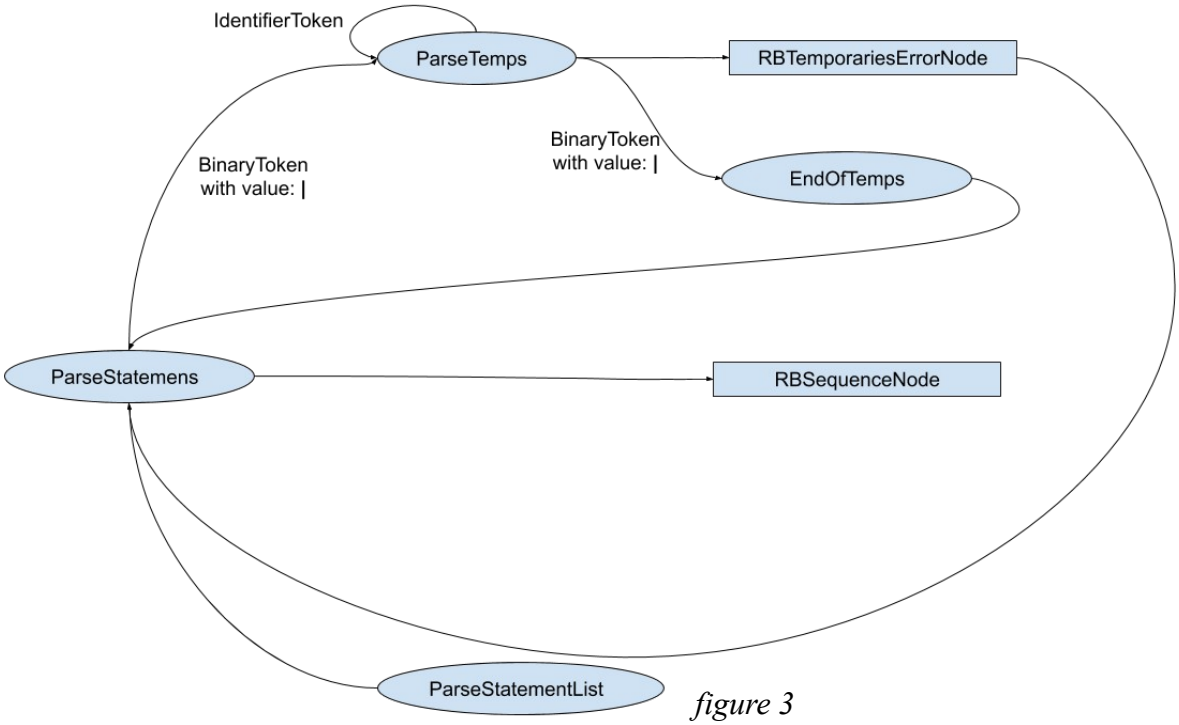


figure 3

La méthode importante, ici, est parseStatementList car elle est utilisée pour parser le code des RBMethodNode, RBBlockNode et RBArrayNode. Le nom complet de cette méthode est parseStatementList:into:et elle parse des déclarations et les ajoute dans une collection. Le premier argument de cette méthode est un booléen indiquant si l'on doit ou non parser les pragmas. Le second argument est un nœud (RBSequenceNode par défault) qui va contenir la collection de déclarations. Avant de retravailler la méthode, l'initialisation de la collection était faite en en créant une nouvelle. Ce qui signifie qu'on ne pouvait pas utiliser cette méthode pour parser à nouveau dans un nœud sans supprimer toutes les déclarations précédemment parsées. J'ai changé l'initialisation pour qu'elle se fasse avec la collection du nœud en argument. Ce qui permet, maintenant, de venir reparser à l'intérieur d'un nœud, ce qui sera utilisé plus tard dans ce rapport.

En simplifiant pour des raisons de compréhension, une déclaration est un RBValueNode que l'on peut retourner ou non. C'est pourquoi parseStatementList regarde avant de parser la déclaration si le token est un caractère spécial et que sa valeur est : ^. Je pense qu'ici, on pourrait créer un RBReturnToken pour éviter les vérifications multiples car je n'ai repéré aucune autre utilisation de ce caractère. Si on trouve un ^, on parse la déclaration, on la place dans un RBReturnNode que l'on met dans la collection et on arrête de parser les déclarations

pour le nœud donné en argument. Sinon, on met la déclaration dans la collection directement et on recommence le parsing avec toutefois une exception lorsque la fin de la déclaration n'est pas suivie par un point, ce qui force l'arrêt.

Tous les nœuds de valeur peuvent être des déclarations donc ce qu'on doit faire est de chercher quel est ce nœud de valeur lors du parsing.

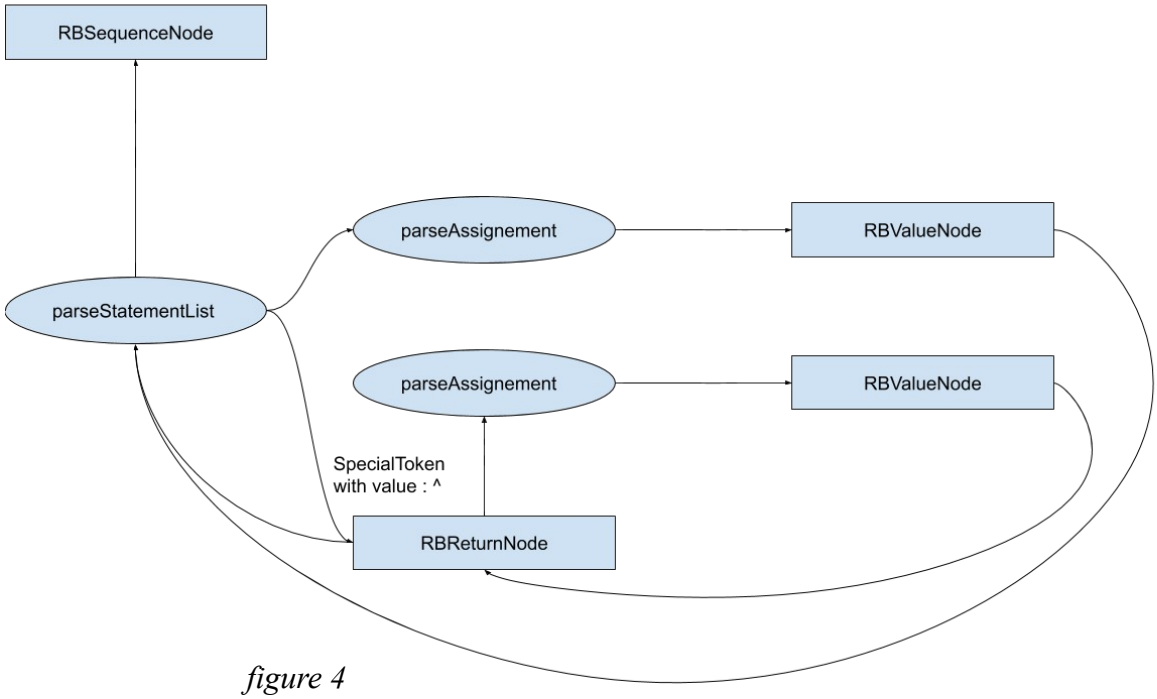


figure 4

De tous les nœuds de valeur, le plus gros et celui de l'assignation car il est composé d'un receveur qui est une variable et une valeur qui peut être n'importe quel nœud de valeur. C'est pourquoi on parse une assignation en premier. Dans la méthode parseAssignement, si on a un RBIdentifierToken suivit d'un RBAssignementToken, cela signifie que l'on a bien une assignation. Dans ce cas là, on parse le premier token pour obtenir un nœud de variable puis on parse un nœud de valeur avec parseAssignement et on crée un RBAssignementNode avec ces deux nœuds. Sinon, on continue la recherche avec le prochain nœud : la cascade.

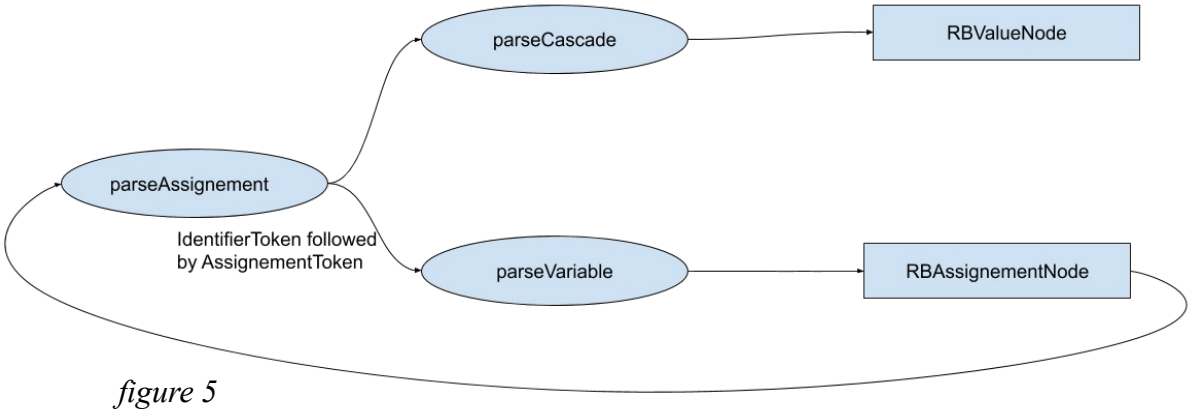


figure 5

La cascade est un nœud de message avec plusieurs sélecteurs séparés par des point-virgules et le receveur peut être soit un nœud de message soit un nœud de valeur plus petit. Pour vérifier si l'on a un nœud de cascade, on essaye de parser un nœud de message et si c'en est un suivi par un point-virgule, on recherche des sélecteurs (avec son argument s'il s'agit d'un mot clé ou d'un binaire)

J'ai réalisé un autre changement ici, parce qu'après avoir rencontré un point-virgule, s'il y avait une erreur dans la cascade, tout le début du nœud était perdu et on renvoyait un nœud d'erreur. Maintenant, s'il y a une erreur dans une cascade alors l'erreur et intégrée à la cascade et le parsing continue de telle façon qu'il renverra un nœud de cascade quoi qu'il arrive.

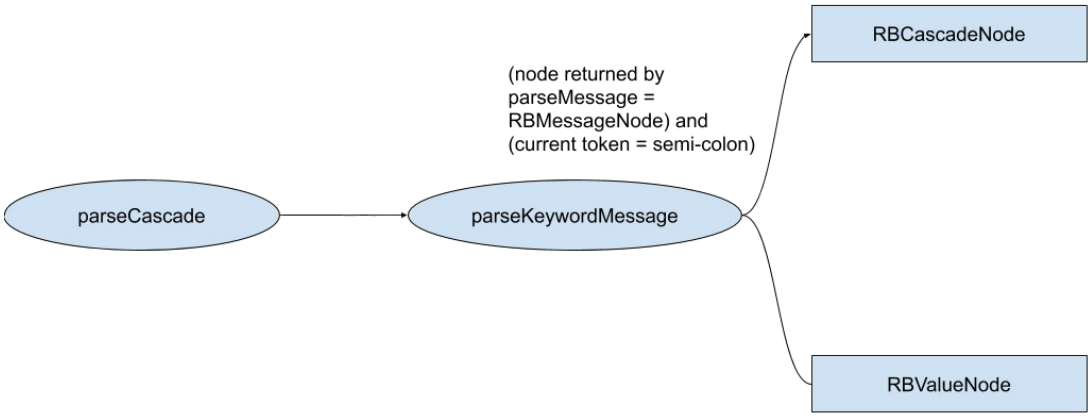


figure 6

(cf Annexe – 4 – parseCascade, p.39)

A l'intérieur de la cascade, le parser essaie tous les types de nœud de message en parsant les messages avec le receveur de celui déjà parsé. Si on obtient pas de nœud de message alors on crée un nœud d'erreur avec le token actuel. (Les méthodes utilisées pour parser les messages à l'intérieur de la cascade sont, dans l'ordre, parseUnaryMessageWith:, parseKeywordMessageWith: et parseBinaryMessageWith:)

C'est la façon dont un message est parsé qui lui donne sa priorité. La méthode parseKeywordMessage est le plus haut niveau de message (le dernier parsé) car elle utilise la méthode parseKeywordMessageWith: avec comme argument, le nœud retourné par parseBinarayMessage. Il se passe la même chose pour cette méthode, elle utilise parseBinaryMessageWith: avec comme argument, le nœud retourné par parseUnaryMessage, qui suit le même schéma.

Il y a deux possibilités, soit on forme un message avec l'argument en receveur soit on retourne l'argument. Dans parseKeywordMessageWith: lorsqu'on a un mot clé, sont argument provient

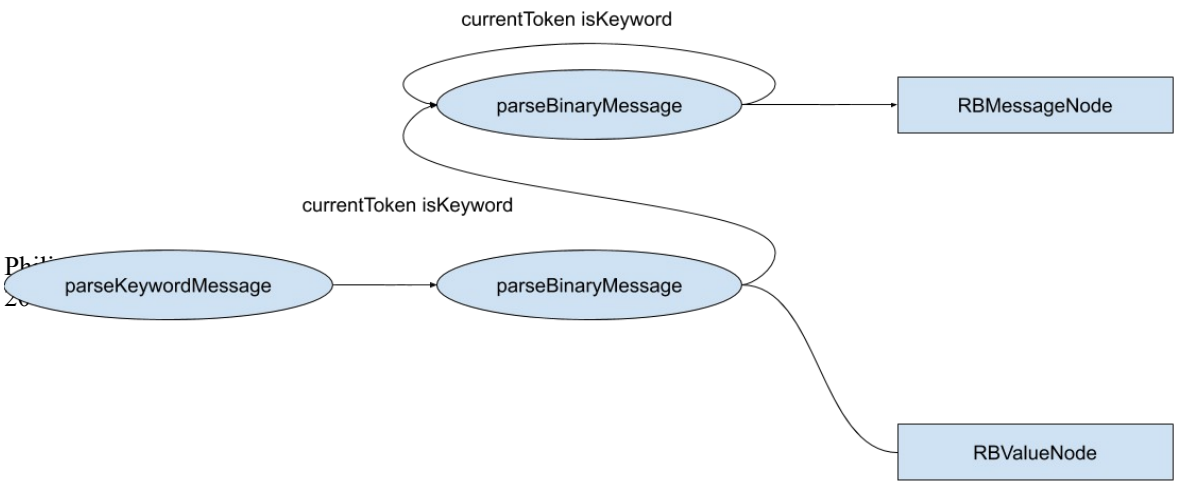


figure 7

aussi d'un parseBinaryMessage.

Il est intéressant de noter que peut importe le type de message, la classe ne change pas.

La méthode parseBinaryMessage fonctionne d'une façon très similaire sauf que là où un message mot clé pourra avoir plusieurs mots clé avec chacun son argument, le message binaire a un receveur, un sélecteur et un argument. Le receveur peut donc être un message binaire. (à chaque tour de boucle, on utilise donc le résultat du précédent comme receveur, voir ci-après)

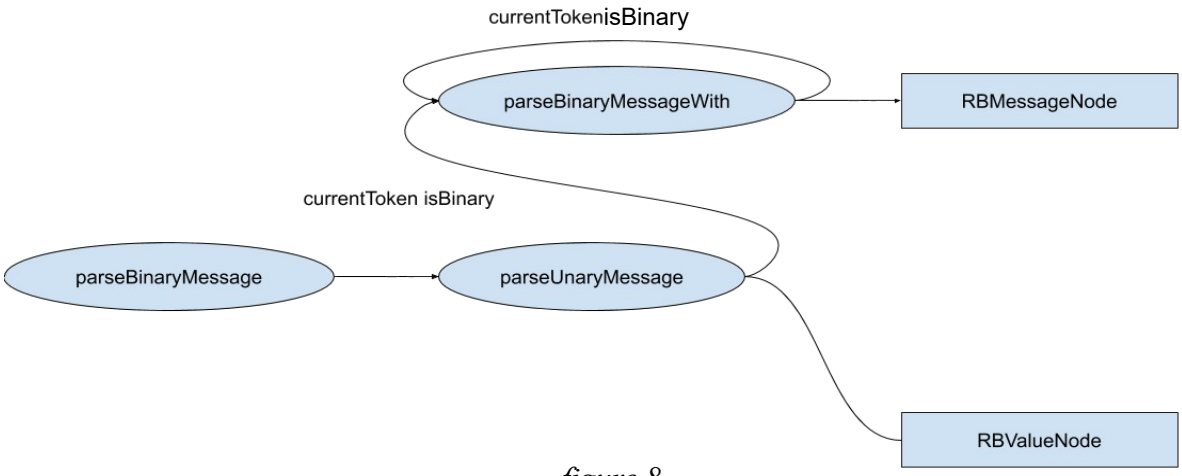


figure 8

Je préciserais, ici, qu'un argument est un nœud de valeur et que si on veut en obtenir un plus haut dans la hiérarchie, il faut utiliser des parenthèses. (c'est ce que l'on va bientôt voir)

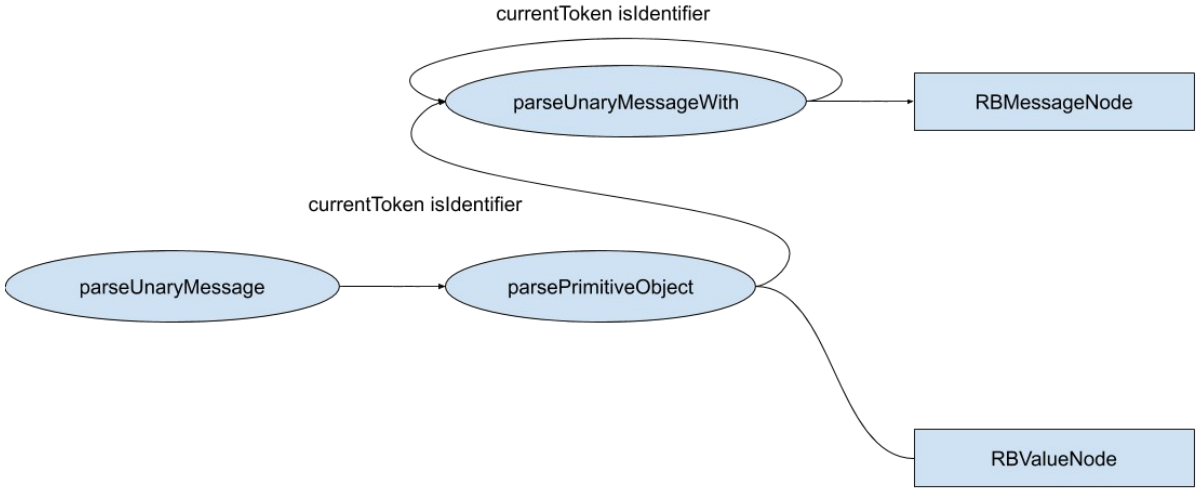


figure 9

Exemple de lecture : 3 plus: 1 + 4 double double – 3 * 4 moins: 1

→ 3 + (((1 + ((4*2)*2)) – 3)*4) – 1

Nous avons atteint un point où il n'y a plus de hiérarchie entre les nœuds de valeur restants. Nous allons donc devoir reprendre le même fonctionnement que le scanner et énumérer les cas possibles.

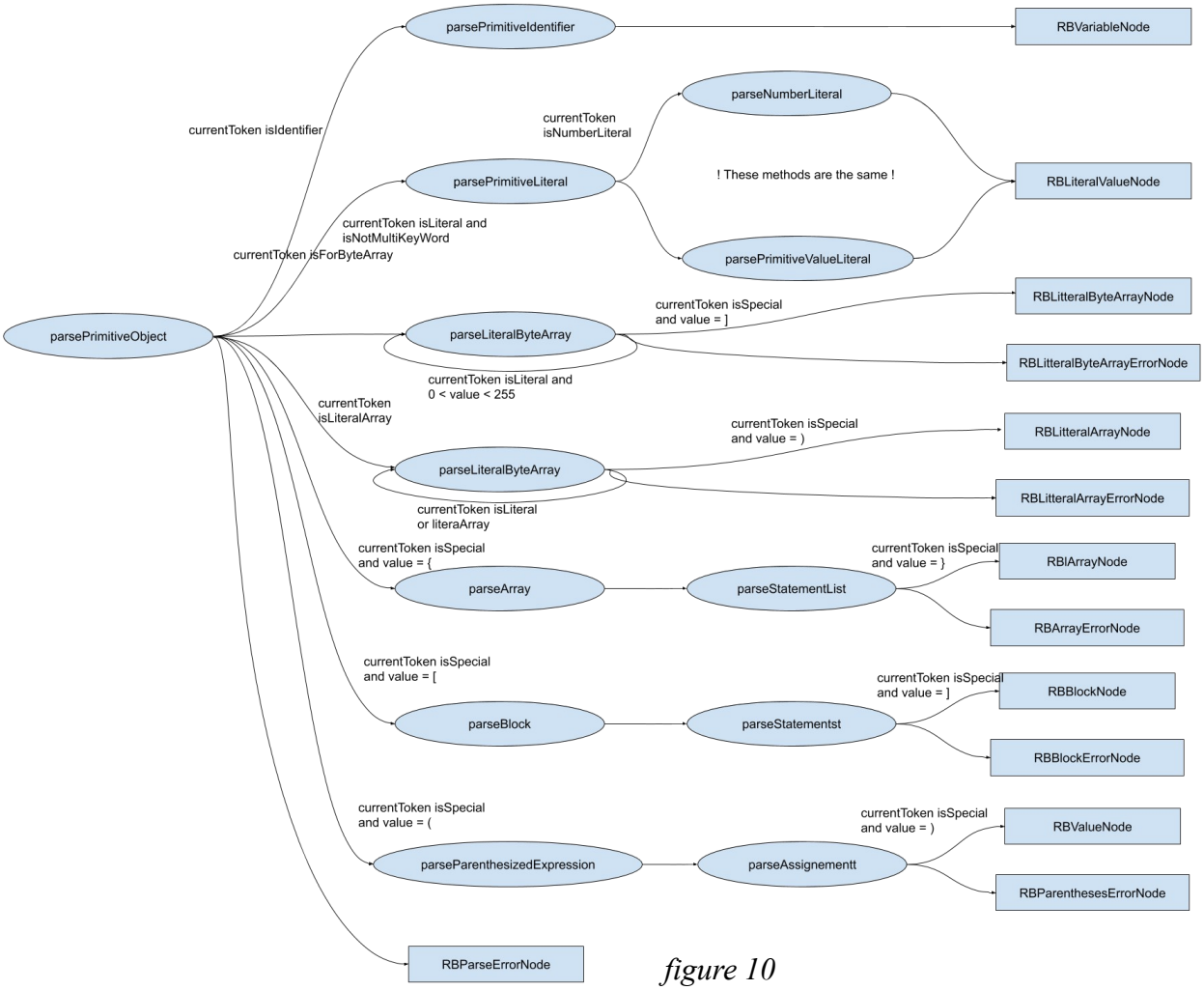


figure 10

Maintenant, il nous faut revenir à parseMethod et parseExpression car si une fois que tout cela est fini, on n'est toujours pas à la fin du code source, on utilise parseIncompleteExpression pour créer un nœud d'erreur. Auparavant, le nœud d'erreur était créé avec le reste du code mais j'ai modifié cette méthode pour qu'elle crée le nœud d'erreur avec le token actuel puis redémarre le parsing tant qu'elle n'est pas à la fin.

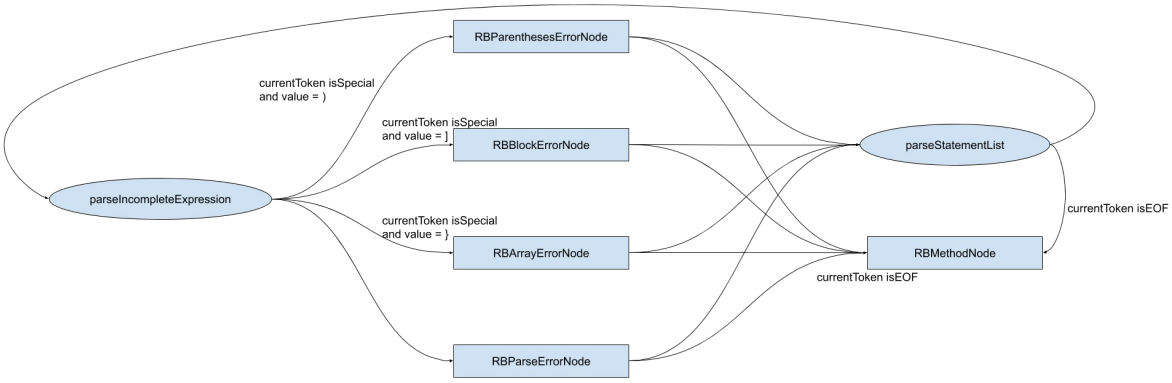


figure 11

On peut voir, ici, le plus grand changement dans le parser : les nouveaux nœuds d'erreur. Le parser avait seulement une classe d'erreur, RBParseErrorNode. (cf Annexe – 5 – parseIncompleteExpression, p.40)

4. Noeuds ajoutés

1) Noeuds d'erreur

Après avoir étudié le parser et le scanner, j'en ai conclu qu'il y avait trois types principaux d'erreur syntaxique.

Le premier type est celui des caractères erronés. Ils sont repérés par le scanner et dont des caractères interdits. Ils sont soit des caractères inconnus soit un # suivi d'un séparateur. Pour les caractères inconnus, on ne peut rien faire d'autre que les supprimer, les placer dans une chaîne de caractères, dans un commentaire ou d'ajouter un $ avant pour réparer l'erreur. # est plus complexe car si le token suivant remplis les critères pour être un littéral, on peut aussi supprimer les séparateurs pour réparer l'erreur.

Le second type est celui des erreurs de token inattendu. Cette erreur est repérée par le parser et a un message d'erreur tel que : « Variable name expected », « Literal constant expected »....
Ce type d'erreur signifie qu'en essayant de parser un nœud, le token reçu n'allait pas avec ceux attendus pour ce nœud. Malheureusement, cette erreur est très difficile à réparer car il y a beaucoup trop de possibilités de réparations différentes.
Le troisième type est celui des ouverture/fermeture manquantes :],),},>,|,', »,(,[,{.
Il y a deux types de fermeture manquantes : ' et » qui sont reconnues par le scanner et les autres reconnues par le parser. ' et » sont spéciales car ce qu'il y a à l'intérieur n'est pas du code et on ne peut donc pas déterminer un point d'arrêt autre que la fin de fichier. Cependant, les autres types peuvent être délimités par leur contenu, par la méthode utilisée pour parser le noeud.
Exemple : `(parentheses error. normal message.`
Et puisque le code à l'intérieur est censé être du code valide, on peut donc faire l'analyse syntaxique et sémantique pour les nœuds contenus dans l'erreur. C'est pour cela que j'ai créé un nœud d'erreur qui garderait ces nœuds parsés pour pouvoir en faire l'analyse et déterminer leur validité.

J'ai également modifié la méthode parseErrorNode. Elle agissait différemment pour le premier et le second type d'erreur. Pour le premier, elle utilisait le token d'erreur pour créer le nœud d'erreur avec sa valeur et son début. Pour le second type, elle recréait la valeur à partir de l'endroit où elle détectait l'erreur jusqu'à la fin du code source. Maintenant la valeur est celui du token actuel. Le troisième type d'erreur n'est maintenant plus géré par cette fonction mais parseEnglobingError. (cf Annexe – 6 – parsing of errors, p.40-41)

Pour finir, il y a des erreurs qui ne font partie d'aucune de ces catégories comme « cascaded message not allowed » ou « invalid token » qui sont à mi-chemin entre le premier et le second type puisque ce sont des nœuds interdits dans le contexte.

2) Noeuds de sélecteur

Dans l'AST, les sélecteurs sont un cas spécial car ce ne sont pas des nœuds. Il s'agit de chaînes de caractères (ByteString pour être plus précis) contenues dans le nœud de message. Les sélecteurs unaires et binaires sont utilisés tel quel mais les mot-clés sont collés les uns aux autres et doivent être redécoupés à l'aide d'une fonction présente dans la classe ByteString elle même. Cette implémentation est gênante pour la réparation car, pour les erreurs sémantiques, nous avons besoin de pouvoir les séparer. Par exemple :

doit devenir `(node isNumber: 3) ifTrue: [ true ] ifFalse: [ false ]`

Nous avons pensé, même si ce n'est pas encore implémenté que le sélecteur pourrait contenir un nœud d'erreur pour les réparations mais la façon dont fonctionne le parser rend cela difficile. J'ai donc ajouté un nœud sélecteur : RBSelecteurNode que j'ai ensuite divisé en trois sous-classes : RBUnarySelectorNode, RBBinarySelectorNode et RBKeywordSelectorNode. Unaray et Binary n'ont pas encore de différence mais Keyword possède une collection contenant chaque mot clé.

II – Retour en temps réel

1. Calypso

Dans la machine virtuelle de Pharo, c'est un module nommé Calypso qui se charge de l'interface utilisateur du navigateur de classe. Elle permet de naviguer entre les paquets, les classes et les méthodes. Lors de la sélection de ces classes ou ces méthodes, l'interface ouvre des onglets permettant la création ou la modification de méthodes. Ces onglets, ouverts par Calypso, sont gérés par des outils qui permettent l'affichage des données en temps réel. Dans le navigateur de classe, il y a 3 types d'onglets. D'abord l'onglet « Comment » qui contient la documentation de la classe. Il a pour but de donner une brève description de cette dernière et n'est donc pas régis par une quelconque contrainte de langage. Ensuite l'onglet classe qui permet de créer de nouvelles classes et contient des informations telles que son nom, le nom de la superclasse, du paquet auquel elle appartient, ses variables d'instances et ses variables de classe. Enfin, les onglets de méthode, ce sont eux qui nous intéressent. Ces onglets sont gérés par la classe abstraite « ClyMethodToolMorph ». Il y a deux types d'onglet pour les méthodes : il y a l'onglet de création de méthode (ClyMethodCreationToolMorph) et l'onglet de modification de méthode (ClyMethodCodeEditorToolMorph). Dans la pratique, il y a très peu de différence entre ces deux onglets car la syntaxe à utiliser est la même et qu'ils ne peuvent être ouverts que lorsqu'une classe est sélectionnée car ils sont liés à celle-ci. La seule différence notable est que l'un possède du code déjà compilé sur lequel s'appuie le menu contextuel alors que l'autre non.
Pour le moment, le seul retour en temps réel sur les erreurs est la stylisation du texte. La stylisation du texte est gérée par un visiteur (SHRBTexteStyler) qui parcourt l'arbre syntaxique du code courant et attribue des couleurs au texte suivant le nœud visité. Ainsi, un nœud de variable verra son texte colorié en bleu, un sélecteur en noir et une erreur en rouge.

2. IconStyler

Les méthodes disposent déjà d'un outil pratique qui est l'IconStyler. Celui est utilisé lors de la compilation pour fournir des informations visuelles sur des éléments importants. Il peut s'agir

d'avertissements pour dire que la méthode va s'arrêter en cours d'exécution, par exemple s'il y a un breakpoint ou un appel à une méthode inconnue. Ce qui permet de rapidement identifier la portions du code en question et de pouvoir la réparer rapidement. Mais il peut s'agir également de points d'attention sur des éléments douteux dans la méthode, par exemple si on définit une variable temporaire et qu'on ne l'utilise pas.
Il y a deux éléments utilisés pour la stylisation visuelle : l'icône qui est affiché dans la marge à gauche de la fenêtre et un styliseur de texte qui permet de le surligner ou le souligner.
IconStyler est un outil pratique car il s'agit d'un visiteur qui va parcourir l'AST et qui va décider si le nœud visité doit être stylisé ou non. Ce sont ses sous-classes qui vont se charger de définir quels nœuds doivent être stylisés et comment. Lors de son utilisation, IconStyler va donc faire parcourir l'AST à toutes ses sous-classes.
Chaque sous-classe a pour responsabilité de redéfinir la méthode « shouldStyleNode : ». Cette méthode prend un nœud en argument et doit renvoyer un booléen pour dire si le nœud valide ou non le(s) critère(s) pour être styliser. Par exemple, si on veut styliser les erreurs syntaxiques, on lui demande s'il s'agit d'un nœud d'erreur. Ensuite on peut personnaliser la sous-classe en changeant l'icône à afficher, si le texte doit être surligné ou souligné et en quelle couleur.

3. Modifications apportées

Pour pouvoir modifier les méthodes à mon grès sans abîmer l'UI, j'ai créé des copies des classes responsables de la création et de la modification de méthodes. J'ai fais de même pour l'IconStyler, cette fois-ci pour avoir le moins de sous-classe possible et éviter de trop nombreuses visites de l'AST.
Dans les outils de gestion de méthode, il y a une méthode nommée « textChanged : » qui reçoit un caractère comme argument. Cette fonction est chargée d'intégrer le caractère au texte déjà présent à chaque fois qu'il y a une entrée sur le clavier. Je l'ai donc modifiée pour y ajouter mon IconStyler. Comme dis précédemment, c'est un visiteur qui parcours l'AST et comme à chaque caractère on peut modifier tout l'AST, il faut parser le code à chaque fois.
A chaque entrée de clavier, on reparse donc l'AST, on refait son analyse sémantique et on le visite avec l'IconStyler. (cf Annexe – 7 – styling, p41-42)

III – Réparation d'erreur

1. Recherche de réparation

Pour la réparation d'erreur, j'ai réalisé une recherche primaire sur les erreurs communes réalisées durant l'écriture de code, à la fois syntaxique et sémantique.

1) Analyse sémantique

self, super et thisContext ne peuvent être rien d'autre que des variables. C'est pourquoi, si on les rencontre en tant que sélecteur dans un message, cela signifie qu'il y a probablement un point manquant juste avant.

Exemple :

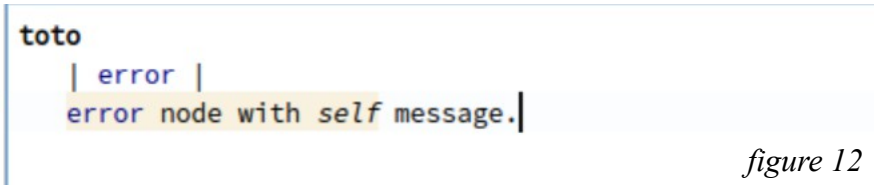


figure 12

Réparation :

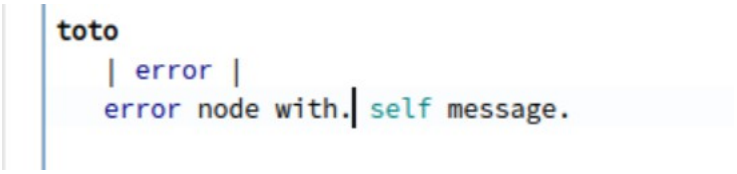


figure 13

Les sélecteur mot clé peuvent être difficiles car si on a un autre message mot clé en argument, il faut obligatoirement des parenthèses. Les cas les plus simples sont les méthodes ifTrue:ifFalse: et assert:equals: car on sait qu'il ne peut pas y avoir d'autres méthodes utilisant ces mots clés.

Pour ifTrue:ifFalse:, les arguments sont des blocks donc on a juste à vérifier s'ils sont précédés par un autre mot clé. Auquel cas, il faut simplement rajouter des parenthèses au début de noeud de message et après l'argument précédant ifTrue:.

Exemple :

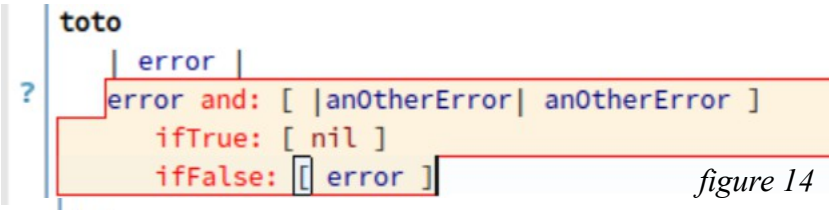


figure 14

Réparation :

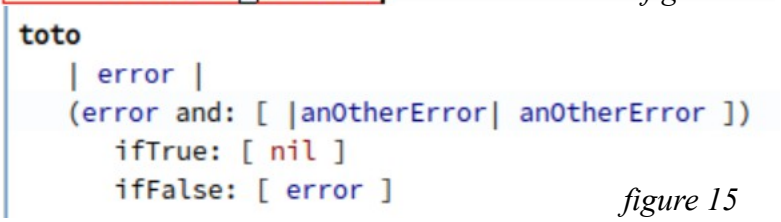


figure 15

Pour assert:equals:, on a pas cette chance il faut donc vérifier que ni le receveur ni les arguments ne contiennent de mot clé. Donc on vérifie si le sélecteur commence par assert. Si non, on fait comme pour ifTrue:ifFalse:. Puis on vérifie que assert: est suivi de equals:. Si non, on met des parenthèses entre les deux. Enfin (s'il y a un equals car assert: existe aussi), on vérifie s'il y a quelque chose après. Cependant le sélecteur equals: existe et il faut donc vérifier la possibilité que ce ne soit pas un assert:equals:.

Exemple :

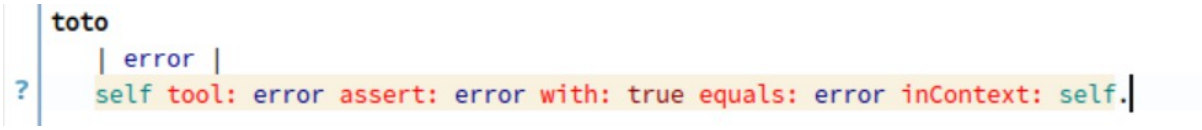


figure 16

Réparation :

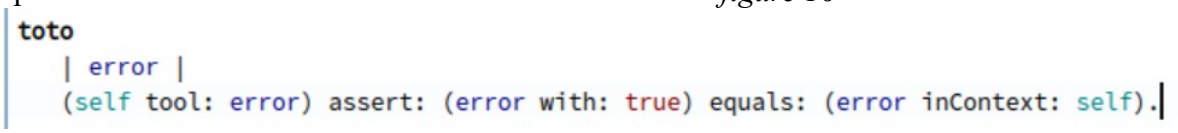


figure 17

Autre réparation :

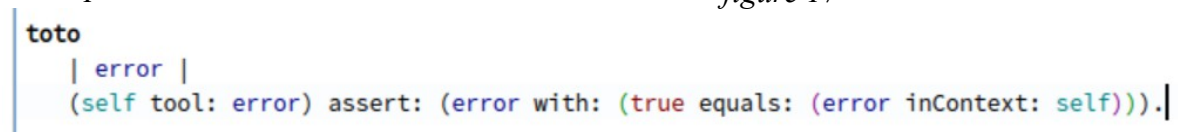


figure 18

2) Analyse syntaxique

Le meilleur indice qu'il manque un point est la présence de plusieurs noeuds d'erreur qui se suivent. Car si la déclaration s'arrête au mauvais endroit, il y a création d'une chaîne d'erreurs.
Exemple : Réparation :

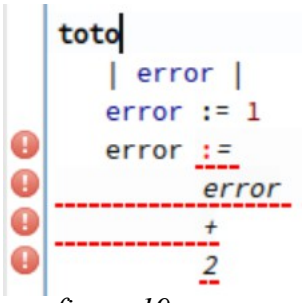


figure 19

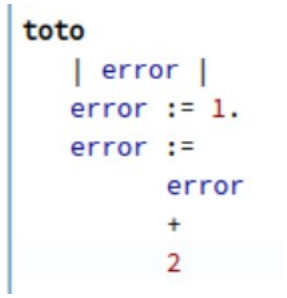


figure 20

Les noeuds d'erreurs englobantes contiennent des noeuds qui peuvent parfois être leur ouverture ou fermeture supposément manquante. Dans ce cas, il y a probablement un point qui manque avant.

Exemple 1 :

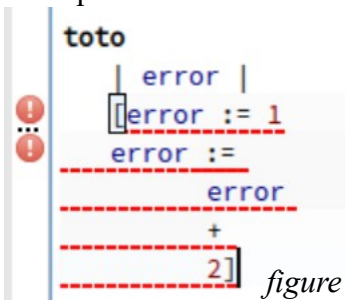


figure 21

Exemple 2 :

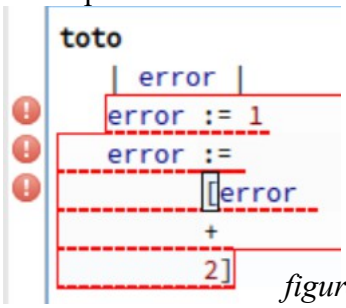


figure 22

Réparations :

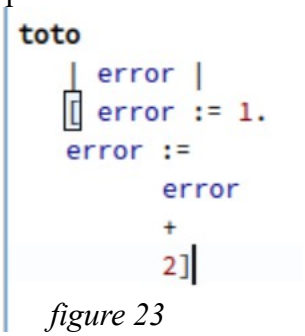


figure 23

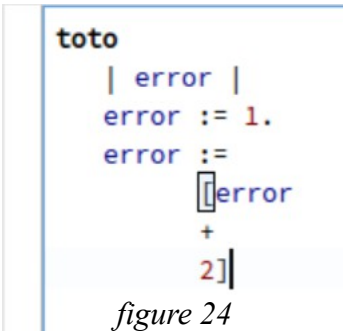


figure 24

Parfois, cette erreur n'est pas si simple et il y a un mot clé manquant ou un deux-points pour faire de ce noeud l'argument d'un message.

De même, quand il y a une erreur de type fin de déclaration, il manque probablement un point. La recherche la moins coûteuse sera alors de chercher juste avant et à chaque passage de ligne si l'ajout d'un point répare l'AST.

Exemple :

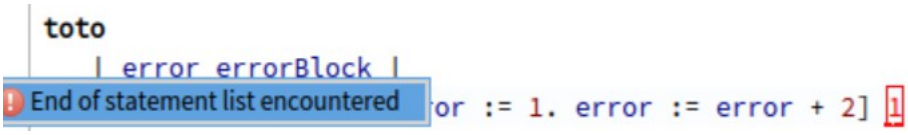


figure 25

Réparation :

```
toto
    | error errorBlock |
    errorBlock with: [ error := 1. error := error + 2]. 1
```

figure 26

Quand l'erreur fait partie du second type, cela signifie que soit le noeud doit être changé, soit on doit ajouter un noeud soit un point.

Exemple 1 :

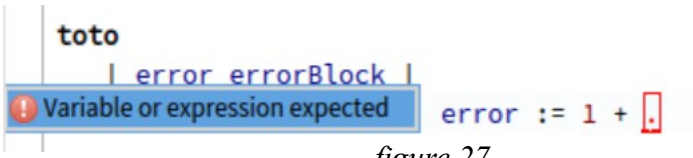


figure 27

Réparation :

```
toto
    | error errorBlock |
    error := 1 + 2.
```

figure 28

Exemple 2 :

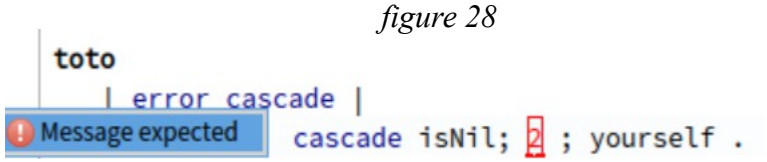


figure 29

Réparation :

```
toto
    | error cascade |
    cascade isNil; isCharacter ; yourself .
```

figure 30

Exemple 3 :

figure 31

Réparation :

```
toto
    | error cascade |
    cascade isNil; + 2; yourself .
```

figure 32

Ensuite, il y a les erreurs du premier type.

Exemple :

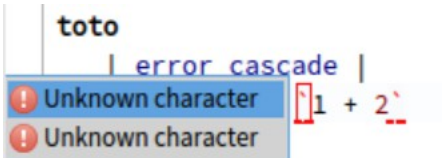


figure 33

Réparation 1 :

```
toto
    | error cascade |
    '1 + 2'
```

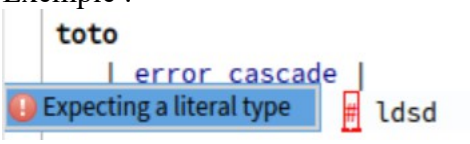
Réparation 2 :

```
toto
    | error cascade |
    1 + 2
```

figure 34

figure 35

Exemple :



Réparation:

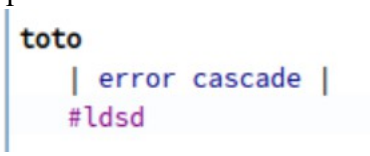


figure 36

figure 37

Pour finir, si on rencontre une erreur englobante, sans sa contrepartie à l'intérieur, on recherche à fermer le noeud à l'intérieur de son contenu.

Exemple:

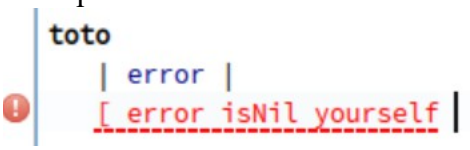


figure 38

Réparation 1 :

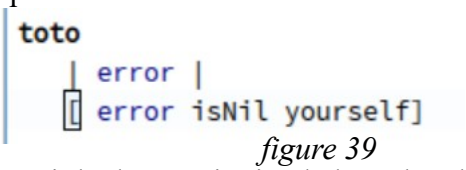


figure 39

Réparation 2 :

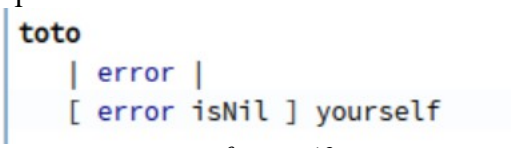


figure 40

Il s'agit là du premier jet de la recherche de réparation qui devra être complété par la suite.

Pour moi, la recherche de réparation doit être implémentée dans un visiteur qui visite l'AST de façon efficace pour récupérer le noeud à changer ou adjacent au changement. Une fois le noeud choisi, il est envoyé à une classe de réparation chargé de modifier le code source au niveau de ce noeud.

2) Classes de réparation

J'ai réalisé des classes de réparation, une pour chaque réparation sémantique qui ajoute la sémantique au contexte et quatre autres réparant le code. Il y a une classe pour enlever le noeud donné en argument, un pour ajouter du code après, un pour ajouter du code avant et un pour ajouter du code à deux positions différentes dans l'AST.

Les classes de réparation sémantique réutilisent tout simplement le code qui existait déjà pour faire ces réparations mais l'idée, ici, est de pouvoir réparer le code avec un raccourci clavier au lieu d'avoir une fenêtre qui s'ouvre et sur laquelle on doit sélectionner l'option et valider. (cf Annexe – 8 – semantic reparation, p42)
Les autres classes partitionnent le code et ajoutent la réparation entre ces parties. Une fois la réparation réalisée, on reparse l'AST, fournit le contexte de compilation et on refait l'analyse sémantique. (cf Annexe – 8 – code reparation, p43)
Si j'ai choisi cette méthode, c'est que je pense que l'on peut, à terme, dans la recherche de réparation, utiliser les différents AST récupérable pour réaliser une réparation statistique par rapport aux erreurs présentes. Ou encore, utiliser ces AST dans des tests préalablement écrits, permettant de définir si la réparation réalisée permet le bon fonctionnement de la méthode, en accord avec les résultats attendus.

IV - Automatisation des messages imbriqués

Pour la gestion des messages imbriqués, il y a deux choses à modifier. Premièrement, ce qu'on appelle les "smartCharacters". Dans Pharo, lorsque l'on sélectionne un pan de texte, si l'on tape un caractère, Pharo peut réagir de 2 façons différentes.
Si on tape une lettre, par exemple, la sélection va être effacée (comme dans tous les logiciels d'écriture) mais si on tape une parenthèse ouvrante alors Pharo placera une parenthèse

ouvrante au début de la sélection et une parenthèse fermante à la fin de la sélection. C'est cela les "smartCharcters", si on attend une fermeture au caractère tapé, on la place à la fin de la sélection.
Leur problème était qu'ils ne s'occupaient pas de ce qui est à l'intérieur de la sélection donc pour le cas des commentaires et des chaînes de caractères, s'il y en avait une dans la sélection, on se retrouvait avec du code cassé et pour le réparer il fallait les échapper manuellement. Ce que faisaient les "smartCaracters" était de copier la sélection, ajouter les "smartCaracters" au début et à la fin de la copie puis remplacer la sélection par cette nouvelle chaîne.
Dans Pharo, échapper des commentaires et des chaînes n'est pas compliqué. Il suffit de doubler chaque guillemet ou apostrophe (les guillemets sont pour commenter et les apostrophes pour chaîner).
Ce que j'ai fait est donc, en soit, très simple. J'ai tout simplement créé un cas particulier si le caractère tapé est une apostrophe ou une guillemet. Si on a l'un des deux, on double tous les caractères identiques au caractère tapé dans la copie de la sélection avant de lui ajouter l'ouverture et la fermeture.
J'aurais également pu vérifier si la sélection est à l'intérieur d'un commentaire ou d'une chaîne pour doubler l'insertion des "smartCaracters", cependant je pense que c'est beaucoup compliquer le processus pour un résultat mitigé. En effet, savoir si on se trouve dans un commentaire ou une chaîne nécessite de parser l'AST pour identifier le noeud dans lequel on se trouve. Néanmoins, on ne pourra pas détecté si notre sélection est sur une partie déjà imbriqué et donc on ne pourrait pas de combien de caractères il faudrait échapper. En plus de cela, il peut être intéressant de pouvoir scinder un commentaire en deux en sélectionnant la partie à décommenter.

Pour décommenter ou déchaîner avec les "smartCaracters", on se situe dans la même méthode. La sélection doit concerner tout ce qui se situe entre les "smartCaracters" et on doit entrer le caractère ouvrant ou le fermant. ("toto" en appuyant sur " on obtient : toto)
Il se passe alors la même chose, la sélection est copiée et on remplace la sélection et le caractère précédent et suivant par le code copié. J'ai donc également appliqué le traitement inverse sur cette chaîne. (cf Annexe – 9 – smartCharacters, p.43)

L'imbrication concerne aussi copier et coller. Le traitement de la sélection ne change pas des "smartCaracters". La différence est de repérer quand l'appliquer. Comme expliqué précédemment, il faut, ici, vérifier le noeud dans lequel on est placé.
Pour copier, si on se trouve à l'intérieur d'un commentaire ou d'une chaîne, il faudra alors enlever la moitié des caractères correspondant de la sélection avant de la mettre dans le clipboard.
Il y a tout de même une subtilité pour coller car lorsque l'on colle, la sélection courante est supprimée. Il ne faut donc pas savoir si la sélection est dans un commentaire mais si une fois la sélection supprimée, on se situe dans un commentaire.
(cf Annexe – 10 – copy paste, p.44)

V – Intégration

J'ai également réalisé l'intégration de mon travail à Pharo 9. Tout au long de mon stage, pour ne pas à avoir à gérer tous les problèmes liés au fait de modifier le système centrale du langage, j'ai travaillé sur une copie partielle du parser dans un package séparé. Il a donc fallu refaire tous les changements. Avec mon tuteur professionnel, nous avons alors établi un plan visant à ne pas se retrouver, au cours d'un changement, avec un parser cassé et donc une

impossibilité de faire quoi que ce soit. Pour cela, nous avons créé un petit outil permettant de comparer les méthodes de deux classes (cf annexe) et donc de repérer les méthodes ajoutées, changées et supprimées.
Logiquement, nous avons commencé par les changements les moins conséquents soient ceux du scanner et l'ajout des Tokens. Puis les ajouts de tous les nouveaux noeuds et des nouvelles méthodes sans les utilisés. Suivis par les changements des méthodes du parser et enfin, l'utilisation des nouveaux noeuds.
La modification du sélecteur est ce qui a le plus posé problème que c'est un objet extrêmement utilisé et il a donc fallu adapter le nouveau noeud avec des changements temporaires pour qu'il agisse comme un ByteString le temps que toutes les utilisations du noeud soient implémentées dans le parser.
Une fois cela fait, il a fallu régler tous les conflits entre la nouvelle implémentation du parser et l'ancienne, notamment au niveau des tests. Il a fallu faire en sorte que tous les tests, de toutes les classes, de tous les packages cassés par mon implémentations soient réparés, soit en changeant le test car le résultat attendu n'était plus le même soit en ajustant le code du parser pour ne pas avoir d'impact négatif sur d'autres outils de Pharo.

Conclusion

Grâce à ce stage, j'ai pu comprendre en pratique l'importance de ce qui m'a été appris en cours. Tout d'abord au niveau de la création de tests, que ce soit avec la création des tests du parser ou avec les tests suivants permettant de savoir les résultats que l'on souhaite obtenir. J'ai également appris à faire des tests significatifs qui, à leur exécution, permettent d'identifier précisemment le problème. En somme, faire de nombreux tests spécifiques, compréhensibles

et adaptable car malgré les changements de l'objet, les tests resteront et devront pouvoir être relus et compris par d'autre. L'absence de commentaires dans le parser pour comprendre le code est également quelque chose qui a beaucoup freiné mon travail car j'ai du en refaire une étude complète. J'ai donc pu constater le temps que l'on peut passer à essayer de comprendre du code non commenté, et vu son importance. J'ai pu mettre mon enseignement à profit tout au long de ce stage que ce soit le cours de META qui m'a appris la syntaxe de Pharo, les cours de Programmation et Conception Orienté Objet qui mon permit de bien organiser mon code dans ce langage objet notamment avec les design pattern et les principes de programmation SOLID ainsi que le cours de programmation fonctionnel qui nous avait fait travailler sur les langages et les parser. Je suis également fier d'avoir pu participer à l'amélioration de ce système et de, je l'espère, avoir pu faciliter la vie de nombreux utilisateurs en leur offrant un feedback plus précis et avec plus d'information. Bien que je n'ai pas eu le temps de mettre en place un système de réparation, j'ai dégrossi la recherche d'erreur et réalisé des supports qui une fois finalisés et connectés deviendront un système de réparation fonctionnel. Le support de nested comments a été très amusant à réaliser et ma permis de prendre une pause nécessaire sur l'autre projet. Au delà de mes productions, grâce à ce stage, j'ai appris à appréhender un système "open source" comme celui-ci et voir et participer à de l'intégration de code. Mais je pense que ce que j'ai appris et qui me sera le plus utile dans le futur est à debugger du code. Surtout pendant l'intégration, j'ai appris à debbuger que ce soit grâce à l'outil fourni par Pharo ou, dans les cas le plus graves, en allant voir le fichier de log. Pour finir, j'ai appris à travailler avec github qui contient son lot de subtilités et difficultés, notamment lors du branchage et de la navigation entre différentes branches crées et maintenues sur différentes machines.

Bilan

Avec le confinement lié au Covid-19, les premiers mois de stage se sont passés en télétravail et cet isolement, avec les limites des moyens de communication ont fini par entraîner une grosse baisse de productivité et des retards dans le projet. Je suis donc légèrement déçu de ne pas avoir pu mener l'entièreté du projet à bien. Je suis cependant fier du travail accompli, je pense avoir bien respecté les principes SOLID et ai veillé à avoir les processus qui demandent

le moins de ressources possibles.

Mis à part la fin de la situation de confinement, qui m'était très pénible, j'ai beaucoup aimé ce stage, il m'a permis de voir les points que je maîtrise en gestion de projet et ceux qu'il faut que je travaille.

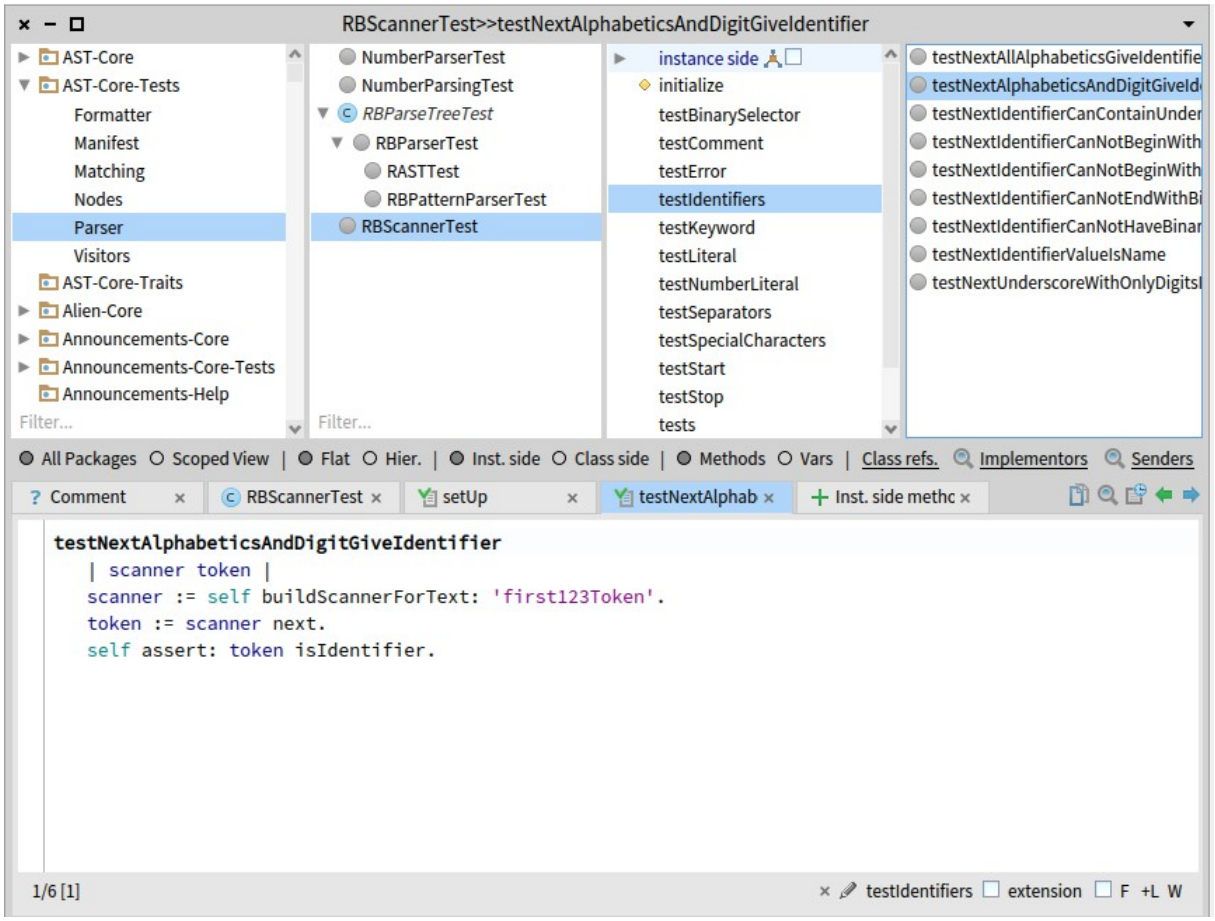
Cette expérience a renforcé beaucoup de thèmes vus en cours en voyant le côté pratique. Je pense qu'elle a également permis de développer des connaissances et de compétences peu vues en cours comme le debuggage.

Enfin, les compétences développées par ce stage sont basiques, ce qui en fait un très bon stage, peu importe ce que je fais derrière. Il s'agit des bases d'un langage, de programmation objet, d'intégration de code dans un système existant et d'utilisation d'outils répendus tel que github.

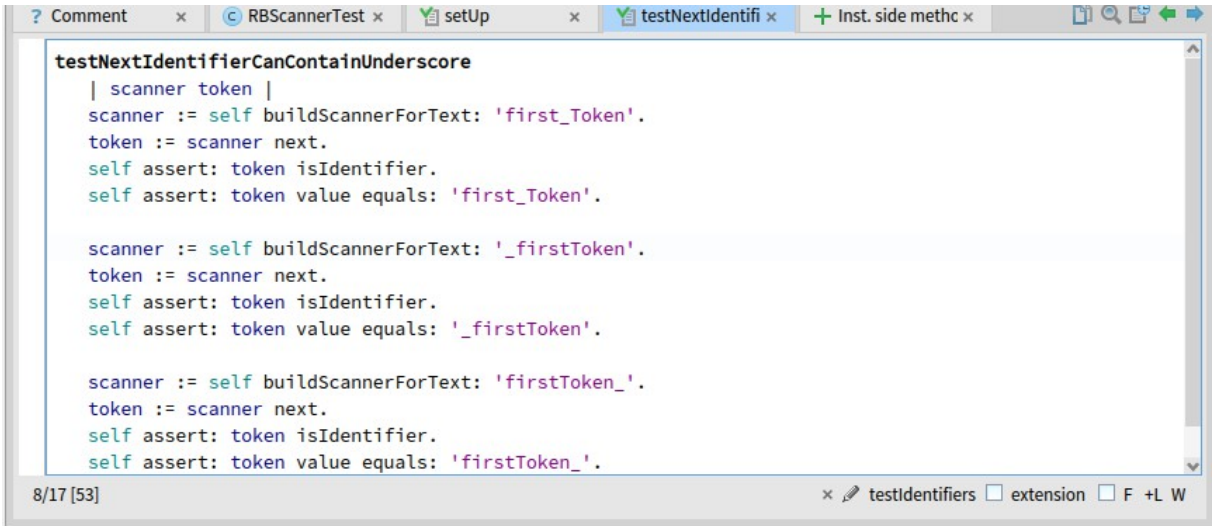
Annexe

Code :

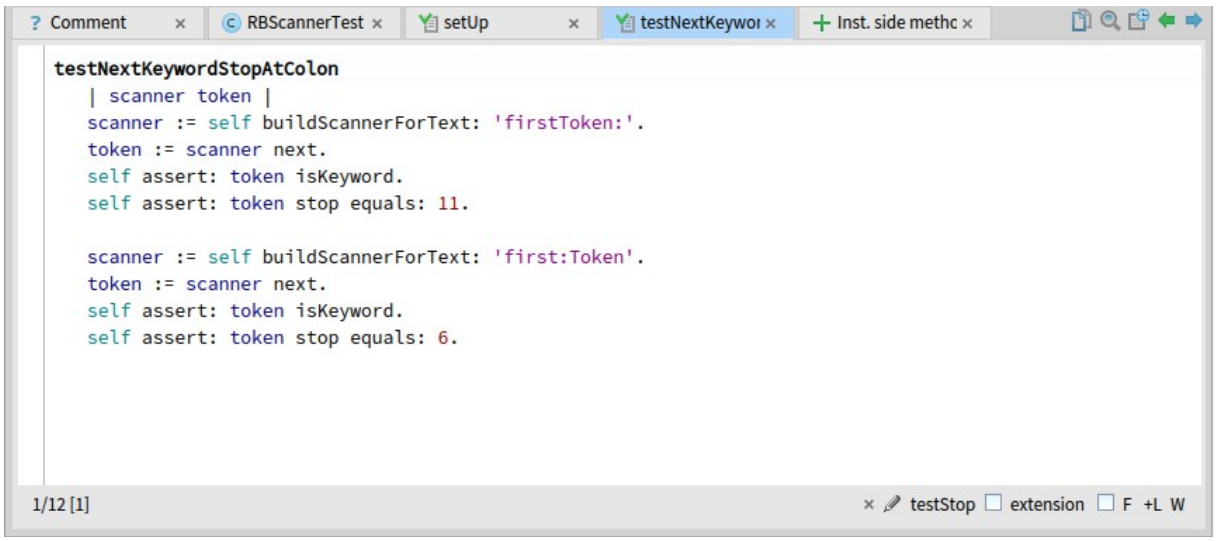
1 - token type :



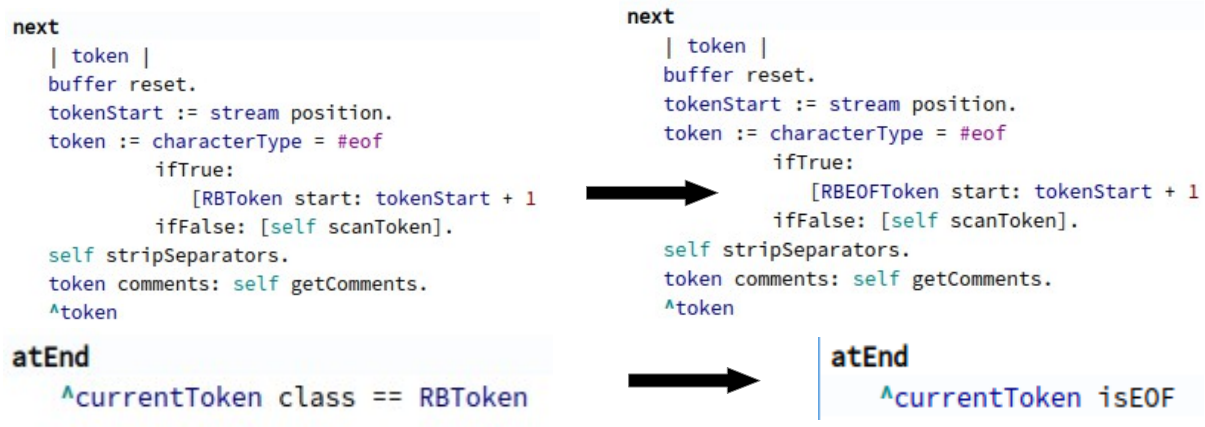
1 – token value :



1 – start and stop :



2 – EOF token :



3 – scanComment :

```
scanComment
    | start stop |
    start := stream position.
    buffer reset.
    self step.
    self atEnd
       ifTrue: [ ^ self scannerError: 'Unmatched " in comment.' ].
    [ currentCharacter = $" and: [ buffer nextPut: currentCharacter. self step ~= $" ] ]
       whileFalse: [ characterType = #eof
             ifTrue: [ ^ self scannerError: 'Unmatched " in comment.' ].
          buffer nextPut: currentCharacter.
          self step ].
    stop := self atEnd
       ifTrue: [ stream position ]
       ifFalse: [ stream position - 1 ].
    comments add: (RBCommentToken value: (buffer contents copyFrom: 1 to: buffer contents size -1)
start: start stop: stop).
```

4 – parseCascade :

```
parseCascadeMessage
   | node receiver messages semicolons |
   "Parse of the first message. It can be either unary, binary or keyword."
   node := self parseKeywordMessage.
   (currentToken isSpecial and: [ currentToken value = $; ])
      ifTrue: [ (node isVariable or: [ node hasParentheses ])
            ifTrue: [ self parserError: 'cascaded message not allowed' ] ].
   (currentToken isSpecial and: [ currentToken value = $; and: [ node isMessage ] ])
      ifFalse: [ ^ node ].
   receiver := node receiver.
   messages := OrderedCollection new: 3.
   semicolons := OrderedCollection new: 3.
   messages add: node.
   [currentToken isSpecial and: [currentToken value = $;]] whileTrue:
         [|newMessage| semicolons add: currentToken start.
         newMessage := nil.
         self step.
         self saveCommentsDuring:[
         newMessage := currentToken isIdentifier
                  ifTrue: [self parseUnaryMessageWith: receiver]
                  ifFalse:
                     [currentToken isKeyword
                        ifTrue: [self parseKeywordMessageWith: receiver]
                        ifFalse:
                           [| temp |
                           currentToken isLiteralToken ifTrue: [self patchNegativeLiteral].
         "Upon encountering an error in the cascade, it stores an error node as it would store a message node
          This can allow the possibility of fixing the cascade simply by replacing the false node by another."
                           currentToken isBinary ifFalse:
                              [ temp := self parserError: 'Message expected'.
                                (currentToken isSpecial and: [currentToken value = $;]) ifFalse: [self step.]
                                temp.]
                           ifTrue:
                              [temp := self parseBinaryMessageWith: receiver.
                               temp == receiver ifTrue:
                                 [ self parserError: 'Message expected']. temp]]]].

      self addCommentsTo: newMessage.
      messages add: newMessage].
   ^self cascadeNodeClass messages: messages semicolons: semicolons
```

5 – parseIncompleteExpression :

```
parseIncompleteExpression: priorStatementsNode
   "some incomplete expressions followed. Add a parserError node
 to the prior statements, but not if prioipriorStatementsNode  is already
 an errorNode"
    ^ priorStatementsNode isFaulty
      ifTrue: [ priorStatementsNode ]
      ifFalse:
         [ | errorNode |
         errorNode := self parserError: 'Unknown input at end'.
         errorNode ifNotNil: [ priorStatementsNode statements: priorStatementsNode statements , {errorNode} ].
         priorStatementsNode ]
```

```
parseIncompleteExpression: priorStatementsNode
   " This method is used when parsing an expression or a method, if the parsing of the list of
statements finishes but there are some tokens left to parse."
    | statements error |
   statements := priorStatementsNode statements copy.
   " Checks if the current token is a closure to create an englobing error node or a parse error node
otherwise. "
   error := self getErrorFromClosuresWithMissingOpenings: statements.
   error parent: priorStatementsNode.
   statements add: error.
   self step.
   priorStatementsNode statements: statements.
   ^self atEnd ifTrue: [ priorStatementsNode ]
               ifFalse: [ self parseStatementList: false into: priorStatementsNode.
                          self atEnd ifFalse: [ self parseIncompleteExpression: priorStatementsNode ]
                                     ifTrue: [ priorStatementsNode ] ]
```

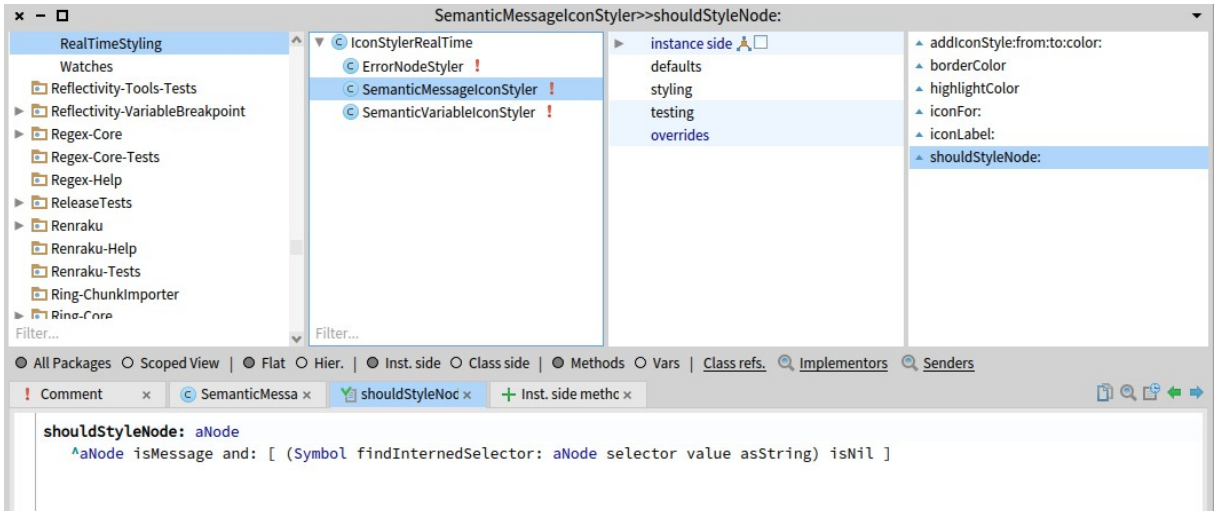
6 – parsing of errors :

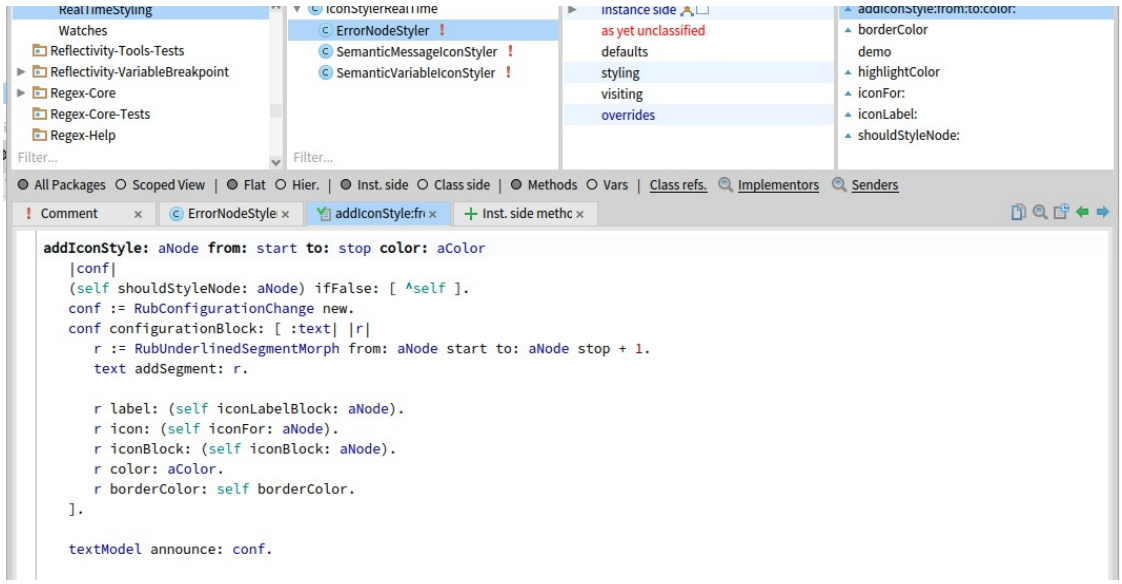
```
parseErrorNode: aMessageString
   | sourceString errorPosition |
   currentToken isError
      ifTrue: [ ^ RBParseErrorNode errorMessage: currentToken cause value: currentToken value at: currentToken
start ].
   errorPosition := self errorPosition.
   "the error at the end means in some cases that the start of the error is before"
   aMessageString = ''')'' expected'
      ifTrue: [ errorPosition := source findLastOccurrenceOfString: '(' startingAt: 1 ].
   aMessageString = '''|'' expected'
      ifTrue: [ errorPosition := source findLastOccurrenceOfString: '|' startingAt: 1 ].
   (errorPosition = 0) ifTrue: [ errorPosition := self errorPosition].
   sourceString := source copyFrom: errorPosition to: source size.
   ^ RBParseErrorNode errorMessage: aMessageString value: sourceString at: errorPosition
```

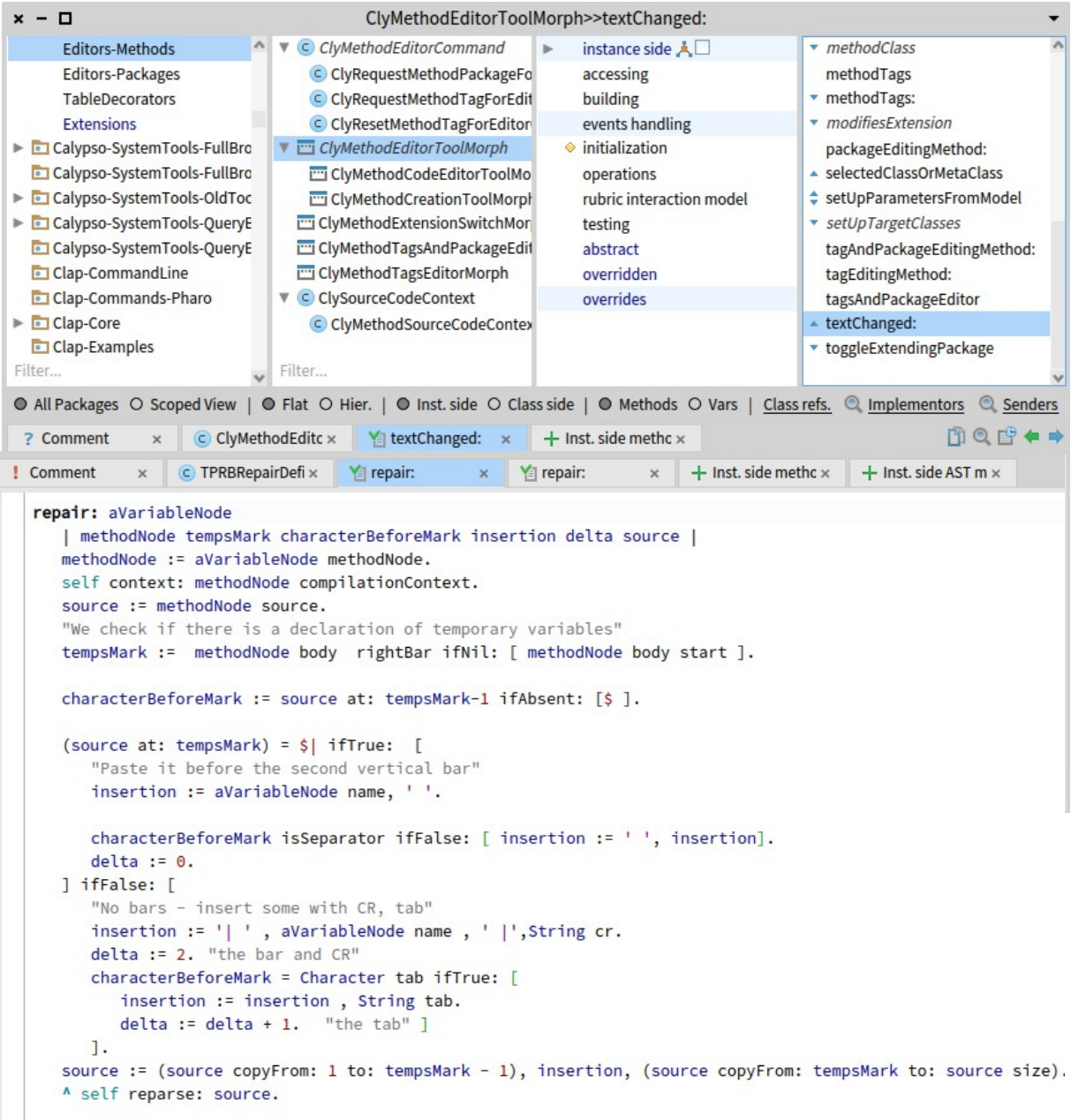
```
parseErrorNode: aMessageString
   | errorPosition|
   currentToken isError
      ifTrue: [ | errorNode|
                errorNode := RBParseErrorNode errorMessage: currentToken cause value: currentToken
value at: currentToken start.
                self step.
                ^errorNode].
   errorPosition := self errorPosition.
   (errorPosition = 0) ifTrue: [ errorPosition := self errorPosition].
   ^ RBParseErrorNode errorMessage: aMessageString value: currentToken value asString at: errorPosition
```

```
parseEnglobingError: aCollection with: aToken errorMessage: anErrorMessage
   | firstParse |
   firstParse := self parserError: anErrorMessage.
   ^[firstParse isParseError
      ifTrue: [ self englobingErrorNodeClass error: aToken withNodes: aCollection ]
      ifFalse: [ firstParse ]] on: Error do: [ firstParse ].
```

7 – styling :




RealTimeStyling
Watches
Reflectivity-Tools-Tests
Reflectivity-VariableBreakpoint
Regex-Core
Regex-Core-Tests
Regex-Help
ErrorNodeStyler
SemanticMessageIconStyler
SemanticVariableIconStyler
instance side
as yet unclassified
defaults
styling
visiting
overrides
borderColor
demo
highlightColor
iconFor:
iconLabel:
shouldStyleNode:
All Packages Scoped View | Flat Hier. | Inst. side Class side | Methods Vars | Class refs. Implementors Senders
Comment
ErrorNodeStyle
addIconStyle:fr
Inst. side methc
addIconStyle: aNode from: start to: stop color: aColor
|conf|
(self shouldStyleNode: aNode) ifFalse: [^self].
conf := RubConfigurationChange new.
conf configurationBlock: [:text| |r|
r := RubUnderlinedSegmentMorph from: aNode start to: aNode stop + 1.
text addSegment: r.
r label: (self iconLabelBlock: aNode).
r icon: (self iconFor: aNode).
r iconBlock: (self iconBlock: aNode).
r color: aColor.
r borderColor: self borderColor.
].
textModel announce: conf.


ClyMethodEditorToolMorph>>textChanged:
Editors-Methods
Editors-Packages
TableDecorators
Extensions
Calypso-SystemTools-FullBro
Calypso-SystemTools-FullBro
Calypso-SystemTools-OldToc
Calypso-SystemTools-QueryE
Calypso-SystemTools-QueryE
Clap-CommandLine
Clap-Commands-Pharo
Clap-Core
Clap-Examples
ClyMethodEditorCommand
ClyRequestMethodPackageFo
ClyRequestMethodTagForEdit
ClyResetMethodTagForEditor
ClyMethodEditorToolMorph
ClyMethodCodeEditorToolMo
ClyMethodCreationToolMorph
ClyMethodExtensionSwitchMor
ClyMethodTagsAndPackageEdit
ClyMethodTagsEditorMorph
ClySourceCodeContext
ClyMethodSourceCodeContex
instance side
accessing
building
events handling
initialization
operations
rubric interaction model
testing
abstract
overridden
overrides
methodClass
methodTags
methodTags:
modifiesExtension
packageEditingMethod:
selectedClassOrMetaClass
setUpParametersFromModel
setUpTargetClasses
tagAndPackageEditingMethod:
tagEditingMethod:
tagsAndPackageEditor
textChanged:
toggleExtendingPackage
All Packages Scoped View | Flat Hier. | Inst. side Class side | Methods Vars | Class refs. Implementors Senders
Comment
ClyMethodEditc
textChanged:
Inst. side methc
Comment
TPRBRepairDefi
repair:
repair:
Inst. side methc
Inst. side AST m
repair: aVariableNode
| methodNode tempsMark characterBeforeMark insertion delta source |
methodNode := aVariableNode methodNode.
self context: methodNode compilationContext.
source := methodNode source.
"We check if there is a declaration of temporary variables"
tempsMark := methodNode body rightBar ifNil: [methodNode body start].
characterBeforeMark := source at: tempsMark-1 ifAbsent: [$].
(source at: tempsMark) = $| ifTrue: [
"Paste it before the second vertical bar"
insertion := aVariableNode name, ' '.
characterBeforeMark isSeparator ifFalse: [insertion := ' ', insertion].
delta := 0.
] ifFalse: [
"No bars - insert some with CR, tab"
insertion := '| ' , aVariableNode name , ' |',String cr.
delta := 2. "the bar and CR"
characterBeforeMark = Character tab ifTrue: [
insertion := insertion , String tab.
delta := delta + 1. "the tab"]
].
source := (source copyFrom: 1 to: tempsMark - 1), insertion, (source copyFrom: tempsMark to: source size).
^ self reparse: source.

8 – semantic reparation :

8 – code reparation :

```
repair: aNode
   | beginning end |
   self context: aNode methodNode compilationContext.
   beginning := aNode start = 1 ifFalse: [ (aNode source copyFrom: 1 to: aNode stop) ]
                              ifTrue: [ '' ].
   end := aNode stop = aNode source size ifFalse: [ (aNode source copyFrom: aNode stop + 1 to: aNode source size) ]
                                         ifTrue: [ '' ].
   ^self reparse: (beginning , self replacement , end).
```

9 – smartCharacters

```
encloseWith: aMatchingPair
   "Insert or remove bracket characters around the current selection."

   | left right startIndex stopIndex oldSelection text newString |
   self closeTypeIn.
   startIndex := self startIndex.
   stopIndex := self stopIndex.
   oldSelection := self selection.

   left := aMatchingPair key.
   right := aMatchingPair value.
   text := self text.
   ((startIndex > 1 and: [stopIndex <= text size])
         and: [ (text at: startIndex-1) = left and: [(text at: stopIndex) = right]])
      ifTrue: [
         "already enclosed; strip off brackets"
         newString := (self shouldNestCharacter: left)
                  ifTrue: [ self unNesting: oldSelection with: left ]
                  ifFalse: [ oldSelection ].
         self selectFrom: startIndex-1 to: stopIndex.
         self replaceSelectionWith: newString ]
      ifFalse: [
         " Checks if the characters inside the selection need to be escaped or not. "
         newString := (self shouldNestCharacter: left)
                  ifTrue: [ self nesting: oldSelection with: left ]
                  ifFalse: [ "not enclosed; enclose by matching brackets"
                           (String with: left), oldSelection string, (String with: right) ].
         self replaceSelectionWith:
            (Text string: newString attributes: self emphasisHere).
         "we add the difference of the newString and the oldSelection, here, to ajust to eventual nesting.
         self selectFrom: startIndex+1 to: stopIndex+(newString size - oldSelection size - 2)].
   ^true
```

10 – copy paste

```
paste
   "Paste the text from the shared buffer over the current selection and
   redisplay if necessary.  Undoer & Redoer: undoAndReselect."
   | node charac nestedText |
   "Research of selected node for eventual nesting of comment or string."
   node := self bestNodeInTextAreaWithoutSelection.
   charac := self getNestingCharacterFromAst: node.
   nestedText := self stringNestedWith: charac.
   "Check if it is a Symbol that needs extra apostrophes to receive"
   "The commented code is a hint at how to nest strings in symbols. The solucion has not been found yet."
bestNodeInTextAreaWithoutSelection
   "Find the best node in the editor text area trimming the selection to prepare for eventual deletion"

   | node start stop intervalTrimmedText |

   start := self textArea startIndex.
   stop := self textArea stopIndex.
   intervalTrimmedText := start = stop
               ifFalse: [  | beginning end |
                           beginning := end := ''.
                           start <= 1 ifFalse: [ beginning := self textArea string copyFrom: 1 to: start - 1 ].
                           stop >= self textArea text size ifFalse: [ end :=  self textArea string copyFrom: stop + 1 to: self textArea text size ].
                           beginning,end ]
                ifTrue: [ self textArea string ].
   [node := RBParser parseMethod: intervalTrimmedText  onError: [
      RBParser parseFaultyExpression: intervalTrimmedText]
   ] on: Error do: [^nil].

   ^node bestNodeFor: (start to: start).
```

Ci-après, le rapport que j'ai réaliser pour l'étude du parser :

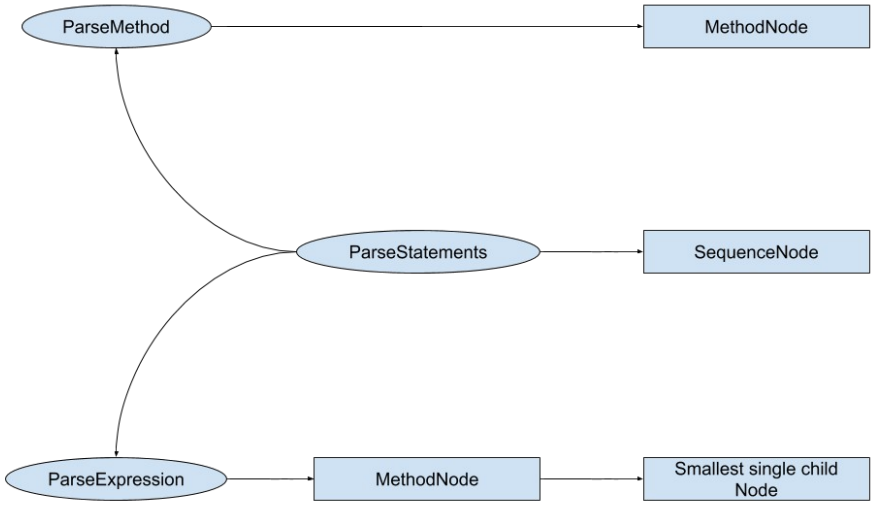
The Parser of Pharo

Pharo's parser is called RBParser and found in the package AST-Core. The goal of the parser is to analyse a string of code and create an abstract syntax tree, only containing the syntactic analysis of the code. The parser uses a tool called RBScanner to divide and classify the code into tokens and then uses the tokens to create the nodes composing the AST.

Summary

1) Construction of the AST:

The parser uses 6 public methods to create an AST (4 are often used: parseExpression, parseFaultyExpression, parseMethod, parseFaultyMethod).
The difference between parseExpression and parseMethod lies in its premise. ParseMethod is used in context of method creation or modification, so it expects the selector of the method and eventually pragmas. Whereas parseExpression is used in context like the one of the playground, where the goal of the code is only to be executed, not stored in a method. Nevertheless, the expression is kept in a generic method node with #noMethod for selector, but doesn't accept pragmas.
The returned node is also different parseMethod will always return a method node, but parseExpression will return the lowest node with a unique path.



NB: The child of a node is a subnode of this one. SequenceNode is always a single child of a MethodNode so ParseExpression never returns a MethodNode.

The difference between parse and parseFaulty is very simple. When using parse, if you encounter a syntax error, the parsing stops, no AST is returned and a SyntaxErrorNotification is raised.
With parseFaulty, if you encounter a syntax error, you create a parseErrorNode which can be put in the AST and the faulty AST is returned.

2) Scanner:

1- Tokens:

Like said previously, the parser uses a scanner to make a primary partition of the code in the form of tokens. Tokens are made of 4 important elements : their identification (Literal, Identifier, Keyword, Binary, EOF, etc...) represented by sub-classes of RBToken, their value,

their start and their stop.
Concerning identification, the abstract class RBToken implements a verification method for each sub-class. The method returns a boolean set by default to 'false'. Each sub-class overrides its corresponding method to set it to 'true'.
Concerning value, this is the most sensible part of the token because the scanner has very few cases of error. Scanner doesn't necessarily use separators to part tokens. If the current character doesn't fit the token being created, then it returns the token as is and start another one. This particularity allows a flexible programmation syntax. For example, 1+2, 1halt, (parentheses node) are valid messages. The value is the sub-string that compose the token.
The start and the stop refer to the position of the substring in the scanned code. For the most part, stop is not registered but deduced from the start point and the size of the value. However, there is an exception for the literals. Number literals are an exception in the scanner because the value is not a sub-string of the source, but an analysed numeric value by a class named NumberParser.

2- Going through the code:

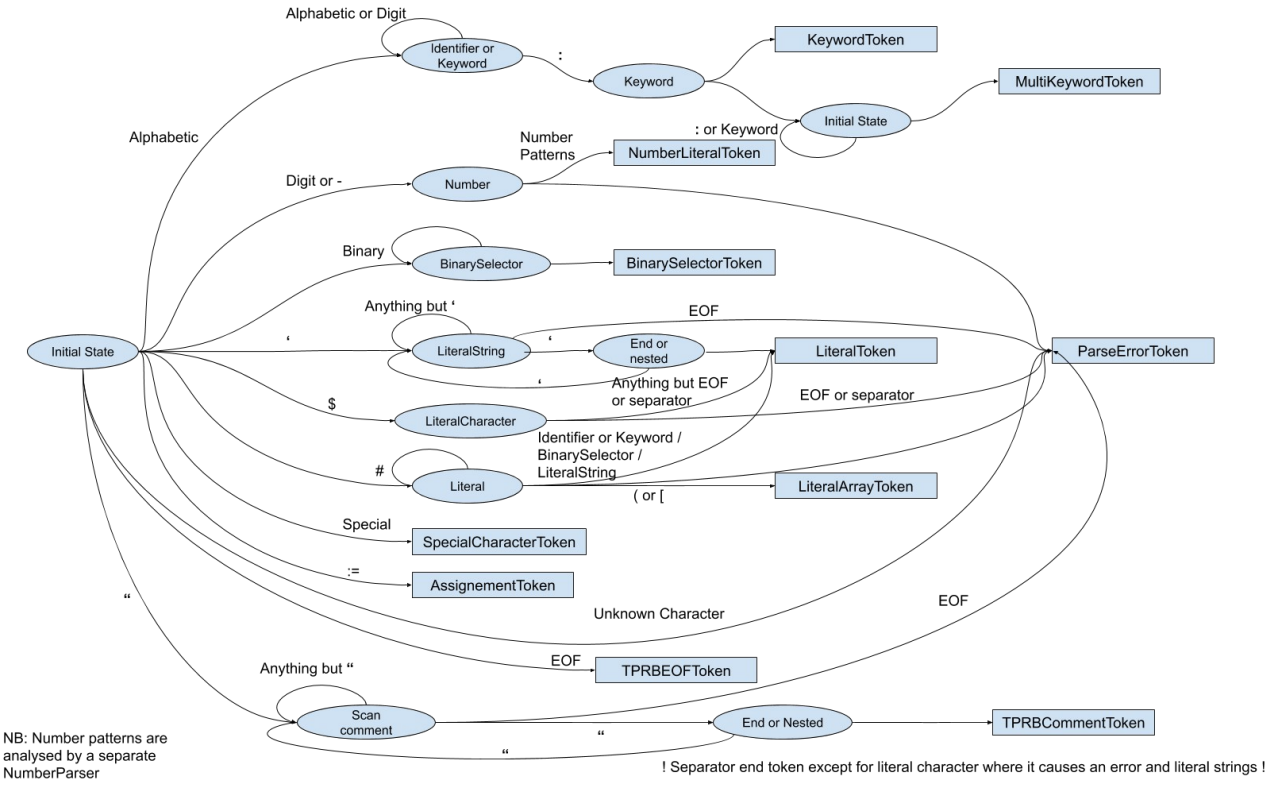
The Scanner is an automaton initialized by a ReadStream which is then analysed character by character. To advance in the scanning, there are 2 methods. One method is used by the parser to get the next token called 'next'. The other is used by the Scanner inside the scanning methods to get the next character called 'step'.

Step changes two instance variables: currentCharacter and characterType. The first is the analysed character and the second is its specification in Pharo. For example, '(' is a SpecialCharacter, 'a' is an alphabetical character, '>' is a binary character, etc.... I would like to precise that in Pharo, the underscore is considered as an alphabetic character.

The method next launches the creation of a new token. For each token, the scanner stores the value in a buffer so it is reset before doing anything. Then we store the current position in the stream to know the starting point of the token. If the current character type is EOF it means there are no more character to scan so we return an EOF token.
(Before it was the Abstract Class RBToken that was used, but for comprehension and simplification purposes I've changed it to an RBEOFToken. This had an impact on the method atEnd of the parser which verified the class the current token. This was changed to a call to a method called 'isEOF'.)
At last, the method scans the next token, strip the separators and adds the comments to the token.

3- Scanning a token:

The method used to create a token is scanToken. The following image represents all the paths towards creating a token. When the next character type doesn't fit in the current branch, the token is returned.



4- Stripping separators.

The method used is 'stripSeparators' which considers the comments as being separators. It works by advancing in the stream while the current character is a separator. When it is not anymore, if the character is a quote, it scans the comment and restarts stripping separators and comments.
This also is a change I made. Before, the comment wasn't scanned but stripped and instead of having a RBCommentToken fed to the previously created token, it was fed the interval of the comment position in the source code. The scanning of a comment works exactly like the scanning of a string (see above and replace apostrophes by quotes).

3) Parser:

We've now seen how tokens are created and the parser acts the same way the scanner does using tokens instead of characters. However, if the scanner is linear, the parser is not.

1- Going through the code:

Like the scanner, the method used to go through the tokens is 'step'. The first thing it does is to extract the current token's comments to construct comment nodes, which are stored in an instance variable to be then attached to another node.
Then it checks if there is a next token already built, if not the scanner is used to create a new one. By default the next token hasn't been scanned yet. The nextToken instance variable is only used by patching methods so a manually created token can be inserted before the current one without losing it (the current becomes next and the created becomes current) or inserted after (the created becomes next).

2- Nodes:

RBMethodNode : it is the highest node in the AST. Every node of the AST can find this one using the method 'methodNode'.
Useful variables : the rest of the AST (body), the name of the method (selector), the arguments that
can come with the selector (arguments), the pragmas (pragmas)
the original code used to make the AST (sourceCode) or the context of where to
compile the code (compilationContext).
Useful method : 'doSemanticAnalysis' this method can't be called from other nodes and allow to
link variables and messages to values and methods.
This node can be returned by any sub-node with the use of the method : 'methodNode'.
It has always a single child, which is a sequence node.

RBSequenceNode : this node is used by method and block.
Concretely, this node is designed to contain multiple statements, hence multiple nodes.
Useful variables : the collection of sub-nodes (statements), the temporary variables of the method or
the block (temporaries) and the interval they are contained in : start (leftBar) and
stop (rightBar).

RBComment : this node is always contained in another one, in a collection inside the variable (comments). This is the only node that can't contain comments.
Useful variable : the content of the comment (contents)

RBReturnNode : this node must contain a value node that is going to be returned by the

method.
Its biggest purpose is to signify the end of the method when encountered.
Useful variable : the value node to be returned (value).

RBValueNode : it is an abstract class that regroups all nodes that represent a usable object.
All the following classes are sub-classes of this one.
Useful variable : the position of the paretheses if there are any (parentheses)

RBArrayNode : this node is an array that stores the value returned by each statement it is composed of.
Useful variable : the statements composing the array (statements)
RBAssignementNode : it is composed of two parts, a variable and the value it is going to be affected by. Inside the AST, the value is necessarily a value node, but all types of value nodes are fine.
Useful variables : the variable that will receive the affectation (variable), the value that is going to be affected (value).

RBBlockNode : this node is a tiny method inside the method that return either the return node's value or the last statement's value. It can contain its own temporaries but not its own pragmas.
Useful variable : the sequence node

RBMessageNode : this node is composed of a receiver and a selector. The receiver can be any value node except error nodes and the selector is the message sent to the receiver. The selector can be accompanied by an argument, if it is a binary and even multiple, if it is a keyword (one selector per keyword).

RBCascadeNode : this node is like a shortcut for messages. It allows the user to write the receiver only once.
Warning : This node creates the messages with the first receiver which means it is the same object for all messages. The method 'nodesDo:' used on a cascade will visit the receiver for every message it contains therefor the receiver as many times as there are messages.

RBLiteralNode : this node is the abstract superclass of value containers.

RBVariableNode : this node contains the name of the variable a value or a message will be sent to.

RBParseErrorNode : this node contains the value of the token that caused the error.

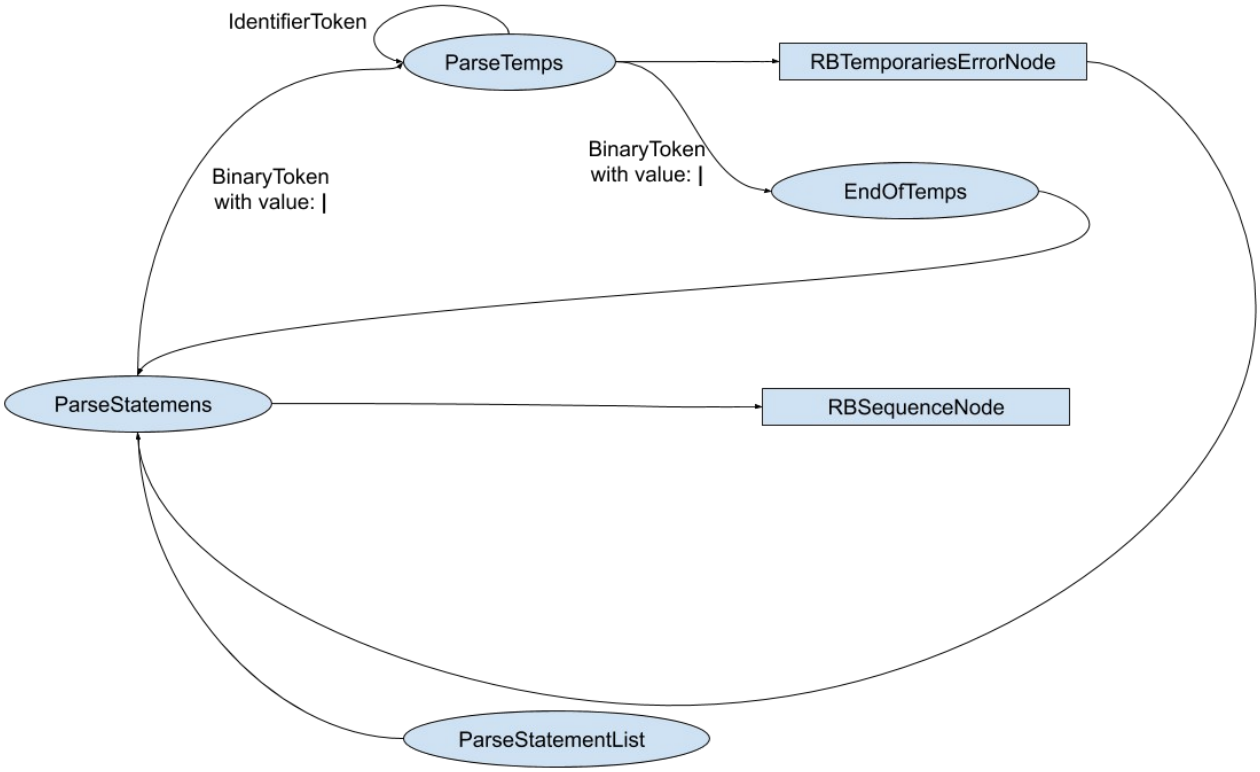
RBEnglobingErrorNode : this node is a sub-class of RBParseErrorNode destined to missing closure or opening errors. Its sub-classes are a specialisation of those errors.

RBParenthesesErrorNode : this node can be chosen from a missing '(' or ')'.
RBLiteralArrayErrorNode : this node comes from a missing ')' after a '#('.
RBLiteralByteArrayErrorNode : this node comes from a missing ']' after a '#['.

RBBlockErrorNode : this node can be chosen from a missing '[' or ']'.
RBArrayErrorNode : this node can be chosen from a missing '{' or '}'.
RBPragmaNode : this node comes from a missing '>'.
RBTemporariesErrorNode : this node comes from a missing '|'.

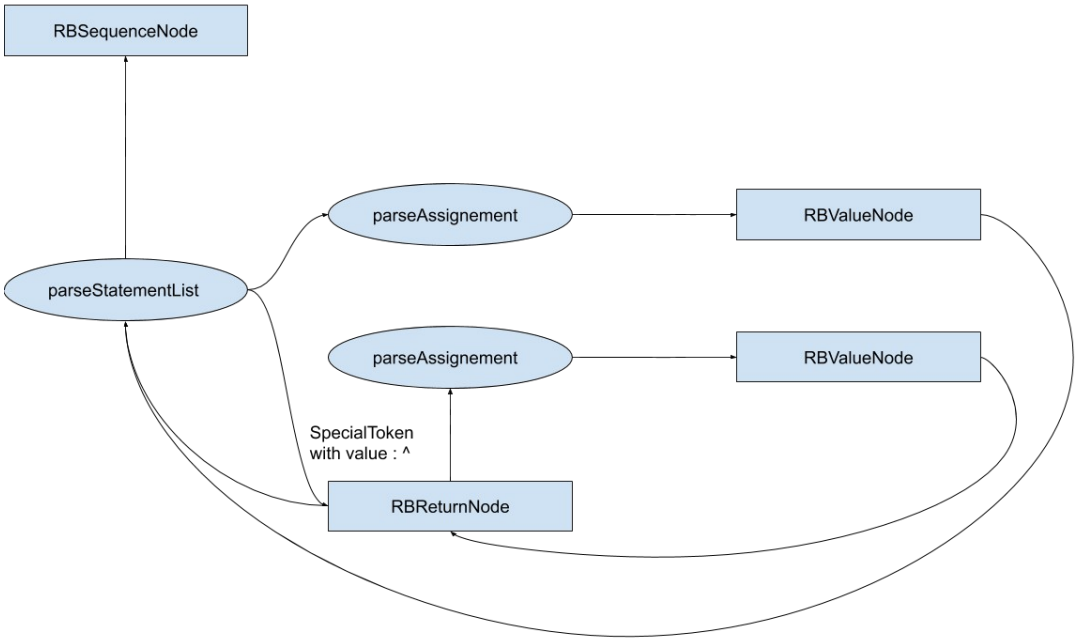
3- Parsing

We saw in the first chapter that both parseExpression and parseMethod use parseStatements to create the method nodes content. What it does is parsing assignements and storing them in a sequence node. But they are not the only ones, as blocks have the same content as an expression.

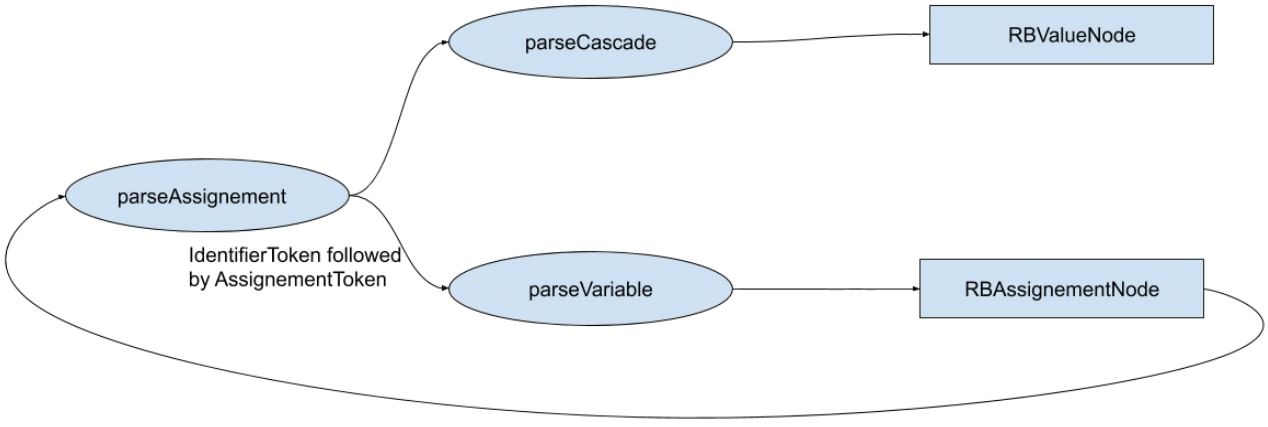


The important method, here, is parseStatementList because it is a method used to parse the code inside RBMethodNode, RBBlockNode and RBArrayNode. This may come as a shocker, but parseStatementList:into: (full name of the method) parses statements and put them in an orderedCollection. The first argument expected by parseStatementList:into: is a boolean saying if we can parse pragmas or not. The second argument is the node (RBSequenceNode by default)that will contain the statements. In the first implementation, the statements were initialized as an empty collection. Which meant you could not parse statements into the same node without erasing the first ones. I modified the initialisation to get the statements from the node. So now, we can parse statements inside a node, then stop for some reason, then continue the parsing again later.

To simplify for comprehension, a statement is a value node which we can return or not. That is why parseStatementList checks, before parsing the statement, if the current token is a special token with the value '^'. I think we could do a similar thing as the assignment token and create a return token because I didn't find any other use to '^'. This would allow us to ask a simple isReturnToken and not stock the character. If it finds a return, then this statement will be the last statement and the content of the statement will be put in a return node. Else it will be directly parsed. And if the end of a statement is not a dot then it means there is an error and we stop parsing the statements.
All value nodes meet the requirements to be a statement so what we have to do to parse one is to search which value node it is.

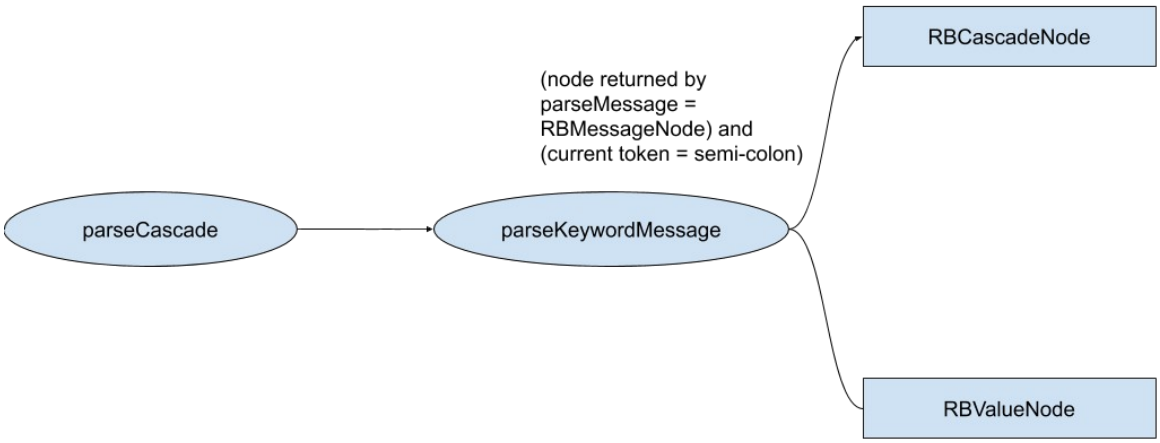


Between all the value node, the biggest is the assignment node because it is composed of a variable and a value node. That is why we try to see if it is an assignment first. In parseAssignement, if we have an IdentiferToken follow by an AssignementToken, it means we have an assignment. In which case, we parse the identifier to get a variable node and we create a RBAssignementNode with the variable as receiver and to get the value, we do another parseAssignement. Else we search for the next biggest value node, the cascade.



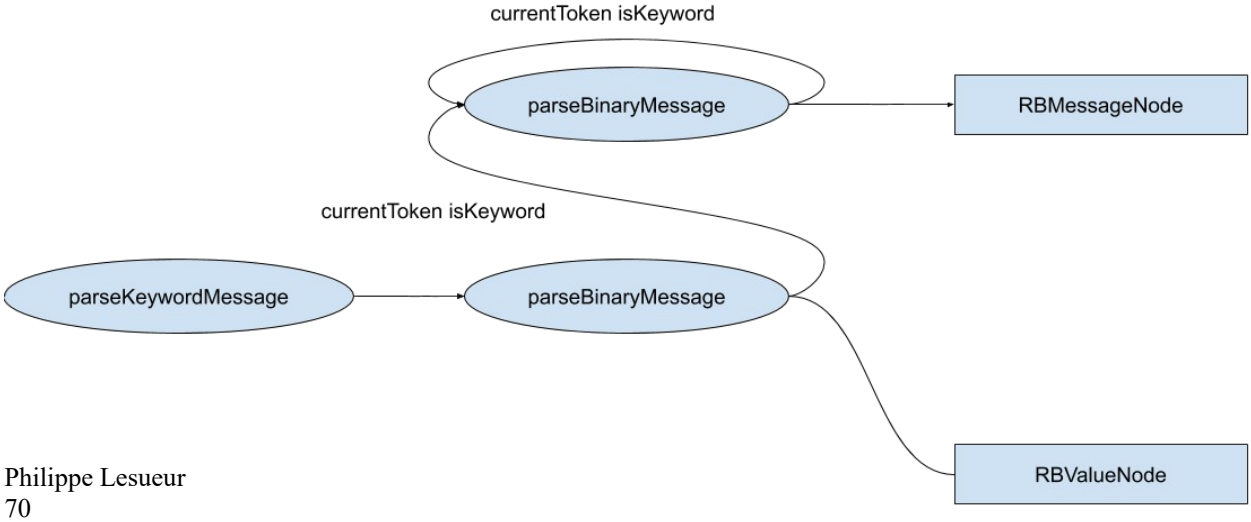
The cascade is a MessageNode with multiple selectors separated by semi-colons and the receiver of a message node can be either a message node or one of the remaining value nodes. So, to verify if we have a cascade node first, we try to parse a message node and then, if we have a message node and semi-colons, we look for selectors (if the selector is a keyword, we also look for its argument or a binary).

I made another change here because after encountering the semicolon, if there was an error in the cascade then the previously parsed parts of the cascade would be lost and the parser would return a parseErrorNode. Now, if there is an error in the cascade then the error is put inside it and the parsing continues, so the parser always returns a cascade node.

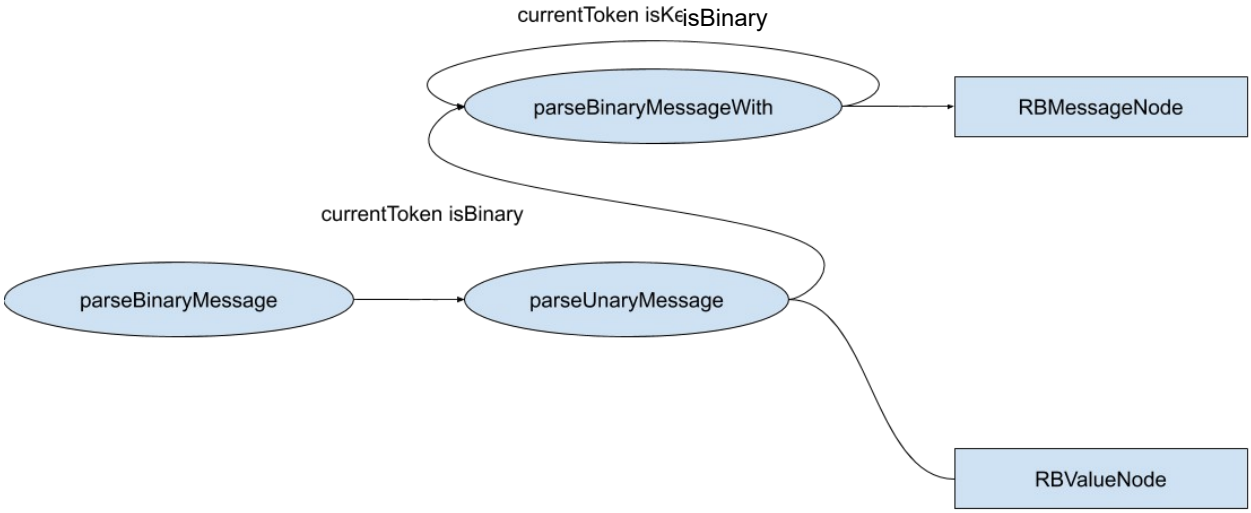


Inside the cascade, the parser tries to parse every type of message node with the receiver of the already parsed message node and the token as selector. If you don't obtain a message node, then you create a parseErrorNode with the currentToken. (The methods used to try to parse messages are : parseUnaryMessageWith:, parseKeywordMessageWith: and parseBinaryMessageWith:.)

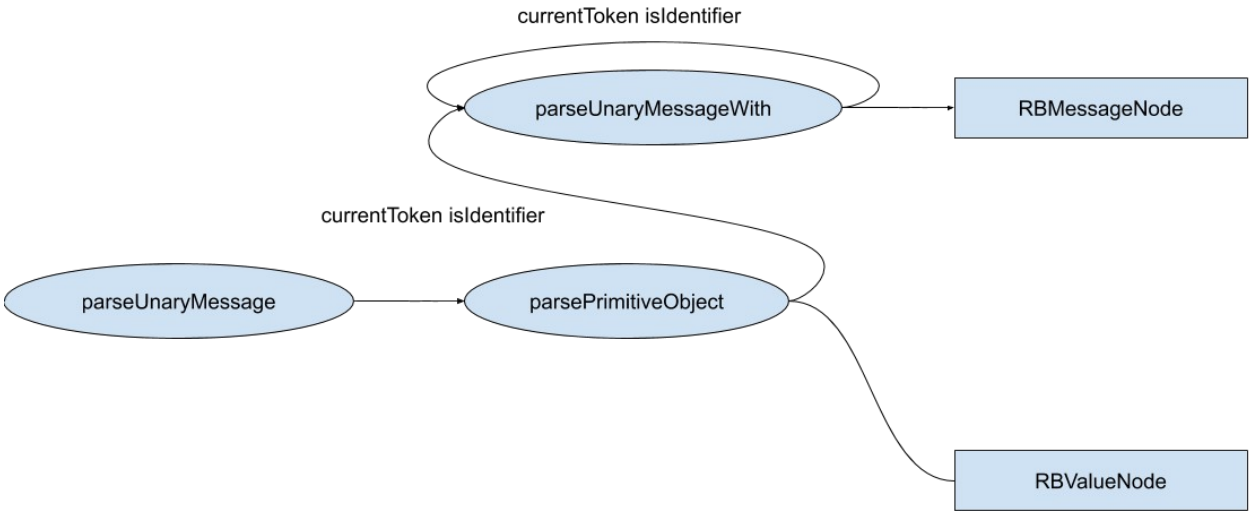
It is the way a message is parsed that gives it its priority. ParseKeywordMessage is the highest level of a message (the last one parsed) because the method parseKeywordMessage uses the method parseKeywordMessageWith: with, for argument, the node return by parseBinaryMessage. And of course, parseBinaryMessage uses parseBinaryMessageWith: with the node returned by parseUnaryMessage and it's the same for this one.
The argument of those methods will either become the receiver of the message node or the returned node. In parseKeywordMessageWith: when we have keywords, their arguments are also the returned node of parseBinaryMessage.



It is interesting to note that message nodes are the same class for keyword, binary and unary messages.
The method parseBinaryMessage works in a very similar way except a keyword message can have a selector composed of multiple keywords (and arguments) where a binary message has one receiver, one selector and one argument. Since we know there is only one selector, the receiver can be a binary message too.

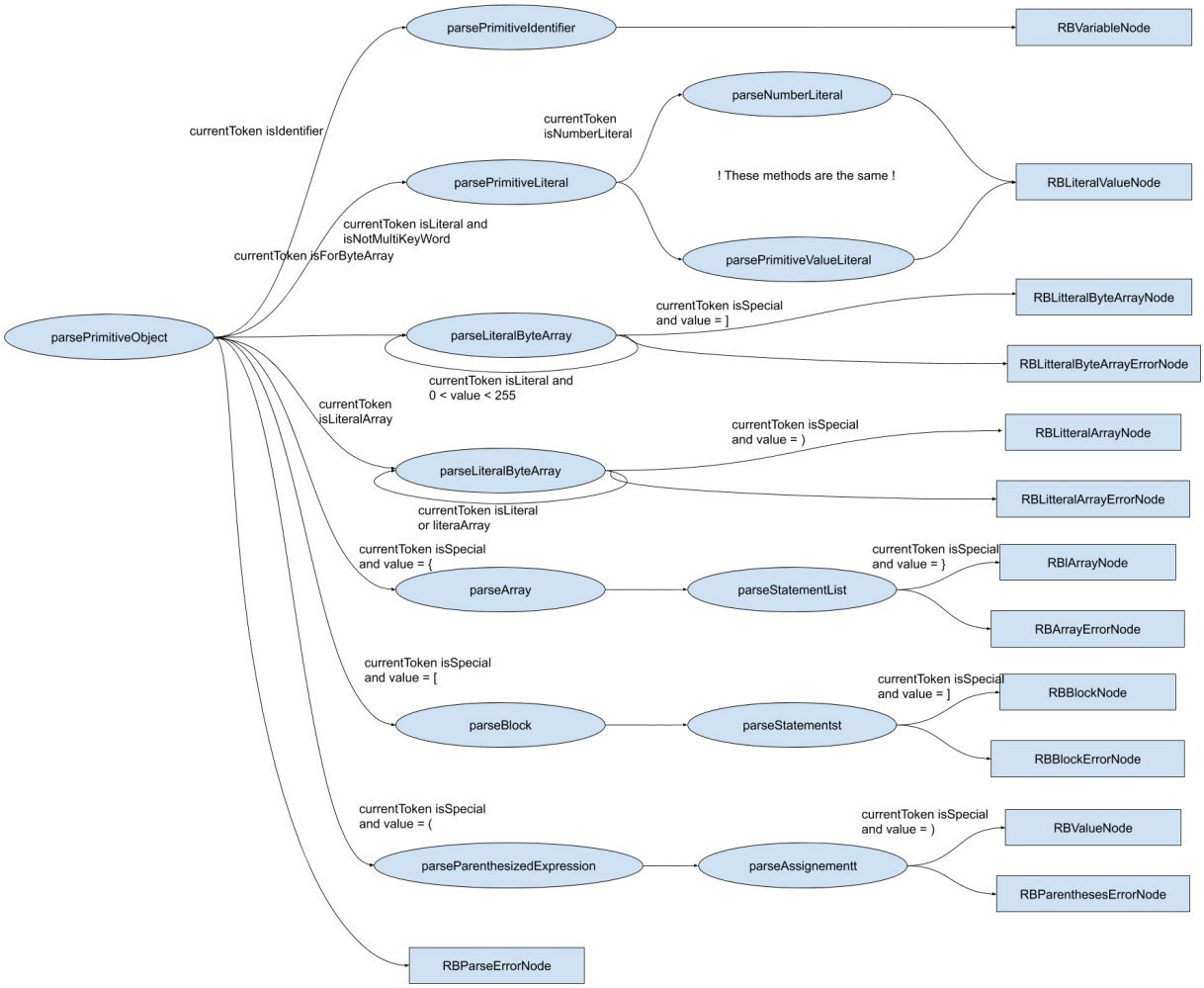


I will precise here that an argument is a value node and if we want to have a keyword message for argument (as an example), we have to add parentheses which we will see after parseUnaryMessage.

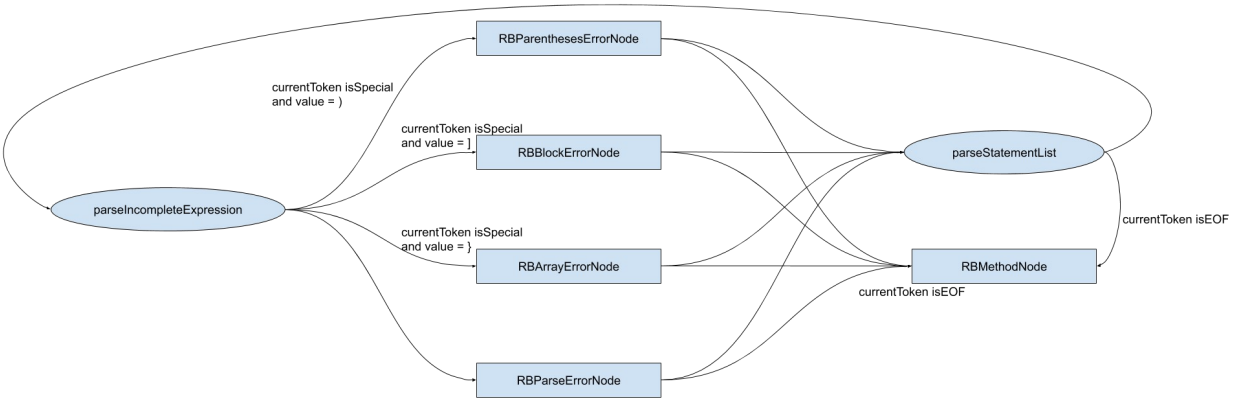


This construction is why messages are read from left to right because, each time we use parseUnaryMessageWith: or parseBinaryMessageWith: the node given in argument is the one received from the previous loop.
Exemple: 3 plus: 1 + 2 doubled – 3 * 4 minus: 1will be created like 3 + (((1 + (2*2)) – 3) * 4) - 1.

We have attained a point where there is no hierachy between the remaining value nodes anymore, so we have to enumerate the possible cases of what is left (like we did in the scanner).



Now, I would like to come back to parseMethod and parseExpression because if the parsing of the statements is over and we still aren't at the end of file the we use parseIncompleteExpression to create an error node with the rest of the code and add it to the statements. I modified this method to create an error with the current token and restart the parsing until the end of file.



We can see, here, the biggest improvement of the parser : the new error nodes. The parser had only one class of error, RBParseErrorNode. But during my study of the parser, I also studied the errors to classify them into types.

4) Added Nodes:

1- Error nodes:

After studying the parser and the scanner, I concluded that there were 3 major types of syntactic error.
The first ones are the character errors. They are spotted by the scanner and are forbidden characters. They are either unknown character or # followed by a separator which is forbidden. For unknown characters, we can't do anything but delete them, put them in a string or in a comment to repair the error or add a $ before. # is more complex since it is the beginning of a litteral, so the next token must fit literal specifications, if the next token fits then there is a probability to fix the error by suppressing the separators too.

The second ones are the unexpected token errors. This error is spotted in the parser and has a message like : Variable name expected, Variable or expression expected, Literal constant expected... This type of error means that in trying to parse a node, the token received didn't match with the expected ones. Unfortunately, this error is extremely tough to fix because is can pretty much be any kind of error : wrong token, missing token, missing dot, unwanted separator etc...

The third ones are the missing closure/opening errors :],),},>,|,',",(,[,{.
Here, there are 2 types of missing closures : ' and " which are recognized by the scanner and the others recognized by the parser. ' and " are special because a string and a comment don't contain code, but text therefor we don't have to parse what is inside and we can directly put the text inside. That is why the only way to detect the error is : encounter the end of file and the reparation must be manual or it would be random.
However, the other ones can be delimited by their context. When you have missing closure it is limited by the parsing method, it has been initiated in.
Example : `(parentheses error. normal message.`

Here, the parentheses utilises parseParenthesizedExpression which can only contain one statement. That is why we know the parentheses can't close further than the period. An array or a block can contain multiple statements and will stop at the end of file (or at the beginning). Furthermore, we can assume that the inside is more or less correct code. The openings are limited to parentheses, block and array because | and > are binary selectors so we can't detect the error and literal arrays and literal byte arrays have the same closure as, respectively, a parenthesized node and a block node. Since they are more common, we assume it is the latter.

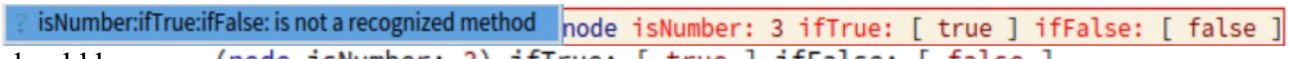
This is why I created a new type of node that would keep this parsed code to be able to check its validity. This new nodes are RBEnglobingErrorNodes which a subclass of RBParseErrorNode. Then there is a subclass for each type of error. This has 2 practical advantages (though they are not yet implemented). First, you can know which type of node you expected to search for reparations. Second, you can further the content you keep. For example, a blockNode can have its own temporaries and so could a blockErrorNode.

I also modified the method parseErrorNode. It was acting differently for the first and the second type of errors. For the first one, it created a RBParseErrorNode with the value of the erroneous token. For the second one, it created a RBParseErrorNode with a value constructed with the source code from the error position to the end of the source. Now, since the handling of error is much more refined, we can use the value of the current token and not lose any information in the tree compared to the source.

Finally, there are some errors that don't belong to any of these categories like 'cascaded message not allowed' which is almost a message expected because it concerns a variable followed by a semicolon or a parenthesized message followed by a semicolon when we expect a message without parenthesis. Or 'invalid token' which seems to be something in between the first and the second type of errors.

2- Selector Nodes:

In the AST, selectors are a special case because they are not nodes. Selectors are a ByteString contained inside the concerning node. Unary and binary selectors are used as stored, but keywords are different as keyword selectors can contain an indefinite amount of keywords parts (this is illustrated by the multiKeywordToken). In fact, there is a method inside the class ByteString to separate them called findSelector. This implementation is bothering for reparation purposes because, for semantic errors we would need to separate some parts. For example :



should become `(node isNumber: 3) ifTrue: [ true ] ifFalse: [ false ]`

We thought, even though it is not yet implemented, it would also be great to place error nodes in the selector variable for reparation purposes but the way the parser works makes it difficult. That is why I added a selector node : RBSelectorNode. This node has 3 sub-classes : RBUnarySelectorNode, RBBinarySelectorNode, RBKeywordSelectorNode.

Unary and binary don't have a difference yet, but we could attach the argument to the selectorNode instead of the messageNode.

The keyword also owns its keyword parts in a collection. The object is called RBKeywordPart and contains the keyword part, its start position and its argument.

4) Error Reparation:

1- Reparation research:

For error reparation, I did some research on what common mistakes are made during the writing of a method both for semantic and syntactic errors.

Semantic analysis :

self, super, thisContext can't be taken for anything else than variable. That is why, if we encounter them as a selector of a message, it means that there probably is a dot missing before them.

Example:

```
toto
    | error |
?   error node with self message.
```

Reparation:

```
toto
    | error |
    error node with. self message.
```

keyword selectors can be tricky because if you have another keyword in your argument, then you need to add parentheses. The most simple of those cases are the ifTrue: ifFalse: situation and the assert: equals:. You just need to add the parentheses at the right place.
When doing ifTrue: ifFalse:, the arguments are blocks so, the only focus must be to verify that we have blocks as arguments and if there is a prior selector, to put parentheses before the first "if"
Example:

```
toto
    | error |
?   error and: [ |anOtherError| anOtherError ]
        ifTrue: [ nil ]
        ifFalse: [ error ]
```

Reparation:

```
toto
    | error |
    (error and: [ |anOtherError| anOtherError ])
        ifTrue: [ nil ]
        ifFalse: [ error ]
```

When doing assert: equals: we have to watch before, after and in between. This is a more tricky situation because other keyword selectors can use equals. That is why we must verify that assert: alone creates another semantic analysis.

Example:

```
toto
    | error |
?   self tool: error assert: error with: true equals: error inContext: self.
```

Reparation:

```
toto
    | error |
    (self tool: error) assert: (error with: true) equals: (error inContext: self).
```

Other Reparation:

```
toto
    | error |
    (self tool: error) assert: (error with: (true equals: (error inContext: self))).
```

Syntactic Errors:

The best hint there is a dot missing is the presence of multiple errors following themselves because the statement stops at the wrong place, therefor the error creation chains.

Example:

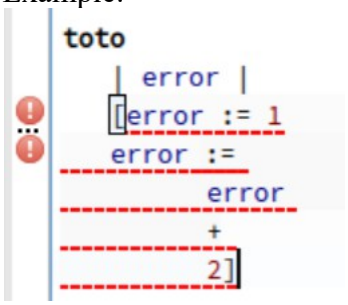
```
toto
    | error |
    error := 1
    error :=
          error
          +
          2
```

Reparation:

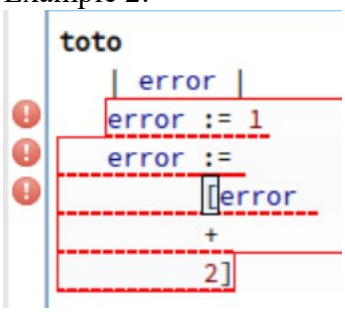
```
toto
    | error |
    error := 1.
    error :=
          error
          +
          2
```

Englobing error nodes contain nodes, which can sometimes be there closing or opening, in this case, we can determine that there is probably a dot missing inside the node before the closure or before the opening.

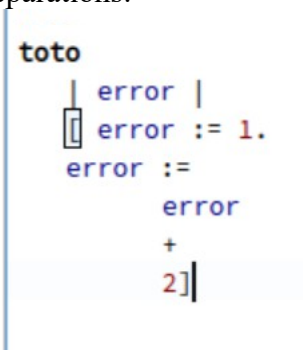
Example:

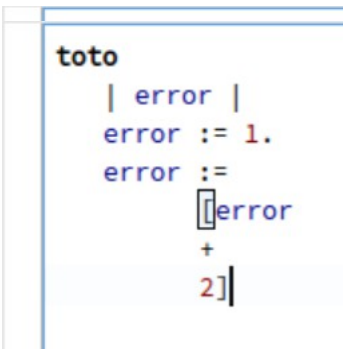


Example 2:



Reparations:

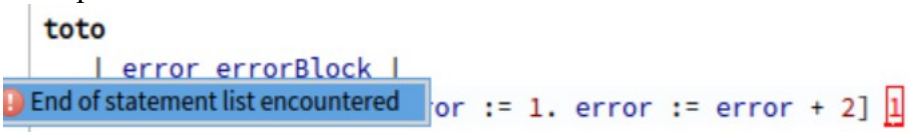




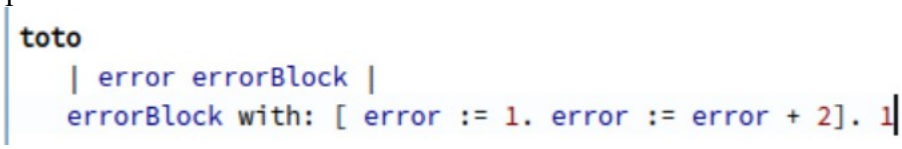
Sometimes this error is not that simple and there is a missing keyword or colon (at the end of previous node) to make the node argument of a message.

When there is an end of statement error, then we probably forgot a dot somewhere. The research will then be to see if placing dots at the end of the lines repairs anything.

Example :

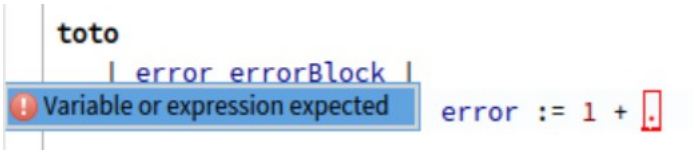


Reparation:

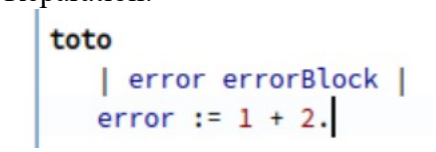


When you have an error saying "variable expected" or "Message pattern expected" it means one of your nodes is not correct or you forgot what is expected. I found that the possible reparations can be to change the node, add a node before or place a dot.

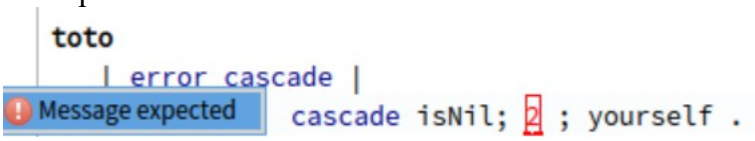
Example:



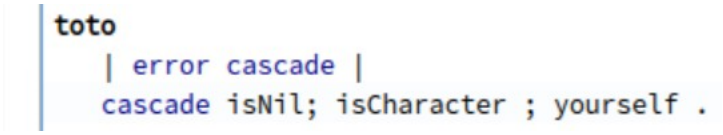
Reparation:



Example:

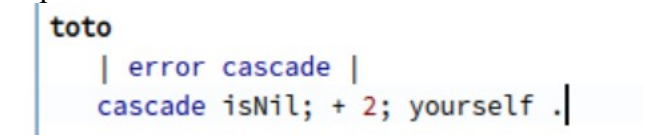


Reparation:



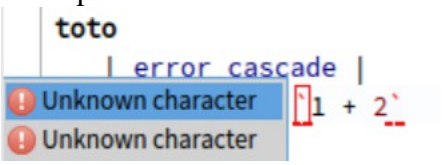
Example:

Reparation:



There is a specific type of error found by the scanner which is wrong character. If it is an unknown character then we can change or delete it. If it is a #, we can try to connect them to the next node by trimming the separators.

Example:



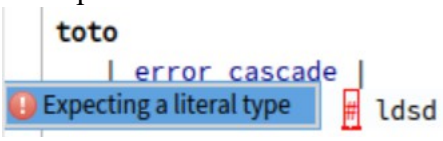
Reparation:

```
toto
    | error cascade |
    '1 + 2'
```

Reparation 2:

```
toto
    | error cascade |
    1 + 2
```

Example:

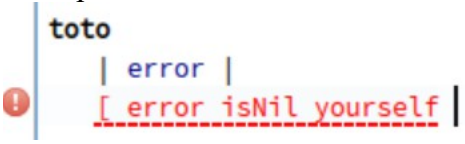


Reparation:

```
toto
    | error cascade |
    #ldsd
```

Finally, if we encounter an englobing error without an opening or a closure inside, we make a research for opening or closure inside its content.

Example:



Reparation:

```
toto
    | error |
    [ error isNil yourself]
```

Reparation 2:

```
toto
    | error |
    [ error isNil ] yourself
```

This is a first draft of reparation research which should be completed by Pharo users.

For me, the reparation research should be implemented inside a vistor that efficiently visits the tree to recuperate the node to modify. We would then do the research on this node which could either be quickly repaired (the most pertinent choice) or the reparation choices could be displayed in a pop up window. Once the option chosen, the node would be sent to a reparation class that would modify the code inside the method tab and reparse it.

2- Reparation classes:

I have done the reparation classes, one for each semantic reparation that is adding the semantic to the context and four for fixing the code. There is one to erase the given node, one to add code before the node, one to add code after the node and one to add code at given positions.

The semantic reparation classes simply reuse the already implemented code, but the goal here is to be able to quickly fix the error using a hotkey.
The other classes basically divide the code in parts and add the reparation code between those parts.
Then, once the reparation is done, it reparses the AST and does semantic analysis.
I chose not to modify the code in the tab at this stage because I think that we could do a lot of things with the returned AST.
For example, I think we could find a way to run the tests of this method using the AST to see if the result is the one we expected. Or use a tool that allows us to execute code on this AST.
So, with an automated process, we could make it part of the reparation research.