Projet de compilation *[IF204]*

## Rapport de projet 21/12/2007

Compilation avec *LEX* et

*YACC*

Encadrant :

Binˆomes

### 



### 

ENSEIRB i2 2007/2008 Semestre n˚3 IF204

# Table des mati`eres

1. Introduction 3
2. Analyse du probl`eme 4
   1. [Grammaire 4](#_TOC_250018)
   2. [Modifications apport´es sur la grammaire 5](#_TOC_250017)
      1. Au niveau de IF...THEN 5
      2. Au niveau de WHILE 5
      3. Au niveau des BLOCS 5
3. Conception 6
   1. [Les symbols 6](#_TOC_250016)
      1. [Structure 6](#_TOC_250015)
      2. [Actions sur les symbols 7](#_TOC_250014)
   2. [Les expressions 7](#_TOC_250013)
      1. [Structre 7](#_TOC_250012)
      2. [Actions sur les expressions 7](#_TOC_250011)
   3. [Les blocs 8](#_TOC_250010)
      1. [Structure 8](#_TOC_250009)
4. Analyse s´emantique 9
   1. [Table de symboles 9](#_TOC_250008)
   2. [Table des expressions 10](#_TOC_250007)
5. Production du code `a trois adresses 11
   1. [Traitement des expressions 11](#_TOC_250006)
      1. [Traitement des conditionnelles 11](#_TOC_250005)
      2. [Traitement des boucles 12](#_TOC_250004)
      3. [Traitement du bloc 12](#_TOC_250003)
6. Probl`emes rencontr´es 13
   1. [Gestion de blocs 13](#_TOC_250002)
   2. [Gestion des conditionnelles 13](#_TOC_250001)
   3. [Gestion des imbrications 13](#_TOC_250000)
7. Tests 14
   1. Test d´eclaration 14
   2. Test de la conditionnelle *if-then* 14
   3. Test de la conditionnelle if-then-else 15
   4. Test de la boucle while-do 16
   5. Test des blocs 16
   6. Test while-repeat-imbriqu´e 17
8. Conclusion 18

Chapitre 1

Introduction

Le projet consiste `a cr´eer, `a l’aide de *Yacc* et de *LEX*, un compilateur d’un lan- gage relativement simple qui ne poss`ede que des fonctionnalit´es basiques telles que les d´eclarations locales, les types... (pas des types complexes tels que les tableaux ou les fonctions).

Ce compilateur doit fournir un code cible `a trois-adresses. Les d´eclarations doivent ˆetre en tˆete de ce code.

Dans ce rapport, nous allons pr´esenter, dans un premier temps, l’analyse du probl`eme. Ensuite, nous nous int´eressons `a la phase de conception pendant laquelle on va d´etailler le probl`eme d’analyse s´emantique de notre compilateur, en particu- lier les techniques mises en jeu pour mettre en ´evidence cette phase cruciale.

Dans le troisi`eme chapitre, nous introduisons le code cible produit tout en ex- pliquant les probl`emes rencontr´es lors de la r´ealisation de ce compilateur.

A` la fin, un jeu de tests a ´et´e effectu´e dans le but de v´erifier que le compilateur arrive effectivement `a compiler un code source introduit.

Chapitre 2

Analyse du probl`eme

Durant cette ´etape nous avons identifi´e les besoins de notre application dans le cadre des contraintes impos´ees par le cahier de charge `a savoir :

* Pour le code source :
  + Un typage minimum (int, bool, float) avec v´erification de type.
  + Un m´ecanisme de d´eclarations locales avec masquage et port´ee.
  + Une gestion des structures de contrˆoles (les conditionnelles et les boucles).
  + Une gestion des expressions complexes.
* Pour le code cible :
  + Un code trois adresses (des goto pour les branchements).
  + La d´eclaration des variables est globale et en tˆete de programmes.
* Un jeux de tests simples pour chaque fonctionnalit´e (avec les r´esultats des tests).

## Grammaire

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

|  |  |  |
| --- | --- | --- |
| prog1 | → | decl list2 pv 3  | decl list inst list4 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 |

## Modifications apport´es sur la grammaire

* + 1. Au niveau de *IF...THEN...*

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

* + 1. Au niveau de *WHILE...*

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

* + 1. Au niveau des *BLOCS*

|  |  |  |
| --- | --- | --- |
| bloc | → | da prog fa |
| da | → | DA |
| fa | → | FA |

Chapitre 3

Conception

L’analyse lexicale utilis´e un g´en´erateur d’analyseur lexicaux : *LEX*. L’analyse syntaxique utilis´e un g´en´erateur d’analyseur syntaxiques : *YACC*.

## Les symbols

### Structure

Un symbole est represent´e 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 r`egles 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 caract`ere en sortie du compilateur.
* *bloc* : est un pointeur sur la structure *bloc*.
* *previous* : pointe sur le symbole pr´ec´edent pour former une liste chain´ee dans le sens inverse pour pouvoir effectuer une recherche `a partir du dernier ´el´ement ajout´e (ainsi la recherche sera plus rapide).

le shema explique la structure des symboles que nous avons utilis´e.

1 Pour pouvoir y pointer facilement .Apr`es on convertira la cahine en entier pour r´ecup´erer la valeur

2 Voir la structure bloc 3.3.1

3 C’est une liste simplement chaˆın´ee

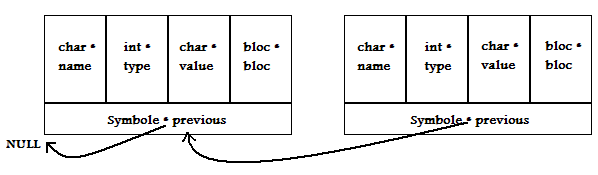


Fig. 3.1 – Structure des symboles

### 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 |

## Les expressions

### Structre

Une expression est represent´ee 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 utilis´e.

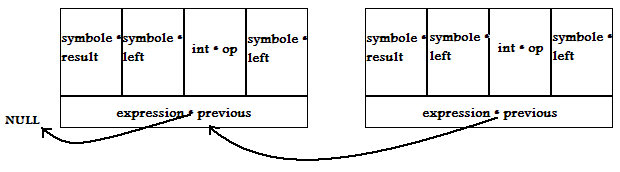


Fig. 3.2 – Structure des expressions

### 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 chaˆın´ee

## Les blocs

### Structure

Un bloc est repr´esent´e sous forme de structure contenant :

int id int level

Chapitre 4

Analyse s´emantique

L’analyse s´emantique permet d’analyser et d’identifier les diff´erents mots du lan- gage. De plus, elle permet de v´erifier que les types des diff´erentes variables utilis´ees dans le programme sont corrects. Pour traiter cette phase, nous avons d´efini une table de symboles et table d’expressions. Nous allons `a pr´esent voir la structure de ces deux tables.

## Table de symboles

Comme son nom l’indique, cette table permet de contenir toutes les informa- tions associ´ees aux symboles rencontr´es dans le programme. Chaque symbole est d´efini par son *nom*, son *type*, sa *valeur un marqueur bloc* qui indique `a quel bloc appartient et *un pointeur au symbole precedent*. (voir figure 3.1)

Dans cette table, nous n’ins´erons que les symboles diff´erents des mots cl´es du lan- gage choisi (les mots cl´es sont d´ej`a d´efinis dans le fichier *projet.l* ).

Les nom des variables interm´ediaires est de la forme *vari j* ou` *i* et *j* indiquent respectivement le num´ero de la variable et le num´ero du bloc ou` elle se trouve.

* *i* est un compteur qui s’incr´emente `a chaque variable rencontr´ee pour distin- guer entre les variables.
* *j* est aussi un compteur qui s’incr´emente si on rentre dans un nouveau bloc et qui se d´ecr’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;

## Table des expressions

Cette table permet de fournir les noms des variables interm´ediaires qui seront utilis´ees dans la production du code cible (sp´ecifique `a la machine) : code `a trois adresses.

Chaque expression est constitu´ee d’un *symbol* r´esultat, un *symbol* gauche, un *op´erateur* un *symbol* droit et un pointeur vers l’expression pr´ec´edente. (voir figure 3.2)

Chapitre 5

Production du code `a trois adresses

## Traitement des expressions

La g´en´eration du code 3@ consiste tout d’abord `a stocker chaque argument d’un op´erateur donn´e dans une variable. Ensuite, les variables produites seront ´ecrites suivant un enchaˆınement s´equentiel ou` nous retrouvons le mˆeme op´erateur mis en jeu.

Toute construction d’expression du langage est donc associ´ee `a une expression

s´emantique ou`

une variable interm´ediaire est cr´e´ee pour contenir le r´esultat de

l’´evaluation d’une sous-expression de cette forme.

Le code synth´etis´e d’une expression complexe est une composition s´equentielle des codes produits pour les sous-expressions correspondant `a l’op´erateur de la construc- tion consid´er´ee.

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 ;

### Traitement des conditionnelles

Le principe de la g´en´eration du code 3-adresses est le mˆeme : cr´eer des variables interm´ediaires et les mettre sous une s´equence ad´equate qui traduit le code de base. Mais avec les structures de contrˆole, il y a, en plus, l’utilisation des branchements d´efinis par “goto”.

Voici la traduction en code 3@ de deux ´ecritures 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’´ecriture des labels au bon endroit est faite lors des lectures et des r´eductions des non-terminaux formant la structure de contrˆole tout en utilisant des attributs synth´etis´es et h´erit´es dans le fichier *projet.y*

Par exemple, le branchement vers un label donn´e est fait lors de la r´eduction de : else → ELSE

### Traitement des boucles

Pareil que les conditionnelles, la traduction des boucles en code 3@ introduit l’utilisation des branchements.

Ce tableau r´esume la traduction de deux d´erivations possibles du non-terminal *loop*

(pour d´esigner 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’´ecrire les labels de d´ebut (label0 dans ce cas). Pour la boucle “while. . .do”, la lecture du non terminal *do* introduit l’´ecriture de la condition. Quant au *repeat*, c’est la lecture de *until* qui permet d’´ecrire la condition.

Pour les deux cas, c’est la r´eduction qui permet d’´ecrire les labels de fin.

### Traitement du bloc

Lors de la lecture d’un d´ebut bloc caract´eris´e par la lecture du non terminal *da* qui d´esigne une accolade ouvrante, nous avons choisi de cr´eer une structure bloc dans laquelle on incr´emente la variable du champ *id* et *level* si on passe dans un nou- veau bloc et on d´ecr´emente le champ *level* si on sort du bloc et on passe au bloc p`ere.

Le programme `a l’int´erieur des accolades est trait´e comme pr´ec´edemment.

Chapitre 6

Probl`emes rencontr´es

## Gestion de blocs

Chaque lecture doit ˆetre suivie d’une recherche dans le bloc courant et les blocs pr´ec`edents afin de d´ecider si l’identificateur vient d’ˆetre lu ou bien qu’il est d´ej`a lu. Cette v´erification nous permet d’´eviter les probl`emes engendr´es par les redondances.

## Gestion des conditionnelles

Le non terminal *cond* d´efini dans le langage fourni a deux d´erivations possibles :

* + - if . . .then
    - if . . .then. . .else

Pour l’utilisation des labels au d´ebut du code cible, l’id´ee de remplacer le termi- nal *IF* par un non terminal *if*, comme pour les autres structures de contrˆole, n’est pas pratique `a cause du conflit li´e `a l’existence de *IF* dans les deux d´erivations.

Pour r´esoudre ce probl`eme, nous avons pens´e `a remplacer le terminal *ELSE* par un non terminal *else*, comme pour les autres structures de contrˆole,.

## Gestion des imbrications

La gestion des boucles et des conditionnelles imbriqu´ees a pos´e probl`eme au niveau de l’utilisation des labels. En effet, il y avait un chevauchement entre les labels utilis´es pour les diff´erentes structures. Pour bien g´erer les labels, nous avons utilis´e la d´erivation du terminal *ELSE* en terminal *else* comme expliqu´e pr´ecedemment.

Chapitre 7

Tests

* 1. Test *d´eclaration*

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;

* 1. 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;

* 1. 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 ´evidence la r´ealisation des branchements `a l’aide des *goto* et les labels *L0* et *L1*. Le code source propos´e teste entre autres la port´ee : la variable *z* (*z 0* d´eclar´ee au d´ebut du programme) est bien visible dans le bloc qui suit le *then*.

* 1. 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;

* 1. 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 diff´erence entre la variable *x* d´eclar´e `a l’ext´erieure du bloc et la variable *x* (qui est de type diff´erent que la premi`ere) et qui est d´eclar´e `a l’interieure du bloc.

* 1. Test *while-repeat-imbriqu´e*

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 ´etait une occasion pour mettre en oeuvre les concepts ´el´ementaires de compilation de langage de programmation moderne. Nous nous sommes familiaris´ees avec des outils d’analyse lexicale tel LEX et d’analyse syntaxique tel YACC.

Le compilateur d´evelopp´e, qui colle aux sp´ecifications impos´ees par le cahier des charges, peut certainnement ˆetre am´elior´e.