Projet de compilation des langages

Rapport

Sacha Lemonnier
Louise Pount
Gautier Raimondi
Maxime Riere

Année 2017–2018

Encadrant : Suzanne Collin

Déclaration sur l'honneur de non-plagiat

Je soussigné(e),

Nom, prénom : Lemonnier, Sacha

Élève-ingénieur(e) régulièrement inscrit(e) en 2e année à TELECOM Nancy

Numéro de carte de l'étudiant(e) : numcarte

Année universitaire : 2017–2018

Auteur(e) du document, mémoire, rapport ou code informatique intitulé :

Rapport de Projet de Compilation des Langages

Par la présente, je déclare m'être informé(e) sur les différentes formes de plagiat existantes et sur les techniques et normes de citation et référence.

Je déclare en outre que le travail rendu est un travail original, issu de ma réflexion personnelle, et qu'il a été rédigé entièrement par mes soins. J'affirme n'avoir ni contrefait, ni falsifié, ni copié tout ou partie de l'œuvre d'autrui, en particulier texte ou code informatique, dans le but de me l'accaparer.

Je certifie donc que toutes formulations, idées, recherches, raisonnements, analyses, programmes, schémas ou autre créations, figurant dans le document et empruntés à un tiers, sont clairement signalés comme tels, selon les usages en vigueur.

Je suis conscient(e) que le fait de ne pas citer une source ou de ne pas la citer clairement et complètement est constitutif de plagiat, que le plagiat est considéré comme une faute grave au sein de l'Université, et qu'en cas de manquement aux règles en la matière, j'encourrais des poursuites non seulement devant la commission de discipline de l'établissement mais également devant les tribunaux de la République Française.

Fait à Villers-lès-Nancy, le 18 mai 2018

Signature :

Déclaration sur l'honneur de non-plagiat

Je soussigné(e),

Nom, prénom : Pount, Louise

Élève-ingénieur(e) régulièrement inscrit(e) en 2e année à TELECOM Nancy

Numéro de carte de l'étudiant(e) : 31618903

Année universitaire : 2017–2018

Auteur(e) du document, mémoire, rapport ou code informatique intitulé :

Rapport de Projet de Compilation des Langages

Par la présente, je déclare m'être informé(e) sur les différentes formes de plagiat existantes et sur les techniques et normes de citation et référence.

Je déclare en outre que le travail rendu est un travail original, issu de ma réflexion personnelle, et qu'il a été rédigé entièrement par mes soins. J'affirme n'avoir ni contrefait, ni falsifié, ni copié tout ou partie de l'œuvre d'autrui, en particulier texte ou code informatique, dans le but de me l'accaparer.

Je certifie donc que toutes formulations, idées, recherches, raisonnements, analyses, programmes, schémas ou autre créations, figurant dans le document et empruntés à un tiers, sont clairement signalés comme tels, selon les usages en vigueur.

Je suis conscient(e) que le fait de ne pas citer une source ou de ne pas la citer clairement et complètement est constitutif de plagiat, que le plagiat est considéré comme une faute grave au sein de l'Université, et qu'en cas de manquement aux règles en la matière, j'encourrais des poursuites non seulement devant la commission de discipline de l'établissement mais également devant les tribunaux de la République Française.

Fait à Villers-lès-Nancy, le 18 mai 2018

Signature :

Déclaration sur l'honneur de non-plagiat

Je soussigné(e),

Nom, prénom : Raimondi, Gautier

Élève-ingénieur(e) régulièrement inscrit(e) en 2e année à TELECOM Nancy

Numéro de carte de l'étudiant(e) : 31521501

Année universitaire : 2017–2018

Auteur(e) du document, mémoire, rapport ou code informatique intitulé :

Rapport de Projet de Compilation des Langages

Par la présente, je déclare m'être informé(e) sur les différentes formes de plagiat existantes et sur les techniques et normes de citation et référence.

Je déclare en outre que le travail rendu est un travail original, issu de ma réflexion personnelle, et qu'il a été rédigé entièrement par mes soins. J'affirme n'avoir ni contrefait, ni falsifié, ni copié tout ou partie de l'œuvre d'autrui, en particulier texte ou code informatique, dans le but de me l'accaparer.

Je certifie donc que toutes formulations, idées, recherches, raisonnements, analyses, programmes, schémas ou autre créations, figurant dans le document et empruntés à un tiers, sont clairement signalés comme tels, selon les usages en vigueur.

Je suis conscient(e) que le fait de ne pas citer une source ou de ne pas la citer clairement et complètement est constitutif de plagiat, que le plagiat est considéré comme une faute grave au sein de l'Université, et qu'en cas de manquement aux règles en la matière, j'encourrais des poursuites non seulement devant la commission de discipline de l'établissement mais également devant les tribunaux de la République Française.

Fait à Villers-lès-Nancy, le 18 mai 2018

Signature :

Déclaration sur l'honneur de non-plagiat

Je soussigné(e),

Nom, prénom : Riere, Maxime

Élève-ingénieur(e) régulièrement inscrit(e) en 2e année à TELECOM Nancy

Numéro de carte de l'étudiant(e) : numcarte

Année universitaire : 2017–2018

Auteur(e) du document, mémoire, rapport ou code informatique intitulé :

Rapport de Projet de Compilation des Langages

Par la présente, je déclare m'être informé(e) sur les différentes formes de plagiat existantes et sur les techniques et normes de citation et référence.

Je déclare en outre que le travail rendu est un travail original, issu de ma réflexion personnelle, et qu'il a été rédigé entièrement par mes soins. J'affirme n'avoir ni contrefait, ni falsifié, ni copié tout ou partie de l'œuvre d'autrui, en particulier texte ou code informatique, dans le but de me l'accaparer.

Je certifie donc que toutes formulations, idées, recherches, raisonnements, analyses, programmes, schémas ou autre créations, figurant dans le document et empruntés à un tiers, sont clairement signalés comme tels, selon les usages en vigueur.

Je suis conscient(e) que le fait de ne pas citer une source ou de ne pas la citer clairement et complètement est constitutif de plagiat, que le plagiat est considéré comme une faute grave au sein de l'Université, et qu'en cas de manquement aux règles en la matière, j'encourrais des poursuites non seulement devant la commission de discipline de l'établissement mais également devant les tribunaux de la République Française.

Fait à Villers-lès-Nancy, le 18 mai 2018

Signature :

Projet de compilation des langages

Rapport

Sacha Lemonnier
Louise Pount
Gautier Raimondi
Maxime Riere

Année 2017–2018

Sacha Lemonnier
Louise Pount
Gautier Raimondi
Maxime Riere
sacha.lemonnier@telecomnancy.eu
louise.pount@telecomnancy.eu
gautier.raimondi@telecomnancy.eu
maxime.riere@telecomnancy.eu

TELECOM Nancy
193 avenue Paul Muller,
CS 90172, VILLERS-LÈS-NANCY
+33 (0)3 83 68 26 00
contact@telecomnancy.eu

Encadrant : Suzanne Collin

Projet de compilation des langages

Rapport

Sacha Lemonnier
Louise Pount
Gautier Raimondi
Maxime Riere

Année 2017–2018

Encadrant : Suzanne Collin

Remerciements

“Nous tenons à remercier l’intégralité du corp enseignants du module de Traduction des Langages de TELECOM Nancy, à savoir Mme. Suzanne Collin, M. Sébastien Da Silva et M. Pierre Monnin, pour leur investissement dans l’enseignement de la matière ainsi que pour les conseils prodigués tout au long de ce projet. ”

– Sacha Lemonnier, Louise Pount, Gautier Raimondi, Maxime Riere

Table des matières

1 Introduction

Le projet que nous avons réalisé nous a été proposé dans le cadre de notre cursus à TELECOM Nancy. Il nous a permis de mettre en pratique les compétences que nous avons acquises au cours de cette année, notamment au sein du module de TRAD. Il fait aussi appel au cours de PFSI que nous avons suivi en première année.

L'objectif de ce travail est la création d'un compilateur du langage miniRust décrit ci-dessous.

```
FICHIER      ⟶  DECL*
DECL         ⟶  DECL_FUNC | DECL_STRUCT
DECL_STRUCT  ⟶  struct idf { [ idf : TYPE (, idf : TYPE)* ] }
DECL_FUN     ⟶  fn idf ( [ ARGUMENT (, ARGUMENT)* ] ) [ -> TYPE ] BLOC
TYPE         ⟶  idf | idf < TYPE > | & TYPE | i32 | bool | Vec
ARGUMENT     ⟶  idf : TYPE
BLOC         ⟶  { INSTRUCTION* [EXPR] }
INSTRUCTION  ⟶  ;
                | EXPR ;
                | let [mut] idf = EXPR ;
                | let [mut] idf = idf { [ idf : EXPR (, idf : EXPR)* ] }
                | while EXPR BLOC
                | return [EXPR] ;
                | IF_EXPR
IF_EXPR      ⟶  if EXPR BLOC [ else (BLOC | IF_EXPR) ]
EXPR         ⟶  cste_ent | true | false
                | idf
                | EXPR BINAIRE EXPR
                | UNAIRE EXPR
                | EXPR . idf
                | EXPR . len ( )
                | EXPR [ EXPR ]
                | idf ( [ EXPR (, EXPR)* ] )
                | vec ! [ [ EXPR (, EXPR)* ] ]
                | print ! ( EXPR )
                | BLOC
                | ( EXPR )
BINAIRE      ⟶  + | - | * | / | && | || | < | <= | > | >= | == | !=
UNAIRE       ⟶  - | ! | * | &
```

Figure 1.1 – Langage miniRust

Le compilateur que nous proposons a été construit en trois phases. La première a consisté à définir la grammaire du langage en l'adaptant au logiciel ANTLR. Ce logiciel nous a également servi à générer des arbres syntaxiques utiles pour la seconde étape du développement. Cette deuxième étape a, quant à elle, consisté à définir des contrôles sémantiques chargés de prévenir l'utilisateur s'il commet une erreur sémantique. Pour réaliser ces contrôles nous avons utilisé le langage JAVA. Enfin, la dernière étape fut la génération de code. C'est-à-dire la traduction de miniRust vers du code assembleur afin que l'appel à notre compilateur soit fonctionnel.

Ce rapport détaille ces trois étapes de conception.

2 Grammaire

Dans ce chapitre nous détaillons la grammaire fournie, les modifications que nous lui avons apporté afin qu'elle réponde davantage à nos besoins ainsi que la manière dont nous l'avons implémenté à l'aide du logiciel ANTLR.

2.1 Présentation de la grammaire

La grammaire qui nous a été fournie propose une variante du langage Rust : miniRust. Cette proposition présentait cependant quelques problèmes parmi lesquels la récursivité gauche. Pour pallier ces soucis, nous avons choisi de modifier le miniRust fourni. Nous lui avons également ajouté du code permettant la génération d'un Arbre Syntaxique Abstrait.
Vous trouverez la grammaire que nous proposons en Annexe A.

2.2 Mise en oeuvre et arbre syntaxique

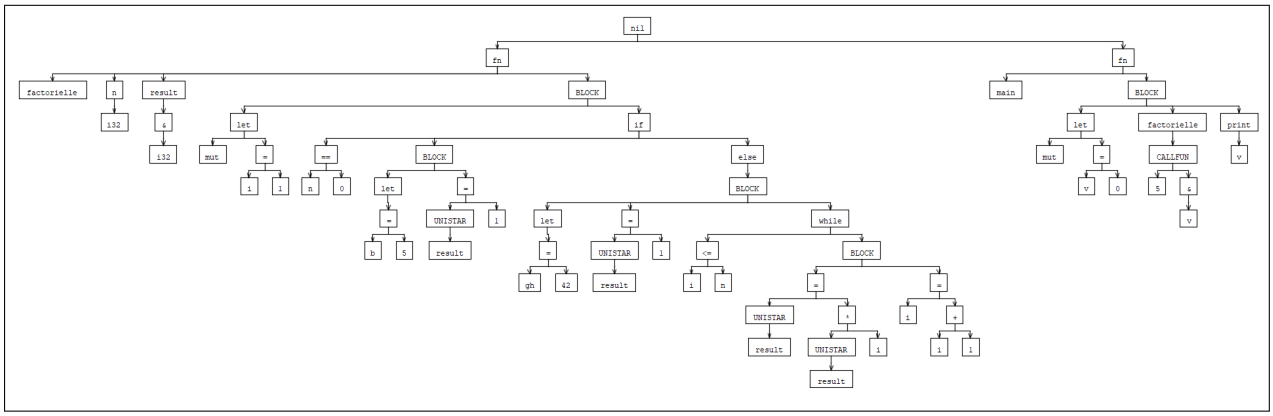
Les ajustements que nous proposons ainsi que nos choix d'implémentation sont présentés ci-dessous. Vous trouverez également le détail du calcul de l'arbre syntaxique ainsi que les résultats que nous obtenons.

2.2.1 Calcul de l'arbre syntaxique

Lorsque nous soumettons un fichier d'entrée à ANTLR, le logiciel commence par le parser. La génération de l'arbre syntaxique s'effectue au cours de cette étape. Les indications de génération d'arbre sont ainsi présentes au sein même de notre grammaire. Ces instructions sont précédées, comme la syntaxe ANTLR l'impose, par '->'. Nous présentons ci-dessous un exemple de génération d'arbre syntaxique depuis un fichier d'entrée présentant une fonction factorielle.

```
1  fn factorielle(n : i32, result : &i32)
2  {
3    let mut i = 1;
4    if n==0
5    {
6      let b = 5;
7      *result = 1;
8
9    } else{
10     let gh = 42;
11         *result = 1;
```

```
    while i <= n
    {
      *result = *result * i;
      i = i+1;
    }
  }
}

fn main(){
  let mut v = 0;
  factorielle(5,&v);
  print!(v);
}
```



Arbre généré après soumission du code ci-dessus

2.2.2 Définition du programme

L'extrait de grammaire présenté dans la figure ci-dessous correspond au commencement du programme. On remarque qu'un programme est défini comme une liste de déclaration de fonctions et de structures, et qu'au moins une des fonctions doit s'appeler "main".

```
axiom   : fichier EOF {if (!mainFound){System.err.println("main not found");}} -> fichier //Ok !
;

fichier : decl* //Ok !
;

decl : declFun //Ok !
  | declStruct
;
```

2.2.3 Déclaration de structure

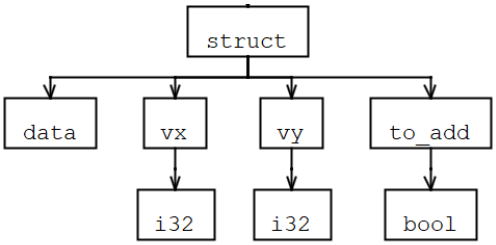
La définition d'une structure se fait selon la syntaxe suivante :

struct structName (attr1 : **attr1Type**(, attri : attriType)*

Ce qui donne l'extrait de grammaire ci-dessous. Ici, la liste d'attributs a été ramené vers une règle différente pour pouvoir mettre en commun avec les arguments de fonctions.

```
decl : declFun //Ok !
 | declStruct
;

declStruct : 'struct' IDF '{' args? '}' -> ^('struct' IDF args?) //Ok !
;

args : IDF ':' type (',' IDF ':' type)* -> (^(IDF type))*//Ok !
;
```

Concernant l'arbre syntaxique, on définit 'struct' comme la racine de notre branche et chaque nom d'attribut comme un noeud de nouvelle branche. On obtient par exemple l'arbre :



2.2.4 Déclaration de fonction

La définition d'une fonction se fait selon la syntaxe suivante :

fn fnName (param1 : **param1Type**(, param : paramType)*)

Ce qui donne l'extrait de grammaire ci-dessous.

```
declFun : 'fn' (IDF '(' args? ')' ('->' type)? block -> ^('fn' IDF ^('->' type)? args?  block)
                |                {mainFound = true;}MAIN '(' ')' block -> ^('fn' MAIN block))
;
```

Concernant l'arbre syntaxique, on définit 'fn' comme la racine de notre branche. Comme fils gauche du noeud racine, on a le nom de la fonction, puis le noeud '->' correspondant au type de retour, ensuite les paramètres et enfin le fils droit et le noeud 'BLOCK' qui va contenir le corps de la fonction. On obtient par exemple l'arbre :

2.2.5 Déclaration de variable

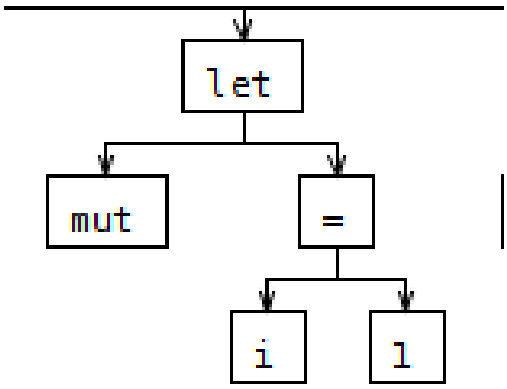
La définition d'une variable se fait selon la syntaxe suivante :

let varname (: **type**)? = expr ;

Ce qui donne l'extrait de grammaire ci-dessous :

```
| 'let' 'mut'? dotIDF (':' type)? '=' bigExpr ';' instruct?-> ^('let' 'mut'? (type)? ^('=' dotIDF bigExpr)) instruct?
```

Concernant l'arbre syntaxique, on définit 'let' comme la racine de notre branche. On peut avoir un ou deux fils sur ce noeud racine selon que la variable déclarée est mutable ou non. Si elle l'est, le noeud racine aura comme fils gauche le noeud 'mut' et comme fils droit le noeud '='. Si la variable n'est pas mutable, alors le noeud 'let' n'aura qu'un seul fils : '='. On obtient par exemple l'arbre :



2.2.6 Instruction

L'appel à une instruction correspond à plusieurs possibilités. C'est le bloc de base de l'écriture du programme.

Cela correspond à l'extrait de grammaire ci-dessous.

```
instruct :
                                expr instrBoucle
 | ';' instruct?-> instruct?
 | 'let' 'mut'? dotIDF (':' type)? '=' bigExpr ';' instruct?-> ^('let' 'mut'? (type)? ^('=' dotIDF bigExpr)) instruct?
 | 'while' expr block instruct?-> ^('while' expr block) instruct?
 | 'return' expr? ';' instruct?-> ^('return' expr?) instruct?
 | 'loop' block instruct?-> ^('loop' block) instruct?
 | 'break' ';' instruct?-> 'break' instruct?
 | ifExpr instruct?
;
```

Une instruction étant ce qu'elle est, on ne peut pas vraiment ressortir un arbre syntaxique général, celui-ci étant propre à l'instruction appelée.

2.2.7 Conditionnelle

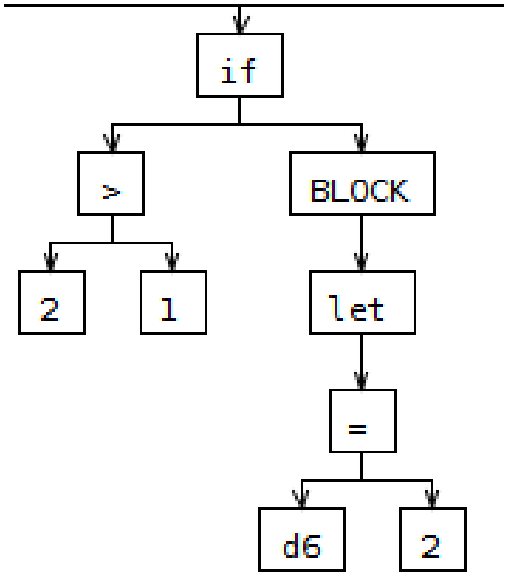
La définition d'une conditionnelle se fait selon la syntaxe suivante :

if (condition) { BLOCK } (else { BLOCK })?

Ce qui donne l'extrait de grammaire ci-dessous.

```
ifExpr : 'if' expr block ('else' block )? -> ^('if' expr block ^('else' block)?);
```

Concernant l'arbre syntaxique, on définit 'if' comme noeud racine de la branche. Le fils gauche est le noeud racine de la branche qui contient la condition, tandis ce que le fils droit est le noeud racine de la branche qui contient le 'BLOCK' d'opérations à réaliser. On obtient par exemple l'arbre :



2.2.8 Boucle "while"

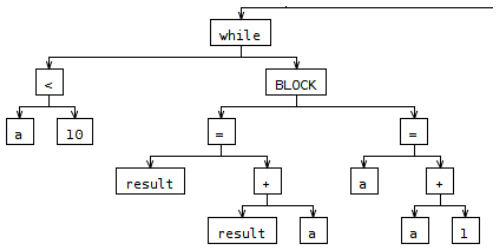
La définition d'une boucle "while" se fait selon la syntaxe suivante :

while (condition) { BLOCK }

Ce qui donne l'extrait de grammaire ci-dessous.

```
| 'while' expr block instruct?-> ^('while' expr block) instruct?
```

Concernant l'arbre syntaxique, on définit 'while' comme noeud racine de la branche. Le fils gauche est le noeud racine de la branche qui contient la condition d'arrêt du "while", tandis ce que le fils droit est le noeud racine de la branche qui contient le 'BLOCK' d'opérations à réaliser dans la boucle. On obtient par exemple l'arbre :



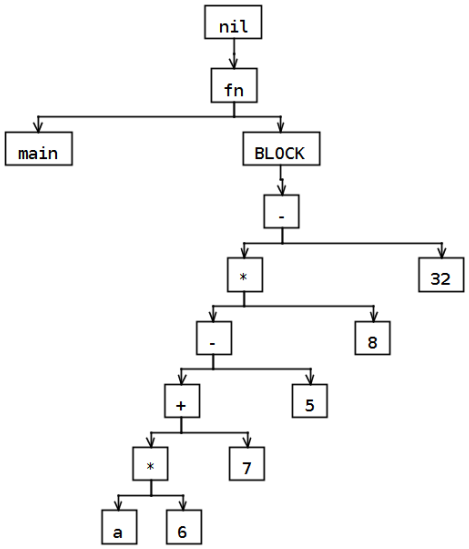
2.2.9 Expression

Les expressions sont définies selon les règles de grammaires suivantes :

```
binExpr1 : binExpr2 (EQUAL^ binExpr2)*;

binExpr2 : binExpr3(ORBOOL^ binExpr3)*;

binExpr3 : binExpr4(ANDBOOL^ binExpr4)*;

binExpr4 : binExpr5((PREV^|OPBOOLEQ^|NEXT^) binExpr5)*;

binExpr5 : binExpr6((ADD^|SUB^)  binExpr6)*;

binExpr6 : unExpr ((STAR^|DIV^) unExpr)*;

vectExpr : starExpr ('['^ expr ']'!)*;

 starExpr
                :               STAR moinsExpr -> ^(UNISTAR moinsExpr)
                | moinsExpr;

 moinsExpr
                :               SUB moinsExpr -> ^(UNISUB moinsExpr)
                | atom;


dotExpr : vectExpr ('.'^ (IDF | 'len' '('!')'! ))?;

unExpr : (UNAIRE^|EPERLU^)? dotExpr;

atom : INT
 | BOOL
 |              IDF^ ((callFun))?
 | block -> ^(ANOBLOCK block)
| '('expr')'-> expr;

expr : 'vec' '!' '[' expr(',' expr)* ']' -> ^('vec' expr*)
 |'print' '!' '(' exS  (',' exS)* ')' -> ^('print' exS*)
|               binExpr1;
```

Il est bon de noter qu'il existe deux branches parallèles d'expressions : expr et bigExpr, la deuxième étant strictement identique à la première à la différence près qu'elle est capable de gérer les instanciations de structures.

On peut par exemple obtenir l'arbre syntaxique :



pour le code :

```
fn main() {
    ((a*6+7)-5)*8-32;
}
```

3 Tables des symboles

Pour réaliser la génération de code à partir de l'AST, un certain nombre de données supplémentaires sont nécessaires, parmi lesquelles on peut citer l'existence de chaque identificateur, leur portée, leur déplacement qui sera à appliquer pour les retrouver en mémoire ...

Ces différentes données sont, pour une question de simplicité, calculées au préalable et insérées dans une structure de données bien particulière que l'on appelle Table des Symboles.

3.1 Architecture de la structure de données

L'interprétation que l'on peut faire de la table des symboles étant assez libre, plusieurs organisations sont possibles pour sa génération et son utilisation.

Nous avons choisi de profiter des particularités du langage Java et de se tourner vers de la Programmation Orientée Objet pour notre implémentation. On dispose alors de deux classes distinctes : Scope et TDS.

La première, tout d'abord, équivaut à une portée. On dispose d'un scope par fonction, par structure, par bloc, anonyme ou non, etc ... Il dispose d'un ancêtre, pour pouvoir remonter au bloc précédent, d'une origine (est-ce une fonction, un bloc if ...) pour pouvoir différencier les traitements en fonction, d'un nom, d'un déplacement initial pour calculer plus aisément le déplacement des variables, et de 3 HashMap qui contiennent soit les informations des sous-scopes (if, bloc anonyme ...) soit celles des variables définies dans le bloc.

Le second est plus simple, et ne dispose que de l'information du premier Scope et de celui en cours.

3.2 Remplissage de la table

Pour remplir la table, un simple parse de l'arbre décoré est effectué.

Pour chaque noeud rencontré, si une nouvelle création de scope doit avoir lieu ("fn", "struct", "if" ...), alors on la réalise. Sinon, on ajoute au scope courant les informations relatives au noeud.

Il est également bon de noter que l'on profite de ce parcours de l'arbre pour effectuer les différents contrôles sémantiques.

3.3 Affichage

Notre architecture de classes utilisant l'imbrication nous permet également de simplifier l'affichage de la TDS.

Si par exemple on reprend le programme défini au chapitre précèdent :

```
fn factorielle(n : i32, result : &i32)
{
  let mut i = 1;
  if n==0
  {
    let b = 5;
    *result = 1;

  } else{
    let gh = 42;
        *result = 1;
    while i <= n
    {
      *result = *result * i;
      i = i+1;
    }
  }
}

fn main(){
  let mut v = 0;
  factorielle(5,&v);
  print!(v);
}
```

On obtient la table des symboles :

```
/usr/local/logiciels/java-1.8/bin/java ...
Scope General
    factorielle : [function, Void]
    main : [function, Void]
    Scope factorielle
        n : [param, i32, 0, false]
        result : [param, &i32, 4, false]
        i : [var, i32, 6, true]
        Scope if1
            b : [var, i32, 10, false]
        Scope else1
            gh : [var, i32, 10, false]
    Scope main
        v : [var, i32, 0, true]

Process finished with exit code 0
```

où à chaque variable est associée un tableau [typeDeVariable, typeDeDonnée, Déplacement, mutable ?].

4 Contrôles sémantiques

Dans l'idée de simplifier la génération du code et pour ne pas avoir à effectuer des vérifications à la volée, un certain nombre de contrôles sémantiques sont effectués. Nous avons également décidé de générer des warnings lorsque le code est accepté mais peut conduire à un résultat inattendu.

4.1 Liste des contrôles

Voici une liste des contrôles :

Type de return correspond à renvoi de fonction
Lors d'une affectation, le membre de gauche est bel et bien une variable définie
Lors d'une affectation, le membre de gauche est bel et bien mutable
Lors d'une affectation, les types des membres de droite et gauche sont les mêmes
Lors d'une déclaration de variable, le nom n'est pas déjà pris par une fonction.
Lors d'une déclaration, si le type est renseigné, il est cohérent avec les données d'initialisation
Lors de la déclaration d'un tableau de tableau, la taille de chaque sous-tableau est identique
Lors de la déclaration d'un tableau, les types de tous les éléments du tableau sont identiques
Lors de la déclaration d'un tableau, les types des éléments correspond au sous-type du tableau
Lors de l'appel d'attribut ("x.a"), la variable x est bien définie
Lors de l'appel d'attribut ("x.a"), l'attribut a existe bien dans la structure de x
Lors de l'appel à un tableau ("t[i]"), l'index i est plus petit que la taille de t
Lors de l'appel à un tableau ("t[i]"), la variable t doit être définie
Lors de l'appel à un tableau ("t[i]"), la variable t doit être de type "vec ?"
Lors de l'appel à un tableau ("t[i]"), la variable i doit être de type i32
Lors de la déclaration d'une variable de structure ("x = data a :5,y :false"), chaque attribut doit être du même type que défini dans la structure
Lors de la déclaration d'une variable de structure ("x = data a :5,y :false"), chaque attribut doit être présent dans la structure
Lors de la déclaration d'une variable de structure ("x = data a :5,y :false"), tous les attributs de la structure doivent être renseignés
L'opération & ne peut s'appliquer qu'à des variables
Lors d'une opération binaire entière (+,-,*,/) , les types des membres de droite et gauche sont les mêmes, et i32
L'opération ! ne s'applique qu'à des booléens
Lors d'une opération binaire booléenne (&&, ||), les types des membres de droite et gauche sont les mêmes, et bool
Lors d'une opération de comparaison d'égalité (==, !=), les types des membres de droite et gauche sont les mêmes, et i32 ou bool
Lors d'une opération de comparaison (>,<,<=,>=), les types des membres de droite et gauche sont les mêmes, et i32
L'opération unaire * ne peut s'appliquer qu'aux adresses
L'opération unaire - ne peut s'appliquer qu'aux i32
Lors de l'appel d'une fonction ("f(a,b)"), il ne doit pas y avoir plus de paramètres que défini par la fonction
Lors de l'appel d'une fonction ("f(a,b)"), chaque paramètres doit être le même ordre et du même type que dans la définition de la fonction
Lors de l'appel d'une fonction ("f(a,b)"), tous les paramètres définis par la fonction doivent être renseignés
Lors de l'appel à une variable, elle doit être définie
Lors de la déclaration d'une fonction, il est impossible de passer des tableaux en paramètres (utilisation de pointeur)
Lors de la déclaration de fonction, le nom de fonction ne doit pas être déjà utilisé
Lors de l'appel à un type "struct", il doit être défini (en tant que struct)
Lors de la déclaration de structure, le type de chaque paramètre est un type existant
Lors de la déclaration de structure, le nom ne doit pas être déjà utilisé
Lors de l'appel à while ou if, la condition doit être un bool

Et on a les warnings suivants :

Prévient lors de la surcharge d'une variable par une autre d'un Scope parent Prévient lors du "shadowing" d'une variable (variable déjà déclarée dans ce scope) Regarde quelles fonctions n'ont pas de "return"

4.2 Exemple

Si l'on prend le code

```
fn puissance(a : i32, b: i32) -> i32
{
  if b == 0
  {
    let c=5;
  }
  let mut i = 1;
  let mut result = a;
  while i < b
  {
    result = result * a;
    i = i+1;
  }
}

fn main(){
  print!(puissance());
}
```

On dispose alors de l'affichage :

```
/usr/local/logiciels/java-1.8/bin/java ...
WARNING ! These functions does not have a proper return : [puissance]
Error : "Not enough parameters for function puissance" at 17:8

Process finished with exit code 6
```

5 Génération de code

5.1 Outils utilisés

La génération de code, développée en JAVA, a nécessité l'utilisation de plusieurs outils, parmi lesquels ANTLR et microPIUP

5.1.1 ANTLR

L'utilisation d'ANTLR se fait de la génération de la grammaire à la définition des types utilisés par le code Java.

5.1.2 MicroPIUP

"Le MicroPIUP a été conçu pour obtenir une visualisation du fonctionnement d'un processeur 16 bits comportant un compteur ordinal noté PC, 16 registres notés de R0 à R15. Le registre R15 est utilisé comme pointeur de pile. Et un registre d'état SR comportant 6 indicateurs notés WF, IF, ZF, ZF, CF, NF de 1 bit chacun. Les 4 derniers indicateurs sont positionnés lors de l'exécution de la plupart des instructions.

Il s'agit d'un simulateur. MicroPIUP possède ses propres programmes d'exécution qui permettent d'observer le comportement du processeur au cours de l'exécution.

Il s'adresse à tous, aussi bien un public averti qu'un novice de l'informatique.

MicroPIUP a été conçu dans le cadre de l'enseignement de Projet de l'IUP GMI première année de l'Université Henri Poincaré de Nancy sous la conduite de K. Proch. Cette version date de 2005."

Nous utilisons ce simulateur comme cible de notre compilateur et donc pour tester le code machine généré.

5.2 Généralités et utilisation des registres

Pour assurer le bon fonctionnement du code généré, un grand nombre de registres sont associés uniquement à une tâche.

Par exemple, le registre R15 est redéfini comme SP, le sommet de piles, R14 est un registre de travail, où l'on stock des données de tous types et R13 est BP, le pointeur d'environnement.

Ces trois registres sont standards mais nous avons également défini nos propres règle. Par exemple, R5 contient le type de la dernière opération réalisée (entière ou booléenne) au cas où un affichage est souhaité. De manière générale, le registre R0 est utilisé pour les calculs, avec le registre R1.

5.3 Routines prédéfinies

Pour simplifier la génération, mais surtout rendre le code généré plus lisible, deux routines ont été créées, respectivement pour évaluer un booléen (run) et pour effectuer un print (print_). Pour revenir d'une routine au code principal, nous utilisons le code assembleur suivant avant de JUMP vers la routine :

```
MPC WR

ADQ 6,WR
```

5.3.1 run

La routine run permet de stocker 1 dans R0. Elle est utilisée comme emplacement de saut lors de l'évaluation d'opérations booléennes. Lors de l'appel à cette routine, le registre WR contient l'adresse où le code reprend après l'évaluation booléenne.

```
run LDW R0,#1
JEA (WR)
```

5.3.2 print_

La routine print_ permet l'affichage de la valeur stockée dans R0, dans le mode d'affichage spécifié par R5. On définit les modes d'affichage de la façon suivante : 0 : entier, 1 : boolean, 2 : string, 3 : CRLF Étant donné que le registre R0 contient la valeur qui nous intéresse, on commence à travailler avec le registre R6. Le registre R4 contiendra lui le signe de la valeur stockée en R0.

L'algorithme est le suivant :

```
empiler(0)
si R5!=0
    si R5!=2
        si R5 != 3
            si R0 = 1
                empiler("TRUE")
            sinon
                empiler("FALSE")
        sinon
            empiler("\n")
    sinon
        afficher(R0)
sinon
    tant que R0>0
        R1 = R1%10
        R0 = R0/10
        empiler(texte(R1))
```

```
R0 = depiler() jusquà' 0
afficher(R0)
```

Sa conversion en langage assembleur est :

```
TRUE string "true"

FALSE string "false"

print_

    STW BP,-(SP)
    LDW BP, SP
    STW R0,-(SP)
    LDW R6,#0
    STB R6,-(SP)
    LDW R4, #0
    CMP R5,R4
    JEQ #ent-$-2
    LDW R4, #2
    CMP R5,R4
    JEQ #str-$-2
    LDW R4, #3
    CMP R5, R4
    JEQ #CRLF-$-2
    CMP R0,R4
    JEQ #false-$-2
    LDW R0,#TRUE
    TRP #WRITE_EXC
    JMP #fin-$-2
    false
    LDW R0,#FALSE
    TRP #WRITE_EXC
    JMP #fin-$-2
    str
    JMP #fin_str-$-2
    CRLF
    LDW R6,#0x000a
    STB R6,-(SP)
    JMP #fin-$-2
    ent
    CMP R0,R4
    JNE #nonzero-$-2
    LDW R6,#0x0030
    STB R6,-(SP)
    JMP #fin-$-2
    nonzero
    CMP R0,R4
    JGE #finsigne-$-2
    LDW R4,#1
    NEG R0,R0
    finsigne
    LDW R7,R0
    boucle
    LDW R0,R7
    LDW R6,#0
    CMP R0,R6
    JEQ #finboucle-$-2
    LDW R6,#10
    DIV R0,R6,R6
```

```
56      STW R6,R7
57      LDW R6,#0x0030
58      ADD R6,R0,R6
59      STB R6,-(SP)
60      JMP #boucle-$-2
61      finboucle
62      LDW R6,#1
63      CMP R4,R6
64      JNE #fin-$-2
65      LDW R0,#0x002d
66      STB R0,-(SP)
67      JMP #fin-$-2
68      fin_str
69      TRP#WRITE_EXC
70      fin
71      LDW R0,SP
72      TRP #WRITE_EXC
73      LDW R0,(SP)+
74      LDW SP,BP
75      LDW BP, (SP)+
76      JEA (WR)
```

5.4 Changement de scope

Au fur et à mesure que l'on avance dans les instructions du programme, il va parfois falloir changer de portée, pour par exemple appeler une fonction, ou simplement pour définir une variable locale dans un bloc if par exemple.

Pour cela, nous avons défini des fonctions JAVA qui automatisent les instructions assembleurs à réaliser pour chaque entrée/sortie d'un scope.

Dans le cas de l'entrée dans un scope, il faut tout d'abord empiler la valeur actuelle de BP, puis calculer le déplacement total du scope pour réserver de la place en mémoire aux variables locales et/ou les paramètres. Enfin, on met la valeur de BP à SP.

Cela donne le code :

```
1 STW BP,-(SP)
2 ADQ -dep,SP
3 LDW BP, SP
```

Pour la sortie, on applique les opérations inverses et on trouve :

```
1 LDW SP, BP
2 ADQ dep,SP
3 LDW BP, (SP)+
```

5.5 Les opérations

Pour réaliser les opérations, nous avons utilisé les registres R0 et R1. Nous avons créé une fonction generateOperation(BaseTree t) qui nous permet d'effectuer toutes les opérations contenues dans

l'arbre t, ou d'en donner la valeur si c'est une des feuilles de l'arbre. Nous avons dans cette fonction séparé toutes les opérations que nous traitons, c'est-à-dire +, -, *, / , > , < , <= , == , >= , != , UNISUB , UNISTAR ,! , & , && , || , avec UNISTAR l'appel à la valeur d'un pointeur et UNISUB l'opposé. Nous avons créé pour chacune de ces opérations une fonction pour générer le code assembleur associé.

Par exemple pour une multiplication le cheminement est le suivant : D'abord nous calculons l'opération à gauche du signe multiplié, avec generateOperation appliqué au fils gauche de notre noeud. Nous mettons la avaleur ainsi obtenu au dessus de la pile :

```
    STW R0, -(SP)
```

Puis nous calculons l'opération à droite du signe multiplié, avec generateOperation appliqué au fils droit de notre noeud. Et enfin, nous récupérons la valeur que nous avons empilée, le résultat de l'expression à gauche de notre signe multiplié et nous faisons notre opération, en prenant soin d'en mettre le résultat dans R0.

```
    LDW R1, (SP)
    MUL R1, R0, R0
```

Nous utilisons le registre R5 pour le print comme précisé précédemment.

```
    LDW R5,#0
```

Nous voyons bien avec cet exemple de code qu'à la fin de la génération d'une opération, le résultat reste stocké dans R0 ce qui justifie que nous utilisions directement R0 après l'appel à generateOperation.

Pour l'exemple :

```
fn main () {
  print !( 3*4 +3 );
}
```

On obtient le code :

```
LDW R0, #3

LDW R5,#0

STW R0, -(SP)

LDW R0, #4

LDW R5,#0

LDW R1, (SP)+

MUL R1, R0, R0

LDW R5,#0

STW R0, -(SP)

LDW R0, #3

LDW R5,#0
```

```
22
23 LDW R1, (SP)+
24
25 ADD R1, R0, R0
26
27 LDW R5,#0
```

Nous avons aussi en plus de toutes ces opérations la fonction generateValue, qui permet lorsque notre arbre est unaire de charger la valeur. Elle apparaît 3 fois dans le code précédent :

```
1 LDW R0, #3
```

```
1 LDW R0, #4
```

```
1 LDW R0, #3
```

C'est avec cette fonction que nous chargerons plus tard les valeurs des variables.

5.6 Affectations des variables

L'affectation des variables est au final assez simple du moment que la TDS est bien construite et que l'on sait calculer une expression.

En effet, on commence par évaluer le fils droit du noeud, puis, on le stocke dans la case mémoire correspondant au déplacement enregistré dans la TDS

5.6.1 Cas d'une constante

De fait, à partir du code :

```
1 fn main(){
2   let a = 52;
3 }
```

On obtient le code assembleur :

```
1 LDW R0, #52
2
3 STW R0, (BP)4
```

5.6.2 Cas d'une opération plus complexe

Et, pour le code :

```
1 fn main(){
2   let a = 52;
3   let b = 15;
4   let c = a+b;
5 }
```

On obtient le code assembleur :

```
LDW R0, #52

STW R0, (BP)12

LDW R0, #15

STW R0, (BP)16

LDW R0, (BP)12

STW R0, -(SP)

LDW R0, (BP)16

LDW R1, (SP)+

ADD R1, R0, R0

STW R0, (BP)20
```

5.7 Conditionnelles

Pour les if, nous effectuons d'abord l'opération qui détermine la condition, que ce soit un simple booléen ou une opération donnant un booléen. Selon le résultat nous effectuons un saut vers les opérations à réaliser ou vers la fin du if. Les changements de scope que nous avons présentés auparavant sont bien gérés à l'entrée et à la sortie, mais nous ne les présentons pas dans l'exemple.

Avec le code suivant

```
fn main () {

   if (1> 2) {
     print!(2);
   }

}
```

On obtient le code suivant :

```
LDW R0, #1

LDW R5,#0

STW R0, -(SP)

LDW R0, #2

LDW R5,#0

LDW R1, (SP)+

MPC WR

ADQ 10,WR

```

```
CMP R1, R0

JGT #run-$-2

LDW R0, #0

LDW R5,#1

LDW R1, #0

CMP R0,R1

JEQ #if1-$-2

STW BP, -(SP)

ADQ -0, SP

LDW BP, SP

LDW R0, #2

LDW R5,#0

MPC WR

ADQ 6,WR

JMP #print_-$-2

LDW SP,BP

ADQ 0, SP

LDW BP,(SP)+

if1
```

5.8 Boucles While

Nous le réalisons basiquement de la même manière que pour le if :

Exemple :

```
fn main () {

  while (1> 2) {
    print!(2);
  }

}
```

On obtient alors le code :

```

STW BP, -(SP)

```

```
ADQ -0, SP

LDW BP, SP

While1

LDW R0, #1

LDW R5,#0

STW R0, -(SP)

LDW R0, #2

LDW R5,#0

LDW R1, (SP)+

MPC WR

ADQ 10,WR

CMP R1, R0

JGT #run-$-2

LDW R0, #0

LDW R5,#1

LDW R1,#0

CMP R0,R1

JEQ #EndWhile1-$-2

LDW R0, #2

LDW R5,#0

MPC WR

ADQ 6,WR

JMP #print_-$-2

JMP #While1-$-2

EndWhile1
```

5.9 Fonctions

5.9.1 Génération

Pour la génération de fonctions nous avons séparé le cas "main" et les autres, mais ils se font globalement de la même manière.

Nous créons un label après lequel nous écrivons le code correspondant à la routine de notre fonction.

Avec le code :

```
fn f(x:i32) -> i32 {
  print!(4);
    return 4;
  }
```

```
f_

STW BP, -(SP)

ADQ -0, SP

LDW BP, SP

LDW R0, #4

LDW R5,#0

MPC WR

ADQ 6,WR

JMP #print_-$-2

LDW R5, #3

MPC WR

ADQ 6,WR

JMP #print_-$-2

LDW R0, #4

LDW SP,BP

ADQ 0, SP

LDW BP,(SP)+

LDW WR, (SP)+

JEA (WR)

LDW SP,BP

ADQ 0, SP
```

```
42
43 LDW BP,(SP)+
44
45 LDW WR, (SP)+
46
47 JEA (WR)
```

5.9.2 Appel

Pour l'appel à la fonction, nous allons utiliser la routine que nous avons créée précédemment. Nous enregistrons nos paramètres dans le haut de la pile, comme si c'était des variables locales puis nous lançons la routine.

Avec l'exemple :

```
1 fn f(x:i32) -> i32 {
2   print!(4);
3     return 4;
4   }
5
6
7 fn main () {
8
9   f(4);
10 }
```

Nous obtenons le code suivant pour l'appel à f :

```
1 LDW R0, #4
2
3 STW R0, -(SP)
4
5 LDW R0, #f_
6
7 MPC WR
8
9 ADQ 6, WR
10
11 STW WR, -(SP)
12
13 JEA (R0)
```

5.10 Pointeurs et adresses

La génération de code des pointeurs et adresses n'était pas encore réalisée lors de l'écriture de ce rapport. Néanmoins, des recherches ont été réalisées.

5.11 Tableaux

La génération de code des tableaux n'était pas encore réalisée lors de l'écriture de ce rapport. Néanmoins, des recherches ont été réalisées.

5.12 Structures

La génération de code des structures n'était pas encore réalisée lors de l'écriture de ce rapport. Néanmoins, des recherches ont été réalisées.

5.13 Print

Étant donnée que nous avons modifié la grammaire pour supporter le formatage dans les print, la génération de code pour print est un peu plus complexe que ce qui devrait être nécessaire.

En effet, pour chacun des paramètres de print, on va appeler la routine avec la valeur de R5 adéquate, avant de finalement effectuer un print("\n") pour inclure artificiellement un saut de ligne.

6 Répartition du travail

	Gautier	Louise	Sacha	Maxime
Grammaire	8h	4h	4h	4h
Contrôles sémantiques et TDS	20h	10h	10h	10h
Génération de code	20h	15h	15h	20h
Rapport	6h	3h	4h	3h
Total	54h	32h	33h	37h

7 Conclusion

Ce projet nous a permis de renforcer les compétences que nous avons acquises en modules de TRAD et de PFSI ainsi que de pouvoir les recouper.
Il nous a paru très intéressant de pouvoir appréhender le projet dans sa globalité et ainsi d'avoir un aperçu du fonctionnement des compilateurs que nous utilisons quotidiennement.
Nous sommes néanmoins un peu déçus de ne pas avoir eu suffisamment de temps pour implémenter les boucles 'for' ainsi qu'ajouter une interface graphique à notre compilateur, les parties non réalisées du sujet étant encore dans les plans pour la soutenance.

Annexes

A Grammaire

La grammaire que nous utilisons est la suivante :

```
grammar Grammar;

options {
  backtrack      = false;
  k      = 1;
  output     = AST;
  ASTLabelType   = CommonTree;
}

tokens {
  BLOCK;
  NEW;
  VEC;
  CALLFUN;
  UNISUB;
  UNISTAR;
  ANOBLOCK;
  RES;
}

@members{
boolean mainFound = false;
}

axiom : fichier EOF {if (!mainFound){System.err.println("main not found");}}
    -> fichier //Ok !
;

fichier : decl* //Ok !
;

decl : declFun //Ok !
| declStruct
;

declStruct : 'struct' IDF '{' args? '}' -> ^('struct' IDF args?) //Ok !
;

args : IDF ':' type (',' IDF ':' type)* -> (^(IDF type))*//Ok !
;

declFun : 'fn' (IDF '(' args? ')' ('->' type)? block -> ^('fn' IDF ^('->' type
    )? args?  block)
  | {mainFound = true;}MAIN '(' ')' block -> ^('fn' MAIN block))
;

```

```
type : | 'i32'
| 'bool'
|IDF
  | 'vec' ('<' type '>') -> ^('vec' type) //Ok !
| '&' type -> ^('&' type)
;

block : '{' instruct'}'-> ^(BLOCK  instruct) //Voir pour le dernier return (si
    expr)
;


callFun : '(' (expr (',' expr)*)? ')' -> ^(CALLFUN expr*);

newStruc : '{' IDF ':' bigExpr (',' IDF ':' bigExpr)* '}' -> ^(NEW ^(IDF
   bigExpr)*);

instrBoucle
  : ';' instruct? -> instruct?
  | -> RES;

instruct :
    expr instrBoucle
| ';' instruct?-> instruct?
| 'let' 'mut'? dotIDF (':' type)? '=' bigExpr ';' instruct?-> ^('let' 'mut'? (
   type)? ^('=' dotIDF bigExpr))  instruct?
| 'while' expr block instruct?-> ^('while' expr block) instruct?
| 'return' expr? ';' instruct?-> ^('return' expr?) instruct?
| 'loop' block instruct?-> ^('loop' block) instruct?
| 'break' ';' instruct?-> 'break' instruct?
| ifExpr instruct?
;

dotIDF  :
IDF ('.'^ IDF)?;

ifExpr : 'if' expr block ('else' block )? -> ^('if' expr block ^('else' block)
   ?);

binExpr1 : binExpr2 (EQUAL^ binExpr2)*;

binExpr2 : binExpr3(ORBOOL^ binExpr3)*;

binExpr3 : binExpr4(ANDBOOL^ binExpr4)*;

binExpr4 : binExpr5((PREV^|OPBOOLEQ^|NEXT^) binExpr5)*;

binExpr5 : binExpr6((ADD^|SUB^)  binExpr6)*;

binExpr6 : unExpr ((STAR^|DIV^) unExpr)*;

vectExpr : starExpr ('['^ expr ']'!)*;

starExpr
  :  STAR moinsExpr -> ^(UNISTAR moinsExpr)
  | moinsExpr;

moinsExpr
  :  SUB moinsExpr -> ^(UNISUB moinsExpr)
  | atom;
```

```
101
102
103 dotExpr : vectExpr ('.'^ (IDF | 'len' '('!')'! ))?;
104
105 unExpr : (UNAIRE^|EPERLU^)? dotExpr;
106
107 atom : INT
108 | BOOL
109 |  IDF^ ((callFun))?
110 | block -> ^(ANOBLOCK block)
111 | '('expr')'-> expr;
112
113 expr : 'vec' '!' '[' expr(',' expr)* ']' -> ^('vec' expr*)
114 |'print' '!' '(' exS  (',' exS)* ')' -> ^('print' exS*)
115 | binExpr1;
116
117 exS : expr
118 | STRING;
119
120 bigbinExpr1 : bigbinExpr2 (EQUAL^ bigbinExpr2)*;
121
122 bigbinExpr2 : bigbinExpr3(ORBOOL^ bigbinExpr3)*;
123
124 bigbinExpr3 : bigbinExpr4(ANDBOOL^ bigbinExpr4)*;
125
126 bigbinExpr4 : bigbinExpr5((PREV^|OPBOOLEQ^|NEXT^) bigbinExpr5)*;
127
128 bigbinExpr5 : bigbinExpr6((ADD|SUB)^ bigbinExpr6)*;
129
130 bigbinExpr6 : bigunExpr ((STAR^|DIV^) bigunExpr)*;
131
132 bigvectExpr : bigstarExpr ('['^ bigExpr ']'!)*;
133
134 bigstarExpr
135   :  STAR bigmoinsExpr -> ^(UNISTAR bigmoinsExpr)
136   | bigmoinsExpr;
137
138 bigmoinsExpr
139   :  SUB bigmoinsExpr -> ^(UNISUB bigmoinsExpr)
140   | bigatom;
141
142
143 bigdotExpr : bigvectExpr ('.'^ (IDF | 'len' '('!')'! ))?;
144
145 bigunExpr : (UNAIRE^|EPERLU^)? bigdotExpr;
146
147 bigExpr
148 : 'vec' '!' '[' expr (',' expr)*']' -> ^('vec' expr*)
149 |' print' '!' '(' exS (',' exS)* ')' -> ^('print' exS*)
150 | bigbinExpr1;
151
152 bigatom : INT
153 | BOOL
154 |  IDF^ (newStruc|callFun)?
155 | block -> ^(ANOBLOCK block)
156 |'('bigExpr')' -> bigExpr;
157
158
159 EQUAL : '=';
160
```

```
161 ORBOOL : '||';
162
163 ANDBOOL : '&&';
164
165 PREV : '<';
166 NEXT : '>';
167
168 OPBOOLEQ : '==' | '!=' | '<=' | '>=';
169
170 ADD : '+';
171
172 STAR : '*';
173
174 DIV : '/';
175
176 UNAIRE : '!';
177
178 EPERLU : '&';
179
180 IF : 'if'
181 ;
182
183 SUB : '-';
184
185 MAIN : 'main'
186 ;
187
188 BOOL : 'true' | 'false'
189 ;
190
191 IDF : ('a'..'z') ('a'..'z' | 'A'..'Z' | '0'..'9' | '_')*
192 ;
193
194
195 INT : '0'..'9'+
196 ;
197
198 WS : ( ' ' | '\t' | '\r' | '\n' ) {$channel=HIDDEN;};
199
200 STRING
201 : '"' ~('\r' | '\n' | '"')* '"'
202 ;
203
204 COMMENT : '/*' ( options {greedy=false;} : . )* '*/' {$channel=HIDDEN;};
205
206 ATTRIBUTE : '#' ( options {greedy=false;} : .)* ('\n' | '\t') {$channel=HIDDEN
    ;};
```