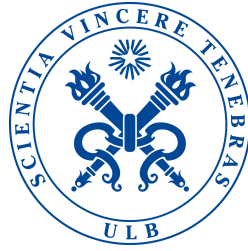


UNIVERSITÉ LIBRE DE BRUXELLES



INFO-F403 - INTRODUCTION TO LANGUAGE THEORY AND
COMPILING

Rapport 3ème Partie

Auteurs :

Yasin ARSLAN

Jacky TRINH

Titulaires :

Gilles GEERAERTS

Assistants :

Marie VAN DEN BOGAARD

Leo EXIBARD

Table des matières

1	Introduction	2
2	Langage LLVM	2
3	Initialisation des variables	2
4	Expression Arithmétique	3
5	If, else, while, for	3
6	Print & Read	3
7	Implantation Java	4
7.1	Classe	4
7.1.1	CodeGenerator	4
7.1.2	NotationConverter	4
7.2	Rules	4
7.3	Main	4
8	Exécution & Makefile	4
9	Exemple d'exécution	5
10	Conclusion	6

1 Introduction

Dans le cadre du cours INFO-F403, il nous a été demandé d'implanter un compilateur pour le langage IMP. Ce projet est divisé en 3 parties. La troisième partie consiste en l'implantation d'un générateur de code qui va générer du code de bas niveau en langage LLVM. Pour ce faire, nous avons dû utiliser ce que nous avons implanté lors de la première et seconde partie ainsi que des notions et concepts vus aux cours théoriques et pratiques

2 Langage LLVM

Tout d'abord, afin de pouvoir faire cette troisième partie du projet, nous avons dû premièrement nous familiariser avec le langage LLVM qui est plus au moins un mélange entre l'assembleur et le langage C#. Nous avons remarqué que ce langage utilisait un système de variable temporaire avec des nombres qui s'incrémentaient de 1 à chaque fois (ie : %0, %1, %2, ..). Cela nous a permis d'utiliser le résultat d'une opération binaire pour une prochaine opération plus aisément. Ou encore de stocker la valeur d'une variable dans une variable temporaire, etc...

3 Initialisation des variables

Après de multiples essais avec le langage LLVM, nous avons remarqué qu'il était préférable pour les variables du code IMP d'être transformées en variables dont l'espace était alloué en langage LLVM. En effet, comme il est possible d'assigner à plusieurs reprises une valeur à une variable, il fallait utiliser les pointeurs en LLVM.

Une deuxième constatation était d'utiliser la liste des *identifiers* provenant de la partie 1 du projet afin de pouvoir initialiser toutes les variables en début de code afin de ne pas avoir de soucis. En effet, initialiser les variables seulement lorsque nous les rencontrons pour la première fois était une mauvaise approche :

```
if x = 3 then
— y = 2
else
— y = 1
endif
```

Ici, en retranscrivant ce code en langage LLVM, nous aurions initialisé la variable `%y` et assigné la valeur 3 dans le premier cas (qui est un label). Dans le second cas, nous aurions seulement assigné la variable à 1 puisque dans le code, nous avons déjà initialisé la variable. Mais le problème est que si jamais nous sautons au deuxième label, la variable n'aurait pas été initialisée. Afin de palier ce genre de problèmes, nous avons donc décidé de cette solution.

4 Expression Arithmétique

N'utilisant aucun arbre (Parse Tree), nous nous sommes rendus compte d'une petite difficulté. En effet, nous n'avions pas de "sauvegarde" de priorités des opérations. Comme nous utilisons simplement des fonctions récursives, c'est comme si nous lisions l'expression tel quel, de gauche à droite. Afin de palier à ce soucis, nous avons décidé de transformer les expressions arithmétiques qui sont écrites en notations dites *infix* en notations dites *postfix* (Notation Polonaise Inversée). Un simple $5 + 2 * 3$ équivaldrait à $5\ 2\ 3\ *+$. Ceci permettait, en effet, de produire une expression en plusieurs sous opérations LLVM. Dans l'exemple, nous pouvons donc calculer en premier lieu $2 * 3$ et le stocker dans une variable temporaire $\%i$ (où i est le nombre actuel de variables temporaires) qui est utilisé par la suite pour l'opération d'addition $5 + \%i$.

5 If, else, while, for

Les conditions étaient un jeu de labels. En effet, si nous avions une simple condition et qu'elle s'avère vrai, nous sautions au label contenant le code du *if*, dans le cas contraire, nous sautions dans le label contenant ce qu'il se passe après le *if*. Si un *else* était présent, c'est simplement ici que nous sautions. Si des conditions étaient présentes avec un *and*, il suffisait simplement d'avoir un label par condition et de sauter à la condition suivante (label suivant) lorsque la condition avait été vraie, sinon nous sautions dans le label après l'instruction (celui du *else* ou bien celui après le *if*). Dans le cas des conditions avec des *or*, nous avions simplement deux choix : soit nous passons au label du code, soit nous passons au label suivant afin de vérifier la condition. Dans le cas de la dernière condition, nous sautions soit au label du code, soit au label suivant qui est celui du *else* ou celui qui contient ce qui se passe après le *if*. En suivant ces logiques, il n'a pas trop été compliqué de pouvoir mélanger ces deux opérateurs en gardant les priorités.

Le *while* est simplement un label qui contient un *if* dans lequel nous resautons une fois que nous atteignons la fin.

Le *for* est également un simple *while* dans lequel nous incrémentons à chaque fois la variable à comparer.

Afin de n'avoir aucune embrouille, nous utilisons un stack afin de sauvegarder les contextes, c'est-à-dire un stack qui va contenir les labels pour permettre de faire des instructions imbriquées.

6 Print & Read

La définition de ces deux fonctions ont été prises sur l'Université Virtuelle.

7 Implantation Java

7.1 Classe

De nouvelles classes ont vu le jour et d'autres ont été modifiées afin de pouvoir générer du code LLVM.

7.1.1 CodeGenerator

Cette classe va écrire dans le fichier *file.ll* le code utilisé pour le langage LLVM.

7.1.2 NotationConverter

Comme nous utilisons une notation postfix afin de gérer les expressions arithmétiques, c'est elle qui va se charger de la conversion infix -> postfix.

7.2 Rules

A été légèrement modifié afin de faire appel aux méthodes de *CodeGenerator* pour générer du code.

7.3 Main

A également été modifiée afin d'afficher le code généré.

8 Exécution & Makefile

Afin de pouvoir exécuter notre projet, il faut se situer dans le dossier */dist/* et rentrer la commande suivante dans un terminal : *java -jar part3.jar sourceFile* où le *sourceFile* est supposé être un fichier *.imp*.

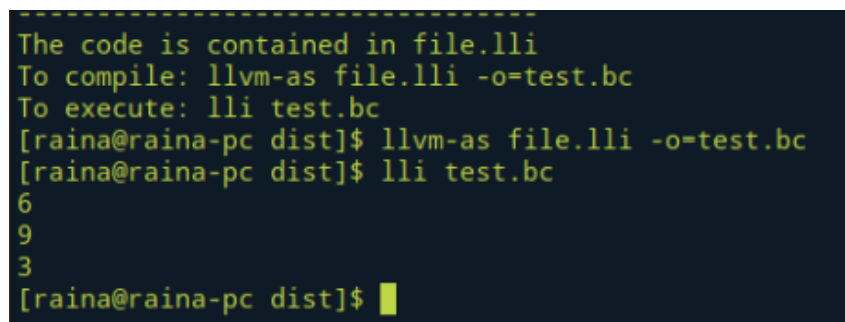
Le code généré sera retranscrit dans le fichier *file.ll* du dossier courant. Afin de compiler ce fichier, il suffit de taper *llvm-as file.ll -o=test.bc* qui va donc compiler notre code et produire le fichier *test.bc* qu'il va falloir ensuite exécuter à l'aide de *lli test.bc*. Nous avons également créé un makefile afin de rendre les choses beaucoup plus simple. Pour l'utiliser, il faut se rendre dans le dossier */more/*.

- *make* permet de compiler le projet.
- *make euclid* permet de tester les fichiers *.class* avec le fichier d'input *Euclid.imp* provenant de l'Université Virtuelle.
- *make test* même chose qu'au-dessus mais avec le fichier d'input *Test.imp*.
- *make aa* même chose qu'au-dessus mais avec le fichier d'input *aa.imp*.

- *make jar* permet de créer un fichier *jar* dans le dossier */dist/* ainsi que de l'exécuter avec le fichier de base *Euclid.imp*
- *make clean* supprime tous les fichiers *.class*.

9 Exemple d'exécution

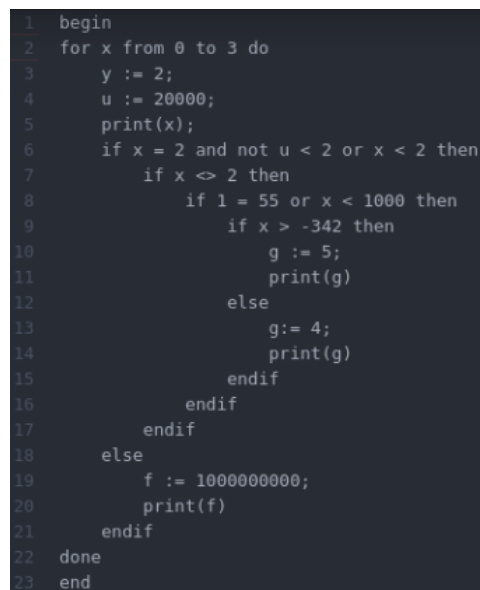
Afin de vérifier si notre projet fonctionnait correctement, nous l'avons testé sur plusieurs fichiers dont un étant le fichier fourni par l'Université Virtuelle, à savoir *Euclid.imp*. Comme l'algorithme d'Euclid permettait de connaître le plus grand commun diviseur, il a été facile de savoir si cela fonctionnait :



```
The code is contained in file.lli
To compile: llvm-as file.lli -o=test.bc
To execute: lli test.bc
[raina@raina-pc dist]$ llvm-as file.lli -o=test.bc
[raina@raina-pc dist]$ lli test.bc
6
9
3
[raina@raina-pc dist]$
```

FIGURE 1 – PGCD(6,9) = 3

Nous avons créé plusieurs autres fichiers *.imp* afin de tester les if,else imbriqués, les while imbriqués ainsi que les for imbriqués. Le fichier *aa.imp* contient à peu près toutes les instructions mélangées.



```
1 begin
2   for x from 0 to 3 do
3     y := 2;
4     u := 20000;
5     print(x);
6     if x = 2 and not u < 2 or x < 2 then
7       if x <= 2 then
8         if 1 = 55 or x < 1000 then
9           if x > -342 then
10            g := 5;
11            print(g)
12          else
13            g := 4;
14            print(g)
15          endif
16        endif
17      endif
18    else
19      f := 10000000000;
20      print(f)
21    endif
22  done
23 end
```

FIGURE 2 – aa.imp

10 Conclusion

Cette troisième partie nous a montré à quel point la génération d'un code provenant d'un langage de haut niveau en un celui de bas niveau est à la fois simple et compliquée. Comme à la seconde partie, nous n'avions pas pensé à utiliser un Parse Tree, nous avons décidé de continuer sur notre façon de faire. Il est très fort probable qu'avec un Parse Tree, nous aurions eu plus (voire beaucoup plus) de facilité mais ne voulant pas complètement changer la structure de notre code, nous avons décidé de garder ce que nous avons fait. Dans l'ensemble, ce projet fut intéressant et agréable. En effet, n'ayant auparavant aucune connaissance sur la manière dont cela fonctionnait concrètement, il a été très captivant de s'y plonger. Nous sommes assez contents du résultat produit.