Université Paul Sabatier
Toulouse III
IUT « A »
Département Informatique

Rapport de stage

Compalgo 2.0

Compilation et contrôle d'exécution interactif

Damien Leroux

Année Spéciale 2006-2007

« Loi de Hofstadter : il faut toujours plus de temps que prévu,
même si l'on tient compte de la Loi de Hofstadter. »
Douglas Hofstadter, *in Gödel, Escher, Bach,*
les brins d'une guirlande éternelle

« It's not a bug, it's a feature ! »
Anonyme

Remerciements

Je tiens tout d'abord à remercier mon maître de stage Max Chevalier pour m'avoir permis d'effectuer ce stage au sein de l'IRIT, ainsi que l'ensemble du personnel de l'IRIT pour l'ambiance sympathique et chaleureuse du laboratoire.

Je remercie également Guillaume Cabanac, doctorant, co-auteur de la première version de Compalgo, pour ses conseils en Java et typographie, Ronan Tournier, doctorant, pour ses conseils concernant UML et l'utilisation avancée de MS Word, et Sabine Carbonnel, postdoc, pour ses pistes concernant les patrons de conception et l'utilisation de TeX.

Fiche Technique

Etudiant	Damien Leroux
Année	2006-2007
Raison sociale de l'entreprise	Institut de Recherche Informatique de Toulouse (Université Toulouse III)
Maître de stage	Max Chevalier
Enseignant responsable	Christine Julien
Sujet du stage	Correction et évolution du langage d'initiation à l'algorithmique Compalgo.
Plate-forme informatique ou technique	Java 1.5
Outils logiciels	- Eclipse 3.x : environnement de développement Java. - Coco/R Java : générateur de compilateurs en Java. - RDP : générateur de compilateurs en C ANSI et validateur de grammaires LL(1). - Bash et GNU Binutils : utilitaires de traitement de textes en ligne de commandes. - YourKit Java Profiler : profileur de code Java.

Sommaire

Partie I - Présentation de l'entreprise et de l'environnement professionnel du stage 6
1. Introduction 6
1.A. IUT « A » Paul Sabatier – Université Toulouse 3 6
1.B. Institut de Recherche Informatique de Toulouse 6
1.C. Environnement de travail 7
Partie II - Nature de la tâche et travail réalisé 8
1. Présentation 8
1.A. Problématique du stage 8
1.B. Eléments de théorie des langages et de compilation 8
1.C. Compalgo 1.0 9
1.D. Compalgo 2.0 9
2. Matériel et outils logiciels 10
2.A. Rédaction des documents 10
2.B. Java 1.5 10
2.C. Eclipse 3.3 10
2.D. Générateur de compilateurs Coco/R 11
2.E. Autres outils 11
3. Travail réalisé 11
3.A. Emploi du temps 11
3.B. Revue et comparaison des outils existants 12
3.C. Réécriture de la grammaire pour obtenir la nature LL(1) 12
3.D. Définition de l'AST de Compalgo 12
3.E. Génération de l'AST avec le générateur Coco/R 13
3.F. Abstraction de parcours d'arbre 13
3.G. Applications avec les vérifications sémantiques et l'interprétation 14
3.H. Contrôle d'exécution pour l'interprétation 14
3.I. Mécanisme d'extension de la bibliothèque native du langage algorithmique 14
3.J. Interface graphique 15
3.K. Profilage et optimisation de l'interprétation 15
3.L. Intégration des nouveautés dans l'éditeur existant 15
3.M. Tests de non-régression 15
Partie III - Bilan 17
1. Bilan professionnel 17
2. Bilan technique et informatique 18
3. Bilan de la formation initiale 18
Bibliographie 19
Journal de stage 20

Partie I -
Présentation de l'entreprise et de l'environnement professionnel du stage

1. Introduction

L'objet de mon stage est de faire évoluer un outil pédagogique permettant aux étudiants en informatique d'acquérir les notions d'algorithmique, nommé Compalgo. Cet outil est utilisé par l'équipe enseignante de l'IUT « A » Paul Sabatier, de l'Université Toulouse III. Les professeurs, et mon maître de stage en particulier, étant enseignants-chercheurs, j'ai effectué mon stage dans le laboratoire où il effectue ses recherches, l'Institut de Recherche Informatique de Toulouse (IRIT), au sein de l'équipe Systèmes d'Information Généralisés (SIG).

Cette section présente l'IUT « A » Paul Sabatier et l'IRIT, les organisations impliquées dans le projet.

2. IUT « A » Paul Sabatier – Université Toulouse 3

« L'Institut Universitaire de Technologie « A » est un des deux IUT de l'Université Paul Sabatier, l'autre étant l'IUT de TARBES.

Les IUT forment au niveau « bac + 2 » des spécialistes immédiatement opérationnels en entreprise, mais qui ont aussi de nombreuses possibilités de poursuites d'études. Ils sont aussi opérateurs pour un certain nombre de licences professionnelles.

L'IUT « A » est constitué de 15 départements d'enseignement, répartis sur 4 sites : 10 départements sur les 2 sites de Toulouse (Ponsan, Rangueil, 2 départements à Auch et 3 à Castres.

L'IUT est dirigé par un Directeur et est administré par le Conseil de l'Institut.

Les départements sont dirigés par un Chef de Département assisté par un Conseil de Département.

A l'IUT « A », près de 4000 étudiants préparent un DUT en formation initiale et 225 en Promotion Supérieure du Travail (cours du soir). 335 étudiants préparent une Licence Professionnelle. Par ailleurs, 71 étudiants préparent un Diplôme d'Université Post DUT. » [IUT]

L'équipe pédagogique enseignant dans le pôle Algorithmique et Programmation (AP) à l'IUT « A » est dirigée par Christian Percebois. Dans ce pôle sont notamment enseignées l'algorithmique, les langages Java, C, C++, ainsi que les technologies associées à Internet.

3. Institut de Recherche Informatique de Toulouse

« L'IRIT, Institut de Recherche en Informatique de Toulouse, est une Unité Mixte de Recherche, UMR 5505, commune au Centre National de la Recherche Scientifique (CNRS), à l'Institut National Polytechnique de Toulouse (INPT), à l'Université Paul Sabatier (UPS) et à l'Université des Sciences Sociales Toulouse 1 (UT1).

L'IRIT, créé en 1990, représente l'un des plus forts potentiels de recherche en informatique en France, fédérant plus de 190 chercheurs et enseignants chercheurs, relevant non seulement de ses tutelles mais aussi de l'Université Toulouse Le Mirail (UTM).

[…]

L'IRIT est organisé autour des sept thèmes suivants, auxquels sont rattachées 25 équipes :

- thème **1** : Analyse et synthèse de l'information
- thème **2** : Indexation et recherche d'informations
- thème **3** : Interaction, autonomie, dialogue et coopération
- thème **4** : Raisonnement et décision
- thème **5** : Modélisation, algorithmes et calcul haute performance
- thème **6** : Architecture, systèmes et réseaux
- thème **7** : Sûreté de développement du logiciel

Cet ensemble de thèmes met en évidence la couverture scientifique de l'IRIT sur l'ensemble des problématiques de recherche de l'informatique actuelle. » [IRIT]

(*source : site web de l'IRIT, http://www.irit.fr/-Le-mot-du-directeur-*)

4. Environnement de travail

L'IRIT, et particulièrement l'équipe SIG, fournit aux stagiaires et aux doctorants qui ne disposent pas de bureaux dédiés des machines en libre service regroupées dans des « salles des machines ». L'équipe SIG dispose de deux salles machines comptant une vingtaine de machines. On y trouve des stations Sun ainsi que des PC Linux et Windows.

Utilisant Eclipse pour faire un développement multiplateformes en Java, j'ai alterné l'utilisation des différents systèmes disponibles.

Partie II -
Nature de la tâche et travail réalisé

1. Présentation

Après avoir exposé les axes de la problématique de l'évolution de Compalgo et les éléments théoriques sous-jacents, je décris la version première existante puis les corrections et nouvelles fonctionnalités de la deuxième version, sujet de mon stage.

1.A. Problématique du stage

Le sujet de mon stage est axé sur la compilation d'un langage formel de programmation et l'exécution de programmes. Le travail à produire englobe toute une chaîne de compilation, depuis l'analyse d'un texte source jusqu'à l'exécution du programme qu'il représente. Il inclut aussi une partie de développement d'interface graphique afin de contrôler cette exécution.

1.B. Eléments de théorie des langages et de compilation

Dans le contexte de l'informatique, l'on distingue deux grandes familles de langages : les langages dit « naturels » et les langages formels. Les langages naturels sont ceux utilisés naturellement par les humains et sont très délicats à manipuler pour les ordinateurs car ils recèlent nombre d'ambiguïtés et de sous-entendus contextuels. Les langages formels, *dixit* Wikipédia, sont « dans de nombreux contextes (scientifique, légal, etc.) […] un mode d'expression plus formalisé et plus précis (les deux n'allant pas nécessairement de pair) que le langage de tous les jours (voir langage naturel) » [WP-LF]. Les langages de programmation font partie des langages formels.

Une grammaire définissant un langage formel est un ensemble de règles de construction et d'éléments lexicaux (*tokens*) comme les mots-clé et les ponctuations.

Dans la théorie des langages, les grammaires sont classées en fonction de leurs propriétés. Les grammaires de la classe dite LL engendrent des langages lisibles de gauche à droite (le premier L, *Left to right*) et structurables de gauche à droite (le deuxième L, *Leftmost derivation*). Ces langages sont aisément *parsés* par l'algorithme de descente récursive. La notation LL(k), k étant un entier défini ou non, indique qu'il est nécessaire de lire au plus k *tokens* pour pouvoir décider de la structure du texte source lu (c'est-à-dire décider des règles de grammaire à appliquer). En particulier, la nature LL(1) d'une grammaire implique qu'à chaque nouvel élément du langage lu, le *parser* sait quelle règle s'applique [WP-LL].

La compilation d'un texte consiste à en extraire l'information pour pouvoir la représenter différemment et la réutiliser ensuite. Dans le cas des langages de programmation, la compilation consiste à transformer un texte source manipulable par un humain en des données équivalentes manipulables par un ordinateur.

Cette transformation s'effectue en plusieurs étapes [C.Wolf], comme représenté sur la Figure 1. Tout d'abord, le texte est découpé en *tokens* par ce qui s'appelle un *lexer*. La séquence de *tokens* ainsi créée est ensuite donnée au *parser* associé qui se charge de structurer l'information qu'ils représentent. Au cours de son traitement, le *parser* effectue des actions dites fonctions de réduction ou actions sémantiques pour générer les données finales.

En fonction de la sortie qu'il génère, le *parser* peut avoir trois appellations. S'il génère un nouveau texte, dans un langage différent ou une simple augmentation du texte d'origine, on parle de traducteur. S'il génère directement les données informatiques correspondant au texte source, on parle de compilateur. Et s'il génère uniquement une représentation abstraite de l'information contenue dans le texte source, sous forme d'arbre (Arbre de Syntaxe Abstraite, ou AST, pour *Abstract Syntax Tree*), on parle de *parser*.

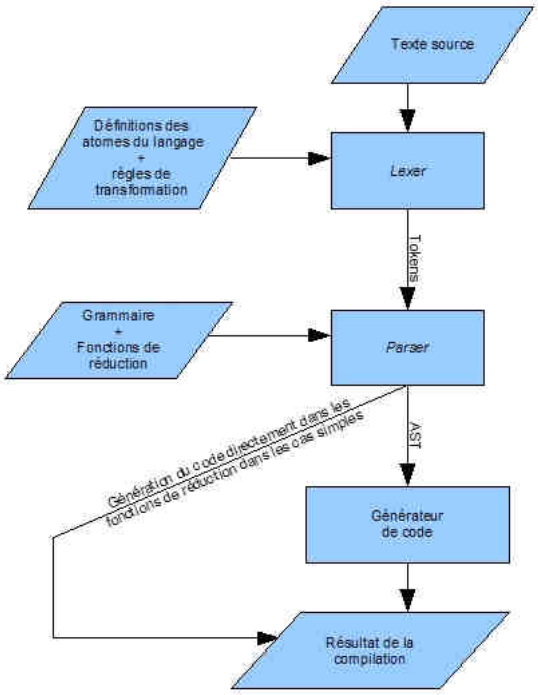


Figure 1 Processus de compilation

1.C. Compalgo 1.0

L'équipe pédagogique de l'IUT « A » a développé, à l'aide d'étudiants de l'équipe SIG, un langage aux fins de l'initiation des étudiants de première année de DUT Informatique à la programmation impérative [CA v1]. La syntaxe de ce langage est très proche du langage Ada (à la traduction en français près). La version existante, utilisée actuellement à l'IUT informatique, offre aux étudiants un éditeur multi-documents capable de mettre en forme les textes sources et un compilateur intégré. Le compilateur en fait traduit les fichiers algorithmiques (écrits en langage algorithmique) en langage Java, puis compile cette traduction via le compilateur fourni par Sun. Un script est généré à l'issue d'une compilation réussie pour permettre l'exécution du fichier compilé sans faire apparaître la couche médiane Java.

Il est malheureusement apparu à l'usage que Compalgo souffrait de quelques défauts, dont certains critiques. Notamment, les priorités des opérateurs arithmétiques n'étaient pas respectées (par exemple « 3+4*5 » évaluait à 35 au lieu de 23), ce qui obligeait l'utilisateur du langage à « sur-parenthéser » les expressions arithmétiques, et l'opérateur unaire « moins » était mal analysé.

Par ailleurs, étant donné la traduction en Java et sa compilation, certaines erreurs (exceptions) étaient difficilement interprétables par les étudiants. Qui plus est il était impossible, dans cette première version, d'exécuter un programme pas à pas ou de surveiller l'état des glossaires (dictionnaires de variables locales).

1.D. Compalgo 2.0

La version 2.0 de Compalgo a pour ambition de corriger ces défauts et lacunes en reprenant à zéro l'intégralité de la chaîne de compilation existante. Tout d'abord, la grammaire doit être réécrite de façon à obtenir la nature LL(1). Le *parser* doit ensuite générer un arbre de syntaxe abstraite. Le choix d'une sortie abstraite facilitera les évolutions futures, et rend le *design* de la version 2.0 plus rigoureux. Max Chevalier a proposé d'interpréter directement le contenu de l'AST généré plutôt que de le compiler, et c'est finalement cette solution qui a été retenue.

Le langage repose sur deux types de fichiers :

- les fichiers d'extension *spec* déclarant uniquement des types, constantes, et entêtes de sous-programmes,
- les fichiers d'extension *algo* déclarant en sus les corps de sous-programmes et le programme principal.

En outre, les enseignants ont souhaité que la gestion d'exceptions, initialement prévue dans le langage mais non implantée intégralement dans la première version, soit effective dans la deuxième version, et que la bibliothèque native du langage soit facilement extensible.

Une fois ces buts principaux atteints, le but secondaire est de fournir aux étudiants un cadre de débogage interactif de leurs programmes. Cela implique de pouvoir contrôler le flot de l'interprétation et de pouvoir visualiser l'état de l'interpréteur (instruction courante, pile d'appels, valeurs des variables locales), le tout à travers une interface ergonomique.

2. Matériel et outils logiciels

Mon travail durant le stage a été réparti entre le développement et la rédaction de documents. Je présente ici les différents logiciels que j'ai été amené à utiliser sur des machines de type PC utilisant des systèmes d'exploitation Linux et Windows.

2.A. Rédaction des documents

J'ai commencé à rédiger les premiers documents avec OpenOffice Writer, qui souffre encore de nos jours de plus de bogues que son concurrent commercial, Microsoft Word. Mon maître de stage tenant à disposer de documents Word, j'ai donc converti les premiers documents et rédigé les suivants en utilisant Word.

2.B. Java 1.5

Le langage d'implantation de Compalgo est Java. La première version de Compalgo utilisait Java 1.4. L'intégralité du développement de la version 2 de Compalgo a été faite pour la version 1.5 de Java, qui apporte de nombreuses améliorations du langage et de la machine virtuelle.

La plus importante des améliorations est l'incorporation dans la machine virtuelle de typages génériques, vaguement comparables aux *templates* de C++ dans la forme. Cela permet d'écrire très simplement du code générique et néanmoins fortement typé, vérifiable dès la compilation. L'inconvénient mineur inhérent à cette amélioration est qu'elle nécessite un minimum de réécriture pour évoluer de Java 1.4 à Java 1.5. En particulier, les collections (classes conteneurs) de Java deviennent fortement typées, tout en permettant l'utilisation à la façon 1.4 sans typage strict.

Une autre amélioration importante est l' « *autoboxing* », qui convertit implicitement et automatiquement les valeurs de type primitif (flottant, entier, caractère…) en des objets et réciproquement (« *unboxing* »). Cette amélioration ne nécessite aucune réécriture pour évoluer d'une version antérieure à la version 1.5.

2.C. Eclipse 3.3

Eclipse est un environnement de développement intégré développé par IBM et entièrement gratuit. Il offre de nombreuses aides au développement comme la reconnaissance syntaxique, la complétion contextuelle semi-intelligente, l'intégration de la documentation « Javadoc » et est extensible par l'ajout de plugins, en général gratuits eux aussi. Eclipse est réalisé en Java et est à l'origine fait pour développer du code Java, mais permet aussi de développer dans bien d'autres langages. Il intègre un système puissant de mise à jour qui permet de modifier facilement sa configuration pour y intégrer les outils nécessaires.

2.D. Générateur de compilateurs Coco/R

Développé à l'Université de Linz (Autriche), Coco/R est un générateur de compilateurs utilisant l'algorithme de descente récursive [Coco/R]. Il intègre la génération de *scanner* et de *parser*, et sa grammaire à attributs permet d'inclure facilement des actions sémantiques au sein des règles de production.

Coco/R existe nativement pour C#, Java, et C++. Ses concepteurs proposent également un plugin Eclipse que j'ai utilisé dans le cadre de mon développement. La justification de ce choix est présentée dans la section 3.B.

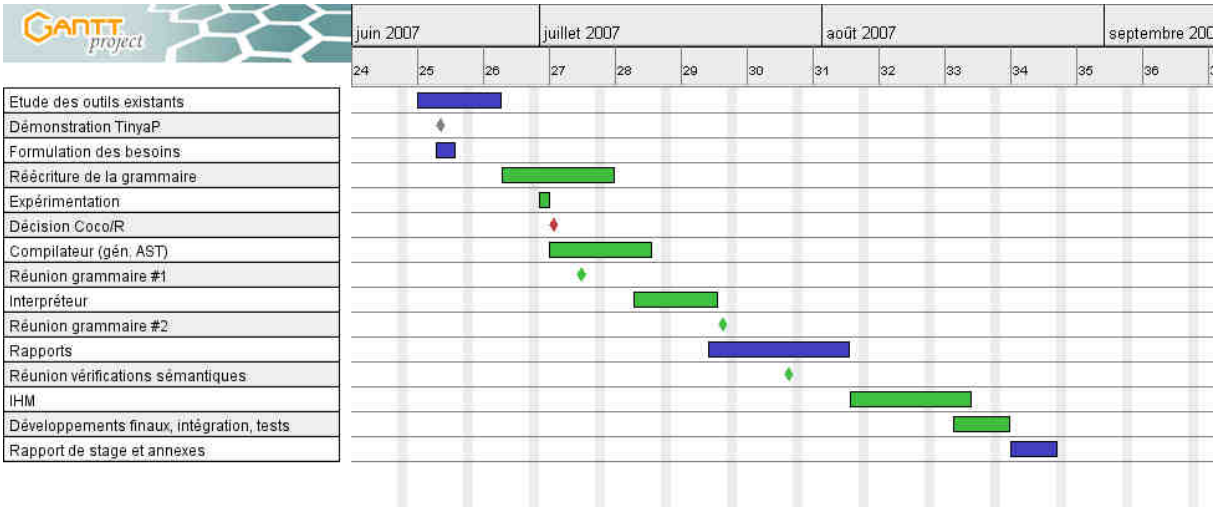
2.E. Autres outils

- RDP Parser Generator : cet outil développé à l'Université de Londres est un générateur de compilateur en C ANSI en même temps qu'un validateur de grammaires LL(1) [RDP], que j'ai utilisé uniquement pour valider la nature LL(1) de la grammaire de Compalgo.
- bash, sed, grep : j'ai utilisé des scripts shell pour automatiser la traduction de la grammaire formelle vers les formats de spécification de parser de RDP, Coco/R, et ANTLR.
- ANTLR : candidat à la génération du compilateur de Compalgo 2.0, ce générateur de compilateurs ne sert finalement qu'à générer les diagrammes syntaxiques [ANTLR].
- YourKit Java Profiler : profileur commercial puissant de code Java, payant pour une utilisation de longue durée [YK-JP].

3. Travail réalisé

Dans cette section je détaille le travail accompli, d'abord par une vue globale de la répartition des tâches dans le temps puis en détaillant mon activité pour chaque tâche.

3.A. Emploi du temps



Les phases de conception préliminaires sont incluses dans les phases de développement, représentées en vert. Les réunions ayant trait au produit final sont également représentées en vert. Les phases de rédaction sont représentées en bleu.

3.B. Revue et comparaison des outils existants

Le tout premier travail qui m'a été demandé a été de synthétiser un tour d'horizon des générateurs de compilateurs existants en Java (au moins générant des classes Java, au mieux eux-mêmes implantés en Java), assorti d'un bref rappel du processus de compilation. La grille d'évaluation *a priori* qui en est ressortie n'aidait pas à décider de l'outil à utiliser, ainsi j'ai complété ce tour d'horizon par un test comparatif de quatre outils sélectionnés : JavaCC, CUP/JLex, ANTLR, et Coco/R.

Le test consistait à écrire avec chaque outil la spécification d'un parser générant un AST pour une grammaire simple mimant le langage LISP, puis à fournir en entrée des *parsers* trois fichiers : un très court écrit à la main, un très gros généré aléatoirement (et valide), et un contenant des erreurs. Cela a permis de tester l'intégration dans l'environnement d'Eclipse, la convivialité des langages de spécification, la facilité de débogage des fichiers de spécification, la robustesse des outils, et la gestion des erreurs. Pour chaque critère, j'ai établi un classement, et Coco/R s'est finalement détaché du lot. Cette revue des outils existants est fournie en annexe page 7.

3.C. Réécriture de la grammaire pour obtenir la nature LL(1)

La grammaire originale de Compalgo [CA v1] est écrite dans le format EBNF [WK-EBNF], qui intègre des opérateurs de cardinalité comme « au moins un », « au plus un », « zéro ou plus ». Cela en rend la lecture difficile, et *a fortiori* la détermination de la nature LL(k) de la grammaire. Mon maître de stage voulait que la grammaire soit réécrite dans le format BNF [WK-BNF] si elle n'était pas de nature LL(1).

J'ai opté pour un *refactoring* complet (réécriture à partir de zéro) pour deux raisons :

- mon maître de stage souhaitait ne plus voir apparaître les constructions EBNF ambigus,
- la grammaire dans sa forme d'origine n'étant pas exempte de bogues, il était beaucoup plus facile de repartir de zéro que de chercher les erreurs et les corriger, sachant qu'il fallait de toutes façons supprimer les constructions EBNF.

Par ailleurs, la grammaire originale comporte au moins une récurrence à gauche. De ce fait elle n'est pas LL(k), quel que soit le *k* considéré.

La grammaire est réécrite dans un format abstrait indépendant de l'outil choisi finalement. Pour guider le *refactoring*, je me suis imposé quelques règles à suivre :

- stratégie « diviser pour mieux régner » : plusieurs règles simples valent mieux qu'une règle alambiquée,
- factorisation à gauche des éléments communs à plusieurs règles (nécessaire pour viser la nature LL(1)),
- toute alternative dans une règle doit être positionnée en partie droite, et uniquement en partie droite,
- des commentaires structurent la grammaire tout au long du fichier.

La grammaire ainsi obtenue est certes longue (649 lignes, 136 lexèmes et règles, 139 commentaires), mais très claire et facile à maintenir et à faire évoluer. Elle est fournie en annexe page 25.

3.D. Définition de l'AST de Compalgo

Un AST est une structure récursive composée de nœuds. Chaque nœud a un label précisant son type et peut contenir d'autres nœuds (fils). Un nœud sans fils est appelé une feuille et son label contient une information provenant du texte source.

La première chose à faire afin de pouvoir construire l'AST de Compalgo était d'énumérer les types de nœuds nécessaires et ce que devait contenir chaque type de nœud (cf. Énumération des types de nœuds de l'AST en annexe page 34).

Par exemple, le nœud représentant la condition « si … alors … sinon … fin si » contient une expression booléenne suivie d'un bloc d'instructions suivi éventuellement d'un autre bloc d'instructions quand le « sinon » est défini.

Au total, avec six instructions, deux types de sous-programme et le programme principal, cinq types primitifs, six opérateurs de comparaison, sept opérateurs arithmétiques, et les trois opérateurs booléens, l'AST de Compalgo comporte plus de 40 types de nœuds différents.

Ma préoccupation a été de m'assurer que toute erreur de construction soit détectée le plus vite possible. En effet, lors d'un traitement récursif, les erreurs se propagent extrêmement vite, et la construction de l'AST pouvait vite devenir un enfer de perte de temps.

3.E. Génération de l'AST avec le générateur Coco/R

Dans la spécification d'un parser, les actions sont imbriquées dans les règles de grammaire et ont lieu quand le parser vient de reconnaître l'élément placé juste à leur gauche (ou dès l'entrée dans la règle si l'action est placée tout à gauche).

Un raisonnement abstrait m'a conduit à réduire la génération de l'AST à partir de la grammaire à quatre opérations distinctes (cf. Rapport d'analyse en annexe) : créer un nouveau nœud courant, ajouter une feuille au nœud courant, valider le nœud courant, et remplacer le nœud courant par un nouveau nœud courant.

L'AST dans son ensemble est construit à partir de ces quatre opérations.

Coco/R génère une classe Parser (et la classe Scanner associée) à partir du fichier de spécification, et permet de définir en entête de ce fichier des attributs et méthodes complémentaires. Lors du parse, un attribut généré par Coco/R représente le *token* courant, et un autre le *token* suivant (*lookahead*). Dans le cadre de notre grammaire LL(1), seul le *token* courant nous intéresse, car il n'y a jamais d'ambiguïté quant au sens à donner à un élément d'une règle.

L'encapsulation des opérations de construction de l'AST dans la classe Parser permet d'obtenir des méthodes simplifiées en rendant implicites certains paramètres. Ainsi, seul le paramètre spécifiant le type de nœud à créer se retrouve dans les appels de méthodes dans les actions du *parser*. Cela rend plus aisé l'écriture et la relecture de ces actions.

Grâce aux mécanismes de vérification de la structure de l'AST préalablement écrits, il a été relativement aisé d'implanter l'ensemble des actions de construction. J'ai ainsi pu me concentrer sur les tâches plus cruciales, notamment la conception du parcours d'arbre et de l'environnement d'exécution.

3.F. Abstraction de parcours d'arbre

Lors de ma première tentative d'implantation de vérification sémantique utilisant l'AST, je me suis heurté à un problème de forme conséquent : le seul moyen d'écrire un comportement spécifique par type de nœud était d'utiliser un *switch*, ou une série de *if*, ce qui dans tous les cas conduisait à des méthodes très longues et illisibles. J'ai donc été obligé de revoir la conception et d'imaginer un moyen alternatif d'écrire les comportements spécifiques à chaque type de nœud.

Je me suis alors inspiré du patron de conception Visiteur [WK-V] en l'adaptant à mon contexte en utilisant la faculté d'introspection de Java. L'énumération utilisée pour les types de nœuds ne permettant pas de surcharge de méthodes, j'utilise l'introspection de Java pour transformer les labels des nœuds, qui sont finalement un typage virtuel, en noms de méthodes qui sont invoquées. Le *switch* se retrouve ainsi éclaté en autant de méthodes que nécessaire, ce qui clarifie énormément le code.

Le mécanisme de parcours d'arbre est indépendant du traitement qu'on effectue sur l'arbre, aussi les déplacements sont gérés par une classe différente de celle qui effectue le traitement, respectivement appelées Arpenteur et Acteur. L'Arpenteur conduit l'Acteur sur un nœud, celui-ci effectue son traitement sur ce nœud, spécifique à son type (le label du nœud), et en retour renvoie à l'Arpenteur la direction à prendre ensuite dans l'arbre.

Avec ce paradigme, chaque classe Acteur doit implanter un traitement bien spécifique, ce qui a l'avantage de bien structurer la conception ensuite. Ce mécanisme abstrait est suffisamment puissant pour servir à implanter tous les traitements nécessaires, des vérifications sémantiques à l'interprétation.

3.G. Applications avec les vérifications sémantiques et l'interprétation

Après un passage dans le *parser*, toute l'information contenue dans le fichier source se retrouve dans l'AST construit, il est donc théoriquement possible de la retrouver en parcourant l'arbre. En partant de ce point de vue, j'ai tenu à éviter de créer des tables de symboles qui auraient été redondantes. Ainsi la résolution de types, la recherche des déclarations de sous-programmes, etc. sont effectuées par des Acteurs spécialisés. Certains Acteurs utilisent les évaluations d'autres Acteurs, voire se réutilisent eux-même.

Pour toutes les évaluations statiques, dont le résultat dépend seulement de la structure de l'arbre (par exemple la résolution du type d'une variable ou de la déclaration de sous-programme correspondant à une instruction d'appel), le résultat est mis en cache afin d'améliorer les temps de traitement.

Lors de l'interprétation, il est néanmoins nécessaire de créer des tables dynamiquement pour stocker l'état des variables locales. Les constantes sont aussi stockées dans des tables par commodité. Le contexte d'exécution pour mener à bien l'interprétation est donc constitué de :

- un dictionnaire de constantes,
- une pile de dictionnaires de variables locales (glossaires), chaque appel de sous-programme empilant un nouveau glossaire,
- une pile d'appels (nécessaire uniquement pour le débogage interactif, mais non requis par le mécanisme d'interprétation).

3.H. Contrôle d'exécution pour l'interprétation

Le débogage interactif requiert d'interrompre et de reprendre l'exécution à la volée (cf. annexe page 60). J'ai utilisé pour ce faire un modèle basé sur deux fils d'exécution (*threads*). Le premier thread est celui du « client », utilisateur du débogage, et le second celui du « moteur », qui effectue l'exécution proprement dite. Entre les deux se trouve un contrôleur, qui en fonction des messages envoyés par le client va autoriser ou non le moteur à exécuter son instruction courante.

3.I. Mécanisme d'extension de la bibliothèque native du langage algorithmique

Mon maître de stage m'avait demandé dès le début du stage de m'intéresser à l'introspection de Java pour voir si cela pouvait servir à étendre simplement la bibliothèque du langage algorithmique. La première utilisation de l'introspection a finalement été l'arpentage de l'AST, et cela a ouvert la voie du mécanisme d'extension de la bibliothèque native.

Dans la première version, les sous-programmes natifs définis étaient implantés « en dur » dans le compilateur, et l'extension du langage nécessitait d'adapter le compilateur au nouveau besoin.

Le mécanisme implanté dans la deuxième version permet d'étendre la bibliothèque du langage algorithmique de façons presque naturelle. L'introspection de Java est utilisée quand une déclaration de sous-programme n'est pas suivie du corps du sous-programme. L'interpréteur va rechercher dans la librairie Java une classe possédant le même nom que le fichier (sans son extension) où est défini le sous-programme, et dans cette classe une méthode possédant le même nom et les mêmes types de paramètres que la déclaration du sous-programme. Le mode de passage est aussi pris en compte, si ce n'est que les modes *sortie* et *màj* sont confondus.

J'ai ainsi défini une classe *graphique* correspondant au fichier graphique.spec et une classe *entréeSortie* correspondant au fichier entréeSortie.spec, ces deux bibliothèques étant définies dans la première version de Compalgo.

J'ai découvert ce faisant une particularité de Java que sont les objets *immutables*. Tous les types de base et leurs *wrappers* de Java 1.5 (Byte, Integer, Float, String, Character…) ne sont pas modifiables une fois créés. Il était donc nécessaire pour gérer les modes de passage *sortie* et *màj* d'implanter des *wrappers* qui conservent une référence sur la variable locale à modifier plutôt que de passer directement les valeurs de ces variables aux méthodes Java.

L'implantation de nouveaux sous-programmes natifs devient légèrement moins naturelle qu'initialement prévu, le programmeur devant utiliser ces types *wrappers* dans les paramètres des méthodes Java.

3.J. Interface graphique

Les cours d'IHM, de programmation événementielle, et d'interface graphique en Java, prodigués pendant l'Année Spéciale, m'ont énormément servi dans cette partie pour concevoir l'interface graphique du débogage et l'implanter avec la librairie Swing de Java. J'ai pu rapidement implanter les grandes lignes de l'interface pour mieux perdre du temps sur les aspects bizarres et mal documentés de la librairie graphique de Java, comme le fait de devoir invoquer *revalidate*() avant *repaint*() après avoir modifié le contenu d'un JPanel pour que l'affichage daigne se mettre à jour.

3.K. Profilage et optimisation de l'interprétation

L'exécution était extrêmement lente du fait de tous les parcours d'arbre et d'invocations de méthodes engendrées.

J'ai d'abord implanté le cache d'évaluations statiques mentionné au point 3.G, puis effectué une série de profilages du code (temps d'exécution par méthode, nombre d'appels pour chaque méthode) afin d'améliorer les performances. J'ai pour ceci utilisé l'outil commercial YourKit Java Profiler, avec une licence d'essai temporaire, et optimisé en réduisant au fur et à mesure les goulots d'étranglement en termes d'occupation du processeur.

3.L. Intégration des nouveautés dans l'éditeur existant

Cette partie a certainement été la plus rapide du stage. En réalité, l'éditeur de la première version a été réutilisé dans le projet de la deuxième version, et j'ai modifié la console d'erreurs de compilation et l'action du bouton de compilation. J'ai aussi ajouté deux boutons, un pour exécuter le programme fraîchement compilé, et un pour lancer l'interface de débogage de ce même programme, ainsi que quelques autres modifications mineures.

3.M. Tests de non-régression

Il était plus qu'important qu'aucune des fonctionnalités de la première version de Compalgo ne soit réduite ou cassée, aussi la totalité des sujets de travaux pratiques donnés en cours d'algorithmique ont été compilés et leur exécution vérifiée, en plus des exemples servant à tester les nouvelles

fonctionnalités telles que la gestion des exceptions et les déclarations préalables d'entêtes de sous-programmes.

Partie III -
Bilan

1. Bilan du projet

Après quelques semaines de stage, j'ai cru pouvoir finir en avance sur la date de fin du stage, mais sans tenir compte de la Loi de Hofstadter. Le temps m'a vite rattrappé, et je réussis finalement à finir juste dans le temps imparti, mais quelques imperfections subsistent.

J'ai notamment perdu beaucoup de temps en phase de rédaction à cause d'une part de ma méconnaissance des éditeurs Microsoft Word et OpenOffice Writer, et d'autre part des bogues dont ils ne sont pas exempts.

Par ailleurs, il y a en général suffisamment de machines pour tout le monde en salles des machines, mais, malheureusement, il arrive qu'il y ait plus d'utilisateurs en présence que de machines disponibles, ce qui peut nuire à la productivité.

Je suis relativement content du *design* objet derrière la nouvelle version de Compalgo, les différents composants sont bien disjoints et coopèrent finement entre eux. Quand je compare la conception de ce projet avec mes précédents développements orientés-objets en C++, j'ai la sensation d'avoir réalisé ici mon premier vrai *design* objet.

Une des difficultés du projet était liée à la quantité de types de nœuds distincts de l'AST. J'ai choisi d'utiliser une énumération au lieu de sous-classer chaque type de nœud. Ce choix a simplifié la vision globale de la construction de l'arbre, mais m'a forcé à imaginer un moyen de singer la surcharge de méthodes qui est au centre du patron de conception Visiteur, qui est un patron idéal pour les traitements sur l'AST.

Je suis néanmoins conscient de quelques lacunes de conception pour la correction desquelles quelques semaines supplémentaires seraient nécessaires.

L'exécution pas à pas est à parfaire sur de nombreux points. L'utilisateur est obligé d'effectuer plusieurs pas au lancement du programme avant d'atteindre la première instruction, deux pas sont toujours nécessaires avant d'entrer dans une boucle, et l'avancée d'un pas « par-dessus » en début de programme exécute l'ensemble.

À côté de cela, le modèle de *threads* utilisé pour le contrôle de l'exécution est certes stable et réactif, néanmoins les deux threads passent la majeure partie de leur temps à attendre le signal de l'autre thread, ce qui consomme inutilement du temps processeur. Cela n'est un problème que lorsque la ventilation des processeurs est bruyante à plein régime.

Je n'ai pas encore eu le temps de finir la documentation des classes, mais le code Java reste facilement compréhensible, et le rapport de conception complète la Javadoc manquante.

2. Bilan professionnel

J'ai eu plusieurs expériences professionnelles avant de reprendre les études en Année Spéciale de DUT Informatique, entre 2001 et 2005, dans lesquelles j'avais été amené à gérer comme dans ce stage un projet informatique de l'analyse des besoins à la réalisation concrète de la solution logicielle.

Ce stage m'a permis de renouer avec ces sensations, réflexions, et décisions, en y apportant une touche supplémentaire de rigueur. Je me sens maintenant pleinement capable dans un rôle d'analyste-programmeur, bien que, contrairement à l'orientation de la formation de DUT, ce stage n'était pas orienté systèmes d'information.

3. Bilan personnel, technique et informatique

Ma culture en théorie des langages, compilateurs, et générateurs de compilateurs, avant ce stage, était superficielle et faite de bric et de broc. J'avais déjà auparavant écrit des *scanners* et des *parsers* utilisant la descente récursive sans vraiment en être conscient, et n'avais jamais utilisé de générateur de *parser*. J'ai pu grâce à ce stage combler mes lacunes théoriques et pratiques en la matière.

Également, ma culture concrète en programmation Java avant l'Année Spéciale se résumait à une simple applet monitorant la progression d'un transfert de fichier, applet qui avait été abandonnée à l'époque à cause des restrictions de sécurité des applets Java et du caractère peu pratique des certificats auto-signés. Le cours d'initiation prodigué pendant l'année m'a donné les bases pour pouvoir accepter le paradigme de Java et progresser dans l'utilisation de ce langage.

4. Bilan de la formation initiale

Le DUT Informatique est très axé sur les systèmes d'information, de leur analyse à la conception et la maintenance des bases de données afférentes. Le sujet de ce stage a surtout porté sur des objets secondaires de la formation tels que la programmation orientée-objets dans le langage Java et la conception d'interfaces graphiques. Néanmoins, loin d'être inutile, l'Année Spéciale m'a habitué à la rédaction de documents « standards », et a rectifié par des bases saines et rigoureuses nombre d'automatismes ou mauvaises pratiques acquis sur le tas.

Bibliographie

[ANTLR] *ANother Tool for Language Recognition*, Terence Parr, 2007
http://www.antlr.org/

[CA v1] *Compalgo première version*, G. Cabanac, M. Chevalier, O. Rouhaud, 2005
http://g.cabanac.free.fr/JAVA/COMPALGO/

[Coco/R] *The Compiler Generator Coco/R User Manual*, Hanspeter Mössenböck, Johannes Kepler, University Linz - Institute of System Software, 2006
http://www.ssw.uni-linz.ac.at/Coco/Doc/UserManual.pdf

[C.Wolf] *Principles of compiler design*, Clifford Wolf, December 22, 2004
http://www.clifford.at/papers/2004/compiler/

[IRIT] *Mot du directeur*, *in* portail de l'IRIT
http://www.irit.fr/-Le-mot-du-directeur-

[IUT] *Présentation générale*, *in* portail de l'IUT « A » de l'Université Toulouse III
http://www.iut-tlse3.fr/iut_pres_tlse.html

[RDP] *RDP Parser Generator*, Royal Holloway, University of London, 1998
http://www.cs.rhul.ac.uk/research/languages/projects/rdp.html

[WK-BNF] Wikipédia, article *Backus-Naur form*, 2007
http://en.wikipedia.org/wiki/Backus-Naur_form

[WK-EBNF] Wikipédia, article *EBNF*, 2007
http://en.wikipedia.org/wiki/Ebnf

[WP-LF] Wikipédia, article *Langage formel*, 2007
http://fr.wikipedia.org/wiki/Langage_formel

[WP-LL] Wikipédia, article *LL Parser*, 2007
http://en.wikipedia.org/wiki/LL_parser

[WK-V] Wikipédia, article *Visitor Pattern*, 2007
http://en.wikipedia.org/wiki/Visitor_pattern

[YK-JP] *Yourkit Java Profiler*, © YourKit LLC
http://www.yourkit.com/

Journal de stage

Semaine 1

- découverte de l'environnement de travail,
- rédaction d'une description de la chaîne logicielle de compilation et d'un comparatif succinct des générateurs de compilateurs existants en Java,
- rédaction de la formulation des besoins.

Semaine 2

- rédaction du test comparatif de quatre générateurs de compilateurs sélectionnés,
- réécriture de la grammaire de Compalgo pour obtenir la nature LL(1) et validation avec le générateur et validateur RDP,
- écriture d'un script de conversion de la grammaire formelle en une spécification de *parser* pour l'outil Coco/R à des fins de test,
- expérimentations sur un *design* de « machine virtuelle virtuelle » en attendant la validation du choix du générateur de compilateur à utiliser.

Semaine 3

- validation par le client du choix du générateur Coco/R,
- fin de la réécriture de la grammaire et début des tests,
- corrections dans la formulation des besoins et le test comparatif,
- réunion avec Max Chevalier et Christian Percebois pour définir les évolutions du langage,
- écriture et documentation de la structure de l'AST de Compalgo,
- début d'écriture du générateur d'AST avec Coco/R,
- réutilisation d'une interface graphique antérieure au stage pour visualiser et déboguer l'AST généré.

Semaine 4

- écriture des classes de parcours de l'AST,
- début d'implantation des vérifications sémantiques,
- début d'implantation de l'interpréteur d'AST de Compalgo.

Semaine 5

- débogage de l'interpréteur,
- intégration de l'interpréteur et de la « machine virtuelle virtuelle »,
- deuxième réunion avec Max Chevalier et Christian Percebois pour valider les choix quant à la gestion des exceptions dans le langage algorithmique,
- début de rédaction du rapport d'analyse.

Semaine 6

- livraison du rapport d'analyse,
- début de rédaction du rapport de conception.

Semaine 7

- livraison du rapport de conception,
- début d'implantation de l'interface de débogage.

Semaine 8

- implantation de l'interface de débogage,
- suite de l'implantation des vérifications sémantiques,
- profilage du code et optimisation de la vitesse d'exécution de l'interpréteur.

Semaine 9

- tests de non-régression,
- dernières corrections mineures dans la grammaire,
- intégration du nouveau compilateur et du débogueur dans l'éditeur de la première version de Compalgo,
- livraison de la beta-version,
- création des styles et de la structure du rapport de stage,
- correction de bugs suite au beta-test.

Semaine 10

- rédaction du rapport de stage,
- corrections finales,
- rédaction de fiches de procédure pour la maintenance de la grammaire formelle et pour l'écriture de librairies natives pour Compalgo.