Traitement des langages

Camille MICHEL

2 janvier 2020

CentraleSupélec

Table des matières

Introduction

L'objectif de ce BE est d'écrire une grammaire ANTLR pour langage (Niklaus), de générer les arbres de syntaxe abstraite (AST), puis de compiler ce langage vers le langage d'assemblage du petit micro-processeur présenté en cours.

Pour ce faire dans un premier temps la grammaire va être écrite à l'aide d'ANTLRWorks qui va générer les fichiers java correspondant à cette grammaire afin d'être utilisé par le compilateur que l'on écrira en java. Le code assembleur ainsi produit sera ensuite assembler grâce à un code python fourni afin de produire un fichier mem que l'on pourra utiliser dans le micro-processeur sur Logisim

1 Grammaire Niklaus

La grammaire Niklaus est défini dans l'énoncé du sujet, et il y a également des fichiers .niklaus qui permettent de mieux la comprendre. Grâce a ces éléments on défini la grammaire comme suit :

```
grammar Niklaus;

options {
   output = AST;
}

expr        :   term (ADDOP^ term)*;

term        :   factor (MULTOP^ factor)*;

factor      :   ID | INT | ( '('! expr ')'! );

comparaison     :   expr COMPOP^ expr;

read        :   READ^ ID ';'! ;

write       :   WRITE^ expr ';'!;

affectation :   ID AFFECT^ expr ';'!;

loop        :   WHILE^ '('! comparaison ')'! '{'! instruction '}'!;

condition   :   IF^ '('! comparaison ')'! '{'! instruction '}'! ELSE '{'! instruction '}'!;

instruction :   (comparaison | read | write | affectation | loop | condition)* ;

program     :   PROGRAM^ ID ';'!  declaration?  '{'! instruction '}'! ;

declaration :   VAR^ ( ID ','! )* ID ';'! ;

INT :   '0'..'9'+
    ;

COMMENT
    :   '//' ~('\n'|'\r')* '\r'? '\n' {$channel=HIDDEN;}
    ;

WS  :   ( ' '
        | '\t'
        | '\r'
        | '\n'
        ) {$channel=HIDDEN;}
    ;

ADDOP   :   '+' | '-' | 'mod' ;

MULTOP  :   '*' | '/';

COMPOP  :   '<' | '<=' | '=' | '<>' | '>=' | '>' ;

READ    :   'read';

WRITE   :   'write';

AFFECT  :   ':=';

WHILE   :   'while';

IF  :   'if';

ELSE    :   'else';

PROGRAM :   'program';

VAR :   'var';

ID  :   ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')*;
```

Figure 1. Grammaire Niklaus

Suite à cela j'ai pu importer les fichier générer par ANTLRWorks dans Eclipse afin de travailler sur les arbres syntaxiques abstraits (AST).

2 Génération du code assembleur

Afin de générer le code assembleur j'ai suivi les niveaux de complétude de l'énoncé. Mon code est séparer en plusieurs partie, au début le main qui va créer l'arbre, la parcourir et renvoyer le code assembleur correspondant puis ajouter les librairies et enfin écrire le fichier .arm. Pour tester un fichier Niklaus il suffit juste de mettre sont nom dans $file_name$ et de changer les paths vers les fichiers. Ici je créé 2 dossiers un pour les fichiers ARM, l'autre pour les fichier Niklaus.

J'ai fait le choix ici de créer un très long string et à la fin de l'écrire dans un fichier pour éviter tout problème d'écriture mais j'aurais pu utiliser un stream directement pour écrire au fur et à mesure de l'avancement. Quand le main parcours l'arbre il fait appel à une méthode $process_bloc$ qui va reconnaitre les léxèmes suivants : VAR, AFFECT, WRITE, READ, IF et WHILE. Au sein de cette méthode un méthode similaire est appelée mais cette fois-ci pour les expressions : INT ? ID, ADDOP, MULTOP. Enfin la comparaison est traitée à part. Enfin tout du long j'ai utilisé des index pour les adresses pour être sûre de ne pas avoir de problème dans mon programme, read et write partage le même index i.

2.1 Complétude 1

Dans cette première partie on souhaite gérer les problèmes de définitions de variables et reconnaître l'instruction *write* qui affiche une valeur sur l'écran. Pour vérifier si une variable a bien été déclaré j'ai choisi de créer une liste global des variables afin que le compilateur vérifie avant l'utilisation d'une variable si elle existe ou lorsqu'on la définie si elle n'a pas déjà été définie. Pour gérer les erreurs j'ai créé une classe $ExceptionVariables$ qui permet de renvoyer un message d'erreur dans les cas de figure présenter, une variable n'est pas définie ou a déjà été définie.

2.2 Complétude 2

C'est a ce moment que j'ai créée $process_expr$ afin de gérer de façon récursive les différentes expressions possibles. J'ai utilisé des if/else afin de séparer les additions, soustractions et modulo tout comme pour multop.

2.3 Complétude 3

Pour ce niveau il faut utiliser la pile en l'initialisant au début du fichier comme vu en cours et en la supprimant à la toute fin, après les librairies. Les différentes opérations sont définies comme dans les slides de cours en utilisant la pile.

2.4 Complétude 3 et 4

Ces niveaux sont assez équivalent puisqu'ils utilisent les comparaisons et des blocs. Pour traiter les blocs j'utilise $process_bloc$ mais pour le if il faut bien séparer les 2 blocs c'est pour cela que j'ai créé une méthode get_bloc qui permet d'obtenir les 2 blocs et de travailler dessus directement.

Le traitement des ces deux niveaux ce fait dans des méthodes $process_while$ et $process_if$. De plus je leur ai créé leurs propres compteur afin d'être sûr que 2 boucles n'aient pas le même numéro ou que la fin de la boucle n'est plus le bon index.

3 Conclusion

Tout les programmes fonctionnent comme demandé dans le micro-processeur, cependant pour le calcul de factoriel lorsque le nombre devient trop grand le micro-processeur affiche 0.