BROSSARD Florian

DJEBRI Maroua

Projet de compilation

Master ILC 1ère année

2017-2018

Table des matières

I. Choix effectués

1. La grammaire

Voici la grammaire que nous avons implémentée dans notre compilateur :

```
axiom:
            fonction //reconnait une fonction main
        |   action //utilisé pour tester rapidement le compilateur
        |   %empty //reconnais un fichier vide
        ;

fonction:
            TYPE MAIN '(' ')' '{'action retour'}' // fonction main
        |   TYPE MAIN '(' ')' '{' retour '}' //function main avec juste un retour
        ;

retour:
            RETOUR ENTIER';' //reconnais une ligne comme return 0;
        ;

action: //permet d'avoir une suite d'instruction
            instruction
        |   instruction action
        ;

instruction:
            declaration     ';' //déclaration d'une variable
        |   operation       ';'
        |   condition       //structure if / else
        |   boucle          //structure for / while / do … while
        |   affectation     ';' //affectation d'une valeur à une variable
        |   incrementation ';'
        |   print_fct ';'  //reconnais les function printf(…) et printi(…)
        ;

print_fct:
            PRINTF '(' STRING ')' //String contient une chaîne de caractère
        |   PRINTI '(' expression ')'
        ;

declaration:
            TYPE ID //reconnais une déclaration
        |   TYPE affectation //reconnais une déclaration avec affectation
        ;

affectation: //affectation avec une valeur directe ou une opération
            ID '=' expression
        ;
```

```
valeur: //on ne gère que les entier et les variables entières
		ENTIER
	|	ID
	;

expression:
		valeur
	|	operation
	;

operation: //paranthèse ou opérateur (sauf comparaison)
		expression '+' expression
	|	expression '-' expression
	|	expression '*' expression
	|	expression '/' expression
	|	'('expression')'
	|	'-' expression %prec UMOINS
	;

condition: //structure if
		IF'('comparaison')' '{'action'}'
	|	IF'('comparaison')' '{'action'}' ELSE '{'action'}'
	;

boucle: //boucles
		WHILE'('comparaison')' '{'action'}'
	|	DO '{'action'}' WHILE '('comparaison')'';'
	|	FOR'('TYPE affectation';' comparaison';' incrementation')' '{'action'}'
	|	FOR'('TYPE affectation';' comparaison';' operation')' '{'action'}'
	;

incrementation: // in-dé-crémentation de 1 ou raccourcis
		ID INCR
	|	ID DECR
	|	ID SHORT_INCR expression
	|	ID SHORT_DECR expression
	;

comparaison: //opérateurs de comparaisons
		expression EGAL expression
	|	expression INEGAL expression
	|	expression INFEG expression
	|	expression SUPEG expression
	|	expression '>' expression
	|	expression '<' expression
	|	TRUE
	|	FALSE
	;
```

2. Représentation du texte d'entrée

Nous avons choisi de travailler avec une représentation du texte d'entrée sous la forme d'un AST. Bien que cela nécessite une charge de travail supplémentaire, les AST sont très souvent utilisé dans les compilateurs actuels, ce qui nous permet d'avoir un avant-goût de ce qu'il se fait réellement.

De plus la structure d'un AST est telle que l'on peut l'utiliser pour presque n'importe quel langage d'entrée, juste la grammaire et les actions changent.

De plus, nous avions déjà une première expérience avec les AST venant d'une matière où nous avions étudié l'analyse syntaxique, en utilisant des AST.

3. Les symboles et les quads

Nous avons premièrement réutilisé les structures qui nous avaient été fournies durant les TP. Nous avons ensuite décidé de changer la structure des symboles pour une liste chaînée de symboles.

Ce changement nous a permis de lever deux limites :

- Le maximum de symboles
- La taille des identificateurs qui était bloqué à 8 caractères

Nous avons effectué ce changement plutôt pour lever la seconde limite. Ayant ensuite été confronté aux joies des fuites mémoire et des erreurs de segmentation, nous avons décidé de ne pas changer de la manière la structure des quads, à savoir un tableau dynamique.

II. Fonctionnalités

1. Reconnaissance du code

Le compilateur reconnait la fonction main sans arguments et renvoyant un entier.

Les commentaires inline et sur plusieurs lignes sont ignorés correctement.

La précédence des opérateurs est correctement gérée, de même que les expressions avec parenthèses.

Le compilateur peu prendre jusqu'à 3 paramètres :

1. Nom du fichier d'entrée (entrée standard si non renseigné)
2. Nom du fichier de sortie MIPS (sortie standard si non renseigné)
3. Nom du fichier de sortie pour rediriger yyout directement (sortie standard si non renseigné)

2. Gestion des erreurs

Le compilateur renvoie une erreur si une fonctionnalité n'a pas encore été implémentée bien que le code soit reconnus (boucle, structure conditionnelle).

Les tokens non reconnus sont affichés par un messages avec flex.

Les erreurs de syntaxes sont gérées par yacc, nous avons juste ajouter quelques options pour que ces erreurs soient plus parlantes (token attendus, ligne de l'erreur). En revanche une erreur syntaxique entraîne immédiatement l'arrêt du compilateur.

Côté sémantique, nous renvoyons une erreur si nous détectons une division par 0, c'est à dire si 0 est utilisé tel, nous ne gérons pas le cas où une variable vaut 0.

Une erreur est également envoyée si un identificateur non-déclaré est utilisé. Cependant on ne gère pas le fait qu'ils ait ou non été initialisé avant leurs utilisations.

3. Fonctionnalités non-implémentées

Les structures conditionnelles (if .. else ..) ainsi que les opération de comparaison n'ont pas été implémentée. De même que les boucles (for, while, do while). Cependant, notre grammaire permet de reconnaître le texte correspondant.

Les stencils et les tableaux n'ont pas été implémentés. De même que les autres types en C.

Les fonctions de l'utilisateurs ne sont pas reconnues, à l'exception de la fonction main.

De même, seuls les appels aux fonctions printf() et printi() sont gérés, printf ne reconnaissant comme argument qu'une chaîne de caractère codé en dur, et printi une expression (entier, opération ou variable).