TELECOM Nancy deuxième année (2018-2019)

Rapport du projet de compilation du langage *Tiger*

Auteurs :
Alexis DIEU,
David FORLEN,
Philippe GRAFF,
Tristan LE GODAIS

Enseignants encadrants :
Suzanne COLLIN,
Sébastien DA SILVA,
Pierre MONNIN

La version originale du programme est disponible à l'adresse :
gitlab.telecomnancy.univ-lorraine.fr/Philippe.Graff/graff24u

Table des matières

1 Introduction **2**

2 Grammaire du langage **3**

2.1 Utilisation du mot-clé *greedy* 3

2.2 Priorisation des conditions 3

2.3 Priorisation des opérations arithmétiques 4

2.4 Priorisation des affectations 5

2.5 Arbres n-aires pour les opérateurs binaires associatifs 6

2.6 Factorisation de ID et $TYID$ 6

3 Structure de l'arbre abstrait **7**

3.1 Règle $letExp$ 7

3.2 Règle $funDec$ 7

3.3 Règle $callExp$ 8

3.4 Règle $ifThenElse$ 8

4 Structure de la table des symboles **10**

5 Jeux d'essais **11**

5.1 Dossier de tests 11

5.2 Tests automatisés 11

6 Gestion de projet **13**

6.1 Gantt 13

6.2 Temps de travail 13

6.3 Colorateur syntaxique pour *Atom* 13

6.4 Comptes-rendus de réunion 14

1 Introduction

Dans ce rapport, nous allons présenter la première étape de la réalisation d'un compilateur pour le langage *Tiger*, projet encadré par Mme Collin, M. Da Silva et M. Monnin. Après avoir mis en lumière quelques subtilités liées à la grammaire du langage et les solutions que nous avons mises en place pour y répondre, nous montrerons la structure de notre arbre abstrait sous la forme d'un diagramme de classes. Enfin, nous évoquerons nos jeux d'essais permettant de tester notre grammaire que nous avons réalisée avec le logiciel *Antlr*.

2 Grammaire du langage

2.1 Utilisation du mot-clé *greedy*

La grammaire fournie par la spécification de Tiger contient des règles relatives aux conditions. Dans le cas d'une condition `if ... then ... else ...`, chaque `else` doit s'appliquer au dernier `if` rencontré. Dans notre grammaire, le non-terminal $ifExp$ est justement chargé de reconnaître les conditions exprimées dans le code.

$$\begin{aligned} ifExp &\rightarrow \texttt{if}\ exp\ \texttt{then}\ exp\ rule \\ rule &\rightarrow \texttt{else}\ exp \\ rule &\rightarrow \end{aligned}$$

Lorsque *Antlr* tente de générer un *parser*, cette règle peut poser problème. En effet, supposons que l'on se donne le programme `if a then if b then c else d`. Plusieurs réécritures différentes pourraient lui correspondre. En effet on peut le lire soit `if a then (if b then c else d)`, soit `if a then (if b then c) else d`. Il y a donc ambiguïté.

L'option *greedy* permet de reconnaître un terminal dès qu'on le peut. Nous entendons par là que dans le cas où le prochain *token* à analyser est `else`, *rule* se réécrira selon la première règle :

$$rule \rightarrow \texttt{else}\ exp$$

Cette manipulation permet de rattacher chaque *exp* au dernier `if` rencontré. Le programme sera donc lu ainsi : `if a then (if b then c else d)`.

2.2 Priorisation des conditions

Les conditions dans *Tiger* comportent un autre problème lié à leur possible valeur retournée. En effet, la section 2.8 de la spécification *Tiger* indique que lors d'un test `if a then b else c`, la valeur `b` est renvoyée si `a` est non nul et la valeur `c` est renvoyée sinon (ce qui est analogue à un opérateur ternaire dans d'autres languages de programmations).

Cette particularité de *Tiger* implique quelques possibles ambiguïtés. En effet, soit la ligne de code *Tiger* suivante.

```
a := if b then c else d + e
```

Sans parenthèses, cette ligne est ambiguë. Doit-on la lire `a := ((if b then c else d) + e)` ou `a := (if b then c else (d + e))` ? Dans le premier

cas, `e` est ajouté au résultat du `if` quelle que soit l'évaluation de `b`, et dans le deuxième cas, il n'est ajouté que si `b` est évalué à zéro.

Nous avons fait le choix de la première possibilité, c'est-à-dire `a := ((if b then c else d) + e)`. Si on souhaite forcer la première possibilité, l'utilisateur écrivant le code source devra utiliser des parenthèses autour du `else`.

2.3 Priorisation des opérations arithmétiques

Pour tenir compte des priorités des opérateurs, la règle $infixExp$ a été scindée en autant de règles qu'il y a de niveaux de priorité. Ces nouvelles règles se présentent ainsi :

$$
\begin{array}{rcl}
orExp & \rightarrow & andExp\ (\ \texttt{|}\ andExp\ )\ * \\
andExp & \rightarrow & compExp\ (\ \texttt{\&}\ compExp\ )\ * \\
compExp & \rightarrow & addExp\ (\ compTerminal\ addExp\ )\ ? \\
addExp & \rightarrow & mulExp\ (\ addTerminal\ mulExp\ )\ * \\
mulExp & \rightarrow & unaryExp\ (\ mulTerminal\ unaryExp\ )\ *
\end{array}
$$

Avec :

- $compTerminal$ les terminaux de comparaison : `<=`, `<`, `=`, `>`, `>=`, `<>`.
- $addTerminal$ les opérateurs d'addition et soustraction : `+` et `-`.
- $mulTerminal$ les opérateurs de multiplication et division : `*` et `/`.

Les règles les plus prioritaires sont donc réécrites en dernier lors de l'analyse descendante, elle seront donc évaluées en premier.

Par exemple, pour la gestion de la priorité de la multiplication et de l'addition (figure 1), on constate bien que l'arbre de l'addition a pour fils droit l'arbre de la multiplication.

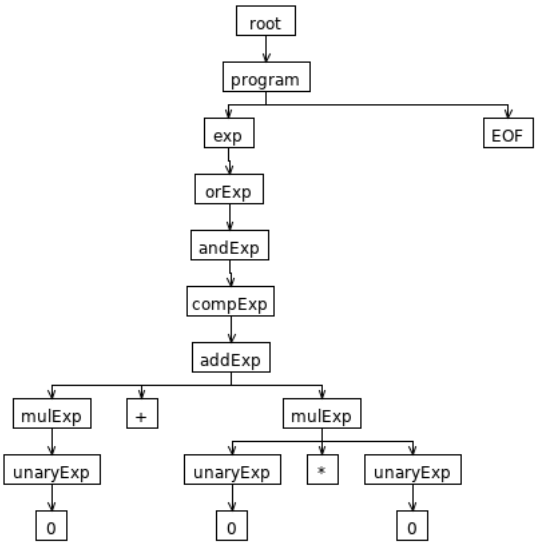

Figure 1: Arbre syntaxique du programme `0 + 0 * 0`

2.4 Priorisation des affectations

Les affectations sont un cas particulier parmi les opérateurs infixes puisque contrairement aux opérateurs arithmétiques, elles n'acceptent une expression qu'à leur droite. À la gauche de l'opérateur se trouve en effet la cible de l'affectation, représentée par le non-terminal $lValue$ dans la spécification *Tiger*, et non une expression.

Alors que les règles reconnaissant les opérations arithmétiques ont été faites de telle sorte que l'opérateur et l'opérande droite soient optionnels, cette stratégie n'est pas applicable à l'opérateur d'affectation `:=` à cause de la particularité éconcée ci-dessus. En effet, lors de l'analyse syntaxique, ce n'est qu'au moment où on rencontre le symbole `:=` que l'on sait que la partie gauche correspond à une cible $lValue$ et non pas à une expression. Or dans le cadre d'une analyse $LL(1)$, ce genre de comportement est impossible.

Pour résoudre ce problème tout en veillant à conserver la priorité de l'opérateur d'affectation (qui a la plus faible précédence), nous avons décidé de calquer la syntaxe d'une affectation sur celle des autres opérations infixes en optant pour une expression comme opérande gauche au lieu d'une cible $lValue$:

$$exp \quad \rightarrow \quad orExp \ (\ \texttt{:=}\ orExp\ )\ ?$$

Ce faisant, notre analyseur syntaxique accepte donc des programmes initialement invalides comme celui ci-dessous, bien qu'ils n'aient aucun sens. C'est pourquoi, conceptuellement, nous avons fait de ce genre d'erreur syntaxique une erreur sémantique qu'il faudra vérifier lors des contrôles sémantiques :

```
a | b := c
```

2.5 Arbres n-aires pour les opérateurs binaires associatifs

Lorsque les opérateurs arithmétiques sont associatifs, nous avons décidé que l'AST correspondant serait, afin de limiter la profondeur de l'arbre, un noeud n-aire. Cela est le cas pour de nombreux opérateurs : `|`, `&`, `+` et `*`. La figure 2 montre l'arbre abstrait d'une addition entre plus de deux termes.

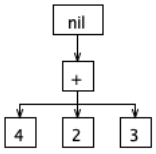

Figure 2: AST d'une addition `4 + 2 + 3`

En revanche, lorsque l'opération ainsi faite n'est pas associative, un arbre n-aire n'aurait pas de sens. Nous utilisons donc un arbre binaire, mais contenant plus de couches, comme l'illustre la figure 3.

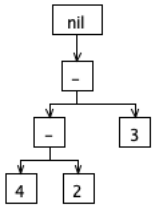

Figure 3: AST d'une soustraction `4 - 2 - 3`

2.6 Factorisation de ID et $TYID$

La grammaire donnée par la spécification *Tiger* distingue les noms de variables et de fonctions de ceux de types en ayant recours à deux types d'identifiants : ID et $TYID$ respectivement. Ceux-ci peuvent parfois être rencontrés au même endroit dans cette grammaire, et en particulier au niveau de *exp* (par exemple au début des réécritures en $lValue$ et $arrayCreate$). Cependant, ces deux terminaux possèdent exactement la même syntaxe, les rendant indiscernables l'un de l'autre, ce qui provoquerait donc des conflits si on essayait de construire tel quel un analyseur syntaxique.

Le choix a donc été fait de les fusionner en un unique terminal ID et par conséquent de factoriser toutes les règles débutant par un identifiant. L'AST généré est toutefois fait de telle sorte qu'on puisse déduire le sens à prêter à un identifiant de l'étiquette de la racine du sous-arbre englobant. Ce sens sera ainsi restitué à un identifiant lors de la génération de la table des symboles par le typage de la classe utilisée pour le représenter.

3 Structure de l'arbre abstrait

Lors de la réalisation de l'arbre abstrait, nous avons réalisé des *tokens imaginaires* permettant de nommer les noeuds de l'arbre lorsque ceux-ci ne correspondent pas déjà à des mot-clés *Tiger*.

3.1 Règle $letExp$

La règle $letExp$ est réécrite en un arbre abstrait *let* qui possède deux fils. Ces fils sont dans l'ordre :

- L'en-tête DEC contenant les déclarations de variables, types et fonctions à prendre en compte ;
- Le corps SEQ contenant la séquence d'expressions dans laquelle les déclarations seront valables.

La figure 4 est l'AST d'un programme déclarant et initialisant plusieurs variables, et faisant plusieurs opérations par la suite.

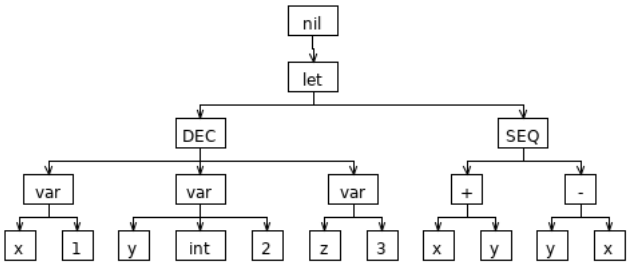

Figure 4: AST d'un programme avec la règle $letExp$

3.2 Règle $funDec$

Concernant le non-terminal $funDec$, nous construisons un arbre abstrait $function$ ayant entre trois et quatre fils. Dans l'ordre :

- Le nom de la fonction que l'on déclare ;
- Le sous-arbre n-aire contenant tous les paramètres de la fonction ;
- Éventuellement le type de retour de la fonction si précisé ;
- Le corps de la fonction.

Le sous-arbre $CALLTYPE$ qui se trouve être le deuxième fils de l'arbre $function$ contient un nombre pair de fils. Pour chaque paire de fils $(2k, 2k+1)$, le premier élément du couple k sert à spécifier l'identifiant du paramètre, tandis que le deuxième représente son type.

La figure 5 est l'AST d'un programme déclarant une fonction sans effet.

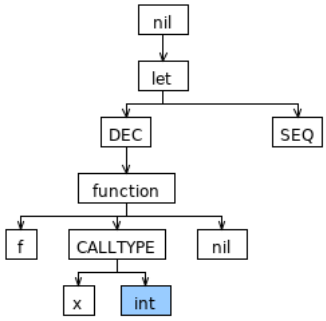

Figure 5: AST d'un programme avec la règle $funDec$

3.3 Règle $callExp$

La règle $callExp$ est réécrite en un arbre $CALL$ qui possède comme au moins deux fils. Dans l'ordre :

- Le nom de la fonction ;
- Les arguments de la fonction un par un.

La figure 6 est l'AST d'un programme appelant une fonction.

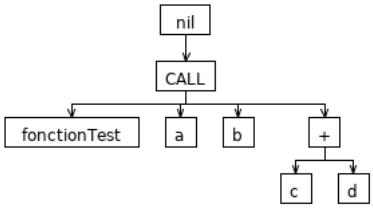

Figure 6: AST du programme `fonctionTest(a, b, c + d)`

3.4 Règle $ifThenElse$

Les règles $ifThen$ et $ifThenElse$ sont réécrites en un arbre if qui possède entre deux et trois fils. Dans l'ordre :

- La condition qui est un *exp* contenant ce qui est à tester pour déterminer si c'est ce qui suit le `then` ou le `else` qui sera exécuté ;
- Le premier embranchement qui est un *unaryExp* contenant ce qui suit le `then` ;
- Éventuellement le second embranchement qui est un *unaryExp* contenant ce qui suit le `else`.

La figure 7 est l'AST d'un programme utilisant la règle $ifThenElse$.

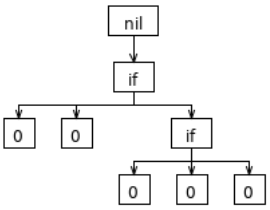

Figure 7: AST du programme `if 0 then 0 else if 0 then 0 else 0`

4 Structure de la table des symboles

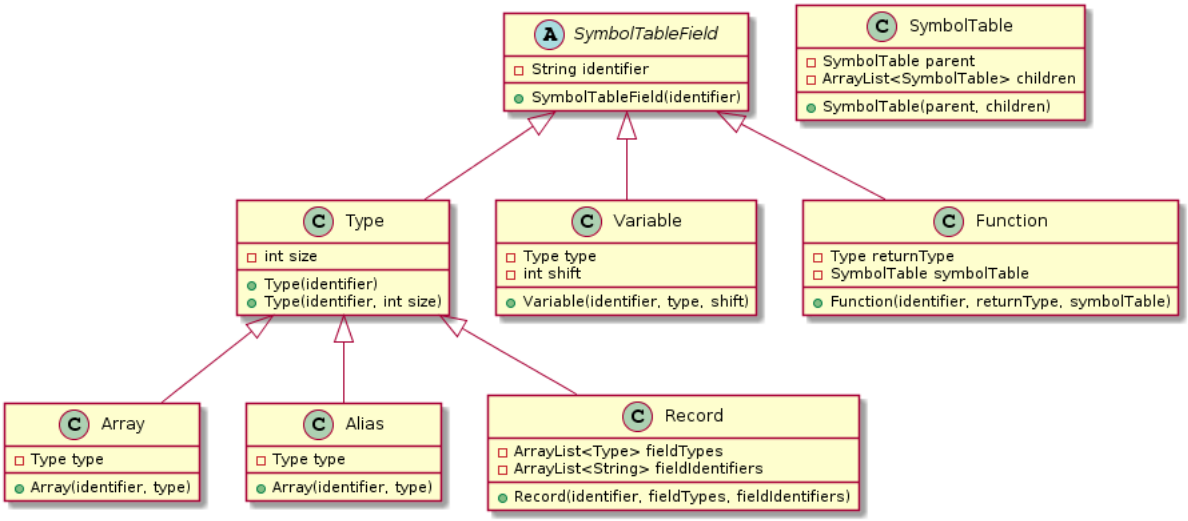

Figure 8: Diagramme de classes de la table des symboles

Nous avons entamé, dans le langage *Java*, la structure de la table des symboles. Le diagramme de classes de cette structure est donnée en figure 8. Afin de préserver la lisibilité, nous n'avons pas écrit les mutateurs et accesseurs et n'avons pas indiqué les types des variables dans les méthodes.

Les deux classes importantes sont `SymbolTable` et `SymbolTableField`. Bien que cela ne soit pas indiqué sur le diagramme, la classe `SymbolTable` hérite de `ArrayList<SymbolTableField>`, car il sera nécessaire de pouvoir facilement ajouter ou supprimer des des lignes dans cette table. Les instances de cette classe possèdent en outre deux attributs : un attribut `parent` qui est une autre table des symboles, et un attribut `children` qui est une liste de tables des symboles. Cela est dû au caractère intrinsèquement arborescent des tables des symboles qui correspondent à des blocs de code pouvant être imbriqués les uns dans les autres. Par exemple, si une fonction est déclarée dans une autre, sa table des symboles sera fille de l'autre.

La classe `SymbolTableField` est une ligne dans une table des symboles. Elle est abstraite car une ligne doit nécessairement correspondre soit à un type, soit à une fonction, soit à une variable.

Une autre subtilité concerne la classe `Function`. En effet, celle-ci comporte un attribut `symbolTable`. Cette table des symboles contiendra (et ne contiendra que) les arguments de cette fonction.

5 Jeux d'essais

Très tôt, nous avons commencé à nous constituer une base de tests. Nous avons par la suite mis au point un processus automatisé parcourant l'ensemble de nos tests. Il nous retourne le nombre d'essais acceptés par notre grammaire.

Après chaque modification apportée à notre grammaire, nous avions donc le moyen de conjecturer si nos changements allaient la faire régresser ou non. Gardons cependant à l'esprit que bien que nos tests soient nombreux, ils sont loin d'être exhaustifs.

Explicitons dès à présent l'organisation de notre dossier de tests, puis abordons ensuite le processus automatisé de contrôle.

5.1 Dossier de tests

Le dossier de tests est composé de trois sous-dossiers :

- Le dossier `/tests/fail` :
  Il contient tous les tests syntaxiquement faux que nous avons élaborés. Ces tests permettent de voir quelques instructions incorrectes étant reconnues comme telles par notre analyseur. Les erreurs figurant dans ces fichiers de tests ne donneront donc pas lieu à des tests sémantiques.
- Le dossier `/tests/pass` :
  Contient tous les tests syntaxiquement corrects que nous avons élaborés. Notons bien qu'ils ne sont pas forcément corrects sémantiquement. Nous avons cherché à y tester chaque règle de notre grammaire au travers de multiples fichiers nommés intelligiblement.
- Le dossier `/tests/prgm` :
  Contient l'ensemble des programmes complexes que nous avons pu rassembler. On y trouve les exemples de programmes *Tiger* communiqués par nos professeurs encadrants, ainsi que d'autres programmes que nous avons codés.

5.2 Tests automatisés

L'automatisation du processus de tests se fait par le biais d'un script *shell* et de commandes *Make* qui permettent de mettre à jour les fichiers `/src/TigerLexer.java` et `/src/TigerParser.java` suite aux dernières modifications de la grammaire. Il nous suffit de taper la commande `make build` pour procéder à cette mise à jour.

Le script, quant à lui, parcourra tous les tests et s'assurera de leur acceptation par notre grammaire. Il vérifie ceci suite au retour de la commande

`java Test` prenant le fichier de test en entrée standard. Deux compteurs sont incrémentés selon le retour de la commande :

- Le compteur `i` des tests s'étant bien passés ;
- Le compteur `j` des tests ne s'étant pas bien passés.

Une subtilité doit être soulignée concernant le sous dossier `/tests/fail` ; on considère que si un de ses fichiers n'est pas accepté par la grammaire, il convient d'incrémenter le compteur `i` car il est normal qu'il ne passe pas. Dans le cas contraire, l'autre compteur sera incrémenté.

6 Gestion de projet

6.1 Gantt

Un Gantt à été réalisé en début du projet pour définir les tâches à effectuer (figure 9). Les responsables de ces tâches ont été déterminés naturellement au cours du projet à mesure que chacun prenait ses marques dans les tâches qui lui étaient attribuées.

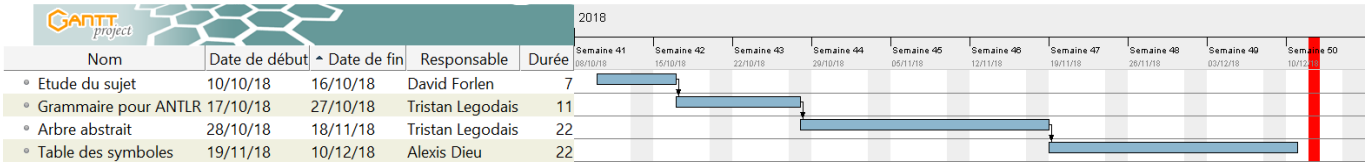

Figure 9: Gantt du projet

6.2 Temps de travail

Le tableau suivant recense le nombre d'heures de travail passées par chacun sur les principales tâches identifiées au cours du projet.

Tâche	David	Tristan	Alexis	Philippe
Prise de connaissance du sujet	3	0.5	0.5	2.5
Écriture de la grammaire	6	11	6	4
Écriture des tests	2	9	4.5	10
Réalisation de la table des symboles	2	1	5.5	3
Réalisation de l'arbre abstrait	6	5	1	1
Participation aux réunions	9	9	9	9
Écriture du rapport	5	5	6	3
Total	33	39.5	32.5	32.5

Figure 10: Répartition du travail

6.3 Colorateur syntaxique pour *Atom*

Au cours de ce projet, nous avons été amenés à écrire un grand nombre de programmes *Tiger*, notamment dans le cadre de la suite de tests. Bien que notre analyseur syntaxique soit en mesure d'indiquer si un programme est valide ou non, il peut être fastidieux d'en faire un usage répété lors de son écriture, juste pour y déceler de potentielles fautes de frappe.

Pour réduire la fréquence des erreurs d'inattention et ainsi améliorer la productivité, un colorateur syntaxique de *Tiger* de type *TextMate* a été réalisé

pour *Atom*, éditeur de texte que nous utilisons fréquemment, mais pour lequel il n'existait alors pas de tel colorateur syntaxique. En pratique, il s'agit d'un simple analyseur lexical qui met en évidence les mot-clés du langage *Tiger*, tel que dans la figure 11.

Afin d'en faire profiter chaque membre du groupe, le colorateur syntaxique a été publié en tant que paquet installable directement depuis *Atom*, et son code peut être consulté à l'adresse atom.io/packages/language-tiger.

```
let
    type vector = {x : int, y : int}
    type data = {v : vector, to_add : int}
    type dataArray = array of data
    var d1 := data {v = vector {x = 0, y = 0}, to_add = 1}
    var d2 := data {v = vector {x = 1, y = 1}, to_add = 1}
    var d3 := data {v = vector {x = 2, y = 2}, to_add = 0}
    var d4 := data {v = vector {x = 3, y = 4}, to_add = 1}
    var l := dataArray [4] of nil
    var result := vector {x = 0, y = 0}
    function add_vectors (l : dataArray, size : int, result : vector) =
        for i := 0 to size - 1 do
            if l [i].to_add then (
                result.x := result.x + l [i].v.x;
                result.y := result.y + l [i].v.y
            )
in
    l [0] := d1;
    l [1] := d2;
    l [2] := d3;
    l [3] := d4;
    add_vectors (l, 4, result);
    print (result.x);
    print (result.y)
end
```

Figure 11: Exemple de coloration syntaxique

6.4 Comptes-rendus de réunion

Ci-après se trouvent les notes prisent lors des neuf réunions, constituant ainsi autant de comptes-rendus de réunion.

Réunion 1

Date et heure. 10 octobre 2018 à 14h30

Prochaine réunion. 17 octobre 2018 à 14h00

Ordre du jour.

1. Prise de connaissance du sujet
2. Chef de projet
3. Gestion du temps
4. Travail à faire

1. Prise de connaissance du sujet

David résume le sujet. Il s'agit de programmer un compilateur *Tiger*. Il faut gérer les erreurs de compilation + numéros de lignes (tout type d'erreurs). Seules les erreurs sémantiques n'arrêtent pas la compilation.

Tout au long du projet, on devra ajouter des tests et les garder.

Pour le 12 décembre.

1. Faire la grammaire (est-elle déjà donnée dans le sujet ?). On la soumettera à ANTLR qui génèrera l'ASD (il fait aussi l'analyse lexicale)
2. Construction de l'arbre abstrait (ce qui a été commencé en TD).
3. Table des symboles. Elle n'a pas besoin d'être utilisée à la fin de la première partie mais doit être donnée dans le PDF dès le premier semestre.

Il faut qu'on ait des jeux d'essais pour montrer le fonctionnement de la grammaire, de l'analyse syntaxique et de la TDS.

Pour le 23 avril

Génération du code assembleur de manière incrémentale.

Il y aura aussi un GANTT qui évoluera au cours du temps et qui devra être présent à la fois dans le rapport partiel, et dans le rapport final.

Discussion autour du nombre de GANTT à réaliser. Il y aura bien *deux* rapports à faire.

David demande quand sera le TP sur ANTLR. Il aura lieu le vendredi 12 octobre de 14 heures à 16 heures.

Alexis et Philippe irons à l'initiation de ANTLR le jeudi 11 octobre à 12 heures.

2. Chef de projet

Philippe GRAFF sera le chef de projet. David sera animateur de réunion, et Alexis secrétaire.

Alexis et Philippe examineront la platforme Slack pour voir s'il est possible de s'en servir comme support pour le projet.

David réalisera le GANTT et Tristan le WBS.

Tristan remarque que la grammaire semble déjà donnée. David propose de faire un GANTT avec seulement 3 étapes. David a fait un drive.

3. Gestion du temps

Une fois le WBS réalisé, on notera dans un fichier (latex ?) le nombre d'heures réalisées pour chaque tâches.

4. Travail à faire

Alexis.

- Lire le texte de cours envoyé mar Mme Colin
- Lecture complète du sujet et des documentations
- Installer ANTLR
- Examiner la plateforme Slack pour voir si on peut s'en servir comme support de gestion de projet

Philippe.

- Lire le texte de cours envoyé mar Mme Colin
- Lecture complète du sujet et des documentations
- Installer ANTLR
- Examiner la plateforme Slack pour voir si on peut s'en servir comme support de gestion de projet

David.

- Lire le texte de cours envoyé mar Mme Colin
- Lecture complète du sujet et des documentations
- Installer ANTLR
- Réaliser le Gantt

Tristan.

- Lire le texte de cours envoyé mar Mme Colin
- Lecture complète du sujet et des documentations

- Installer ANTLR
- Réaliser un WBS

Réunion 2

Date et heure. 17 octobre 2018 à 14h00

Prochaine réunion. 24 octobre 2018 à 14h30

Ordre du jour.

1. Jalons
2. Répartition de la grammaire
3. Travail à faire

1. Jalons

On va voir comment se répartir les règles. On pourrait, d'après David, le faire par groupe de deux.

On commence par le rappel des objectifs. Slack n'apporterait rien en terme de gestion de projet vs. discord ou le drive.

En ce qui concerne le Gantt, David s'en est occupé. Il le mettera en format éditable sur le drive. On aura 10 jours pour effectuer la grammaire, et ensuite vingt-deux jours pour faire chaque tâches.

Il faut absolument penser à garder trace du temps de travail pour chaque tâches.

Il faudra mettre les comptes-rendus sur GIT. Tout le monde doit faire un `git clone`. Pour rédiger n'importe quel document texte, préférer le format markdown lisible directement depuis gitlab.

Bilan sur le cours envoyé pas Mme. Colin. Il y a certains détails qui n'apparaissent pas dans la documentation de Tiger. Tout le monde a lu le sujet.

2. Répartition de la grammaire

Nous allons répartir les tâches pour réaliser la grammaire. Notations de ANTLR.

- L'étoile en exposant signifie *zéro ou plus*
- Le plus en exposant signifie *une ou plus*
- Le `opt`en indice signifie *au plus un*
- `*;` en exposant signifie *zéro ou plus séparés par un point virgule*

Il faudra compléter la grammaire (les règles concernant les additions n'apparaissent pas dans la documentation, par exemple). Il faudra aussi s'occuper des expressions régulières pour reconnaître les tokens.

On devra, dans l'ordre :

1. Factoriser
2. Dérécursiver

3. Prioriser
4. Mettre sous format ANTLR

On verra les commentaires plus tard. On pourra, d'après David, éventuellement télécharger un compilateur Tiger pour l'utiliser comme contrôle.

3. Travail à faire

Alexis.

- Terminer le premier TP de traduction
- Mettre les réunion en ligne
- Écrire les règles : tout ce qui est avant exp exclus plus letexp, et pour les terminaux, *tyid* et *id*

Philippe.

- Écrire les règles : tout ce qui est avant exp exclus plus letexp, et pour les terminaux, *tyid* et *id*

David.

- Écrire les règles : tout ce qui est après exp inclus, tous les autres terminaux (*infixop*, *intlit*, *stringlit*)

Tristan.

- Chercher en ligne un compilateur *Tiger*
- Écrire les règles : tout ce qui est après exp inclus, tous les autres terminaux (*infixop*, *intlit*, *stringlit*)

Réunion 3

Date et heure. 24 octobre 2018 à 14h30

Prochaine réunion. 2 novembre 2018 à 16h00

Ordre du jour.

1. Jalons
2. Grammaire
3. Travail à faire

1. Jalons

Tristan a beaucoup réorganisé la grammaire afin que la priorisation soit faite. Tristan nous explique comment il a fait. Il a remit les terminaux à la fin. Les affectations n'ont pas été faites. Pour ce qui est des `if else`, il y a encore des ambigüités.

2. Grammaire

D'après David, on devrait pouvoir lancer une grammaire, malgré l'ambigüité, car ANTLR prends la première solution qu'il voit (on a toujours des warning, par contre).

Problèmes avec `unaryExp`. On a rendu `expression` synonyme de `infixExp`, qui lui même est synonyme de `orExp`.

Tout le monde ira au tutorat. Il reste trois jours pour faire la grammaire. Après, il faudra commencer à faire l'AST.

3. Travail à faire

Alexis.

- Terminer les règles : tout ce qui est avant exp exclus plus letexp, et pour les terminaux, *tyid* et *id*

Philippe.

- Terminer les règles : tout ce qui est avant exp exclus plus letexp, et pour les terminaux, *tyid* et *id*

David.

- Terminer les règles : tout ce qui est après exp inclus, tous les autres terminaux (*infixop*, *intlit*, *stringlit*)

Tristan.

- Terminer les règles : tout ce qui est après exp inclus, tous les autres terminaux (*infixop*, *intlit*, *stringlit*)

Réunion 4

Date et heure. 2 novembre 2018 à 16h00

Prochaine réunion. 7 novembre 2018 14h30

Ordre du jour.

1. Grammaire
2. Travail à faire

1. Grammaire

Philippe a corrigés quelques bugs avec for. Les tests automatiques ne sont pas encore opérationnels, mais on peut faire les tests depuis antlr. Pas mal de choses ne passaient pas.

S'il y a un problème dans la reconnaissance, il n'y a apparemment pas de code d'erreur. Il faudra tester la sortie standard.

La grammaire n'est pas encore fonctionnelle, il faudra voir quelles règles ne sont pas fonctionnelles. Tristan a décrit les règles qui ne sont pas opérationnelles. tyId est apparemment problèmatique.

Il faudra fusionner tyId et Id. Il s'agit d'un travail assez important, d'autant qu'on a un autre problème : les affectations. Une affectation commence aussi par un Id/typeId.

David propose de faire beaucoup de tests.

Il faut tester les AST sur nos tests. Il n'est pas nécessaire de réécrire toutes les règles. David rappelle qu'il y a un manuel Antlr sur le arche. Il y a tout un chapitre sur les réécritures (chapitre 8, page 191).

Il faudrait commencer la table des symboles, et comment récupérer l'arbre abstrait en Java pour faire les contrôles sémantiques.

2. Travail à faire

Alexis.

- Tester ses réécritures
- Poursuivre l'AST
- Écrire au moins deux tests par règle

Philippe.

- Étudier la table des symboles
- Poursuivre l'AST
- Écrire au moins deux tests par règle

David.

- Poursuivre l'AST
- Écrire au moins deux tests par règle
- Mettre en ligne le PDF de Antlr

Tristan.

- Expliquer aux autres les problèmes restants sur la grammaire
- Poursuivre l'AST
- Écrire au moins deux tests par règle

Réunion 5

Date et heure. 7 novembre 2018 à 14h30

Prochaine réunion. 14 novembre 2018 à 14h30

Ordre du jour.

1. Jalons
2. Assignement de Tristan
3. Travail à faire

1. Jalons

On va commencer par le travail de chacun. Les règles de l'arbre abstraits ont été testées par Alexis, mais pas toutes.

Les règles supplémentaires ont été pushées par David. Il a également ajouté des tests, chacun très atomique. Il a ajouté le PDF de Antlr dans les docs.

Penser à compiler avant de lancer le test avec `make build`.

Philippe remarque que certains tests fail semblent pourtant correct. Tristan rappelle qu'un fichier tiger ne peut pas contenir seulement un commentaire. David demande à Tristan d'expliquer sa nomenclature des fichiers tests pour les commentaires.

Tristan pense qu'Antlr ne provoque pas immédiatement d'erreurs concernants les réécritures, mais en provoque lors des tests.

Philippe a écrit deux tests et en réécrira la semaine prochaine.

Philippe met quelques ressources sur la table des symboles sur le drive.

David se demande à quelle point il faut avancer les tests. Tristan pense qu'il ne faut faire que les tests.

Tristan évoque ses tests. Il en a ajouté pas mal qui testent surtout les opérations arithmétiques. Les affectations font aussi partie des opérations "arithmétiques". Suite à l'ajout des affectation, il y a plus de tests qui passent. Normalement, à son prochain commit, tous ses tests devraient passer, même s'il y a d'autres problèmes.

2. Assignement de Tristan

D'autres points étaient en suspens, notamment les affectations, TYID et ID posant problème. Les affectations sont détaillées sur gitlab dans un fichier ISSUES.md. Il explique pourquoi on avait un problème et comment il a été "résolu". Une affectation s'écrit lValue ':=' exp. Cela pose problème car quand on lit une expression, elle peut commencer par un lValue. Ce lValue, on ne sait pas s'il est

en plein milieu d'une expression arithmétique ou s'il est associé à une affectation. Quand on analyse l'expression globale, on ne sait pas s'il va s'agir d'une affectation ou d'une multiplication, il faut lire des caractères en avance.

Ce que l'on va faire, c'est considérer qu'il est valide d'écrire (a*b):=c d'*un point de vu *syntaxique*, mais que sa *sémantique* est fausse.

3. Travail à faire

Alexis.

- Ajouter des tests plus complexes
- Tester les réécritures

Philippe.

- Ajouter des tests plus complexes
- Effectuer au moins deux tests par règles

David.

- Ajouter des tests plus complexes

Tristan.

- Ajouter des tests plus complexes

Réunion 6

Date et heure. 14 novembre 2018 à 14h30

Prochaine réunion. 21 novembre 2018 à 14h30

Ordre du jour.

1. Jalons
2. Remarques de Tristan
3. Table des symboles
4. Travail à faire

1. Jalons

Alexis a testé toutes ses réécritures et elles sont fonctionnelles. Philippe a ajouté quelques tests ainsi que les exemples envoyés par les enseignants. Philippe a réécrit sa règle `fieldExp` et va écrire `funDec` qui n'avait pas été faite.

David a débloqué Tristan et a fini l'AST du `letExp`, et a ajouté quelques tests pour voir si ça marchait bien.

2. Remarques de Tristan

2.1. Retour sur le tutorat

Ce tutorat a été utile car il a permet de mettre en lumière les problèmes qui n'étaient pas très clair. Philippe remarque que l'AST était particulièrement éclairant.

Mais les problèmes évoqués ont déjà été résolus. Par contre, tout ce qui a été évoqué, que ce soit des problèmes ou des exemples évoqués, sont bien en correspondance avec ce que nous avons fait. Exemple : les arbres n-aires pour les séquences.

Cela valide la liberté que l'on a prise d'autoriser n'importe quelle expression à gauche d'une affectation et pas seulement un identifiant, et de faire le test dans le contrôle sémantique. La structure proposée pendant le tutorat correspond à ce que nous avons fait.

2.2. Remarque sur les crochets et les points

Dans ce que Tristan avait à faire : résolution d'un problème d'AST de David lié aux suites de points et de crochets qui n'incluaient que le dernier. Lorsque l'on faisait `a.b.c`, l'AST n'affichait que `a.c`. David avait déjà rencontré ce problème avant l'AST, mais tout a été résolu.

Tristan a factorisé mercredi dernier les types et les identifiants.

2.3. Tests

Tristan a mit une vingtaine de tests qui échouent, et ce volontairement.

2.4. Coloration syntaxique

Tristan a créé un paquet Atom pour faire de la coloration syntaxique sur Tiger.

On peut l'installer depuis Atom directement.

2.5. Priorisation des tests et des boucles

Soit les tests suivants, écrits en Tiger (un par ligne) :

```
(1) a + if b then c else d + e
(2) a + (if b then c else d := e)
(3) a + if b then c else (d + e)
(4) (a + if b then c else d) + e
(5) if a then b := c else d := f
(6) if a then (b := c) else d := f
(7) if a then (b := c) else (d := f)
```

Comme vu dans le tutorat toute à l'heure, on a deux possibilités de priorité dans le premier tests. Le `if` est-il l'opération la plus prioritaire, ou la moins prioritaire ? Le premier test possède deux interprétations possibles :

1. `if ... (d+e)`
2. `(if ... d) + e`

Dans la première interprétation, seuls les tests (2), (5), (6) et (7) passent. Dans la deuxième interprétation, tous les tests passent.

Le `if` renvoit apparemment bien quelque chose. Exemple.

```
a := if F() then (
    a := y // ici, if renvoit y
) else (
    a := z // ici, if renvoit z
)
```

La question est de savoir : est-ce qu'on doit considérer que le `a :=` au début de ce bloc de code est valide ou non ? Il est possible que cela ait du sens.

Pour l'instant, le premier test ne passe pas et provoque une erreur syntaxique, car on ne peut pas faire `a+if`. Il pourrait être intéressant d'utiliser la deuxième interprétation, c'est à dire celle ou après un `else`, on attend un `unaryExp`. Par contre, on a un petit problème, car on accepte des choses qu'on ne devrait pas accepter, en particulier :

```
a + if b then c else d := e
```

Ce test n'a aucun sens mais passe dans le deuxième cas.

En somme, qu'est-ce qu'on fait ? Pour l'instant, tout ce passe comme si `if` ne renvoyait rien.

2.5. Pour le rapport

- Priorisation du `if`
- Priorisation de l'affectation
- Arbres n-aires pour |, &, +, et *

3. Table des symboles

Il va falloir se lancer dessus pour la prochaine réunion. David remarque qu'il nous reste trois semaines. Il faut qu'on fasse :

- Le rapport
- Table des symbole
- Finir la grammaire (déjà presque finie)

4. Travail à faire

Alexis.

- Écrire l'AST de la règle `funDec` et `fieldDec`
- Expliquer la priorisation du `if` dans le rapport
- Expliquer la partie de l'AST conçue dans le rapport
- Expliquer des arbres n-aires pour |, &, +, * et , (virgule)

Philippe.

- Écrire l'AST de la règle `funDec` et `fieldDec`
- Expliquer l'options *greedy* dans le rapport
- Expliquer la partie de l'AST conçue dans le rapport
- S'occuper du template LaTeX

David.

- Commencer la TDS
- Expliquer la priorisation de opérations arithmétiques dans le rapport
- Expliquer la partie de l'AST conçue dans le rapport

Tristan.

- Commencer la TDS

- Expliquer la partie de l'AST conçue dans le rapport
- Expliquer son package Atom dans le rapport
- Expliquer le EOF dans le rapport
- Priorisation de l'affectation
- Expliquer la factorisation de ID et TYID

Réunion 7

Date et heure. 21 novembre 2018 à 14h30

Prochaine réunion. 28 novembre 2018 à 14h00

Ordre du jour.

1. Jalons
2. Table des symboles
3. Travail à faire

1. Jalons

La partie explication de l'AST n'est pas complètement pertinente. Il faut faire une ou deux lignes pour les règles très simples, et expliquer plus en détails les règles de l'AST qui sont plus compliquées.

Philippe a bien réalisé les règles de l'AST qu'il manquait.

David a cherché dans la documention d'Antlr des notes liées à l'AST, mais il n'y a pas grand chose. Antlr ne propose pas de gestion immédiate de la table des symboles. Il va falloir réfléchir à ce que l'on met dans la TDS, et comment on organise le travail.

Tristan a revu ce qu'avait fait Philippe et a remarqué qu'il manquait l'AST de la règle *recTy*. Ensuite, il a raccourci un peu la grammaire en enlevant les non-terminaux inutiles. David ne comprend pas.

Tristan n'a pas encore touché au rapport mais a lu. Il a changé les noms des sous-arbres que Philippe a utilisé.

2. Table des symboles

Tristan est allé au tutorat qui n'a pas été très long. La TDS a été évoqué. Chacun la fait à sa manière. Tout le monde produira des TDS différentes, que ce soit le nombre de tableaux, le nombre de lignes, etc. Il est possible de réaliser une TDS par bloc, *ie.* par boucle, fonction et let.

Tristan se demande où doivent apparaîtres les fonctions *built-in*, c'est-à-dire les fonctions comme *print*, et les types *int*, *string*...

On pourrait créer une table des symboles *globale* qui pourrait contenir toutes ces fonctions. Le parcours de l'arbre ne peut pas être linéaire.

David remarque qu'il nous reste trois semaines, et qu'il va falloir bosser la TDS. Tout le monde fait Anim'est sauf Tristan qui pourra miner le terrain pour la TDS. Un noeud de notre AST, ses fils sont les colonnes de la TDS.

3. Travail à faire

Alexis.

- Coder un programme Tiger complexe
- Effectuer cinq tests sémantiques

Philippe.

- Coder un programme Tiger complexe
- Effectuer cinq tests sémantiques

David.

- Coder un programme Tiger complexe
- Effectuer cinq tests sémantiques

Tristan.

- Coder un programme Tiger complexe
- Effectuer cinq tests sémantiques
- Se renseigner d'avantage sur la TDS

Réunion 8

Date et heure. 28 novembre 2018 à 14 heures

Prochaine réunion. 5 décembre 2018 à 15 heures

Ordre du jour.

1. Jalons
2. Travail à faire

1. Jalons

David a commencé sa part du rapport, surtout au niveau de la priorisation des règles.

Tristan a réfléchi à la TDS. La dernière fois, il avait évoqué une TDS "d'ordre zéro". C'est-à-dire que le type int doit être déclaré, et pré-existe avant que l'on lance le programme, ainsi que les fonctions de base comme print. Il faut bien noter que *int* n'est pas vraiment défini à partir de *typedef*, vu qu'il ne peut pas être généré à partir des autres types. Il faut aussi préciser la taille de int (32 ?) et string (taille d'un pointeur).

Seulement, il faudra bien noter que si print n'est pas utilisé dans notre code, il ne devra pas être "importé" dans la TDS.

Chose apparamment oubliée dans le tutorat. Dans la TDS qui a été proposée, on met le type de retour pour les fonctions. Sauf qu'en tiger, un type est aussi une forme de variable. Donc dans la TDS, il faut mettre un pointeur.

Tristan propose de mettre les TDS dans un arbres.

On utilisera Intellij.

2. Travail à faire

Alexis.

- Faire la TDS
- Coder un programme Tiger complexe
- Effectuer cinq tests sémantiques

Philippe.

- Faire la TDS
- Coder un programme Tiger complexe
- Effectuer cinq tests sémantiques

David.

- Poursuivre le rapport
- Coder un programme Tiger complexe
- Effectuer cinq tests sémantiques

Tristan.

- Rédiger le manuel d'utilisation
- Poursuivre le rapport
- Coder un programme Tiger complexe
- Effectuer cinq tests sémantiques

Réunion 9

Date et heure. 5 décembre 2018 à 15 heures

Prochaine réunion. *Non plannifiée*

Ordre du jour.

1. Jalons
2. État du rapport
3. Table des symboles
4. Travail à faire

1. Jalons

Alexis doit ajouter des points virgules là où il en faut dans son test complexe.

Philippe a mis des tests sémantiques et a programmé un triangle de Pascal. Il y a quelques erreurs syntaxiques (il manque des parenthèses et des points-virgules).

David a créé un test sémantique corrigé par Tristan. Il peut être amélioré. Il a fait plus de cinq tests sémantiques, assez similaires les uns les autres.

David a avancé dans le rapport.

Tristan a effectué un arbre binaire de recherche en Tiger qui affiche cet arbre avec des `print` de manière indentée.

Tristan remarque qu'il y a un problème au niveau sémantique de l'expression `5 || 4`. Est-ce que cela doit renvoyer `5` ou `4` ? Tristan propose de renvoyer la dernière valeur évaluée.

2. État du rapport

David remarque qu'il y a beaucoup de trous dans le rapport. La gestion de projet est à faire pour la prochaine fois.

3. Table des symboles

David a trouvé des ressources concernant la visualisation de l'arbre. Il l'ajoute sur Discord.

4. Travail à faire

Alexis.

- Corriger son programme complexe
- Mettre sur le rapport son temps de travail
- Mettre les CR dans le rapport

- Corriger la TDS
- Rédiger la partie TDS dans le rapport

Philippe.

- Corriger son test sémantique
- Corriger son programme complexe
- Mettre sur le rapport son temps de travail
- Paragraphe sur les jeux d'essais à rédiger
- Corriger la TDS
- Rédiger la partie TDS dans le rapport

David.

- David va compléter la partie gestion de projet
- Mettre sur le rapport son temps de travail
- Faire la page de garde du rapport

Tristan.

- Manuel d'utilisation des jeux de tests à rédiger
- Mettre sur le rapport son temps de travail
- Faire un tableau pour la gestion du temps de travail