Kilian CORNEC
Léo FILOCHE
Bastien FAISANT
ESIR2 INFO SI

Université de Rennes

TLC - Projet
Compilateur While

Documentation utilisateur & Rapport de projet

ECOLE SUPERIEURE
D'INGENIEURS DE RENNES

le 22 janvier 2023

Table des matières

1. Documentation utilisateur du compilateur et du langage **2**
1.1. Documentation du langage WHILE 2
1.2. Documentation du compilateur 5

2. Rapport de projet **5**
2.1. Description technique 5
2.1.1. Architecture du compilateur 5
2.1.2. L'arbre de dérivation syntaxique et l'AST 8
2.1.3. Table des symboles 8
2.1.4. Code 3 addresses 11
2.1.5. Génération de code 14
2.1.6. Bibliothèque runtime 19
2.2. Validation du compilateur 19
2.3. Méthodologie de gestion de projet 19
2.4. Postmortem 20

1. Documentation utilisateur du compilateur et du langage

1.1. Documentation du langage WHILE

- Construction d'un arbre binaire

Le langage WHILE utilise un unique type de données, ce sont les arbres binaires. Sa construction se réalise à l'aide du mot clé **cons**. Cette expression construit un arbre. Chaque nœud de l'arbre possède 2 sous-arbres, initialisés à **nil** par défaut. L'expression **nil** construit un arbre vide ; c'est-à-dire un arbre où la racine est une feuille.
Un arbre peut être construit à partir de 2 arbres enfants préalablement générés en les incluant à la suite : **(cons A B)**. Dans cet exemple, A est l'arbre enfant gauche et B est l'arbre enfant droit. Voici quelques exemples de construction d'arbre :

- **nil** : Construction d'un arbre vide
- **(cons nil nil)** : Construction d'un arbre avec 2 enfants vides
- **(cons A)** : retourne l'arbre A

Le langage WHILE permet aussi de créer un arbre binaire à partir d'au moins 2 sous-arbres.

- **(cons A B)** : Construction d'un arbre ayant pour enfant gauche l'arbre A et pour enfant droit l'arbre B
- **(cons A B C) = (cons A (cons B C))** : Cette expression construit un arbre composé de A comme enfant gauche et d'un arbre comme enfant droit. Ce dernier construit un arbre avec B et C comme enfant.

- Les types

Ensuite, le langage WHILE n'ayant qu'un seul type de données, les types primitifs sont déclarés à partir d'un arbre binaire :

- Les **entiers naturels** correspondent à la hauteur[1] de l'arbre binaire en ne regardant que la branche droite. Si un arbre est vide alors sa représentation en entier est 0.
  - **nil** vaut 0
  - **(cons nil nil)** vaut 1
  - **(cons nil (cons nil nil))** vaut 2
- Les **booléens** sont aussi simulés avec la hauteur de l'arbre. En effet, si l'arbre est vide ou comporte seulement la racine alors sa valeur est **false** sinon elle est à **true**.
  - **nil** vaut false
  - **(cons nil nil)** vaut true
- Les **chaînes de caractères** sont simulées grâce à la concaténation de toutes les feuilles de l'arbre binaire. Chaque feuille comporte un unique caractère. De plus, un arbre vide correspond à une chaîne de caractères vide.
  - **nil** vaut “”
  - **(cons a b)** vaut “ab”
  - **(cons (cons a b) (cons c d))** vaut “abcd”

D'autres fonctionnalités sont aussi possibles en WHILE.

- Construction d'une liste

Tout d'abord, nous pouvons créer une liste avec l'expression **list**. Une liste vide est équivalente à un arbre binaire vide : **nil**.

[1] hauteur d'un arbre binaire : nombre de nœuds qui constituent la branche contenant le plus de nœuds sans compter la racine

- **(list A)** = **(cons A nil)** : liste contenant 1 élément **A**
- **(list A B … Z)** = **(cons A (cons B … (cons Z nil))** : liste contenant n éléments

- Récupération du premier et dernier élément

Les mots clés **hd** (head) et **tl** (tail) permettent de récupérer respectivement le premier et dernier élément de l'arbre binaire. Si l'arbre est vide ou sa racine est une feuille alors la valeur retournée par **hd** et **tl** est **nil**.

- **(hd (cons A B C))** retourne l'arbre A
- **(tl (cons A B C))** retourne l'arbre C

- Variable

Les variables sont initialisées par défaut à **nil**. Il n'existe pas de variable globale, elles sont toutes locales à une fonction en particulier.
De plus, la déclaration de variables multiples est possible en séparant chaque variable et valeur par une virgule :

```
A, B, C = (cons a), (cons b), (cons c);
```

Dans cet exemple : **A = "a"**, **B="b"** et **C ="c"**

- Structure de contrôle

Comme pour la majorité des langages, WHILE possède les conditions et boucles principales :

- Condition if commençant par le mot clé **if** et terminant par **fi**

```
if E then C1 else C2 fi
```

La variable **E** est interprétée comme un booléen par le langage. Si la condition est validée alors **C1** est exécutée sinon la commande **C2** est exécutée. La partie **else C2** est optionnelle.

- Boucle while commençant par le mot clé **while** et terminant par **od**

```
while E do C od
```

La variable **E** est interprétée comme un booléen par le langage. Tant que le booléen est vrai (= 1), la commande **C** est exécutée

- Boucle for commençant par le mot clé **for** et terminant par **od**

```
for E do C od
```

La variable **E** est interprétée comme un entier par le langage. La commande **C** est exécutée **E** fois.

- Définition de fonction

Une fonction comprend un nom (**function1**), des éventuels paramètres **(p1,..., pN)** et au moins une valeur de retour **(r1,..., rN)**. Sa syntaxe est la suivante :

```
function function1 :
read p1, …, pN
%
  …
%
```

```
write r1, ..., rN
```

- Appel de fonction

Le langage WHILE est fonctionnel et permet donc d'appeler des fonctions depuis une autre fonction. Pour cela, il suffit d'indiquer le nom de la fonction suivi de ses éventuels paramètres. Par exemple :

- **(function1)** : appel de la fonction **function1** ne possédant pas de paramètre
- **(function2 p1)** : appel de la fonction **function1** possédant 1 unique paramètre
- **(function3 p1 … pN)** : appel de la fonction **function3** possédant exactement n paramètres.

De plus, les fonctions peuvent avoir 1 ou plusieurs valeurs de retour, il est donc possible d'assigner une variable à chaque valeur retournée. Par exemple, si **function4** retourne 2 valeurs, la récupération de ces 2 valeurs se réalise de la manière suivante :

```
A, B = (function4)
```

- Fonction principale

La fonction exécutée par défaut est la fonction **main**. Celle-ci appelle donc les éventuelles autres fonctions du programme.

- Paramètres d'entrée
- Valeurs retournées

La ou les valeurs de retour sont affichées selon leur type (entier, booléen, chaîne de caractère). Si la valeur retournée est un arbre vide alors **nil** est affiché.

- Exemple simple de programme WHILE

Voici un programme créant 3 variables A, B, C :

- A produit l'entier 3
- B appelle la fonction **false** produisant un booléen
- C produit la chaîne de caractère "hello"

Ensuite, ces 3 variables sont affichées en sortie.

```
function false:
read
%
    Result := nil
%
write Result

function main:
read
%
    A := (cons nil (cons nil (cons nil nil)))
    B := (false)
    C := (cons h e l l o)
%
write A, B, C
```

1.2. Documentation du compilateur

Le compilateur est composé en 4 packages principaux :

- **Antlr** : C'est un package qui est composé des lexeurs et du parseurs donné par ANTLR.

- **SymbolTable** : Dans ce package on retrouve notamment la classe ParserAST dans laquelle il y a la méthode *getSymbolTable()* qui retourne la table des symboles.

- **ThreeAddressCode** : Ce package contient une classe du même nom qui elle même contient la méthode *generate()* qui renvoie le code 3 adresses.

- **ThreeAddressToCpp** : Enfin ce package contient la classe du même nom qui contient la méthode *writeCpp()* qui prend du code 3 adresses en entré et retourne du code C++

Le Main du projet quant à lui exécute toutes les étapes précédemment décrites pour compiler le code While en code un fichier exécutable.

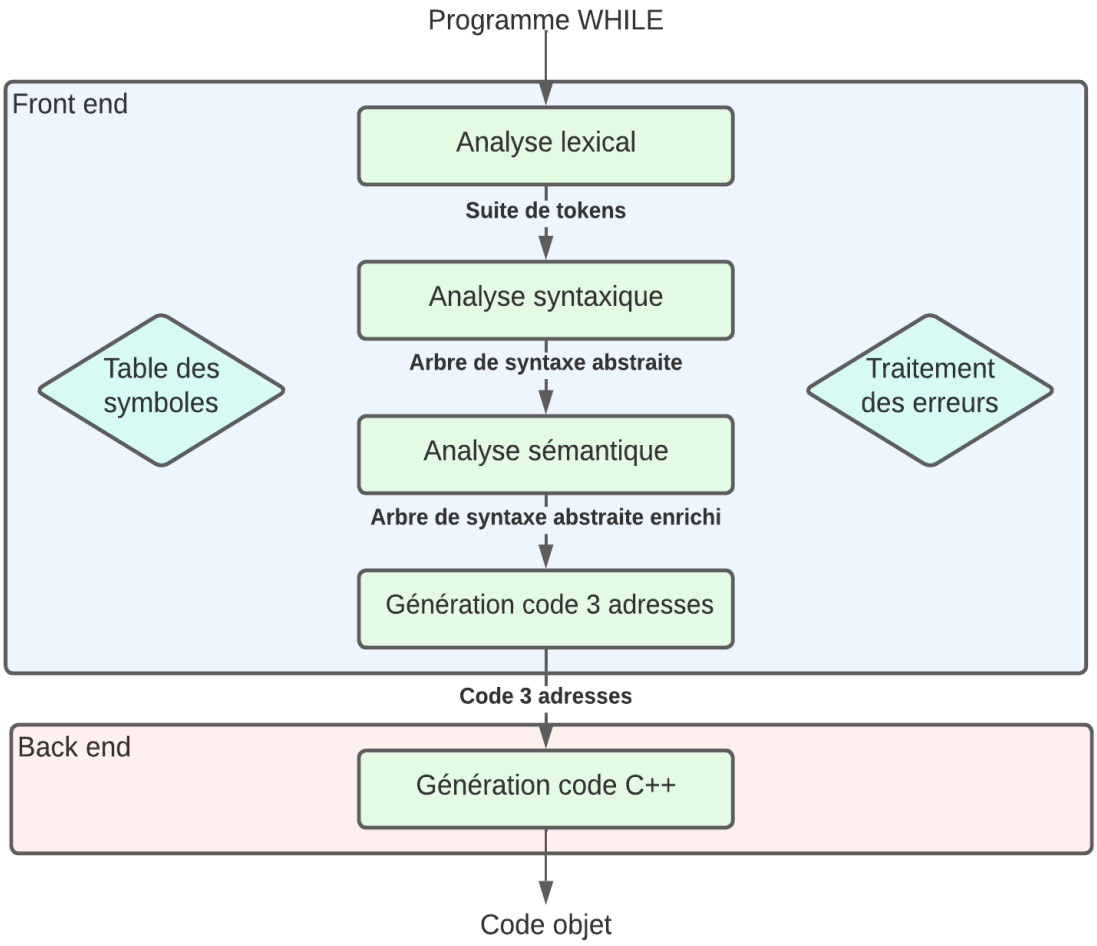
Enfin le dossier *export* il y a un script bash, qui à l'aide du runtime, prend un nom de fichier .while en entrée et exécute le programme. *Pour plus de détails voire README.md.*

2. Rapport de projet

2.1. Description technique

2.1.1. Architecture du compilateur

L'objectif du projet est de concevoir un compilateur pour le langage WHILE. Pour cela, l'architecture de notre compilateur reprend celle vue en cours. Nous devons donc produire un programme C++ (le langage cible de notre projet) à partir d'un programme WHILE (le langage source).



A partir de la documentation du langage WHILE, une analyse lexicale peut être réalisée. En effet, plusieurs séquences de caractères peuvent être identifiées afin de former des tokens :

- les **variables** commençant par une majuscule
- les **symboles** commençant par une minuscule. Ces symboles peuvent être des noms de fonctions ou encore des chaînes de caractères
- **'fonction'** est un mot-clé indiquant le début de déclaration d'une fonction et les deux points '**:**' marque la fin de sa déclaration
- **'read'** est un mot-clé pour récupérer les éventuelles paramètre de la fonction
- **'write'** est un mot-clé pour retourner une valeur en sortie de fonction
- '**%**' délimite le corps des fonctions
- '**:=**' est un symbole d'affectation
- '**if**' est utilisé pour débuter une condition, '**then**' est un mot-clé pour marquer le début du corps de la condition si elle est vraie, '**else**' marque le début du corps si la condition est fausse et '**fi**' marque la fin de la condition
- '**while**' et '**for**' sont des mots-clés pour débuter une boucle, '**do**' marque le début de la boucle et '**od**' indique la fin de la boucle
- '**foreach**' est aussi un mot-clé pour débuter une boucle et '**in**' permet de récupérer chaque élément d'une expression
- '**nil**' est un mot-clé pour représenter un arbre binaire vide
- '**(**' et '**)**' délimitent les expressions
- '**cons**' est un mot-clé permettant de construire un arbre binaire

- '**list**' est un mot-clé permettant de construire une liste d'arbre binaire
- '**hd**' permet de récupérer la partie gauche d'un arbre et '**tl**' d'en récupérer sa partie droite

Ensuite, l'**analyse syntaxique** permet d'identifier la structure du langage WHILE. Pour cela, nous avons transformé la grammaire du langage WHILE fournie en grammaire **EBNF** et **LL** (supporte les grammaires récursives à gauche). Le compilateur de compilateurs ANTLR reconnaît donc ce langage et produit un code pour l'analyseur lexical (lexer) et l'analyseur syntaxique (parser). A l'aide de cette grammaire, un arbre syntaxique est produit. Cet arbre, bien qu'il reconnaisse tout code dans le langage WHILE, possède trop données superflues comme les mots-clés spécifiques à ce langage. Nous devons donc l'améliorer afin qu'il soit plus compréhensible pour tout langage. C'est à ce moment qu'intervient la construction d'un **arbre de syntaxe abstraite (AST)** que nous détaillerons plus en détail dans la prochaine partie.

Après avoir réalisé une première ébauche de notre arbre de syntaxe abstraite, celui-ci peut être amélioré grâce à l'**analyse sémantique**. En effet, cette étape va nous permettre de rejeter des programmes dans le langage WHILE qui sont incorrects. Pour cela, nous avons mis en place une **table des symboles** répertoriant toutes les variables déclarées ainsi que tous les noms de fonctions implémentées. Notre table des symboles correspond à un arbre de tables de correspondances où chaque variable est locale à son contexte. Des vérifications sémantiques peuvent donc être effectuées.
De plus, tout programme ne respectant pas au moins un de ces vérifications est rejeté par notre compilateur avec une erreur ainsi qu'un message précisant les raisons de cette erreur.
Ce traitement des erreurs nous a permis d'enrichir l'arbre de syntaxe abstraite afin de bien respecter les spécifications du langage WHILE.

Ensuite, l'arbre de syntaxe abstraite permet d'en **générer un code intermédiaire** indépendant de notre langage cible C++. Le code intermédiaire choisi est le code 3 adresses. Celui-ci permet de retranscrire l'AST en ligne de code proche de celui de la machine. En effet, la mise en place de ce code intermédiaire nous amène à utiliser les registres et les labels. Le détail sur la génération du code 3 adresses sera présenté dans une partie dédiée.

Le code 3 adresses présente l'inconvénient d'utiliser beaucoup de registres. Pour cela, une **optimisation de code** peut être mise en place afin de diminuer notamment l'occupation de la mémoire en éliminant les expressions communes, le code mort ou encore les propagations de copie. Cependant, cette phase n'a pas pu être mise en place dans notre compilateur.

Enfin, la dernière partie concerne la **génération du code cible C++** à partir du code intermédiaire. Cette transformation de code se retrouve dans la partie backend du compilateur car celle-ci est indépendante du langage WHILE. Le programme peut donc être directement compilé, à l'aide du compilateur C++, en préservant la sémantique du code source WHILE.

2.1.2. L'arbre de dérivation syntaxique et l'AST

La génération de l'arbre de dérivation syntaxique s'est faite en recopiant et modifiant la syntaxe de la grammaire (qui nous était fournie) dans ANTLRWorks.

Ensuite la génération de l'AST s'est effectuée également avec ANTLRWorks juste après l'écriture de la grammaire. En effet l'écriture dans ANTLRWorks de la grammaire nous a permis d'avoir un premier arbre (arbre de dérivation syntaxique) mais bien trop complet, avec beaucoup trop d'informations superflues. Par exemple les symboles comme "%", "," ou encore "(" ne seront pas utiles pour la suite (table des symboles et génération du code 3 adresses).
Pour simplifier cet arbre nous avons utilisé des règles de réécriture. Par exemple à la place de :

```
'read' input '%' commands '%' 'write' output
```

Nous avons écrit :

```
^(BODY input? commands output)
```

Dans ce code (qui correspond à la déclaration d'une fonction) nous y avons retiré les "%", le "read" et "write". Aussi nous avons rajouté un lexème imaginaire "**BODY**". Ces lexèmes imaginaires sont très pratiques pour nommer un nœud et indiquer ce qu'il s'y passe quand on descend dans l'arbre à partir de ce nœud. C'est donc pratique et cela permet aussi de rendre l'AST bien plus lisible.

Il est important de noter que cet AST ne s'est pas fait en une fois. Nous sommes, à plusieurs reprises, retournés dans ces règles de réécriture pour modifier des choses car elles posaient problèmes pour la table des symboles. En effet, il n'y pas une seule bonne façon de faire la table des symboles. Le plus important est qu'elle permet de bien réaliser les étapes suivantes.

Une dernière chose qui n'est pas directement liée à la table des symboles mais qui a été effectuée à ce même moment est la gestion des espaces, des sauts de ligne et des commentaires. Ces éléments ont pu être gérés directement à cet endroit en les exprimant au travers de leur expression régulière et en les envoyant sur le channel *HIDDEN.* Comme cela ANTLR après les avoir détectés va les ignorer. De cette façon, dans l'AST il n'y a pas les commentaires.

2.1.3. Table des symboles

Une fois que l'AST a été généré, la table des symboles a pu être créée. Mais il a d'abord fallu générer le code de l'analyseur lexical et syntaxique depuis ANTLWorks. Une fois cela fait nous avons créé un environnement de travail Java d'où nous avons commencé à créer la table des symboles.

A ce moment nous pouvions manipuler cet AST à l'aide de la classe *CommonTree.* Un *CommonTree* représente un nœud avec son information et ses enfants qui sont eux-mêmes des *CommonTree.*

La première chose que nous avons fait est de créer deux classes :

- Symbol : Représente un symbole avec son nom et sa valeur. La valeur est un type générique T.

- SymbolTable : Comme son nom l'indique elle représente la table des symboles. Nous avons choisi la représentation spaghetti stacks. Un objet SymbolTable possède donc un père et des enfants. Il y aussi une liste de symboles pour garder en mémoire les symboles du bloc parcouru. Et enfin il y a un dictionnaire avec les noms des fonctions (les symboles des fonctions) et leur nombre d'arguments.

Aussi il faut noter qu'en While il n'y a qu'un seul bloc pour les symboles. En effet, une variable déclarée à l'intérieur d'un *for* (par exemple) sera visible à l'extérieur de celui-ci. Le seul bloc qui existe est celui créé par les fonctions. Une variable déclarée dans une fonction ne sera pas accessible à l'extérieur de celle-ci. La table des symboles aura donc une racine et des fils qui eux n'auront aucun fils.

Passons maintenant à la génération de table en elle-même. La première chose que nous avons réalisée est de parcourir de manière récursive l'AST. Puis il a fallu voir à quels moments ce parcours passait par des nœuds qui nous intéressaient pour effectuer le traitement adéquat. Faisons une liste des traitement dans chaque cas :

- *FONCTION_DEF* : C'est donc la déclaration d'une fonction. On :
    - Vérifie que la fonction n'est pas déjà déclarée. Si ce n'est pas le cas on lève une exception
    - Ajoute la fonction à table des symboles
    - Crée un fils pour dire qu'on accède à un nouveau bloc
    - Visite la fonction

- *INPUTSUB* : Ce sont les arguments d'entrée. On ajoute tous ces arguments à la table des symboles.

- *OUTPUT* : Ce sont les variables de sortie. On vérifie que les variables de sortie appartiennent à la table des symboles. Sinon une exception est levée.

- *nop* : On fait rien

- *VARIABLE_DEF* : Permet de déclarer une variable. On :
    - Vérifie qu'il y a un nombre strictement supérieur ou égal de noms de variables que la valeur. Sinon une exception est levée.
    - Ajoute les variables à la table des symboles

Il faut développer ici un point important qui est celui de l'évaluation de la valeur du symbole. Pour ce faire, un autre parcours récursif a été mis en place mais celui-là n'opère que sur les

“r-values” et renvoie la valeur à la fin du parcours. Cette valeur est le plus souvent *null* lorsque cela dépend d’une autre variable. Sinon on a *nil* ou une chaîne de caractères. Enfin ces “r-values” sont aussi traitées dans le parcours principal car pouvant se retrouver par exemple en argument dans un for ou encore d’une fonction. Il y a donc deux traitements différents bien que très proches. Nous allons ici détailler le fonctionnement moyen des deux mais il est possible de retrouver le fonctionnement détaillé dans le code (qui est commenté).

- *FONCTION* : C’est l’utilisation d’une fonction. On :
    - Vérifie que la fonction est déclarée. Sinon une exception est levée
    - Vérifie le nombre d'arguments. Sinon une exception est levée
    - Vérifie que les arguments sont déclarés. Si ça n’est pas le cas, une exception est levée

- *VARIABLE* : C’est l’utilisation d’une variable. On :
    - Vérifie qu’elle est déclarée. Sinon la valeur est mise à 0 et un warning est affiché

- *CONS* : C’est l’utilisation d’un *cons.* On :
    - Vérifie s’il y a des arguments. Sinon renvoie 0.
    - Vérifie que les arguments sont des “r-values”

- *TL* : C’est l’utilisation d’un *tl.* On :
    - Vérifie qu’il y a bien un argument. Sinon une exception est levée
    - Vérifie que l’argument est une “r-values”

- *HD* : Idem que *TL.*

- *LIST :* C’est l’utilisation d’une *list.* On :
    - Vérifie s’il y a des arguments. Sinon renvoie 0.
    - Vérifie que les arguments sont des “r-values”

On a précédemment évoqué le soulèvement d’exception lorsqu'une propriété n’était pas respectée. L’exception en question est CompilationException et c’est une classe qui étend Exception.

De manière générale quand une variable est utilisée sans être déclarée ça valeur est par défaut *nil* et un warning l’indique.

Après toutes ces vérifications, il restait encore un type d’erreur qui n’était pas soulevé : Les erreurs de syntaxe pure qu’un utilisateur pourrait faire. Par exemple écrire la ligne suivante :
A -> 5
Pour lever une exception dans ce genre de cas, nous avons modifié le code du Parser pour lever une exception lorsque c’est le cas.

Aussi la ligne de l’erreur est marquée dans les exceptions grâce à la méthode *getLine()*.

Enfin pour terminer cette partie il est à noter que la table des symboles n’est pas utilisée pour générer le code 3 adresses. Cependant sa création reste très importante car elle

permet de relever, normalement, toutes les erreurs possibles et de soulever une exception. De fait, sa création réalise bien l'analyse sémantique.

2.1.4. Code 3 addresses

Une fois l'arbre de syntaxe abstraite réalisé, le code 3 adresses peut être généré. Ce code intermédiaire comporte la spécification d'avoir, pour chaque ligne de code, **1 cible**, **au plus 2 sources** et **1 opérateur**. Cette spécification entraîne l'utilisation de nombreux registres uniques afin de décrire une expression "complexe".

Le processus de génération de code 3 adresses que nous avons mis en place est le suivant. Chaque nœud de l'AST est parcouru en partant de la racine à ses feuilles. Pour cela, une fonction principale récursive a été implémentée. A chaque fois qu'une fonction est définie (token `FONCTION_DEF`), on le transcrit en code 3 adresses : "**`func begin`**" suivi du nom de la fonction. Ensuite, les éventuels paramètres (token `INPUTSUB`) sont précédés par le code "**`param`**" les plaçant dans la pile. Les valeurs de sorties de fonction (token `OUTPUT`) se font à l'aide du terme "**`return`**" et "**`Return`**" permet le retour à la fonction appelante. Le corps des fonctions sont aussi traités où nous retrouvons les différentes conditions et boucles.

Tout d'abord, les conditions gardent la même syntaxe en code 3 adresses. En effet, l'expression de condition est récupérée et déclarée dans un registre, puis nous utilisons une instruction de saut afin de vérifier si la condition vaut `0` (`= false`) à l'aide du terme "**`ifz`**". Si la condition est valide, un saut de ligne est effectué avec le mot-clé "**`goto`**" suivi d'un label.

```
R3 = MyCondition
ifz R3 goto L2
R4 = v
Result = R4
goto L1
L2:
R5 = f
Result = R5
L1:
```

Quant aux boucles, le code 3 adresses ne permet pas d'utiliser directement les mots-clés "**for**" et "**while**", il faut donc retranscrire ces boucles en instructions conditionnelles.

Boucle **`for`** ajoutant 1 au résultat suivant la valeur de `Expression` dans `R4`

```
R4 = Expression
L1:
ifz R4 goto L2
R6 = 0
R5 = 1 + R6
Result = R5
R4 = R4 - 1
goto L1
L2:
```

Boucle **`while`** ajoutant 1 au résultat suivant la valeur de `MyCondition` dans `R4`

```
L1:
R4 = MyCondition
ifz R4 goto L2
R6 = 0
R5 = 1 + R6
Result = R5
R7 = 0
MyCondition = R7
goto L1
L2:
```

Ensuite, lorsqu'une variable est initialisée (token `VARIABLE_DEF`), celle-ci est enregistrée en mémoire dans un table de hachage liant les noms des variables (`String`) avec leur arbre binaire respectif (`CommonTree`). Les arbres binaires en mémoire sont complets et ne comportent aucune variable afin d'éviter les problèmes de cycles. De plus, on s'assure de la bonne structure de chaque arbre binaire en transformant si nécessaire chaque arbre afin qu'il ne possède pas plus de 2 enfants. Dans la mémoire, nous ne trouvons plus aucun nœud de type : `VARIABLE`, `HD`, `TL` mais ils sont remplacés par leur équivalent avec le nœud **`CONS`**.
Afin de savoir si un arbre binaire est simulé comme un entier, un booléen ou encore un chaîne de caractères, une fonction a été implémentée. En effet, un arbre binaire comportant un nœud **`STRING`** est correspond à une chaîne de caractère. Quant aux autres arbres binaires ce sont des entiers (le type booléen valant `false` pour l'entier `0` et `true` pour l'entier `1`). Voici un exemple d'entier représentant la valeur `3` et de chaîne de caractères représentant la valeur "`hello`".

```
Trois := (cons nil (cons nil (cons nil nil)))

R4 = 0
R3 = 1 + R4
R2 = 1 + R3
R1 = 1 + R2
Trois = R1
```

```
Hello := (cons h e l l o)

R2 = h
R4 = e
R6 = l
R8 = l
R9 = o
R7 = R8 + R9
R5 = R6 + R7
R3 = R4 + R5
R1 = R2 + R3
Hello = R1
```

Quant aux appels de fonction (token `FUNCTION`), le mot-clé "**call**" est utilisé suivi du nombre de paramètres de la fonction appelée.

```
param A
param B
R10 = call add 2
Result = R10
```

Dans cet exemple, nous avons 2 variables initialisées au préalable. Nous ajoutons ces 2 variables à la pile à l'aide du mot-clé "**`param`**" puis nous appelons une fonction "**`add`**" en récupérant les 2 éléments en sommet de la pile. La valeur de retour de la fonction est ensuite enregistrée dans le registre `R10` puis dans la variable `Result`.

Durant le processus de génération du code 3 adresses, certaines difficultés ont été mises au jour. En effet, l'implémentation des listes a été effectuée. Cependant, la boucle **`foreach`** n'a pas été mise en place rendant les listes nettement moins utiles au langage.

A partir d'un exemple de programme WHILE réalisant l'addition de 2 entiers égaux à 2 , le code 3 adresses génère le programme suivant :

```
function add :
read Op1, Op2
%
Result := Op1
for Op2 do
    Result := ( cons nil Result )
od
%
write Result

function main:
read
%
    Deux := (cons nil nil nil)
    Result := (add Deux Deux)
%
write Result
```

```
func begin add
param Op1
param Op2
R1 = Op1
Result = R1
R2 = Op2
L1:
ifz R2 goto L2
R4 = Result
R3 = 1 + R4
Result = R3
R2 = R2 - 1
goto L1
L2:
return Result
Return
func end
func begin main
R7 = 0
R6 = 1 + R7
R5 = 1 + R6
Deux = R5
param Deux
param Deux
R8 = call add 2
Result = R8
return Result
Return
func end
```

2.1.5. Génération de code

Une fois le code 3A généré, il s'agit alors de le convertir en un langage permettant d'exécuter concrètement ces instructions. Pour rendre cela possible, nous avons créé ce traducteur sous la forme d'une classe java (**ThreeAdressToCppGenerator**), au comportement similaire à la classe (**ThreeAdressCode**), mais prenant cette fois-ci en entrée du code 3A pour le transformer (en conservant bien évidemment la sémantique) en un code écrit en C++ exécutable sur la machine de l'utilisateur.

Le choix du langage a été sujet à réflexion, et il s'est principalement sur le fait que nous souhaitions un langage possédant la commande **goto** (présente dans le 3A), car c'est une instruction dont l'équivalence dans les langages modernes n'est pas si évidente (il faudrait construire un système de if/else qui peut être complexe). Après recherches, il s'est avéré que le C++ contient cette commande. En plus de cela, il s'agit d'un langage que nous connaissons ; de ce fait, les erreurs produites lors du développement sont plus facilement interprétables. En outre, le C++ est un langage populaire (donc exécutable sur un grand nombre de machines), et connu pour sa rapidité d'exécution (ce qui est intéressant si on souhaite que le while soit performant).
C'est pour toutes ces raisons que nous avons opté pour ce langage.

Pour le bon fonctionnement du générateur, nous avons décidé de créer un "Lecteur" de code (classe java **InstructionReader**), lisant les lignes du code intermédiaire une à une, traduisant partiellement et stockant des informations essentielles, pour ensuite rendre une traduction complète vers le langage cible. Ces phases de traduction partielle et de mémorisation sont dûes au fait qu'une ligne en 3A n'équivaut pas forcément à une ligne en C++.

```
// Exemple de fonction en 3A
func begin somme //nom de la fonction,
param Op1 // premier argument
param Op2 // second argument
Result = Op1 + Op2 // Corps de la fonction
return Result // Retour
Return
func end // Fin de la fonction
```

```
// Exemple de fonction en C++
int somme(int Op1,int Op2){ // type de retour, nom de la fonction, arguments
int Result = Op1 + Op2; // Corps de la fonction
return Result; // Retour
}// Fin de fonction
```

Comparaison du code d'une fonction en 3A avec celui écrit en C++.

```
// Premières étapes de traduction ligne à ligne du 3A au C++
func begin somme --> int somme(
```

```
param Op1         --> int somme(int Op1
param Op2         --> int somme(int Op1, int Op2)
```

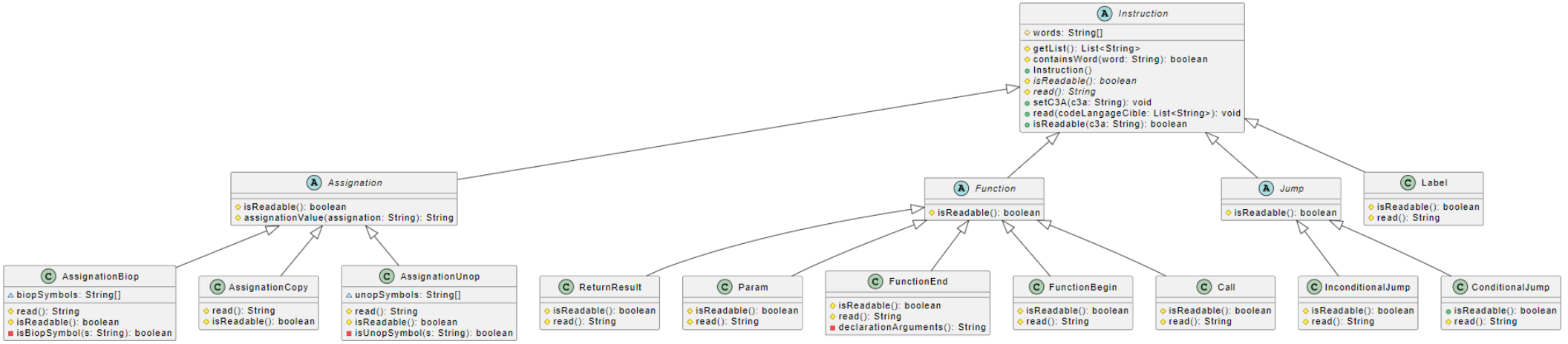
Illustration des première étapes de traduction à effectuer et de la non-équivalence d'une ligne de 3A avec une ligne de C++ montrant la nécessité de l'utilisation d'une mémoire tampon avant d'écrire une ligne définitive vers le code cible

Comme vous l'aurez noté, ce générateur est séparé en deux classes. Nous avons fait cela dans une optique de séparation des préoccupations :

- **ThreeAdressToCppGenerator :** c'est la classe que l'on invoque pour créer le générateur. Elle a pour rôle d'interagir avec le monde extérieur (récupérer le code 3A et renvoyer/écrire (*generateCpp*/*writeCpp*) le code généré après avoir lu chaque ligne avec le lecteur).

- **InstructionReader :** C'est le lecteur, à qui l'on fait lire les lignes une à une (*read*). Cette dernière classe est composée de classes internes, dont la structure est inspirée du patron de conception "Commande". Chaque classe interne est de type **Instruction** : c'est une instruction typique du 3A. Elle est dite "lisible" (*readable*) si le code de la ligne lue en 3A correspond à son motif. Si c'est le cas, on peut enclencher le mécanisme de lecture de la ligne (*read*) pour commencer à traduire la ligne en C++ (traduction partielle) et stocker les informations nécessaires pour la traduction finale.
  Pour réduire au maximum la duplication de code, nous avons regroupé les instructions par nature (Assignation, Saut, Fonction, et Label).

Voir l'image en meilleure qualité



Organisation des différentes classes d'instruction permettant l'identification d'une instruction 3A ainsi que sa traduction vers le C++

Comme dit précédemment, on passe par une phase de “traduction partielle”. Dans le cadre de notre générateur, nous avons fait le choix de finaliser la traduction de la fonction (déclaration des variables, indication du type de la méthode) dès que l’instruction “function end” est détectée.
Il faut avouer que cette phase de typage s’est avérée particulièrement difficile. En effet, le trois adresses ne déclare pas explicitement les types des variables et des fonctions, contrairement au C++. Il a donc fallu mettre en place un système de détection de type.
Pour cela, nous avons déclaré 3 tables d’association[2] (variable, type) : une pour les variables qui sont déclarées dans la fonction (`local_variables`), une pour les arguments de la fonction (`param_variables`), et une pour les fonctions (`known_methods`).
Dès qu’une variable est utilisée pour une assignation, on l’ajoute dans la table, en indiquant : soit le type associé si celui-ci est explicite (b = 3, permet d’en déduire que b est un entier), ou alors la variable associée si le type n’est pas encore déduit (b = a, permet d’en déduire que b est du même type que a si a est une variable connue, sinon on l’interprète comme une chaîne de caractères “a”, et b prend de fait le type `string`). C’est lors de la phase de finalisation que l’on déduira les types finaux (*getType*).
Le type de la fonction est celui de la variable qui est retournée par l’instruction **return**.

Si vous regardez les fonctions C++ qui sont générées, vous observerez que les déclarations de variables sont toutes faites au début de la fonction.

```
int add_[3](int Op1, int Op2){
    int R1;
    int Result;
    int R2;
    int R4;
    int R3;
   //...
```

Exemple du des déclarations en début de fonction dans le code cible C++

Cela est dû au **goto**. En effet, le goto en C++ est plus subtil que le le goto du code 3A : il n’est pas possible de déclarer une variable entre deux labels, sous peine de générer un problème de croisement d’initialisation, c’est à dire qu’on pourrait utiliser une variable qui n’a pas été initialisée. C’est pourquoi, pour éviter ce type d’erreurs, nous avons décidé de déclarer l’ensemble des variables au début de chaque fonction.

[2] Nous avons choisi d’utiliser des `LinkedHashMap`, afin de conserver l’ordre d’ajout dans les listes, pour plus de cohérence.
[3] Le symbole _ est ajouté sur les fonctions pour éviter que les noms des fonctions while soient des mots clés C++

```
// Erreur : le transfert de contrôle ignore l'initialisation de :C/C++(546)
program.cpp(3, 1): variable "a" (déclaré à la ligne 5)
int fonctionAvecDeclarationCroisee()
{
    goto L2;
L1:
    int a = 2;
L2:
    int b = a + 3;
    return b;
}
```

```
// Sans erreur
int fonctionSansDeclarationCroisee()
{
    int a;
    goto L2;
L1:
    a = 2;
L2:
    int b = a + 3;
    return b;
}
```

Illustration du concept de déclaration croisée

Enfin, vous aurez pu noter la présence de la variable `stack_params` qui simule la pile décrite dans le 3A. Dès qu'une instruction **Param** est détectée, on ajoute la variable associée sur la pile, et dès qu'une instruction **Call** est détectée, on récupère dans l'ordre LIFO les n variables demandées (et donc leur nom).

2.1.6. Bibliothèque runtime

La bibliothèque runtime est le fichier *.jar* : *runtime.jar*

Voir README.md

2.2. Validation du compilateur

Pour effectuer la validation du compilateur nous avons écrit deux types de test :

- Des tests qui vérifient si une exception est bien levée lorsqu'un programme incorrect est donné en entrée. Ou à l'inverse qu'un programme ne lève pas d'exception quand le programme est correct. Ces tests ont été écrits comme des tests unitaires Java via JUnit.

- Des tests qui vérifient le bon résultat d'un programme. Pour cela un script bash a été écrit. Celui-ci exécute à un à un les programmes grâce au runtime et vérifie que le résultat est le bon à chaque fois.

Nous savons que chacun de ces types de sont pas exhaustifs. En effet, il aurait été trop long d'écrire un nombre de tests suffisant au vu du temps qu'il nous restait. Cependant il suffit de les compléter, la structure de test étant déjà mise en place.

Aussi loin que l'on a testé, tout fonctionne. Cependant il reste au moins deux choses à améliorer :

- Optimisation de code intermédiaire
- Des tests plus poussés

2.3. Méthodologie de gestion de projet

Afin de mener à bien ce projet, nous nous sommes tout de suite dirigés vers un gestionnaire de versions **Git** où notre code est hébergé sur **GitLab**. Quant à la communication au sein du groupe de projet, la **communication verbale** a été privilégiée notamment lors des TPs. En complément, l'outil de messagerie instantanée **Discord**, nous a permis d'avoir un retour direct sur les éventuels points que l'on souhaitait mettre en place.

Concernant l'aspect technique, les premières séances de TPs étaient consacrées à la compréhension du langage WHILE et à l'implémentation de l'AST. Ces étapes ont été réalisées tous ensemble.
Ensuite, nous nous sommes répartis les tâches en fonction des étapes restantes à faire dans la chaîne de compilation. Donc, Kilian Cornec était chargé de l'analyse sémantique avec la construction de la table des symboles et la validation du compilateur, Bastien Faisant

a généré le code 3 adresses à partir de l'AST et Léo Filoche s'occupait de la génération du code C++.
Enfin, selon la finalité des tâches précédentes, Léo Filoche et Kilian Cornec ont élaboré un script afin de lier la compilation While avec le compilateur C++ et de générer un exécutable.

2.4. Postmortem

- Ce qui a bien fonctionné, moins bien fonctionné dans l'organisation de projet ?

Durant tout le projet, nous étions toujours actifs avec des tâches précises bien qu'au début du projet nous nous sentions un peu perdus devant tout le travail que cela nous demandait. De plus, la construction de l'AST nous a pris beaucoup de temps à cause d'une mauvaise compréhension de l'arbre final que l'on devait obtenir. Cependant une fois que nous nous sommes répartis les tâches pour la suite du projet, nous avons été beaucoup plus efficaces.

Nous nous sommes aussi énormément interrogés sur le lien entre la table des symboles et la génération du code 3 adresses pour finalement comprendre qu'il n'y avait pas de relation directe. En effet, au début, nous pensions stocker les valeurs des variables dans la table des symboles pour ensuite les gérer depuis la génération du code 3 adresses avant de comprendre que la mémoire serait directement dans le générateur de code intermédiaire.

- Avec plus de recul, comment feriez-vous maintenant ?

Avec plus de recul, les premiers TPs ont beaucoup été consacrés à la création de l'AST. Nous aurions dû valider notre AST plus rapidement afin de se consacrer directement aux autres étapes de la chaîne de compilation. Cela nous aurait peut-être permis d'avoir un compilateur plus complet (avec l'optimisation du code intermédiaire notamment) et plus robuste.

Maintenant que nous connaissons précisément les différentes étapes nécessaires à la création d'un compilateur, nous aurions mieux réparti les tâches dès le début. De plus, il aurait été plus aisé d'avancer sur le projet tout au long du semestre sans être découragé par l'inconnu et l'envergure du travail.

Pour conclure, ce cours de théorie des langages nous a apporté énormément de connaissances sur les compilateurs que nous utilisons tous les jours sans connaître réellement leurs fonctionnements et leurs différentes complexités.