Projet de compilation *[IF204]*

Rapport de projet 21/12/2007

Compilation avec *LEX* et *YACC*

Encadrant :

David JANIN

Binômes

Julien LAVERGNE

Mohamed Amine EL AFRIT

Table des matières

Chapitre 1

Introduction

Le projet consiste à créer, à l'aide de *Yacc* et de *LEX*, un compilateur d'un langage relativement simple qui ne possède que des fonctionnalités basiques telles que les déclarations locales, les types... (pas des types complexes tels que les tableaux ou les fonctions).

Ce compilateur doit fournir un code cible à trois-adresses. Les déclarations doivent être en tête de ce code.

Dans ce rapport, nous allons présenter, dans un premier temps, l'analyse du problème. Ensuite, nous nous intéressons à la phase de conception pendant laquelle on va détailler le problème d'analyse sémantique de notre compilateur, en particulier les techniques mises en jeu pour mettre en évidence cette phase cruciale.

Dans le troisième chapitre, nous introduisons le code cible produit tout en expliquant les problèmes rencontrés lors de la réalisation de ce compilateur.

À la fin, un jeu de tests a été effectué dans le but de vérifier que le compilateur arrive effectivement à compiler un code source introduit.

Chapitre 2

Analyse du problème

Durant cette étape nous avons identifié les besoins de notre application dans le cadre des contraintes imposées par le cahier de charge à savoir :

- Pour le code source :
  - Un typage minimum (int, bool, float) avec vérification de type.
  - Un mécanisme de déclarations locales avec masquage et portée.
  - Une gestion des structures de contrôles (les conditionnelles et les boucles).
  - Une gestion des expressions complexes.
- Pour le code cible :
  - Un code trois adresses (des goto pour les branchements).
  - La déclaration des variables est globale et en tête de programmes.
- Un jeux de tests simples pour chaque fonctionnalité (avec les résultats des tests).

2.1 Grammaire

La grammaire sur laquelle on va de reposer est la suivante :

prog[1]	→	decl_list[2] pv [3]
		| decl_list inst_list[4] pv
decl_list	→	decl_list decl
decl	→	type id_aff_list PV
pv	→	PV
id_aff_list	→	id_aff_list VIR id_aff
		| id_aff
id_aff_list	→	id_aff_list VIR id_aff
		| id_aff
id_aff	→	id
		| ID EQ exp
id	→	ID
type	→	INT
		| FLOAT
		| BOOL
		| type STAR
inst_list	→	inst_list PV inst
		| inst
inst	→	affect
		| cond

		| loop
		| bloc
affect	→	ID EQ exp
cond	→	IF exp THEN inst
		| IF exp THEN inst ELSE inst
loop	→	WHILE exp DO inst
		| REPEAT inst UNTIL exp
bloc	→	DA prog FA
exp	→	exp OR exp
		| exp AND exp
		| exp PLUS exp
		| exp MOINS exp
		| exp STAR exp
		| exp DIV exp
		| exp EQL exp
		| exp GRT exp
		| exp LOW exp
		| exp NEQ exp
		| STAR exp
		| MOINS exp
		| NOT exp
		| DP exp FP
		| id
		| const
const	→	NUM
		| TRUE
		| FALSE

2.2 Modifications apportés sur la grammaire

2.2.1 Au niveau de *IF...THEN...*

cond	→	IF exp then inst
		| IF exp then inst else inst
then	→	THEN
else	→	ELSE

2.2.2 Au niveau de *WHILE...*

loop	→	while exp do inst
		| repeat inst UNTIL exp
while	→	WHILE
do	→	DO
repeat	→	REPEAT

2.2.3 Au niveau des *BLOCS*

bloc	→	da prog fa
da	→	DA
fa	→	FA

Chapitre 3

Conception

L'analyse lexicale utilisé un générateur d'analyseur lexicaux : *LEX*.

L'analyse syntaxique utilisé un générateur d'analyseur syntaxiques : *YACC*.

3.1 Les symbols

3.1.1 Structure

Un symbole est representé sous forme de structure contenant :

char * name
int type
char * value [1]
bloc* bloc [2]
struct symbol* previous [3]

- *name* est le nom de la variable tel qu'il est dans le code source.
- *type* est un entier qui indique le type du symbole selon les règles suivantes.
  - 1 pour le type *int*.
  - 2 pour le type *float*.
  - 3 pour le type *bool*.
- *value* est de type *char** car on veut copier directement une chaine de caractère en sortie du compilateur.
- *bloc* : est un pointeur sur la structure *bloc*.
- *previous* : pointe sur le symbole précédent pour former une liste chainée dans le sens inverse pour pouvoir effectuer une recherche à partir du dernier élément ajouté (ainsi la recherche sera plus rapide).

le shema explique la structure des symboles que nous avons utilisé.

[1] Pour pouvoir y pointer facilement .Après on convertira la cahine en entier pour récupérer la valeur
[2] Voir la structure bloc 3.3.1
[3] C'est une liste simplement chaînée

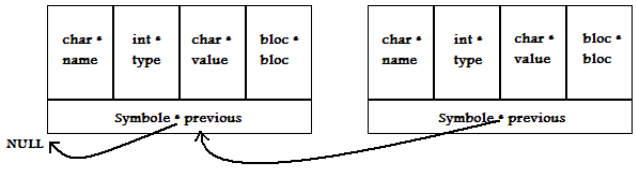

FIG. 3.1 – Structure des symboles

3.1.2 Actions sur les symbols

search_symbol (char* name) → symbol*
print_symbol (symbol* s) → void
create_var (char* name) → symbol*
create_const (char* value) → symbol*
add_symbol (symbol* s) → void
destroy_symbol (symbol* s) → void

3.2 Les expressions

3.2.1 Structre

Une expression est representée sous forme de structure contenant :

symbol* result
symbol* left
int op
symbol* right
struct expression* previous [4]

le shema explique la structure des expressions que nous avons utilisé.

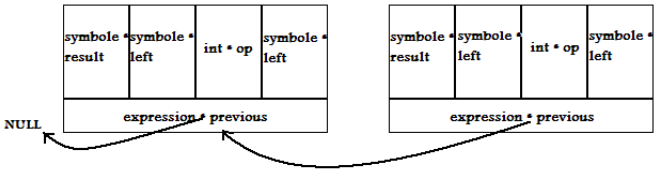

FIG. 3.2 – Structure des expressions

3.2.2 Actions sur les expressions

create_expression (symbol* result, symbol* left, int op, symbol* right) → expression*
print_expression (expression* e) → void
add_expression (expression* e) → void
destroy_expression(expression* e) → void

[4] C'est une liste simplement chaînée

3.3 Les blocs

3.3.1 Structure

Un bloc est représenté sous forme de structure contenant :

int id
int level

Chapitre 4

Analyse sémantique

L'analyse sémantique permet d'analyser et d'identifier les différents mots du langage. De plus, elle permet de vérifier que les types des différentes variables utilisées dans le programme sont corrects. Pour traiter cette phase, nous avons défini une table de symboles et table d'expressions. Nous allons à présent voir la structure de ces deux tables.

4.1 Table de symboles

Comme son nom l'indique, cette table permet de contenir toutes les informations associées aux symboles rencontrés dans le programme. Chaque symbole est défini par son *nom*, son *type*, sa *valeur un marqueur bloc* qui indique à quel bloc appartient et *un pointeur au symbole precedent.* (voir figure 3.1)
Dans cette table, nous n'insérons que les symboles différents des mots clés du langage choisi (les mots clés sont déjà définis dans le fichier *projet.l*).

Les nom des variables intermédiaires est de la forme $vari\_j$ où i et j indiquent respectivement le numéro de la variable et le numéro du bloc où elle se trouve.

- i est un compteur qui s'incrémente à chaque variable rencontrée pour distinguer entre les variables.
- j est aussi un compteur qui s'incrémente si on rentre dans un nouveau bloc et qui se décr'emente si on en sort.

Par exemple pour le bout de code suivant :

```
int x,y;
{x=2+3;};
y =2*3;
```

Nous obtenons les variables suivantes :

```
int x_0;
int y_0;
int var0_1;
var0_1 = 2 + 3;
x_0 = var0_1;
int var1_1;
var1_1 = 2 * 3;
int var2_1;
var2_1 = var1_1;
```

4.2 Table des expressions

Cette table permet de fournir les noms des variables intermédiaires qui seront utilisées dans la production du code cible (spécifique à la machine) : code à trois adresses.

Chaque expression est constituée d'un *symbol* résultat, un *symbol* gauche, un *opérateur* un *symbol* droit et un pointeur vers l'expression précédente. (voir figure 3.2)

Chapitre 5

Production du code à trois adresses

5.1 Traitement des expressions

La génération du code 3@ consiste tout d'abord à stocker chaque argument d'un opérateur donné dans une variable. Ensuite, les variables produites seront écrites suivant un enchaînement séquentiel où nous retrouvons le même opérateur mis en jeu.
Toute construction d'expression du langage est donc associée à une expression sémantique où une variable intermédiaire est créée pour contenir le résultat de l'évaluation d'une sous-expression de cette forme.
Le code synthétisé d'une expression complexe est une composition séquentielle des codes produits pour les sous-expressions correspondant à l'opérateur de la construction considérée.
Exemple :
L'expression suivante $r = x + 2 * z/y$ est traduite comme suit :

```
int x_0 ;
int y_0 ;
int z_0 ;
float r_0 ;
float var0_0 ;
var0_0 = 2 * z_0 ;
float var1_0 ;
var1_0 = var0_0 / y_0 ;
float var2_0 ;
var2_0 = x_0 + var1_0 ;
r_0 = var2_0 ;
```

5.1.1 Traitement des conditionnelles

Le principe de la génération du code 3-adresses est le même : créer des variables intermédiaires et les mettre sous une séquence adéquate qui traduit le code de base. Mais avec les structures de contrôle, il y a, en plus, l'utilisation des branchements définis par "goto".
Voici la traduction en code 3@ de deux écritures suivantes :

Code source	Code 3@
if(exp)then inst	if(not exp) GOTO label0 inst label0
if(exp)then inst1 else inst2	if(not exp) GOTO label0 inst1 GOTO label1 label0 inst2 label1

La gestion de l'écriture des labels au bon endroit est faite lors des lectures et des réductions des non-terminaux formant la structure de contrôle tout en utilisant des attributs synthétisés et hérités dans le fichier *projet.y*
Par exemple, le branchement vers un label donné est fait lors de la réduction de :
else → ELSE

5.1.2 Traitement des boucles

Pareil que les conditionnelles, la traduction des boucles en code 3@ introduit l'utilisation des branchements.
Ce tableau résume la traduction de deux dérivations possibles du non-terminal *loop* (pour désigner la boucle) :

Code source	Code 3@
while (exp)do inst	label0 if(not exp)goto label1 inst goto label0 label1
repeat inst until exp	label0 inst if (exp) goto label0

La lecture du non terminal *while* ou *repeat* nous permet d'écrire les labels de début (label0 dans ce cas). Pour la boucle "while...do", la lecture du non terminal *do* introduit l'écriture de la condition. Quant au *repeat*, c'est la lecture de *until* qui permet d'écrire la condition.
Pour les deux cas, c'est la réduction qui permet d'écrire les labels de fin.

5.1.3 Traitement du bloc

Lors de la lecture d'un début bloc caractérisé par la lecture du non terminal *da* qui désigne une accolade ouvrante, nous avons choisi de créer une structure bloc dans laquelle on incrémente la variable du champ *id* et *level* si on passe dans un nouveau bloc et on décrémente le champ *level* si on sort du bloc et on passe au bloc père.

Le programme à l'intérieur des accolades est traité comme précédemment.

Chapitre 6

Problèmes rencontrés

6.1 Gestion de blocs

Chaque lecture doit être suivie d'une recherche dans le bloc courant et les blocs précèdents afin de décider si l'identificateur vient d'être lu ou bien qu'il est déjà lu. Cette vérification nous permet d'éviter les problèmes engendrés par les redondances.

6.2 Gestion des conditionnelles

Le non terminal *cond* défini dans le langage fourni a deux dérivations possibles :
- if . . .then
- if . . .then. . .else

Pour l'utilisation des labels au début du code cible, l'idée de remplacer le terminal *IF* par un non terminal *if*, comme pour les autres structures de contrôle, n'est pas pratique à cause du conflit lié à l'existence de *IF* dans les deux dérivations.
Pour résoudre ce problème, nous avons pensé à remplacer le terminal *ELSE* par un non terminal *else*, comme pour les autres structures de contrôle,.

6.3 Gestion des imbrications

La gestion des boucles et des conditionnelles imbriquées a posé problème au niveau de l'utilisation des labels. En effet, il y avait un chevauchement entre les labels utilisés pour les différentes structures. Pour bien gérer les labels, nous avons utilisé la dérivation du terminal *ELSE* en terminal *else* comme expliqué précedemment.

Chapitre 7

Tests

7.1 Test *déclaration*

Code Source

```
int x, y=1;
float u,r;
bool z;
z=true;
```

Code Cible

```
int x_0;
int y_0;
y_0 = 1;
float u_0;
float r_0;
bool z_0;
z_0 = true;
```

7.2 Test de la conditionnelle *if-then*

Code Source

```
   bool x=true,y=false;
   float z;
   if (x|y) then {
       z=5;
       y=x;
       };
```

Code Cible

```
bool x_0;
```

```
x_0 = true;
bool y_0;
y_0 = false;
float z_0;
float var0_0;
var0_0 = x_0 || y_0;
if (!var0_0) then GOTO L0;
z_0 = 5;
y_0 = x_0;
L0;
```

7.3 Test de la conditionnelle *if-then-else*

Code Source

```
bool x=true,y=false;
float z;
if (x&y) then{
              int x=2;
              z=x*x;
              y=true;
              }
              else{
                x=y|x;
                x=false;
              };
```

Code Cible

```
bool x_0;
x_0 = true;
bool y_0;
y_0 = false;
float z_0;
float var0_0;
var0_0 = x_0 && y_0;
if (!var0_0) then GOTO L0;
int x_1;
x_1 = 2;
int var1_1;
var1_1 = x_1 * x_1;
z_0 = var1_1;
y_0 = true;
GOTO L1;
L0;
int var2_2;
var2_2 = y_0 || x_0;
x_0 = var2_2;
x_0 = false;
L1;
```

Ces tests mettent en évidence la réalisation des branchements à l'aide des *goto* et les labels *L0* et *L1*. Le code source proposé teste entre autres la portée : la variable *z* (*z_0* déclarée au début du programme) est bien visible dans le bloc qui suit le *then*.

7.4 Test de la boucle *while-do*

Code Source

```
  int i=10;
  while (i>3) do
     i=i-1;
```

Code Cible

```
int i_0;
i_0 = 10;
L0;
int var0_0;
var0_0 = i_0 > 3;
if (!var0_0) then GOTO L1;
int var1_0;
var1_0 = i_0 - 1;
i_0 = var1_0;
GOTO L0;
L1;
```

7.5 Test des *blocs*

Code Source

```
bool x=true,y=false;
{
  int x=3;
  x=23;
};
x=x|y;
```

Code Cible

```
bool x_0;
x_0 = true;
bool y_0;
y_0 = false;
int x_1;
x_1 = 3;
x_1 = 23;
int var0_1;
var0_1 = x_1 ||;
x_1 = var0_1;
```

On voit clairement ici que le compilateur fait la différence entre la variable x déclaré à l'extérieure du bloc et la variable x (qui est de type différent que la première) et qui est déclaré à l'interieure du bloc.

7.6 Test *while-repeat-imbriqué*

Code Source

```
int x = 0;
repeat
  while (x == 99) do
    x = x - 1
until (x == 0);
```

Code Cible

```
int x_0;
x_0 = 0;
L0;
L1;
int var0_0;
var0_0 = x_0 == 99;
if (!var0_0) then GOTO L2;
int var1_0;
var1_0 = x_0 - 1;
x_0 = var1_0;
GOTO L1;
L2;
int var2_0;
var2_0 = x_0 == 0;
if (var2_0) then GOTO L2;
```

Chapitre 8

Conclusion

Ce projet était une occasion pour mettre en oeuvre les concepts élémentaires de compilation de langage de programmation moderne. Nous nous sommes familiarisées avec des outils d'analyse lexicale tel LEX et d'analyse syntaxique tel YACC.
Le compilateur développé, qui colle aux spécifications imposées par le cahier des charges, peut certainnement être amélioré.