Rapport

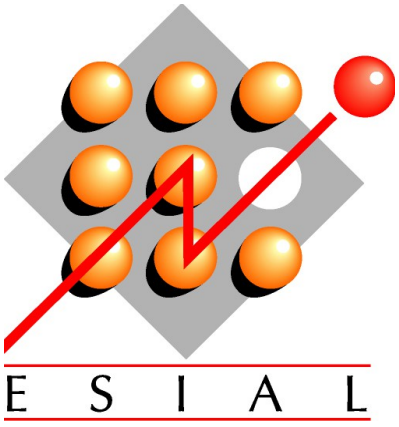
Écriture d'un compilateur

Traduction I ; 2011-2012

Julien VAUBOURG

Pierrick LAUFFENBURGER

2A / TRS



UNIVERSITÉ DE LORRAINE

INSTITUT TELECOM

Introduction

Durant ce projet, nous avons travaillé sur la conception d'un compilateur, qui transforme le code d'un langage inventé pour l'occasion, en code C.

Nous avons utilisé Antlr pour générer le *parser* et le *lexer* en java, à partir d'une grammaire conçue par nos soins à partir des contraintes fournies. Cette étape nous a aussi permise de faire générer l'arbre syntaxique (AST), accessible directement en java pour la suite des opérations.

Nous avons ensuite travaillé directement en java à partir des éléments générés pour concevoir les tables de symboles (organisées en pile de contextes), appliquer des fonctions de contrôle sémantique, et permettre la génération du code C final. Nous aurions pu continuer avec Antlr en utilisant une grammaire d'arbre, mais nous avons préféré supprimer cet intermédiaire qu'on soupçonnait parfois de nous induire en erreur.

Nous avons régulièrement procédé à des tests unitaires pour nous assurer que notre avancement ne mettait pas en péril ce qui avait déjà été fait. Ces tests sont visibles pour les trois dernières parties, grâces aux tests JUnit accessibles dans le projet.

Le plan de ce rapport sera le suivant :

I. Grammaire

II. Parcours de l'arbre syntaxique

III. Tables des symboles

IV. Fonctions sémantiques

V. Génération de code et jeux d'essais

VI. Limites du compilateur et feuilles de temps

Nous finirons en concluant à l'aide de nos impressions personnelles sur le déroulement et l'intérêt que nous avons dégagé de ce projet.

I. Grammaire

La première partie a consisté en l'écriture d'une grammaire au format d'Antlr, qui respecte l'ensemble des contraintes. Elle ne permet pas encore de supporter tout

ce qui était imposé dans le sujet, mais nous avons fait de notre mieux pour obtenir le plus de fonctionnalités supportées.

La grammaire obtenue est la suivante :

```
grammar Expr;

options {
  output=AST;
}

tokens {
  VAR_DEC;
  VAR_AFF;
  TAB_DEC;
  FUN_DEC;
  FUN_DEC_VAR;
  FUN_CALL;
  SEQ;
  BLOCK;
  BLOCK_DEC;
  VOID;
  AFF;
  FOR;
  WHILE;
  IF;
  ID_OR_AFF;
  TAB_ACCESS;
}

prog    :       exp;

vartabdec       :       'var'^ ID ':'! (tabdec | vardec);

vardec :        '=' operation_niv1 -> ^(VAR_AFF operation_niv1)
        |       TYPE ( ':' '=' operation_niv1 )? -> ^(VAR_DEC TYPE operation_niv1?)
        ;

tabdec :        TYPE '[' operation_niv1 ']' ( ':' '=' '{' operation_niv1 (',' operation_niv1)* '}' )?
                -> ^(TAB_DEC TYPE operation_niv1 operation_niv1*);

funcall :       ID '(' (operation_niv1 ( ',' operation_niv1 )*)? ')' -> ^(FUN_CALL ID
operation_niv1*);

fundec :        'function' ID ('()' | '(' id1=ID ':' t1=TYPE ( ',' id2=ID ':' t2=TYPE )* ')') ( ':'
ret=TYPE )? '='NEWLINE? operation_niv1
                -> ^(FUN_DEC ID ^(FUN_DEC_VAR  ($id1 $t1)?($id2 $t2)*)? $ret?
operation_niv1);

seq     :       '('NEWLINE* operation_niv1 ( NEWLINE*';'NEWLINE* operation_niv1  )*
NEWLINE* ')'
                -> ^(SEQ operation_niv1+);

block   :       'let'NEWLINE* (options {greedy=false;} : (vartabdec | fundec) NEWLINE)*
NEWLINE*'in'NEWLINE* (options {greedy=false;} : exp)?  'end'
                -> ^(BLOCK ^(BLOCK_DEC vartabdec* fundec*)? exp?);

exp     :       (operation_niv1 NEWLINE+)* -> operation_niv1*;
```

```
expLeft         :       vartabdec | fundec | block | for_loop | while_loop | if_then | seq |
funcall | lvalueAff
        |       '(')' -> ^(VOID)
        |       STRING
        |       '-'? REAL
        |       '-'? INT
        ;

lvalueAff       :       ID ( '[' op1=operation_niv1 ']' )? (':' '=' op2=operation_niv1)? ->
^(ID_OR_AFF ID ^(TAB_ACCESS $op1)? $op2?);

for_loop        :       'for' lvalueAff  'to' operation_niv1 NEWLINE*'do'NEWLINE*
operation_niv1
                        -> ^(FOR lvalueAff operation_niv1 operation_niv1);

while_loop      :       'while'NEWLINE* operation_niv1 NEWLINE*'do'NEWLINE*
operation_niv1
                        -> ^(WHILE operation_niv1 operation_niv1);

if_then :       'if' operation_niv1 NEWLINE*'then'NEWLINE* operation_niv1 (options
{greedy=true;} : 'else'NEWLINE* operation_niv1)?
                        -> ^(IF operation_niv1 operation_niv1 operation_niv1?);

operation_niv1: operation_niv2 (( '&&' | '||' )^ operation_niv2 )*;
operation_niv2: operation_niv3 (( 'v' | '>' | '<=' | '>=' | '=' | '<>' )^ operation_niv3 )*;
operation_niv3: operation_niv4 (( '-' | '+')^ operation_niv4 )*;
operation_niv4: operation_niv5 (( '*' | '/' )^ operation_niv5 )*;
operation_niv5: expLeft;

NEWLINE: '\r'? '\n';
WS: (' '|'\t')+ {$channel=HIDDEN;};
COM: '/*' ( options {greedy=false;} : . )* '*/' {$channel=HIDDEN;};

INT: ('0'..'9')+;
CHAR: '\'' . '\'';
TYPE: ( 'int' | 'real' | 'string' );
ID : ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;

REAL: (('0'..'9')+ '.' ('0'..'9')+ | ('0'..'9')+);

fragment ESC_SEQ: '\\' ('b'|'t'|'n'|'f'|'r'|'\"'|'\''|'\\');
STRING: '"' ( ~('\\'|'"') | ESC_SEQ )* '"';
```

II. Parcours de l'arbre syntaxique

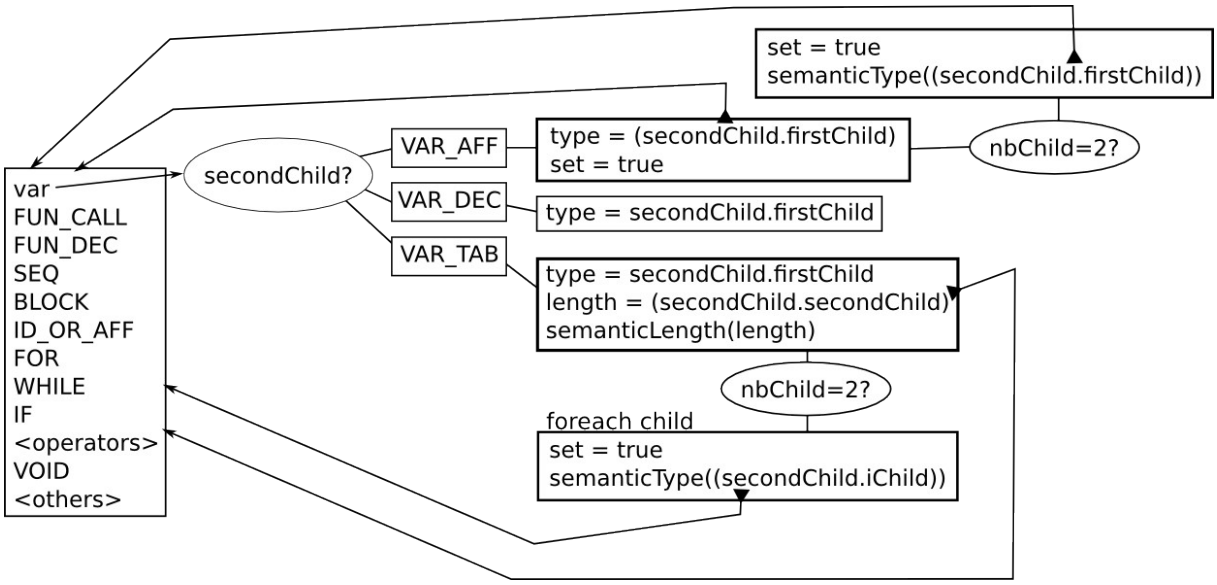
Notre programme parcours l'ensemble des nœuds de l'arbre syntaxique, de façon récursive. Une fonction générale est utilisée avec un *switch* qui appelle d'autres fonctions, selon le type du nœud rencontré.

Cette fonction renvoie systématique le type de retour généré par le sous-arbre qu'elle analyse : cette façon de faire nous permet de faire les contrôles de type.

Par exemple, pour l'affectation d'une variable, nous lançons l'analyse sur le membre droit, et nous exécutons une fonction sémantique de contrôle de type, selon ce que l'analyse a renvoyé et le type du membre gauche.

Si l'analyse détecte une erreur sémantique durant ce contrôle ou durant l'analyse du membre droit, une exception est lancée puis affichée, et le programme passe au sous-arbre suivant.

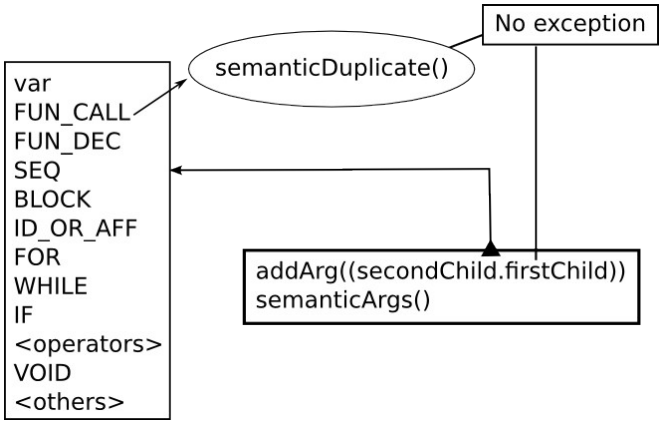
Sur le schéma ci-dessous, la fonction générale qui contient le *switch* est représentée par le bloc de gauche. Il s'agit de l'illustration du traitement d'un nœud *var*, qui représente la définition (avec ou sans initialisation) d'une variable ou d'un tableau. La fonction correspondante est appellée, et suit l'algorithme représenté. On peut remarquer les appels récursifs, qui permettent de descendre dans les sous-arbres et de faire les contrôles sémantiques de type.



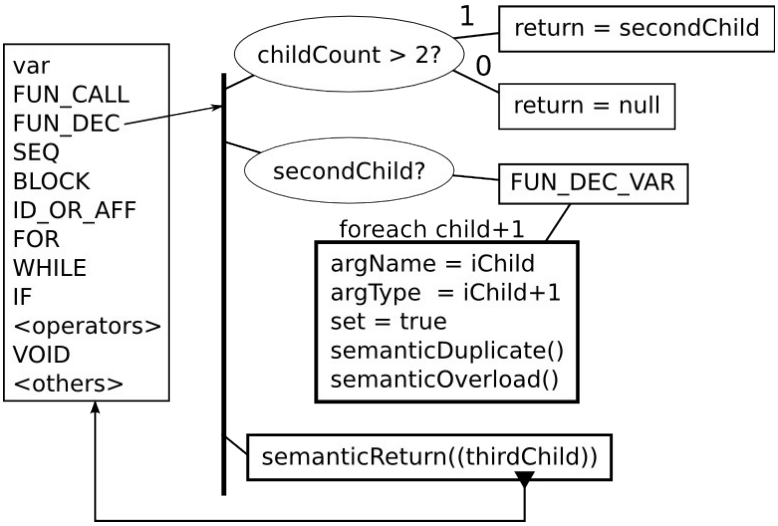
Notre AST a été conçu de telle sorte qu'il n'y ai jamais d'ambiguïté : nous pouvons déduire le type d'opération à partir du nombre et/ou des noms des enfants. Dans cet exemple, on vérifie le type de nœud pour le second fils de notre arbre (le premier fils étant souvent le nom attribué au symbole), et on en déduit le type d'instruction (définition de variable ou de tableau ? avec ou sans initialisation ?).

Voici le fonctionnement des autres fonctions de parcours d'arbre intéressantes, qui permettent de laisser transparaître la structure de notre arbre (nous n'avons pas trouvé de moyen efficace de donner la structure de notre arbre comme demandé, nous espèrons que les algorithmes qui l'exploitent directement seront suffisamment éloquants).

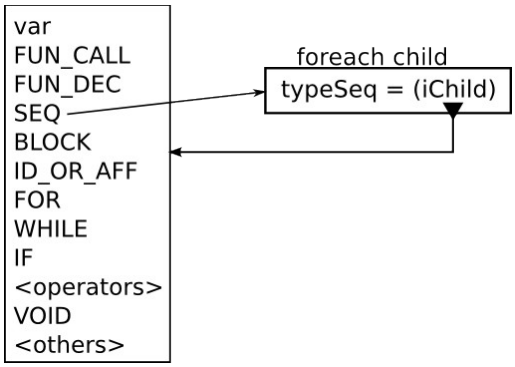
Parcours d'un noeud *FUN_CALL* (appel d'une fonction) :



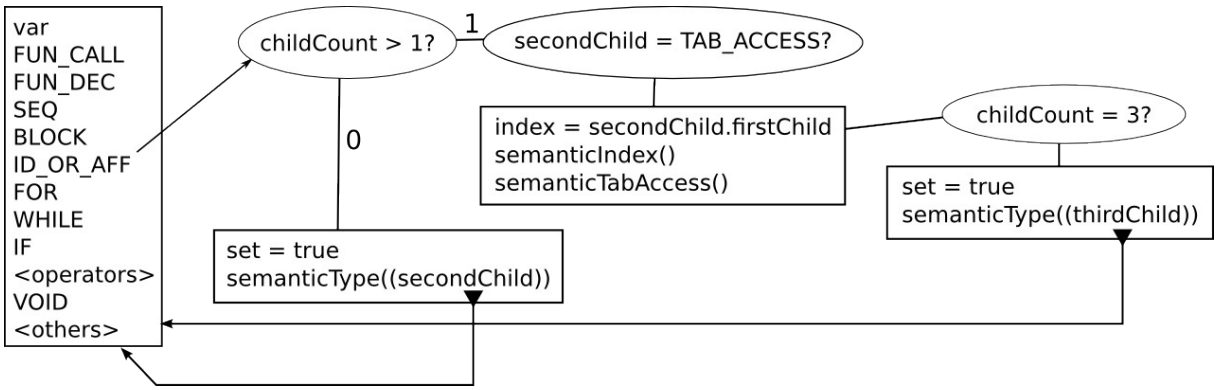
Parcours d'un nœud *FUN_DEC* (définition d'une fonction) :



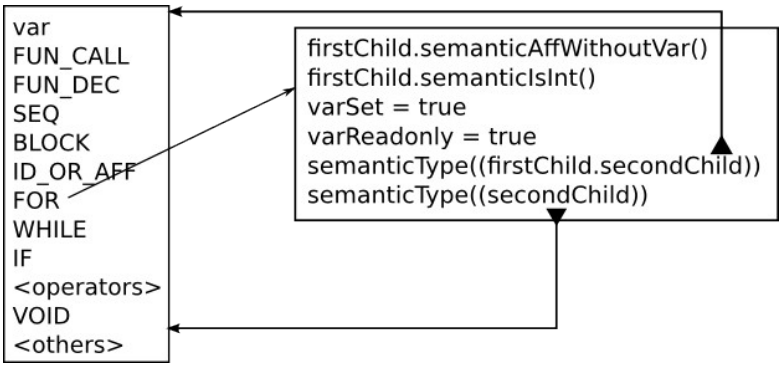
Parcours d'un nœud *SEQ* (utilisation d'une séquence), le *BLOCK* (*let in end*) étant relativement similaire :



Parcours d'un nœud *ID_OR_AFF* (utilisation d'un identifiant, ou affectation de celui-ci) :



Parcours d'un nœud *FOR* (utilisation d'une boucle *for*) :



Concernant le *while* : il s'agit d'un nœud *WHILE* qui contient deux sous-arbres, le premier étant la condition et le second le corps.

Pour le *if* : c'est un nœud *IF* qui contient deux ou trois sous-arbres, le premier correspondant toujours à la condition, le second au bloc *then* et le troisième au *else* (s'il est présent).

Concernant les opérations arithmétiques et booléennes, il s'agit toujours d'un noeud du nom de l'opération, avec deux sous-arbres pour chacun de ses membres.

Le nœud *VOID* présent dans le *switch* représente une instruction vide, qui a été utile pour définir notre grammaire. Le *others* représente un terminal, dont on détermine le type à partir d'une batterie de tests principalement basées sur des tentatives de conversion.

III. Tables de symboles

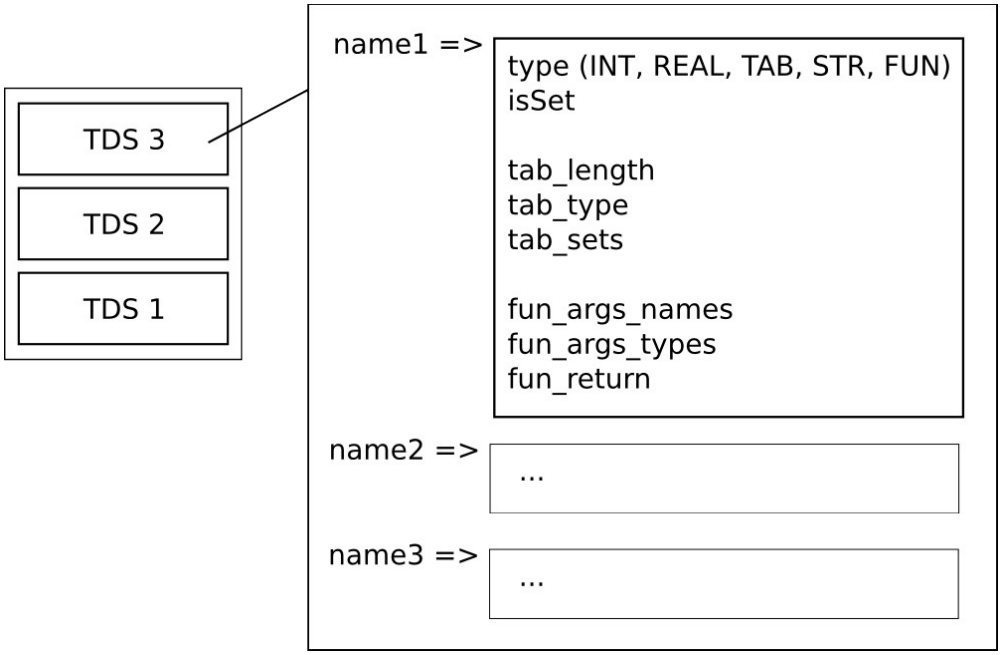
À chaque analyse de sous-arbre vue précédemment, les table de symboles et la pile des contextes qui les rassemble est complétée.

Un contexte est une couche (table de symboles), qui représente la portée de ce qu'elle contient. À chaque fois que notre parcours d'arbre analyse un sous-arbre qui permet la redéfinition de symboles en surchargeant ceux déjà définis, une nouvelle table de symboles est ajoutée à la pile des contextes. Lorsqu'on sort du parcours en question, elle est supprimée, avec tout ce qui a été ajouté dedans entre temps.

Une table des symboles correspond à une table de hashage avec pour clé le nom du symbole et pour valeur un objet qui le représente. Cet objet contient son type (*INT, REAL, TAB, STRING* ou *FUNCTION*) ainsi que tous les éléments qui permettent de le préciser.

Grâce à cet empilement, nous pouvons aisément effectuer des contrôles sémantiques sur leur portée. Par exemple, si on parcours un nœud *var* qui correspond à la définition d'une variable *x*, on vérifie qu'il n'y a pas de symbole de ce nom déjà défini dans le contexte actuel (le plus haut dans la pile). Si c'est le cas on lance une exception, sinon on vérifie qu'il n'existe pas déjà dans les contextes plus profonds. S'il existe dans un contexte plus profond, on se contente d'afficher un avertissement concernant la surcharge d'un symbole plus global. Dans ce dernier cas et dans le cas où le symbole n'est utilisé nul part dans la pile, on l'ajoute au contexte actuel, en récupérant l'ensemble des propriétés qui permettront les contrôles sémantiques suivants.

Schéma de la pile des contextes, et d'un élément d'une table de symboles :



Une table de symboles de base (*TDS 1*) est d'abord empilée dans la pile des contextes, en ajoutant les fonctions pré-existantes (*print*, get**).

À chaque changement de contexte (corps de fonction, séquence, nouveau bloc, etc), une nouvelle TDS est empilée en entrée et dépilée en sortie.

IV. Fonctions sémantiques

On peut noter dans les schémas précédents la présence d'appels de fonctions sémantiques durant les parcours d'arbres. Ils sont partout, et permettent de contrôler tout ce qui est analysé (et éventuellement destiné à être inséré dans une table de symboles).

Les contrôles sémantiques effectués sont les suivants :

- Pas assez ou trop de sous-arbres (ne devrait jamais arriver avec Antlr)
- Taille de tableau incorrecte (autre chose qu'un entier)
- Utilisation d'un tableau sans index (en tant que simple variable), à l'accès
- Utilisation d'un index hors bornes, à l'accès d'un élément d'un tableau
- Élément d'un tableau utilisé alors qu'il n'a pas été initialisé (ou que le tableau lui-même n'a pas été initialisé)

- Nombre incorrect de valeurs lors de l'initialisation d'un tableau, à la définition
- Utilisation d'une variable en tant que tableau, à l'accès
- Variable accédée inexistante dans la pile de contextes
- Variable accédée alors qu'elle n'a pas été initialisée
- Type du retour d'une instruction utilisée lors de l'accès à un élément d'un tableau (ou à l'appel d'une fonction, ou à l'affectation)
- Mauvaise assignation dans la définition d'une boucle *for*
- Utilisation d'autre chose qu'un entier pour le maximum d'une boucle *for*
- Modification d'une variable d'incrément d'une boucle *for*
- Utilisation d'autre chose qu'un entier dans une opération arithmétique (directement, ou par le biais du retour d'une instruction)
- Tentative de redéfinition d'une variable/tableau/fonction dans un même contexte (sinon, surcharge autorisée)
- Avertissement si la définition d'une variable/tableau/fonction surcharge un autre symbole défini dans un contexte plus profond
- Nombre d'arguments lors de l'appel d'une fonction
- Contrôle des types passés à l'appel d'une fonction
- Utilisation d'un autre symbole, en tant que fonction
- Retour incorrect d'une fonction
- Fonction appelée inexistante dans la pile de contextes

Lorsqu'un contrôle sémantique échoue, l'analyse du sous-arbre s'arrête, un message est affiché avec le numéro de ligne correspondant, et le programme continue le parcours.

Les erreurs qui relèvent du domaine syntaxique ou lexical sont automatiquement générées par les classes de Antlr, lors de leurs propres parcours et de la construction de l'arbre syntaxique.

V. Génération de code et jeux d'essais

La génération de code se fait durant le parcours des arbres syntaxiques, au fur et à mesure de la construction des tables de symboles et des contrôles sémantiques. Si tout s'est bien passé, le résultat est écrit dans le fichier cible en fin de parcours.

Une cinquantaine de tests JUnit sont disponibles dans les sources, pour tester les contrôles sémantiques et la génération de code.

Voici quelques exemples de compilation :

TODO : Exemples

Ainsi que quelques jeux d'essais pour tester :

TODO : Jeux d'essais représentatif qui montrent que ça r0xor

VI. Limites du compilateur et feuille de temps

Malgré notre archarnement pour réussir à obtenir un tout cohérent et fonctionnel, nous avons été contraint d'abandonner quelques aspects qui prenaient trop de temps à résoudre dans la situation dans laquelle nous étions :

- Notre grammaire impose les retours à la ligne, il en résulte qu'ils sont parfois obligatoires (ou non) à certains endroits, ce qui rend le code un peu trop rigide
- Nous n'avons jamais compris pourquoi Antlr partait complétement de travers lorsque nous lui imposions la possibilité d'avoir des caractères d'échappement dans les chaînes de caractères
- Nous n'avons pas eu le temps de gérer les tableaux à plus d'une seule dimension
- Problème notoire avec la génération des *return* des fonctions, durant la génération de code. Il s'agit d'un problème lié à l'aspect récursif de notre analyse : puisqu'une fonction peut se définir à l'aide d'un bloc ou d'une séquence qui retourne implicitement la valeur du dernier élément, il faudrait que celui-ci connaisse sa position pour ajouter le mot-clé *return* lors de la génération de son code. Or, en mode récursif, personne ne sait s'il est le dernier ou le premier de la liste. Nous n'avons pas eu le temps nécessaire pour résoudre ce problème

Concernant la feuille de temps, nous avons énormément travaillé ensemble. On dit que travailler à deux cerveaux sur un même clavier est plus efficace que de se répartir le travail. Étant donné la nature du projet (beaucoup de réflexion pour peu de lignes de code à taper), nous avons été parfaitement satisfaits de cette méthode de travail. Nous nous sommes un peu plus séparés pour la conception des fonctions de parcours d'arbres, qui sont plus indépendantes une fois le canevas général trouvé.

Nous estimons avoir passé environ 60 heures pour la conception d'une grammaire fonctionnelle avec l'arbre syntaxique, et environ 30 heures pour la réalisation des fonctions de parcours d'arbres (qui comprennent les TDS, les fonctions sémantiques et la génération de code). Le rapport nous aura pris trois demi-journées et une soirée.

Conclusion

Satisfaits de n'être pas devenus fous à l'issu de ce projet, celui-ci nous aura permis de parfaitement comprendre la chaîne de compilation d'un code.

L'expérience était très intéressante, et nous a donné l'occasion de découvrir énormément de problèmes liés notamment à l'analyse, qui n'étaient pas aussi évidents durant la partie théorique.

Nous ne pensions pas passer autant de temps sur la conception de la grammaire, qui nous a valu de recommencer plusieurs fois et de passer de nombreuses heures à chercher la bonne méthode pour lever les ambiguïtés, sans en soulever d'autres.

Malgré ces difficultés, une fois que notre AST était prêt et pensé sans ambiguïté pour le parcours, nous avons été agréablement surpris de l'efficacité d'un tel travail pour les contrôles sémantiques et la génération du code, qui se font naturellement en respectant les différentes étapes.