Rapport : Projet de compilation 2A 2018

Semestre 1

Ghassen El Mechri - Mohammed Dhouib - Jean-François Diamé - Christopher Alves

21 avril 2019

Table des matières

1 Introduction **3**
1.1 Objectif du projet . . . 3
1.2 Environnement et outils utilisés . . . 3

2 Analyse lexicale et syntaxique **4**
2.1 Langage Tiger . . . 4
2.2 Grammaire du langage . . . 4
2.2.1 Partie déclaration . . . 5
2.2.2 Partie Expression . . . 5
2.3 Arbre abstrait . . . 6
2.3.1 Objectif de l'arbre abstrait . . . 6
2.3.2 associativité a gauche . . . 6
2.4 Jeux d'essais . . . 6

3 Analyse sémantique **10**
3.1 Table des symboles . . . 10
3.1.1 Définition d'un identifier . . . 10
3.1.2 Définition d'un type . . . 10
3.1.3 Notion de sous-table et de parenté entre régions . . . 10
3.2 Diagramme de classes de la TDS . . . 11

4 Gestion de projet **12**
4.1 Objectif et dates limites . . . 12
4.2 Gantt . . . 12
4.3 Compte-rendu de réunion . . . 13
4.4 Évaluation de répartition du travail . . . 21

Chapitre 1

Introduction

1.1 Objectif du projet

Le projet consiste à la réalisation d'un compilateur pour le langage fonctionnel Tiger. A la fin du projet, il est attendu de :

— Pouvoir générer le code assembleur d'un programme correct écrit en Tiger.
— D'afficher les erreurs syntaxiques d'un programme Tiger. S'il y en a, la compilation ne procède pas à l'analyse sémantique.
— D'afficher les erreurs sémantiques d'un programme Tiger, s'il n'y a pas d'erreurs syntaxiques.

Ce rapport concerne la première partie du projet. Celle-ci comprend :

— Elaboration de la grammaire
— Construction de l'AST
— Début de l'anayse sémantique avec un début de TDS

Les attendus spécifiques sont spécifiés dans le sujet. En particulier, la grammaire construite doit être LL(1).

1.2 Environnement et outils utilisés

Nous utiliserons dans ce projet le langage Java. Plus précisément, nous nous servirons du logiciel ANTLR pour nous aider à effectuer l'analyse lexicale et syntaxique. L'analyseur sémantique et la génération de code seront réalisés directement en langage Java.

Pour organiser le code, nous utiliserons l'outil Gradle. Les tests pourront générer des fichiers .dot (en particulier les tests sur la création de l'AST) qui seront générés en fichiers .png par l'utilitaire Graphviz, pour obtenir une représentation graphique de ce que construit le compilateur.

Chapitre 2

Analyse lexicale et syntaxique

2.1 Langage Tiger

Nous ne présenterons pas en détail ici le langage Tiger, se référer plutôt au manuel. Cependant, il vaut mieux rappeler certains points importants.

Tiger est un langage fonctionnel statiquement typé, avec deux types primitifs : les int et les string. On peut créer de nouveaux types, en particulier des types record (cf struct en C) et des types tableau.

Un programme Tiger consiste en une seule expression. Une expression peut notamment être :

- Un type primitif (un entier ou un string)
- Une variable, qui contient une valeur.
- Une séquence d'expression ; la valeur de cette expression correspond à la dernière expression de la séquence.

2.2 Grammaire du langage

Pour écrire la grammaire du langage on s'est basé sur le document **Tiger Specification**, mais la grammaire indiquée dans le document n'était pas LL(1) donc la première étape était de la rendre LL(1).

On va expliquer comment on a réussi a rendre la grammaire LL(1) en modifiant les règles dans l'ordre de niveaux hiérarchique. Tout d'abord on peut séparer un code Tiger en deux grandes parties : les expressions (exp) et les déclarations (dec) avec la hiérarchie suivante pour chacune de ces deux parties :

Hiérarchie Expression :

1. exp
2. letExp | forExp | whileExp | conditionExp (ifThenElse) | infixAssignement
3. infixOr
4. infixAnd
5. InfixAdd
6. infixMult
7. atomExp
8. beginningWithID | INTEGER | STRING | BREAK | NIL | seqExp
9. callExp | recExp

Hiérarchie Déclaration :

1. dec
2. varDec | funDec | tyDec

2.2.1 Partie déclaration

Tout d'abord, on a essayé de rendre la partie déclaration de la grammaire LL(1), les règles qui posaient un problème étaient **funDec** et **varDec** vu qu'elles ont deux règles qui ont le même préfixe. Pour résoudre ce problème on a remplacé la partie qui suit les préfixes (factorisation) par deux nouvelles règles **funDecValue** et **varDecValue**.

2.2.2 Partie Expression

La hiérarchie de reconnaissance d'une expression se découpe en trois partie, qui définissent la priorité :

— Les blocs de région (if, while, for).
— Les opérations infix
— Les expressions atomique (accès à une variable, appel d'une fonction, ...)

Une expression peut avant-tout être un bloc/une région (if, while, for). Si ce n'est pas un bloc, expression peut être une suite d'opération infix (+, -, *, /, ...) où les termes sont des atomes. La suite d'opération peut ne contenir qu'un seul atome.

Règles définissant un bloc

On a choisi de mettre les règles qui crée une région dans le même niveau hiérarchique (cf niveau 2 hiérarchie expression). Pour cela on a commencé par regrouper les deux règles **ifThen** et **ifThenElse** sous la même règle **conditionExp** car elles ont le même préfixe et ça rend la grammaire non-LL(1).
Ensuite on a mis les règles **forExp,** , **whileExp** et **letExp** qui définissent une région chacunes au même niveau que **conditionExp**.

Les opérations infix

Dans le cas où **exp** se transforme en **infixAssignement** On traite les opérations (+, -, /, *, NOT, ..) pour qu'elles respectent la hiérarchie et l'ordre de priorité précisé dans le document **Tiger Specification**

Un atome

Il y a plusieurs atomes possible pour un expression :

— NIL
— BREAK
— Un entier (intLit)
— Une chaîne de caractère stringLit
— Une lvalue/rvalue (x, x[0], x.field1[y.field2[5]], ...)
— Appel d'une fonction
— Instanciation d'un tableau
— Instanciation d'un record

La difficulté ici intervient au niveau des règles lvalue, les règles d'instanciation et appel d'une fonction, puisqu'ils commencent tous par un identifier.
De plus, on ne peut différencier l'instanciation d'un tableau d'un accès à un tableau qu'assez loin de la phrase : arrayofint[5] of 0 contre arrayofint[5].
On crée une règle générique "beginningWithID" qui factorise les règles commençant par un ID à partir d'un atome. Dans cette règle, reconnaître l'appel d'une fonction et l'instanciation d'un record est assez facile. La reconnaissance d'un tableau et d'une lvalue est plus complexe.

Voici la règle beginningWithID de notre grammaire après avoir enlevé la partie de réécriture en AST :

```
beginningWithID:
 callExp -> callExp // appel de fonction
 | recCreate -> recCreate // record
 | ('.' id ) (('[' i5=exp ']') | ('.' i2=id ))* // id.lvalue
 | '[' e1=exp ']' (
                ('of' e2=atomExp) |
                ( ('[' i3=exp ']') | ('.' i2=id) )*
                ) // id[exp] of atomExp ou id[exp]ETC
 | //un simple identifier
 ;
```

2.3 Arbre abstrait

2.3.1 Objectif de l'arbre abstrait

L'arbre abstrait est un arbre qui ne contient pas des nœuds superflus comme l'arbre syntaxique. Il ne contient que des nœuds nécessaire à la génération de code. Il sert à remplir la table des symboles, à faire les contrôles sémantiques puis finalement à générer le code assembleur.

2.3.2 associativité a gauche

On a rendu toutes les operations Tiger (+, -, *, ...) associatif gauche.
Nous avons suivi la hiérarchie présentée dans le document **Tiger Specification**. Ceci est fait par les règles **infixAssignment**, **infixOr**, **infixAnd**, **infixComp**, **infixAddSub**, **infixMulDiv**, **negation**

2.4 Jeux d'essais

Pour s'assurer du bon fonctionnement de la grammaire, on a fait les tests fournis dans le fichier "exemples-tiger.pdf" ainsi que d'autres tests. Pour faire un test il suffit de le mettre dans "GradleProject/src/test/resources" et d'ajouter une méthode au fichier java "TigerLanguageTestASTT.java" "GradleProject/src/test/java/" comme ci-dessous

```
@Test public void testX() throws Exception {
    TestProgram("/nom du fichier test");
}


                    //methode pour faire un test
```

TestProgram lit le fichier .tiger, construit l'arbre abstrait grâce aux classes Lexer et Parser créées par ANTL, la convertit en notation 'dot', utilise la commande 'dot' pour enregistrer cette notation sous forme d'image.
Voici quelques tests et les arbres abstraits correspondants :

```
let var i := a<=2 in if i < 3 then if y = 4 then x else y end
```

```
//test 1 : if then if then else
```

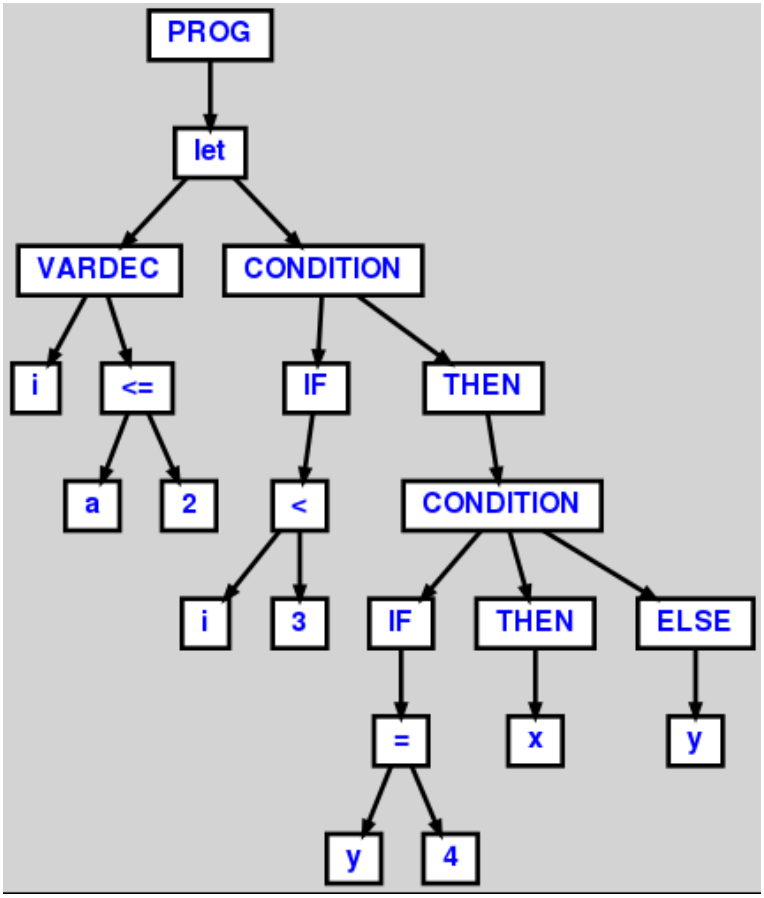

FIGURE 2.1 – Arbre abstrait du test 1

```
let

 var total := 0
 var i := 0
 var n := 10
 var testvar : int := 0

in

 testvar := i <= n ;
 while testvar do
 (
   total := total + i * i ;
   if i = 5 then
      print (5) ;
   testvar := i <= n
 ) ;

 print ( total )
```

```
end
                                    //test 2 : somme des 5 premiers carres
```

D'autres tests peuvent être observés dans le dossier testAST situé sur le dépôt GIT. Ils n'ont pas été inclus dans le rapport dû notamment à la grande taille des AST construit.
Notamment, les tests avaient pu révélé, entre autres :

- Une erreur dans la grammaire, où la phrase "if then x = 3 then i :=2" était reconnu avec la priorité "(if then x = 3 then i) := (2)" (donc la phrase était considéré comme une affectation et non une condition). Ce problème a été rectifié.
- La phrase " x := 1 + if 1 then 1 else 1 " ne fonctionne pas malgré que, selon la spécification, elle devrait, car le bloc condition n'est pas un atomExp. Il faut donc mettre des parenthèses autour du bloc if. Nous n'avons pas trouvé de méthode simple pour résoudre ce problème sans rendre la grammaire non-LL(1), introduire d'autres problèmes ou réécrire toute la grammaire.

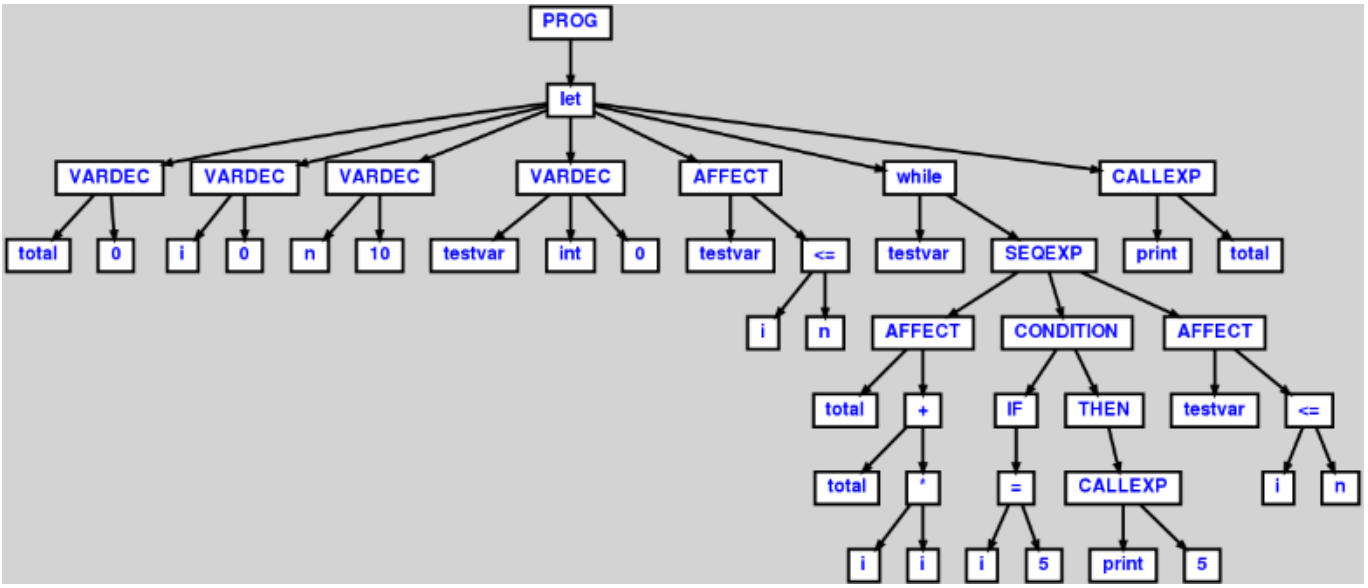

FIGURE 2.2 – Arbre abstrait du test 2

Les tests de code faux :

```
let
var i := 0
in
print(i);
print(i);
end
```

```
//test 3 : La derniere instruction du bloc in se termine par
   un point vergule
```

```
let
var  i := 1
var l := dataArray[3] of nil
in
l[0]=l[1];
l[1]=l[2];
l[*]=l[0];
l[0]=1
end
```

```
//test 4 : L'instruction l[*] n'est pas definie
```

Ces tests provoquent à ANTLR de renvoyer un message d'erreur.
Par exemple, le test 3 affiche :

Chapitre 3

Analyse sémantique

3.1 Table des symboles

Pour remplir la table des symboles on a créé deux interfaces :
— Interface Type : permet de créer un nouveau type
— Interface Identifier : permet de créer un symbole
Une table des symboles contient des Identifier. Un Identifier peut pointer vers un Type.

3.1.1 Définition d'un identifier

Comme on peut le voir dans le diagramme de classes ci-dessous, la table des symboles contient trois types d'identifier (de symbole) qui sont :
— VAR (issus de déclarations de variables)
— FUNCTION (issus de déclarations de fonction)
— TYPE (issus de création de types)
Donc un **Identifier** peut être : une variable **VAR** et dans ce cas on va créer un nouveau objet de la classe **identifierVar**, une fonction **FUNCTION** et donc on va créer un nouveau objet de la classe **identifierFunction** ou un type **TYPE** ce qui va nous mener à créer un nouveau objet de la classe **identifierType** .

3.1.2 Définition d'un type

Il y a 4 types différents :
— INT (entier, type primitif)
— STRING (chaîne de caractère, type primitif)
— RECORD (enregistrement/structure en C, type créé par l'utilisateur)
— ARRAY (tableau, type créé par l'utilisateur)
Un identifier IdentifierType contient un Type (celui qu'il définit). Un IdentifierVar contient un Type (le type de la variable).

3.1.3 Notion de sous-table et de parenté entre régions

La table des symboles est divisée en plusieurs sous-tables où chaque table correspond à une région du programme. Il faut pouvoir, pour chaque région :
— Connaître le nombre d'imbrication (à quel niveau on est)
— Connaître le numéro de la région
— Récupérer et ajouter un nouveau **Identifier** à la région
— Récupérer la région englobante

3.2 Diagramme de classes de la TDS

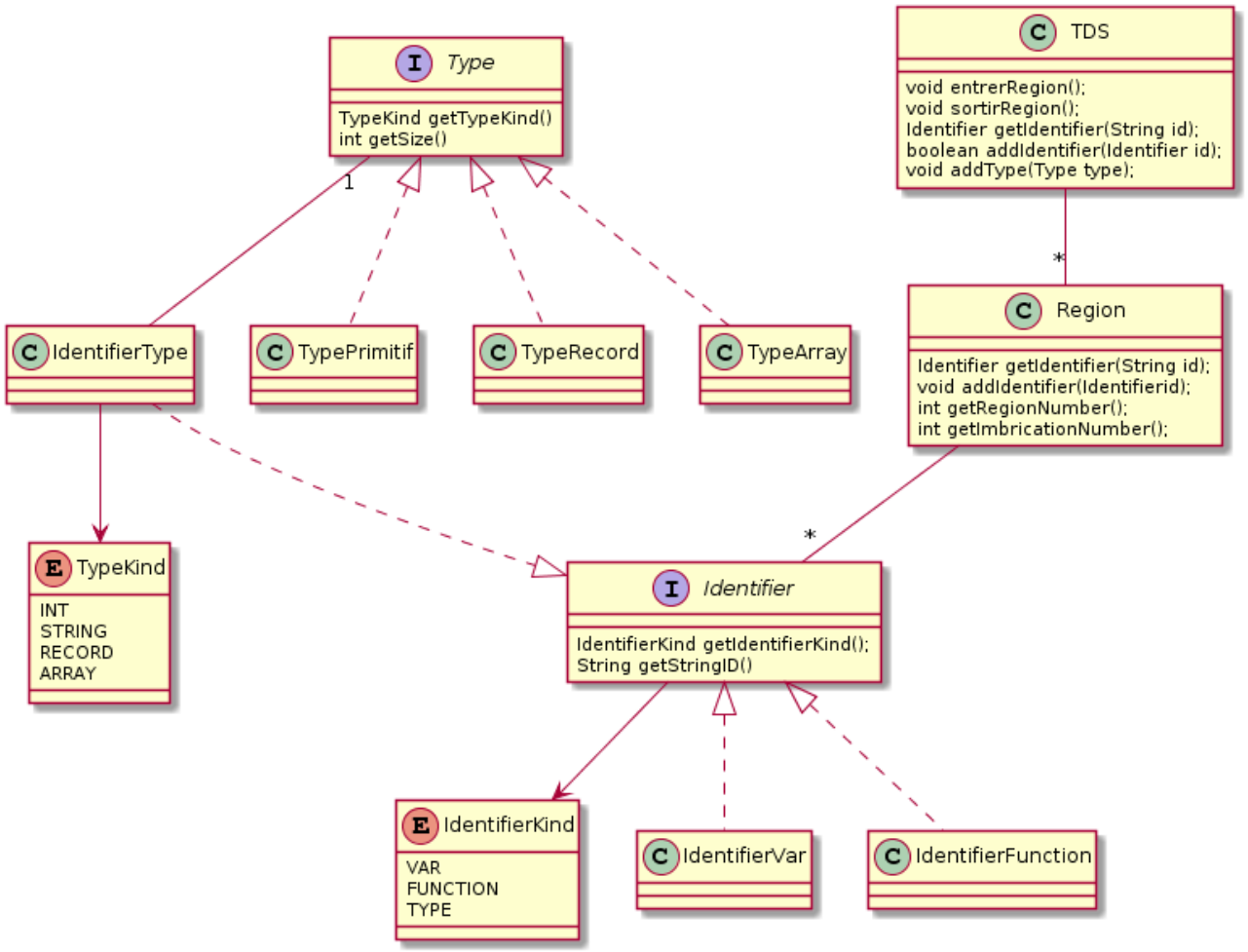

FIGURE 3.1 – Diagramme de classes TDS

Chapitre 4

Gestion de projet

4.1 Objectif et dates limites

L'objectif principal à réaliser est la construction et l'affichage d'un AST en fonction d'un code Tiger en entrée. L'objectif secondaire est le début de l'implémentation de la table des symboles.
Nous pouvons donc séparer cette première partie du projet en deux étapes :

— Ecriture de la grammaire avec les règles de réecriture.
— Utilisation de cette grammaire dans un programme java complet qui permet d'afficher l'AST (sous forme d'image et/ou de texte), avec un début de construction de TDS.

Le travail sur le projet commence le 10 octobre et le rendu est pour le 11 décembre (soutenance le jour d'après).
Nous fixons la fin de la première étape pour mi-novembre, même s'il est toujours possible d'effectuer des modifications mineures après.

4.2 Gantt

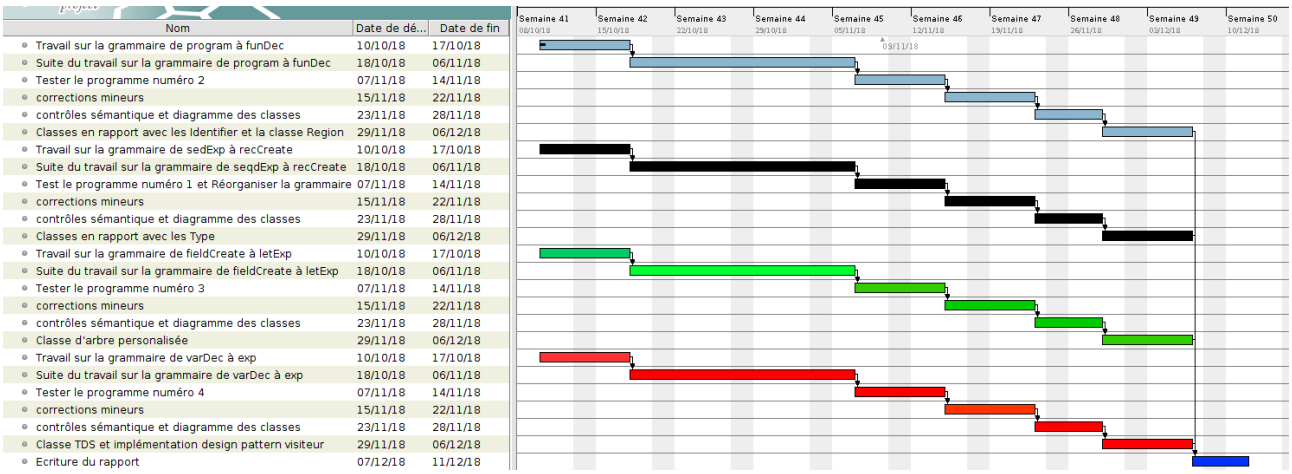

FIGURE 4.1 – GANTT

— Mohammed Hedi Dhouib : bleu
— Jean François Diamé : noir

— Christopher Alves : rouge
— Ghassen El Mechri : vert
— Tâche commune : bleu foncé

4.3 Compte-rendu de réunion

Réunion 1

Minutes for Mercredi, 10 octobre 2018

Present: Jean François Diamé, Ghassen Mechri, Mohammed Hedi Dhouib, Christopher Alves

Ordre du jour

1. Démarrer le projet
2. Discuter du langage TIGER (mieux vaut comprendre un langage avant d'écrire son compilateur)
3. Commencer à répartir les tâches

Information échangées

1. Reprise des modalités du projet dans sa globalité
2. Points importants du langage TIGER et en quoi il diffère des langages déjà connus (C, java)
3. Mise en évidence de ce qui doit être accompli durant la première moitié du projet.
4. En particulier, dans l'immédiat, il faut écrire la grammaire dans la syntaxe de ANTLR avec les réécritures. Une fois cela fait, on peut espérer que ANTLR détermine si la grammaire est LL(1) ou pas et on pourra lever les ambiguïtés soit en modifiant la grammaire soit en y ajoutant des règles supplémentaires (exemple du if if then else).

Décisions

1. Attente du premier TP de PCL, centré sur le logiciel ANTLR, pour mieux comprendre comment cet outil doit être utilisé dans le cadre du projet avant de faire une réel répartition des tâches.
2. Prévoir l'écriture de programmes TIGER de test.
3. Commencer à travailler sur le grammaire : écrire la grammaire dans la syntaxe de ANTLR, y ajouter les réécritures pour l'AST. On 'découpe' la grammaire en 4 et chacun fait une partie.

Next Meeting: Mercredi, 17 octobre. Heure à clarifier.

Actions à suivre / Todo-list

Description	Responsable(s)	Délai	Avancement
Travail sur la grammaire de program à funDec	Mohammed	17/10/2018 au plus tard	0%
Travail sur la grammaire de varDec à exp	Christopher	17/10/2018 au plus tard	0%
Travail sur la grammaire de sedExp à recCreate	Jean François	17/10/2018 au plus tard	0%
Travail sur la grammaire de fieldCreate à letExp	Ghassen	17/10/2018 au plus tard	0%

Réunion 2

Minutes for Mercredi, 17 octobre 2018

Present: Jean François Diamé, Ghassen Mechri, Mohammed Hedi Dhouib, Christopher Alves

Ordre du jour

1. Expliquer ce que chacun a fait
2. Discuter des difficultés et problèmes rencontrés

Information échangées

1. Quelques points difficiles ou importants rencontrés lors de la première tentative d'écriture :
   (a) L'utilisation de l'option greedy pour palier le problème au niveau des règles de condition (exemple problématique : 'if if then else', avec quel if va le dernier else ?)
   (b) La règle 'exp infixOp exp' dans la grammaire du manuel est en fait complexe car elle doit contenir les priorités des opérations.
   (c) L'élimination des récursivité à gauche
   (d) La factorisation de règles pour rendre la grammaire LL(1)
2. Proposition d'une méthode alternative à antlrworks pour afficher les AST (graphviz)

Décisions

1. Continuer le travail sur la grammaire en partant de ce qui a été discuté pendant la réunion.

Next Meeting: Mercredi, 24 octobre. Heure à clarifier.

Actions à suivre / Todo-list

Description	Responsable(s)	Délai	Avancement
Suite du travail sur la grammaire de program à funDec	Mohammed	24/10/2018 au plus tard	50%
Suite du travail sur la grammaire de varDec à exp	Christopher	24/10/2018 au plus tard	50%
Suite du travail sur la grammaire de sedExp à recCreate	Jean François	24/10/2018 au plus tard	50%
Suite du travail sur la grammaire de fieldCreate à letExp	Ghassen	24/10/2018 au plus tard	50%

Réunion 3

Minutes for Mercredi, 24 octobre 2018

Present: Jean François Diamé, Ghassen Mechri, Mohammed Hedi Dhouib, Christopher Alves

Ordre du jour

1. Expliquer ce que chacun a fait durant la semaine
2. Discuter des difficultés et problèmes rencontrés

Information échangées

1. Relecture ensemble de ce qui a été effectué jusqu'à présent sur la grammaire avec explication dans certains cas.
2. Petit conflit sur la règle concernant l'affectation : initialement confiée à Ghassen, l'affectation est en fait un opérateur tombant dans la règle relative à infixOp sous la charge de Jean François.
3. Explication du terminal STRING qui gère les caractères spéciaux.
4. Discussion rapide sur la table des symboles

Décisions

1. Continuer le travail sur la grammaire en partant de ce qui a été discuté pendant la réunion.
2. Finir une première version de la grammaire pour la fin de la semaine.
3. Commencer à réfléchir à la table des symboles, ou du moins relire les parties du cours qui la concerne.

Next Meeting: Date à clarifier (durant la semaine des vacances entre le 29 octobre et 2 novembre)

Actions à suivre / Todo-list

Description	Responsable(s)	Délai	Avancement
Suite du travail sur la grammaire de program à funDec	Mohammed	24/10/2018 au plus tard	80%
Suite du travail sur la grammaire de varDec à exp	Christopher	24/10/2018 au plus tard	80%
Suite du travail sur la grammaire de sedExp à recCreate	Jean François	24/10/2018 au plus tard	80%
Suite du travail sur la grammaire de fieldCreate à letExp	Ghassen	24/10/2018 au plus tard	80%

Réunion 4

Minutes for Mercredi, 7 novembre 2018

Present: Jean François Diamé, Ghassen Mechri, Mohammed Hedi Dhouib, Christopher Alves

Ordre du jour

1. Rappeler ce qui a été fait jusqu'à maintenant pour se remettre des vacances
2. Voir les dernières choses à faire pour finir la grammaire
3. Planifier les tests à réaliser

Information échangées

1. Rappel de ce qui a été fait durant les 2 heures de PCL avant les vacances.
2. Discussion sur la règle concernant les comparaisons (<=,==,>=, ...) qui était incomplète.
3. Discussion sur les quatre exemples de code Tiger qui nous ont été envoyés par e-mail.

Décisions

1. Reorganisation du fichier TigerLanguage.g pour le rendre plus lisible.
2. Finir la grammaire pour la semaine suivante.
3. Tester les quatre programmes Tiger pour la semaine suivante.

Next Meeting: 14 novembre.

Actions à suivre / Todo-list

Description	Responsable(s)	Délai	Avancement
Tester le programme numéro 2	Mohammed	14/11/2018 au plus tard	0%
Tester le programme numéro 4	Christopher	14/11/2018 au plus tard	0%
Test le programme numéro 1	Jean François	14/11/2018 au plus tard	0%
Tester le programme numéro 3	Ghassen	14/11/2018 au plus tard	0%
Réorganiser la grammaire	Jean François	14/11/2018	0%

Réunion 5

Minutes for Mercredi, 14 novembre 2018

Present: Jean François Diamé, Ghassen Mechri, Mohammed Hedi Dhouib, Christopher Alves

Ordre du jour

1. Se concentrer sur les problèmes perçues lors des tests
2. Regarder le cas problématique de infixAssignment
3. Démarrer la partie analyseur sémantique

Information échangées

1. Test 1, 3 et 4 ont bien fonctionné.
2. Le test 2 a eu un problème au niveau d'une instruction du type 'if condition then x := 3 else ...'.
   Cause : l'affectation est une opération et donc x := 3 n'est pas une expression atomique (x et 3 sont atomiques ; l'ajout d'une opération entre les deux donne une expression non atomique). Or, pour régler les problèmes infix (qui nécessiteraient sinon un greedy), nous avons rendu les instructions dans les blocs atomiques. Ainsi, 'if condition then x else ...' et 'if condition then (x := 3) else ...' fonctionnent mais pas ce qu'il y a dans le test.
3. Revue de ce que l'on sait sur la table des symboles et évocation de la table des types (qui ne sera pas forcément utilisée plus tard)
4. Elaboration rapide d'un diagramme des classes concernant la partie table des symboles et table des types.

Décisions

1. Mettre de côté le problème qui ne permet pas de mettre une simple affectation après un if mais qui impose la mise de parenthèses ('if condition then x := 2 else' contre 'if condition then (x := 2) else'. Continuer à y réfléchir mais se concentrer sur la partie sémantique pour le moment.
2. Lister les contrôles sémantique que nous allons réaliser. La grammaire a été coupé en 4 pour permettre à chacun de se concentrer sur sa partie pour la recherche de contrôle sémantique à réaliser.
3. Diagramme des classes 'propre' à réaliser pour la partie TDS

Autre

L'analyseur lexicale et syntaxique est terminée. Nous commençons à travailler l'analyseur sémantique.

Next Meeting: 21 novembre.

Réunion 6

Minutes for Mercredi, 28 novembre 2018

Present: Jean François Diamé, Ghassen Mechri, Mohammed Hedi Dhouib, Christopher Alves

Ordre du jour

1. Diagramme de classe pour la TDS
2. Séparer les tâches pour l'élaboration de la TDS
3. Discussion sur ce qu'il y a à faire pour élaborer la TDS (d'un point de vue implémentation)
4. Rappel de l'architecture Gradle

Information échangées

1. Le problème mis en évidence lors de la réunion précédente a été réglé
2. Discussion sur les structures à utiliser pour mettre en place la TDS
3. La création de la TDS peut se faire en plusieurs étapes :
   (a) Premier parcours pour découvrir les types définis (en réalité plusieurs car les types ont une porté
   (b) Deuxième parcours pour découvrir les fonctions définies.
   (c) Troisième parcours pour découvrir les variables définies. Le type de la variable devra être calculé. On le compare (si possible) au type spécifié (ex : var : int := 3). Il y a plusieurs erreurs sémantiques possibles.
4. Intérêt d'avoir une classe d'arbre personalisée (pour mettre en place un design pattern, pour enregistrer le numéro de ligne, ...).
5. Intérêt de l'utilisation de Gradle (ou tout autre outil similaire) et utilisation dans le cadre du projet.

Décisions

1. Objectif : réaliser les deux premiers parcours
2. Répartition des tâches pour la création des classes

Next Meeting: 5 décembre.

Actions à suivre / Todo-list

Description	Responsable(s)	Délai	Avancement
Classe TDS et implémentation design pattern visiteur	Christopher	5/12/2018	0%
Classes en rapport avec les Identifier et la classe Region	Mohammed	5/12/2018	0%
Classes en rapport avec les Type	Jean-François	5/12/2018	0%
Classe d'arbre personalisée	Ghassen	5/12/2018	0%

Réunion 7 (réunion technique)

Minutes for Mercredi, 5 décembre 2018

Present: Jean François Diamé, Ghassen Mechri, Mohammed Hedi Dhouib, Christopher Alves

Ordre du jour

1. Travail en groupe sur la TDS et autres détails avant la soutenance.
2. Commencer l'écriture du rapport.

Travail effectué

1. Travail sur les classes du package Semantique, conformément à la To-Do list réalisée précédemment.
2. Réalisation du Main pour construire/afficher/sauvegarder l'AST en ligne de commande.
3. Travail sur l'implémentation du pattern visiteur (ou du moins d'une variante, ce qui correspond d'un point de vue pratique aux conditions lors du parcours pour détecter le type de nœud sur lequel nous sommes actuellement).
4. Écriture d'un début de rapport.

Structuration du rapport

Le rapport sera structurée comme suit (susceptible à changement) :

1. Introduction
    (a) Objectif du projet
    (b) Environnement et outils utilisés
2. Analyse lexicale et syntaxique
    (a) Langage Tiger
    (b) Grammaire du langage
    (c) Arbre abstrait
    (d) Jeux d'essais
3. Analyse sémantique
    (a) Table des symboles
    (b) Diagramme de classes provisoire pour la TDS
4. Gestion de projet
    (a) Gantt
    (b) Compte-rendu de réunion
    (c) Évaluation de répartition du travail

Répartition des tâches pour l'écriture du rapport

1. Mohammed : 3, 2.b
2. Christopher : 1, 2.a
3. Jean-François : 4.b, 4.c
4. Ghassen : 2.c, 2.d, 4.a

4.4 Évaluation de répartition du travail

L'évaluation de la répartion du travail se base principalement sur les critères suivants :
— Le respect des delais prévus pour chaque tâche.
— Réalisation des objectifs.
— Efficacité.
— Qualités relationnelles.

Respect des délais	**Réalisation des objectifs**	**Efficacité**	**Qualités relationnelles**
-En moyenne les délais prévus sont respectés par chaque membre du groupe.	-En moyenne les objectifs fixés au cours de chaque réunion sont atteints à temps.	-Bonne organisation du travail pour concevoir et conduire le projet. -Une bonne faculté d'analyse et de synthèse de la part de chaque membre du groupe.	-Très bonne entente au sein du groupe. -Prise en compte des avis de chacun pour aboutir à un consensus.

FIGURE 4.2 – Evaluation de la répartition du travail