Projet de compilation

Modules PCL1

Octobre 2018 - Avril 2019

Membres :

Nathan BARLOY
Quentin MÉRITET
Lucas MINÉ
Guillaume VANNESSON

Superviseur :

Sébastien DA SYLVA

Table des matières

Introduction

Le projet de compilation présenté dans ce rapport s'inscrit dans la continuité du module de Traduction dont l'enseignement est prodigué par Madame Suzanne Collin et dont les travaux dirigés sont encadrés par Madame Suzanne Collin et Monsieur Sébastien Da Silva. Ce projet a pour objectif de développer un compilateur CLIFF (Compilateur d'un Langage Informatique au Fonctionnement Fonctionnel) du langage Tiger, langage décrit par Andrew Appel.

L'équipe ayant collaboré sur ce projet est constituée de quatre élèves de deuxième année à TÉLÉCOM NANCY en les personnes de Nathan BARLOY, Quentin MÉRITET, Lucas MINÉ et Guillaume VANNESSON.

Ce rapport présente la démarche mise en oeuvre et les tâches réalisées dans cette première partie du projet afin d'obtenir une grammaire LL(1), une génération de l'AST et le début de la construction de la table des symboles, étapes nécessaires à la réalisation de la seconde partie du projet, qui consistera à finaliser le compilateur en implémentant le contrôle sémantique et la génération de code assembleur.

1 Grammaire

1.1 Construction de la grammaire

Nous sommes partis de la grammaire donnée dans le sujet, qui n'est pas LL(1), avec comme but de la modifier petit à petit pour la rendre LL(1). Nous avons tout d'abord calculé les symboles directeurs de toutes les règles de la grammaire afin de mettre en évidence les problèmes de la grammaire pour qu'elle soit LL(1) (voir le compte rendu de la réunion du 17 octobre). Nous avons aussi identifié les récursivités à gauche ($exp/infixExp$ et $lValue/subscript/fieldExp$).
Une fois identifiées, nous avons enlevé les cycles de récursivité de la manière suivante :

L'ensemble des règles X de exp prennent la forme :

$$exp \longrightarrow X \quad infixExp$$

La règle $infixExp$ prend la forme :

$$infixExp \longrightarrow \textbf{infixOp} \quad exp \quad infixExp$$

Cette méthode est lourde et introduit d'autres problèmes, donc nous avons utilisé d'autres méthodes. On peut cependant déjà s'occuper de la factorisation des règles de grammaire qui commencent par les mêmes terminaux. Par exemple les règles $ifThen$ et $ifThenElse$ qui commencent par **if** exp **then** exp : on peut les réécrire en

$$ifThen \longrightarrow \textbf{if} \quad exp \quad \textbf{then} \quad exp \quad else$$

$$else \longrightarrow \textbf{then} \quad exp \quad |$$

Il a aussi fallut prendre en compte certaines priorités, notamment pour le if et les opérateurs. En effet, sans donner de priorités, le mot "**if** α **then if** β **then** γ **else** δ" a plusieurs dérivations possibles : le **else** peut se rapporter au premier if ou au second. Si les parenthèses ne précisent rien, on considère qu'il se rapporte au **if** qui le précède immédiatement. Pour implémenter cela, on utilise l'option greedy, qui permet d'associer à gauche immédiatement. Pour la priorité des opérateurs, on sépare ces derniers selon leur degré de priorité, dans cet ordre :

— les opérateurs logiques
— les opérateurs d'addition
— les opérateurs de multiplication

1.2 AST

Les règles de réécriture de la grammaire ont été définies de manière à créer un arbre abstrait utilisable par la suite pour la table des symboles.

Nous allons détailler une sélection pertinente d'arbres abstrait :

Remarque : ne pas considérer le nœud *nil* avant la racine, il est créé par l'outil **ANTLR-Works**.

1.2.1 Let In

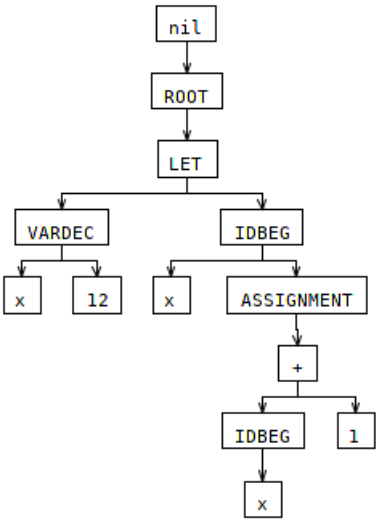


FIGURE 1 – AST : bloc LET

Dans cet arbre, le fils gauche *VARDEC* du nœud *LET* correspond à une déclaration de variable, donc dans le code après le *let* et avant le *in*. Le fils droit *IDBEG* est une expression (en l'occurrence une affectation). Elle est située dans le code après le *in* et avant le *end*.

Généralisation

Les déclarations (type, fonction, variable) seront toujours dans la partie gauche de l'arbre et auront pour nœud respectivement : *TYDEC, FUNDEC, VARDEC*. Si le nœud fils de *LET* n'est pas parmi les des trois sus-cités, cela signifie qu'il s'agit d'une expression et nous sommes donc après le *in*.

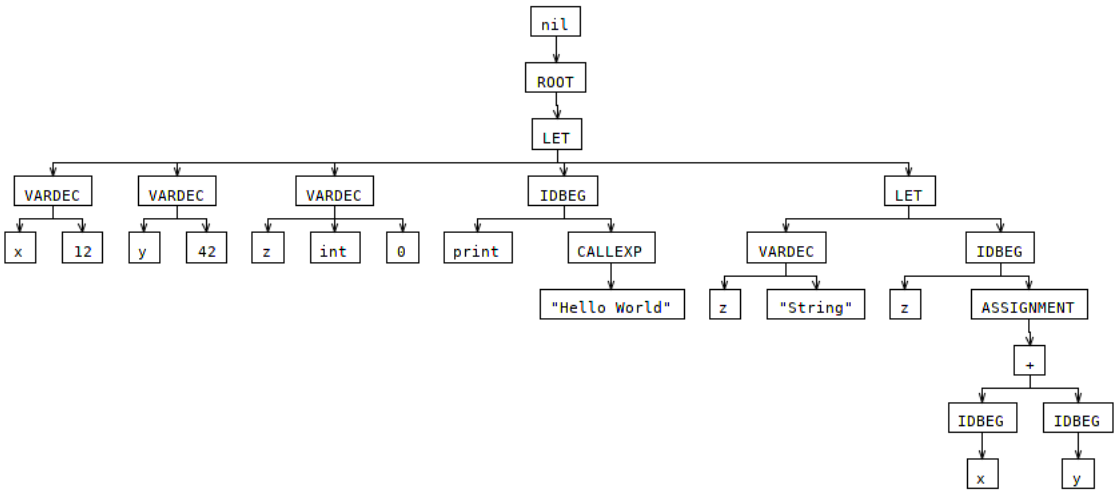


FIGURE 2 – AST : bloc LET généralisation

1.2.2 Déclaration de fonction

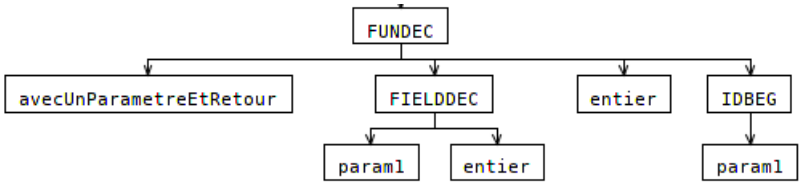


FIGURE 3 – AST : déclaration de fonction

Une déclaration de fonction correspond à un nœud *FUNDEC*. Le fils le plus à gauche est le nom de la fonction. Viens ensuite zéro, un ou plusieurs nœuds *FIELDDEC* correspondant aux paramètres. Deux cas sont alors possibles :

— Il y a encore deux nœuds fils : le premier correspond au type de retour, le second aux expressions effectuées par la fonction.

— Il ne reste qu'un seul nœud fils : il correspond aux expressions de la fonction.

1.2.3 Affectation de variable

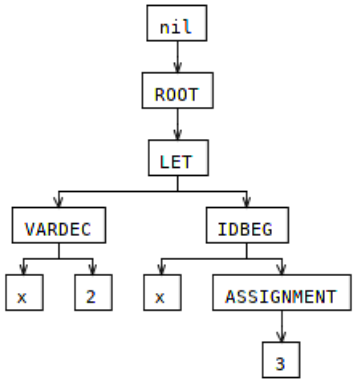


FIGURE 4 – AST : affectation

La partie correspondant à l'affectation est le sous-arbre ayant pour racine le fils droit du nœud *LET*. Le nœud *IDBEG* signifie que l'on effectue une opération sur une variable ou une fonction. Le fils gauche de ce nœud nous indique la variable concernée et le fils droit l'opération à effectuer, en l'occurrence une affectation. Ici, on affecte la valeur 3 à la variable x.

Cas d'un tableau

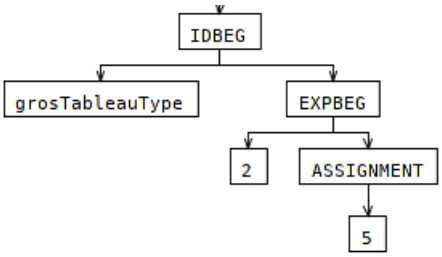


FIGURE 5 – AST : affectation - cas d'un tableau

Comme précédemment la racine est le nœud *IDBEG*. Cependant, le fils droit est *EXPBEG*, cela signifie que la variable $grosTableauType$ est un tableau. Le fils gauche de *EXPBEG* nous donne à quel indice nous souhaitons accéder et le fils droit nous indique l'opération à effectuer. On affecte donc la valeur 5 à l'indice 2 du tableau $grosTableauType$.

1.2.4 Appel de fonction

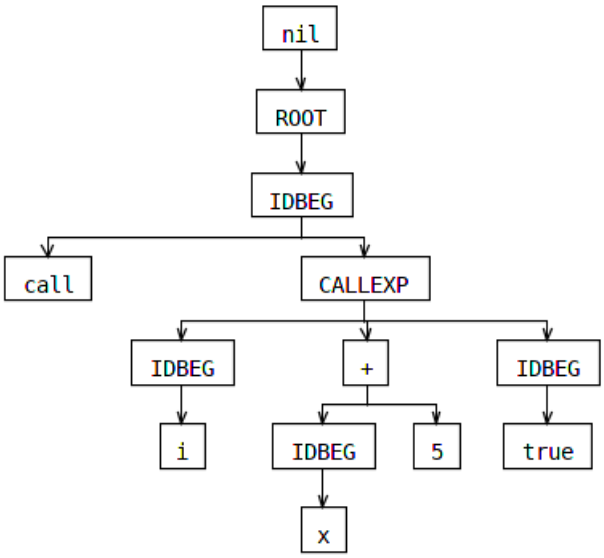


FIGURE 6 – AST : appel de fonction

L'appel de fonction débute avec le nœud *IDBEG*. Comme vu précédemment, le fil gauche nous indique l'identificateur concerné. Le fils droit quant à lui nous informe qu'il s'agit d'un appel de fonction et nous donne les paramètres à utiliser (il peut ne pas y en avoir).

1.2.5 Opérations

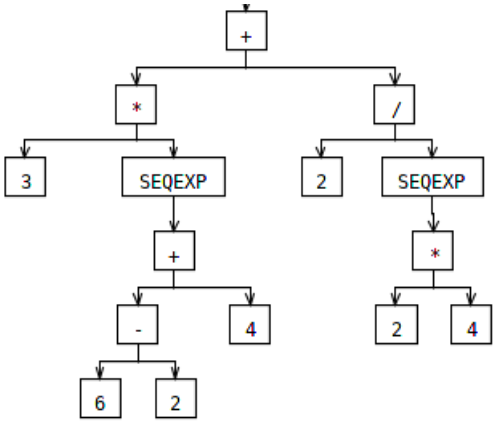


FIGURE 7 – AST : opérations

Cet arbre représente l'opération $3*(6-2+4)+2/(2*4)$. Le nœud *SEQEXP* symbolise l'usage de parenthèses. Les règles de priorités des opérateurs y sont bien respectées : les multiplications et divisions s'effectueront avant les additions ou soustractions.

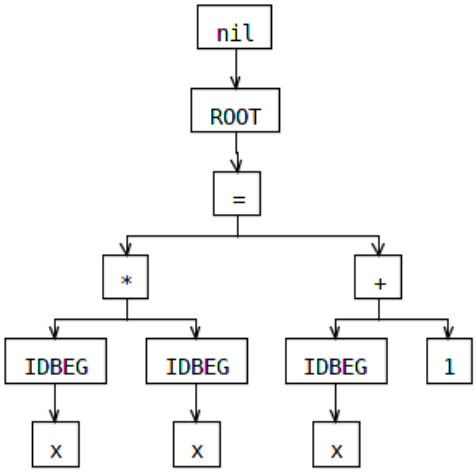


FIGURE 8 – AST : opérations logiques

L'exemple ici illustre la priorité des opérateurs de calcul par rapport aux opérateurs logique.

1.2.6 If Then Else

Le nœud *IFTHEN* possède soit deux fils, soit trois en fonction de la présence ou non du terminal $else$.

Cas 1 : If Then

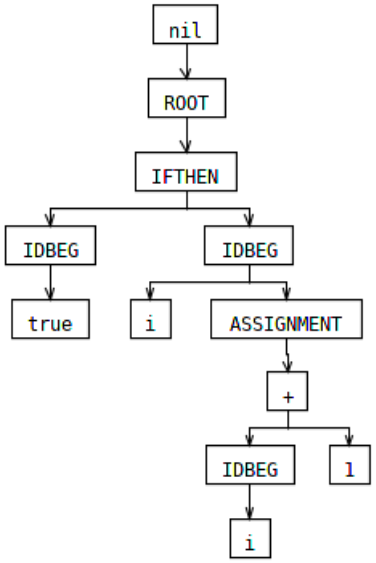


FIGURE 9 – AST : If Then

Le nœud *IFTHEN* a deux fils, il n'y a donc pas de non-terminal $else$ dans le code source. Le fils gauche correspond à la condition et le fils droit aux expressions à effectuer si la condition est vérifiée.

Cas 2 : If Then Else

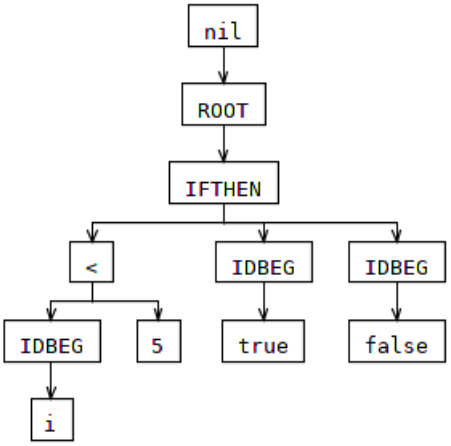


FIGURE 10 – AST : If Then Else

Le nœud *IFTHEN* a trois fils, le cas où la condition n'est pas vérifiée est donc géré. Il correspond au troisième nœud, les deux premiers étant les mêmes que dans le cas précédant.

1.2.7 Boucle While

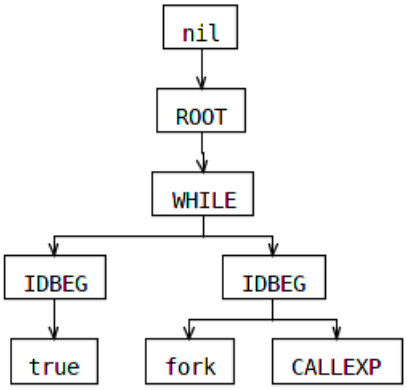


FIGURE 11 – AST : While

Le cas d'une boucle *while* fonctionne sur le même principe que celui du *If Then*. le fils gauche correspond aux conditions et le fils droit aux expressions à effectuer à chaque tour de boucle.

1.2.8 Boucle For

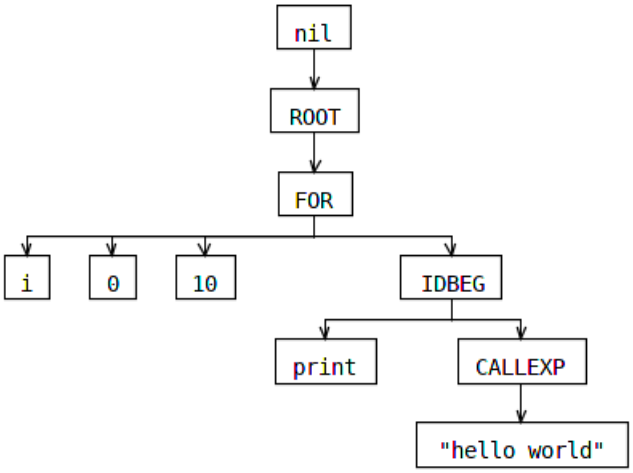


FIGURE 12 – AST : For

L'arbre abstrait d'une boucle *for* aura toujours quatre fils. Les trois premiers indiquent (de gauche à droite) : l'identificateur, sa valeur de début et sa valeur de fin. Le quatrième contient les instructions à effectuer à chaque itération.

1.3 Table Des Symboles

Une fois l'AST construit, il est possible de procéder à la construction de la table des symboles en réalisant un parcours préfixé dans ce dernier. La table des symboles est construite récursivement (de la même façon qu'est réalisé le parcours sur l'AST) en suivant la théorie vue dans le cours de Traduction, c'est-à-dire qu'une table connaît la table du bloc parent (d'imbrication supérieure).

1.3.1 Architecture de la classe TableSymboles

Notre classe JAVA permettant de représenter une table des symboles possède à l'heure actuelle trois attributs :

1. parent : Cet attribut est une référence vers la table des symboles du bloc parent, permettant de gérer les imbrications de blocs. Dans le cas de la table des symboles "générale", c'est-à-dire la table des symboles d'imbrication 0, cet attribut vaut null. Dans le cas d'une table d'imbrication égale ou supérieure à 1, l'attribut pointera donc vers la table du bloc englobant celui que la table actuelle définit.
2. map : Cet attribut associe des chaînes de caractères (type String) représentant les noms des variables à une instance de Variable qui contient les informations de cette variable.
3. fils : Cet attribut est une liste (ArrayList) stockant des références vers les fils de cette table des symboles, c'est à dire les tables des symboles imbriquées dans celle-ci.
4. profondeur : un entier qui indique la profondeur en imbrication de la TDS.

1.3.2 Classe Variable

Cette classe contient les informations nécessaires à la description d'une variable :

— son nom

— son type

— le déplacement dans la pile

Ces informations servent à l'analyse sémantique, ainsi qu'à l'execution.

1.3.3 Construction de la structure de la table des symboles

La table se construit à l'aide d'une méthode récursive qui parcours l'arbre (de manière préfixée donc). Lors du parcours, on détermine en fonction du la valeur portée par le noeud franchi si on entre dans un nouveau bloc, ou s'il faut ajouter une nouvelle variable.
Si le noeud rencontré définit effectivement un nouveau bloc (cas typique des déclarations de fonctions par exemple), on crée un nouvel objet TableSymboles (présenté précédemment) en indiquant la table des symboles actuelle comme parent, puis on continue le parcours avec cette nouvelle table (permettant donc l'imbrication).
Si le noeud déclare une nouvelle variable, on va créer une nouvelle instance de Variable, et la rajouter dans notre TDS.

1.3.4 Analyse sémantique

Cette analyse se fait en même temps que la construction de la TDS : on vérifie les règles sémantiques en fonction du noeud sur lequel on se situe. Par exemple, si on utilise une variable, on vérifie qu'elle a bien été déclarée. Si certaines règles ne sont pas respectées, on renvoie un message d'erreur personnalisé.

1.4 Tests

Afin de s'assurer que l'analyse syntaxique fonctionne, et ce de la même façon que précisé dans la spécification de la grammaire, nous nous sommes appuyés sur la grammaire que nous avions réalisée pour écrire les tests mais aussi sur la spécification de manière à être sûrs que nous n'avions pas ajouté ou supprimé des possibilités à la grammaire donnée dans la spécification.

1.4.1 Écriture des tests

Afin de s'assurer que notre grammaire fonctionne, nous avons réalisés les tests en "remontant" les règles, c'est-à-dire en partant des règles les plus proches des feuilles de l'AST puis en remontant afin de s'approcher de l'axiome. Nous avons donc commencé par les plus petites règles de la grammaire (telles que les déclarations de fonctions simple, variables et de types) puis nous sommes remontés sur les règles plus complexes (exp, ifThen, forExp, whileExp et déclarations de fonctions plus complexes) afin de nous assurer qu'un maximum de possibilités données par l'axiome étaient testées.

Nous avons également pu nous assurer à l'aide des tests fournis par les professeurs que des programmes complexes passaient les tests.

Nous avons également écrit des programmes ayant des erreurs syntaxique et/ou lexicales afin de s'assurer qu'elles ne soient pas acceptées par la grammaire.

1.4.2 Implémentation des tests

Les tests sont réalisés à l'aide de JUnit. Nous avons essayé de rendre les test au maximum génériques, c'est à dire que pour un nouveau code source test, un minimum de ligne de codes soit nécessaires dans le code Java qui réalise les tests. Pour cela, chaque dossier de test contient deux dossiers : un dossier avec des programmes fonctionnels qui sont acceptés par l'analyse syntaxique et un dossier avec des programmes non-fonctionnels. Ainsi, nous avons mis en place une méthode qui parcourt chaque répertoire en testant chaque fichier dans ce répertoire. On utilise les tests unitaires assertEquals ou assertNoEquals en fonction de s'il s'agit d'un fichier dans le répertoire *fonctionnels* ou *nonFonctionnels*. Comme JUnit s'arrête dès qu'un assertEquals pose un problème et en affichant le nom du fichier avant de réaliser un test sur ce dernier, il nous est possible de savoir dans un répertoire quel fichier pose problème et donc de résoudre les problèmes au fur et à mesure qu'ils se présentent. L'appel à assertEquals a quant à lui nécessité de rediriger le flux System.err dans un buffer afin de pouvoir vérifier que ce buffer était vide à la fin du test, le contraire signifiant que l'analyse à rencontré un souci. Ainsi, la mise en place d'un nouveau test ne nécessite que de créer une nouvelle méthode de test et d'appeler la méthode décrite précédemment en précisant dans quel répertoire se trouvent les fichiers sources que l'on veut tester pour ce test, soit 3 lignes de code Java par nouveau test.

1.4.3 Problèmes rencontrés

Les tests nous ont permis de corriger et d'améliorer notre grammaire sur certains points :

1. Les commentaires, que nous pensions ne pas fonctionner, s'avéraient ne pas fonctionner s'ils étaient seuls dans un fichier (car l'analyse pensait analyser un fichier vide et rencontrait directement <EOF>). C'est lors de l'écriture de test que nous avons compris que la règle fonctionnait.
2. Notre règle STRINGLIT, correspondant à une chaine de caractère, ne reconnaissait pas dans un premier temps les caractères accentués et provoquait une erreur à l'analyse. Après nous en être aperçu grâce aux tests, nous avons pu ajouter dans cette règle les caractères accentués et donc résoudre le problème.

2 Gestion de projet

2.1 Fiche de définition de projet

2.1.1 Titre abrégé du projet

Concevoir un compilateur CLIFF pour le langage **Tiger**.

2.1.2 Titre long du projet

L'objectif de ce projet est d'écrire un compilateur CLIFF (Compilateur d'un Langage Informatique au Fonctionnement Fonctionnel) du langage **Tiger**, langage décrit par Andrew Appel (Princeton University) dans son ouvrage sur la théorie de la compilation *Modern Compiler Implementation*. Ce compilateur produira en sortie du code assembleur **microPIUP/ASM**, code assembleur que nous avons étudié dans le module PFSI de première année.

2.1.3 Résumé

Le projet consiste à réaliser un programme réalisant l'ensemble des étapes d'un compilateur. Il utilisera l'outil **ANTLR** interfacé au langage JAVA pour générer l'analyseur lexical et syntaxique descendant. Un code assembleur au format **microPIUP/ASM** sera ensuite généré dans un fichier.

Le projet est séparé en deux parties :

- Partie 1 :
  - Prise en main du logiciel **ANTLR**
  - définition complète de la grammaire du langage
  - mise en place des règle de réécriture pour la construction des arbres abstraits
  - début de la création de la table des symboles
- Partie 2 :
  - Finalisation de la construction de la table des symboles
  - établissement des contrôles sémantiques
  - génération de code assembleur

Chacune des étapes du projet nécessitera des test afin de s'assurer de son bon fonctionnement.

Le projet donnera lieu à un rapport par partie complétée d'une démonstration :

Le premier rapport présentera la grammaire du langage, la structure de l'arbre abstrait et de la table de symboles, des jeux d'essais ainsi que les éléments de gestion de projet.

Le second rapport complétera le premier et rendra compte *a minima* de la structure de la table des symboles, des erreurs sémantiques traitées, des schémas de traduction pertinents, des jeux d'essais, une fiche d'évaluation de la répartition du travail, les divers comptes rendus et le Gantt final.

2.1.4 Finalités du projet

Ce projet à un but principalement éducatif. En effet, il nous permet de mettre en pratique les cours de MI, PFSI et Traduction et ainsi de comprendre comment un compilateur va analyser un code source et le traduire dans un langage compréhensible par un ordinateur.

2.1.5 Résultats attendus

Le programme final prend un code source **Tiger** en entrée et retourne un fichier en langage machine. Il est possible de mesurer la réussite du projet en considérant les différentes étapes du compilateur.

Critères de succès :

- la grammaire est LL(1)

— l'arbre abstrait est cohérent et permet d'établir la table des symbole facilement
— la table des symboles correspond au code source entré
— les erreurs sémantiques sont détectées
— le code assembleur effectue les opérations du code **Tiger**

2.2 Vérification préliminaire - Objectif SMART

Critère	Projet
Spécifique	Le projet s'applique au langage **Tiger** et les attentes sont claires.
Mesurable	Les différentes étapes sont facilement évaluables grâce à des jeux de tests.
Atteignable	Nous avons eu les cours théoriques nécessaire à la réalisation du projet.
Réaliste	L'ensemble du projet répartit sur deux semestres nous assure d'avoir le temps de le réaliser. De plus des séances de TP sont disposées dans l'emploi du temps pour d'éventuelles questions.
Temporellement défini	La première réunion à lieu le 10 octobre 2018. La première partie est due pour le 11 décembre. La seconde pour le 23 avril 2019.

2.2.1 Moyens et ressources

— Les plate-formes $gitlab$ et $shareLaTeX$ sont à disposition.
— Les librairie **ANTLR** et **ANTLRWORKS** servent à compiler et déboguer la grammaire.
— Les mercredi après-midi sont réservés aux réunions pour le projet.

2.3 Savoirs à mettre en oeuvre

— Outils de gestion de projet.
— Théorie des langages.
— Construction d'arbres abstrait.
— Parcours d'arbre.
— Table des symboles.
— Contrôles sémantiques.
— Génération de code assembleur.

2.3.1 Membres de l'équipe

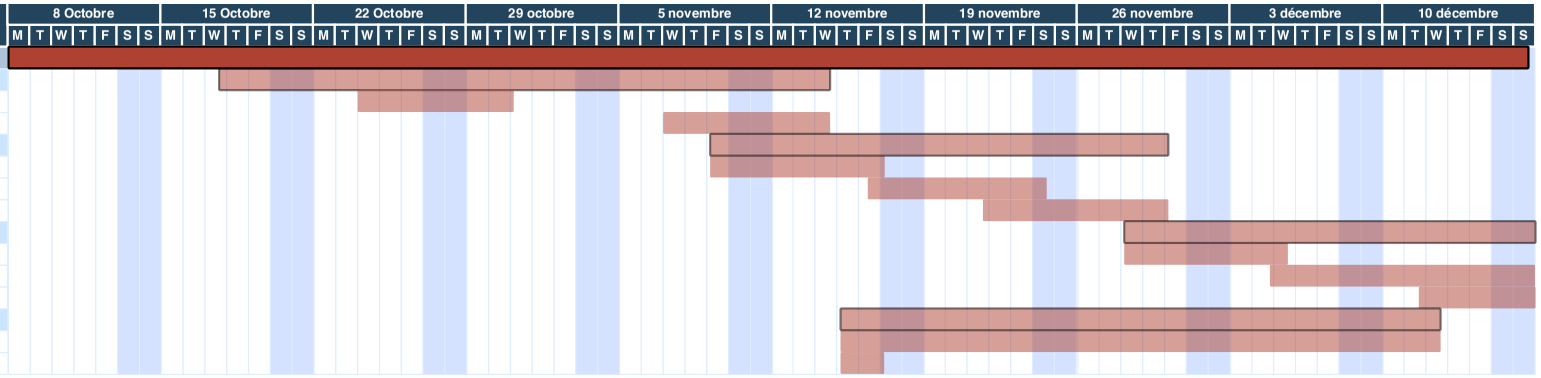
Civilité	Nom	Prénom	Adresse e-mail	Rôle
M.	VANNESSON	Guillaume	guillaume.vannesson@telecomnancy.eu	Chef de projet
M.	MÉRITET	Quentin	quentin.meritet@telecomnancy.eu	Développeur et animateur de réunion
M.	MINÉ	Lucas	lucas.mine@telecomnancy.eu	Développeur et secrétaire
M.	BARLOY	Nathan	nathan.barloy@telecomnancy.eu	Développeur

2.4 Répartition des tâches

Nom de la tâche	Nathan B.	Quentin M.	Lucas M.	Guillaume V.
Suppression récursivité à gauche	4 heures	4 heures	4 heures	2 heures
Priorité des opérateurs	6 heures	6 heures	6 heures	4 heures
Mise en place de l'AST	3 heures	4 heures	4 heures	3 heures
Améliorations et corrections	2 heures	2 heures	2 heures	2 heures
Parcours de l'AST	1 heure			1 heure
Création des TDS lors du parcours	1 heure		1 heure	1 heure
Ajout des informations dans la TDS				
Écriture de code source Tiger à tester	1 heure		3 heures	3 heures
Générisation des méthodes de test JAVA				2 heures
Rédaction du rapport	2 heures	4 heures	4 heures	3 heures
Rédaction des éléments de gestion de projet		2 heures	1 heure	1 heure
Total	20 heures	22 heures	25 heures	22 heures

2.5 GANTT

	Nom des tâches	Durée	Début	Fin
1	Projet fini	70 jours	08.10.18	17.12.18
2	Mise en place d'une grammaire LL(1)	28 jours	17.10.18	14.11.18
3	Suppression récursivité à gauche	7 jours	24.10.18	31.10.18
4	Priorité des Opérateurs	8 jours	07.11.18	15.11.18
5	AST	21 jours	09.11.18	29.11.18
6	Mise en place de l'AST	8 jours	09.11.18	16.11.18
7	Règle de réécritures dans la grammaire	8 jours	16.11.18	24.11.18
8	Amélioration et correction	8 jours	21.11.18	29.11.18
9	Table des symboles	19 jours	28.11.18	17.12.18
10	Parcours de l'AST	8 jours	28.11.18	05.12.18
11	Création des TDS lors du parcours	13 jours	05.12.18	17.12.18
12	Ajout des informations dans la TDS	6 jours	11.12.18	17.12.18
13	Tests	28 jours	14.11.18	12.12.18
14	Écriture de code source Tiger à tester	28 jours	14.11.18	12.12.18
15	Générisation des méthodes de test JAVA	2 jours	14.11.18	16.11.18



2.6 SWOT

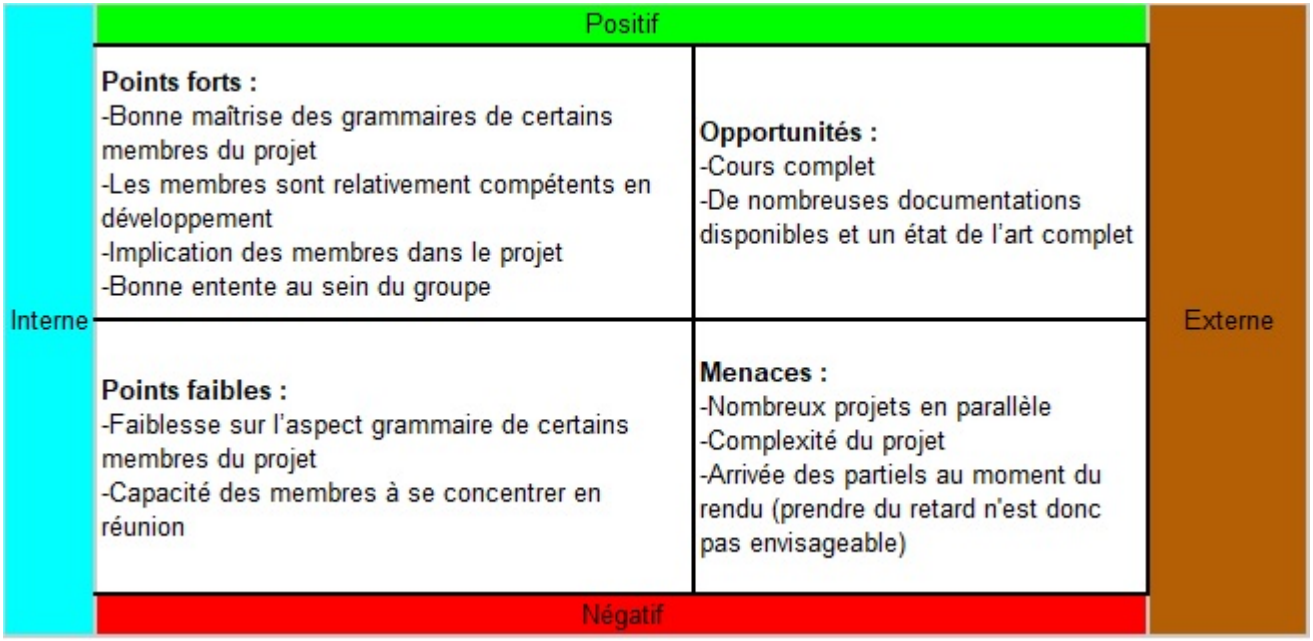


FIGURE 13 – Matrice SWOT

Conclusion

Dans ce rapport, nous avons pu voir la démarche mise en oeuvre ainsi que les difficultés rencontrées afin d'obtenir une grammaire LL(1), de mettre en place la génération des AST et de réaliser la création de la table des symboles.

L'objectif de cette première partie du projet étant de mettre en place les éléments présentés précédemment afin d'assurer une base stable pour la suite du projet, c'est-à-dire la mise en place de l'analyse sémantique et la génération de code, et au vu de l'avancement du projet et du fonctionnement des éléments nécessaires, tout semble être en place pour réaliser la suite du projet dans de bonnes conditions.

3 Annexes

Grammaire Tiger.g sans les règles de réécriture.

```
grammar Tiger;

options {
//language = JAVA;
output = AST;
backtrack = false;
k=1 ;
}

tokens {
        //tokens des r gles de r  criture
        MULTEXP ;
        FIELDEXP ;
        LETEXP;
        IFTHEN ;
        RECTY ;
        ARRTY ;
        TYDEC ;
        ASSIGNMENT ;
        ROOT ;
        WHILE ;
        VAR;
        FOR ;
        LET;
        NEGATION ;
        RETURNTYPE ;
        VARDEC ;
        VD ;
        EXPSTOR ;
        IDSTOR ;
        STRINGLIT ;
        NIL ;
        BREAK ;
        IDBEG ;
        EXPBEG ;
        FIELDDEC ;
        BRACBEG ;
        FIELDCREATE ;
        IFTHEN ;
        LET ;
        ID ;
        INTLIT ;
        SEQEXP ;
        CALLEXP ;
        FUNDEC;
}

program
        : exp
        ;

dec
        : tyDec
```

```
        | varDec
        | funDec
        ;

tyDec
        : 'type' tyid '=' ty
        ;

ty
        : tyid
        | arrTy
        | recTy
        ;

arrTy
        : 'array' 'of' tyid
        ;

recTy
        : '{' (fieldDec (',' fieldDec)*)? '}'
        ;

fieldDec
        : ID ':' tyid -> ^(FIELDDEC ID tyid)
        ;

funDec
        : 'function' ID '(' (fieldDec(',' fieldDec)*)? ')' returnType? '=' exp
        ;

returnType
        : ':' tyid
        ;

varDec
        : 'var' ID vd? ':=' exp
        ;

vd
        : ':' tyid
        ;

lValue
        : '[' exp ']' lValue
        | '.' ID lValue
        | assignment
        |
        ;

assignment
        : ':=' exp
        ;

exp
        : e (options{greedy=true;}: logop e)*
        ;

e
```

```
        : multExp (options{greedy=true;}: addop multExp)*
        ;

multExp
        : atom (options{greedy=true;}: multop atom)*
        ;

atom
        : 'nil'
        | INTLIT
        | STRINGLIT
        | seqExp
        | negation
        | ID idBegin
        | ifThen
        | whileExp
        | forExp
        | 'break'
        | letExp
        ;

seqExp
        : '(' (exp (';' exp)*)? ')'
        ;

negation
        : '-' exp -
        ;

idBegin
        : '[' exp ']' bracBegin
<<<<<<< HEAD
        | '.' ID lValue
        | '{' (fieldCreate(',' fieldCreate)*)? '}'
=======
        | '.' ID lValue
        | '{' (fieldCreate(',' fieldCreate)*)? '}'
        | assignment
        | '(' (exp(',' exp)*)? ')'
        |
        ;

bracBegin
        :  'of' exp
        | lValue
        ;

fieldCreate
        : ID '=' exp
        ;


ifThen
        : 'if' exp 'then' exp (options{greedy=true;}: 'else' exp)?
        ;

whileExp
        : 'while' exp 'do' exp
```

```
        ;

forExp
        : 'for' ID ':=' exp 'to' exp 'do' exp
        ;

letExp
        : 'let' dec+ 'in' (exp(';' exp)*)? 'end'
        ;

tyid
        : ID
        ;

addop
        : '+'
        | '-'
        ;

multop
        : '*'
        | '/'
        ;

logop   : '='
        | '<>'
        | '>'
        | '<'
        | '>='
        | '<='
        | '&'
        | '|'
        ;

//definition des expressions regulieres reconnaissant les tokens

ID      :       ('a'..'z' | 'A'..'Z') ('a'..'z' | 'A'..'Z' | ('0'..'9') | '_')*
        ;

INTLIT
        :       ('0'..'9')+
        ;

STRINGLIT
        :       '"' ('a'..'z'
        | 'A'..'Z'
        | '0'..'9'
        |'!'
        |'#'..'@'
        |' '
        |'\u00C0'..'\u00D6' //caract res accentu s
        |'\u00D8'..'\u00F6'
        |'\u00F8'..'\u02FF'
        |'\u0370'..'\u037D'
        |'\u037F'..'\u1FFF'
        |'\u200C'..'\u200D'
    |'\u2070'..'\u218F'
    |'\u2C00'..'\u2FEF'
```

```
    |'\u3001'..'\uD7FF'
    |'\uF900'..'\uFDCF'
    |'\uFDF0'..'\uFFFD')*  '"'
        ;


WS : (' ' | '\t' | '\n' | '\r' | '/*'.*'*/' | '//'.*('\r'|'\n'))+ {$channel = HIDDEN; }
   ;
```

Grammaire Tiger.g avec les règles de réécriture.

```
grammar Tiger;

options {
//language = JAVA;
output = AST;
backtrack = false;
k=1 ;
}

tokens {
        //tokens des regles de reecriture
        MULTEXP ;
        RECCREATE ;
        FIELDEXP ;
        LETEXP;
        IFTHEN ;
        RECTY ;
        ARRTY ;
        TYDEC ;
        ASSIGNMENT ;
        ROOT ;
        WHILE ;
        VAR;
        FOR ;
        LET;
        NEGATION ;
        RETURNTYPE ;
        VARDEC ;
        VD ;
        EXPSTOR ;
        IDSTOR ;
        STRINGLIT ;
        NIL ;
        BREAK ;
        IDBEG ;
        EXPBEG ;
        FIELDDEC ;
        BRACBEG ;
        FIELDCREATE ;
        IFTHEN ;
        LET ;
        ID ;
        INTLIT ;
        SEQEXP ;
        CALLEXP ;
        FUNDEC;
}

program
        : exp -> ^(ROOT exp)
        ;

dec
        : tyDec
        | varDec
        | funDec
```

```
        ;

tyDec
        : 'type' tyid '=' ty -> ^(TYDEC tyid ty)
        ;

ty
        : tyid
        | arrTy
        | recTy
        ;

arrTy
        : 'array' 'of' tyid -> ^(ARRTY tyid)
        ;

recTy
        : '{' (fieldDec (',' fieldDec)*)? '}'  -> ^(RECTY fieldDec*)
        ;

fieldDec
        : ID ':' tyid -> ^(FIELDDEC ID tyid)
        ;

funDec
        : 'function' ID '(' (fieldDec(',' fieldDec)*)? ')' returnType? '=' exp -> ^(FUND
        ;

returnType
        : ':' tyid -> tyid
        ;

varDec
        : 'var' ID vd? ':=' exp -> ^(VARDEC ID vd? exp)
        ;

vd
        : ':' tyid -> tyid
        ;

lValue
        : '[' exp ']' lValue -> ^(EXPSTOR exp lValue?)
        | '.' ID lValue -> ^(IDSTOR ID lValue?)
        | assignment
        |
        ;

assignment
        : ':=' exp -> ^(ASSIGNMENT exp)
        ;

exp
        : e (options{greedy=true;}: logop^ e)*
        ;
/*      : 'nil'
        | INTLIT
        | STRINGLIT
        | seqExp
```

```
        | negation
        | ID idBegin
        | ifThen
        | whileExp
        | forExp
        | 'break'
        | letExp                    */

e
        : multExp (options{greedy=true;}: addop^ multExp)*  //-> ^(multExp (addop multEx
        ;

multExp
        : atom (options{greedy=true;}: multop^ atom)*  //-> ^(atom (multop atom)*)
        ;

atom
        : 'nil'
        | INTLIT
        | STRINGLIT
        | seqExp
        | negation
        | ID idBegin  -> ^(IDBEG ID idBegin?)
        | ifThen
        | whileExp
        | forExp
        | 'break'
        | letExp
        ;

seqExp
        : '(' (exp (';' exp)*)? ')' -> ^(SEQEXP exp*)
        ;

negation
        : '-' exp -> ^(NEGATION exp)
        ;

idBegin
        : '[' exp ']' bracBegin                                          -> ^(EXPBEG exp
        | '.' ID lValue                                                        -> ^(FIE
        | '{' (fieldCreate(',' fieldCreate)*)? '}'      -> ^(RECCREATE fieldCreate*)
        | assignment
        | '(' (exp(',' exp)*)? ')'                                       -> ^(CALLEXP exp
        |
        ;

bracBegin
        :  'of' exp     -> ^(BRACBEG exp)
        | lValue
        ;

fieldCreate
        : ID '=' exp    -> ^(FIELDCREATE ID exp)
        ;


ifThen
```

```
        : 'if' exp 'then' exp (options{greedy=true;}: 'else' exp)?      -> ^(IFTHEN exp
        ;

whileExp
        : 'while' exp 'do' exp  -> ^(WHILE exp exp)
        ;

forExp
        : 'for' ID ':=' exp 'to' exp 'do' exp -> ^(FOR ID exp exp exp)
        ;

letExp
        : 'let' dec+ 'in' (exp(';' exp)*)? 'end' -> ^(LET dec+ exp*)
        ;

tyid
        : ID
        ;

addop
        : '+'
        | '-'
        ;

multop
        : '*'
        | '/'
        ;

logop   : '='
        | '<>'
        | '>'
        | '<'
        | '>='
        | '<='
        | '&'
        | '|'
        ;

//definition des expressions regulieres reconnaissant les tokens

ID      :       ('a'..'z' | 'A'..'Z') ('a'..'z' | 'A'..'Z' | ('0'..'9') | '_')*
        ;

INTLIT
        :       ('0'..'9')+
        ;

STRINGLIT
        :       '"' ('a'..'z'
        | 'A'..'Z'
        | '0'..'9'
        |'!'
        |'#'..'@'
        |' '
        |'\u00C0'..'\u00D6' //caracteres accentues
        |'\u00D8'..'\u00F6'
        |'\u00F8'..'\u02FF'
```

```
        |'\u0370'..'\u037D'
        |'\u037F'..'\u1FFF'
        |'\u200C'..'\u200D'
    |'\u2070'..'\u218F'
    |'\u2C00'..'\u2FEF'
    |'\u3001'..'\uD7FF'
    |'\uF900'..'\uFDCF'
    |'\uFDF0'..'\uFFFD')* '"'
        ;


WS : (' ' | '\t' | '\n' | '\r' | '/*'.*'*/' | '//'.*('\r'|'\n'))+ {$channel = HIDDEN; }
    ;
```

Compte rendu réunion

Minutes for 10/10/18

Present: Guillaume VANNESSON Nathan BARLOY Quentin MÉRITET Lucas MINÉ

Réunion tenue à Télécom Nancy

Ordre du jour

- Attribution des rôles
- Compréhension global du sujet
- Organisation du code
- Rétro-planning
- Gestion de projet

L'animateur de la séance est Quentin MÉRITET et le secrétaire : Lucas MINÉ. La séance a débuté à 14h30.

Attribution des rôles

Chef de projet : Guillaume VANNESSON
Secrétaire : Lucas MINE
Animateur de réunion : Quentin MERITET
Développeur : Nathan BARLOY

Compréhension global du sujet

Le sujet a été lu et les étapes principales ont été soulignées :

- Définition de la grammaire,
- analyse lexicale,
- analyse syntaxique,
- construction de l'arbre abstrait,
- table des symboles.

Organisation du code

Un drive d'équipe et un dépôt Gitlab ont été créés.
Les commits doivent être explicite.
Chaque fonctionnalité implémentée va de paire avec des tests.
La présentation du code doit être la suivante :

- Les noms de variables et fonctions suivent la convention : nomDeVariable,
- les noms de variables et fonctions sont en anglais,
- chaque fonction dispose d'une documentation,
- retour à la ligne avec l'accolade ouvrante de nouveau block.

Rétro-planning

La découverte du logiciel ANTLR se fera vendredi 12 octobre. On se laisse une semaine pour prendre en main le logiciel.

La deadline est le 12 décembre.

Gestion de projet

Seront réalisé :

— Fiche de définition projet,

— GANTT,

— matrice SWOT,

— matrice RACI

Tous les sujets à l'ordre du jour ont été traités. Fin de la réunion à 15h20.

Next Meeting: Mercredi 17 Octobre à 14h

Compte rendu réunion technique

Minutes for 17/10/18

Present: Guillaume VANNESSON Nathan BARLOY Quentin MÉRITET Lucas MINÉ

Réunion tenue à Télécom Nancy

Ordre du jour

— Étude de la grammaire donnée dans les spécifications du langage

L'animateur de la séance est Quentin MÉRITET et le secrétaire : Lucas MINÉ. La séance a débuté à 14h00.

Étude de la grammaire donnée dans les spécifications du langage

Le but est de mettre en évidence ce qui gène dans la grammaire pour qu'elle soit LL(1) et d'identifier les récursivités à gauche. On saura par la suite les modifications à effectuer.

Les premiers et les suivant de la grammaire ont été déterminés.

Non terminaux	Premiers	Suivants
program	id nil intList StringLit (- tyId if while for break let	$
dec	type var function	in type var function
tyDec	type	in type var function
ty	tyId array {	in type var function
arrTy	array	in type var function
recTy	{	in type var function
fieldDec	id	,) }
funDec	function	in type var function
varDec	var	in type var function
lValue	id	end ; to do then else , }] infixOp) in type var function $ [; :=
subscript	id	end ; to do then else , }] infixOp) in type var function $ [; :=
fieldExp	id	end ; to do then else , }] infixOp) in type var function $ [; :=
exp	id nil intList StringLit (- tyId if while for break let	end ; to do then else , }] infixOp) in type var function $

seqExp	(	end ; to do then else , }] infixOp) in type var function $
negation	-	end ; to do then else , }] infixOp) in type var function $
callExp	id	end ; to do then else , }] infixOp) in type var function $
infixExp	id nil intList StringLit (- tyId if while for break let	end ; to do then else , }] infixOp) in type var function $
arrCreate	tyId	end ; to do then else , }] infixOp) in type var function $
recCreate	tyId	end ; to do then else , }] infixOp) in type var function $
fieldCreate	id	, }
assignement	id	end ; to do then else , }] infixOp) in type var function $
ifThenElse	if	end ; to do then else , }] infixOp) in type var function $
ifThen	if	end ; to do then else , }] infixOp) in type var function $
whileExp	while	end ; to do then else , }] infixOp) in type var function $
forExp	for	end ; to do then else , }] infixOp) in type var function $
letExp	let	end ; to do then else , }] infixOp) in type var function $

Les règles posant des problèmes de récursivités à gauche sont les suivantes :

— exp / infixExp

— lValue / subscript / fieldExp

Les symboles directeurs sont à faire pour la semaine prochaine.

Fin de la réunion à 15h45.

Next Meeting: Mercredi 24 Octobre à 14h

Compte rendu réunion

Minutes for 24/10/18

Present: Guillaume VANNESSON Nathan BARLOY Quentin MÉRITET Lucas MINÉ

Réunion tenue à Télécom Nancy

Ordre du jour

— Enlever les cycles de récursivité
— Factorisation

L'animateur de la séance est Quentin MÉRITET et le secrétaire : Lucas MINÉ. La séance a débuté à 13h1à.

Enlever les cycles de récursivité

Deux cycles de récursivité ont été identifiés : $exp/infixExp$ et $lValue/subscript/fieldExp$
Pour enlever le cycle de $exp/infixExp$ nous procéderons de la manière suivante :
L'ensemble des règles de de exp prennent la forme :

$$exp \longrightarrow X \quad infixExp$$

La règle $infixExp$ prend la forme :

$$infixExp \longrightarrow \textbf{infixOp} \quad exp \quad infixExp$$

Factorisation

Factorisation de ifThen et ifThenElse

La règle suivante est supprimée :

$$exp \longrightarrow ifThen$$

Le non terminal ifThenElse est supprimé. La règle $ifThen$ est ré-écrite en :

$$ifThen \longrightarrow \textbf{if} \quad exp \quad \textbf{then} \quad exp \quad else$$

Le non terminal else est introduit :

$$else \longrightarrow \textbf{else} \quad exp \quad |$$

Sur le même principe, on réalise une factorisation pour les non terminaux :

— $funDec\, arrCreate$ et $recCreate$,
— $varDec$,
— $lValue$, $callExp$ et $assignment$.

A réfléchir : Gérer la priorité des opérateurs (infixOp) ?
Fin de la réunion à 14h35.

Next Meeting: Mercredi 05 novembre à 14h

Compte rendu réunion

Minutes for 07/11/18

Present: Guillaume VANNESSON Nathan BARLOY Quentin MÉRITET Lucas MINÉ

Réunion tenue à TELECOM Nancy

Ordre du jour

— INFIXOP
— else
— TYID

L'animateur de la séance est Quentin MÉRITET et le secrétaire : Lucas MINÉ. La séance a débuté à 14h.

INFIXOP

Le but est de gérer les priorités des opérateurs infixes. Pour cela de nombreuses propositions ont été testées inspirées du premier TP de PCL sur la grammaire $Expr.g$. Cependant, nous n'avons pas réussis à adapter la solution vue en TP à notre grammaire.
Après discussion avec M. DA SILVA, il s'avère que dérécursiver avant de gérer les priorités des opérateurs n'était pas une bonne idée. La solution est donc de recommancer la gestion de INFIXOP.

else

Il y a un problème d'indécision dans le cas de ifThen imbriqués. La solution trouvée est d'utiliser l'option "greedy" qui permet d'indiquer à l'analyseur que le $else$ se rapporte au dernier ifThen.

TYID

Les tokens TYID et ID sont définis de la même manière, ceci entraîne un problème avec $ANTLR$. Pour résoudre ce problème, on ajoute un non terminal tyId :

$$tyId \longrightarrow ID$$

Fin de la réunion à 15h35.
Next Meeting: Mercredi 14 Novembre à 15h30