Rapport du projet de compilation

Groupe :

Ben Brahim Dhouha
Wang Miao

27 janvier 2013

Introduction

Le projet consiste à créer un compilateur d'un langage objet proche du langage Ruby. Le langage sera compilé en du code intermédiaire du compilateur LLVM. L'appel au compilateur LLVM permettra ainsi de finir la compilation en binaire et de générer du code efficace.

Dans ce rapport, il sera question de présenter , en premier temps, l'analyse du problème, puis la phase de conception dans laquelle nous allons présenter les structures utilisées pour les symboles et les expressions. Ensuite, nous introduirons le code cible produit. A la fin, un jeu de test a été effectué afin de vérifier que le compilateur arrive effectivement à compiler un code source introduit.

Table des matières

1 Analyse du problème

1.1 La grammaire

Voici la grammaire utilisée :

program → topstmts opt_terms

topstmts → topstmt
| topstmts terms topstmt

topstmt → CLASS ID term stmts END
| CLASS ID < ID term stmts END
| stmt

stmts → ϵ
|stmt
| stmts terms stmt

stmt → IF expr THEN stmts terms END
| IF expr THEN stmts terms ELSE stmts terms END
| FOR ID IN expr TO expr term stmts terms END
| WHILE expr DO term stmts terms END
| lhs = expr
| RETURN expr
| DEF ID opt_params term stmts terms END

opt_params → ϵ
| ( )
| (params)

params → ID , params
| ID

lhs → ID
| ID . primary
| ID (exprs)

exprs → exprs , expr
| expr

primary → lhs
| STRING
| FLOAT
| INT
| (expr)

expr → expr AND comp_expr
| expr OR comp_expr
| comp_ expr

comp_expr → additive_expr < additive_expr
| additive_expr > additive_expr
| additive_expr LEQ additive_expr
| additive_expr GEQ additive_expr
| additive_expr EQ additive_expr
| additive_expr NEQ additive_expr
| additive_expr

additive_expr → multiplicative_expr
| additive_expr + multiplicative_expr
| additive_expr - multiplicative_expr

multiplicative_expr → multiplicative_expr * primary
| multiplicative_expr / primary
| primary

opt_terms → ϵ
| terms

terms → terms ;
| term '\n'
| ;
| '\n'
| ' '

term → ;
| '\n'
| ' '

Voici un exemple d'un mot reconnu par la grammaire :

```
def set_params()
   a = 5
   return 0
end
```

On obtient l'arbre de dérivation suivant :

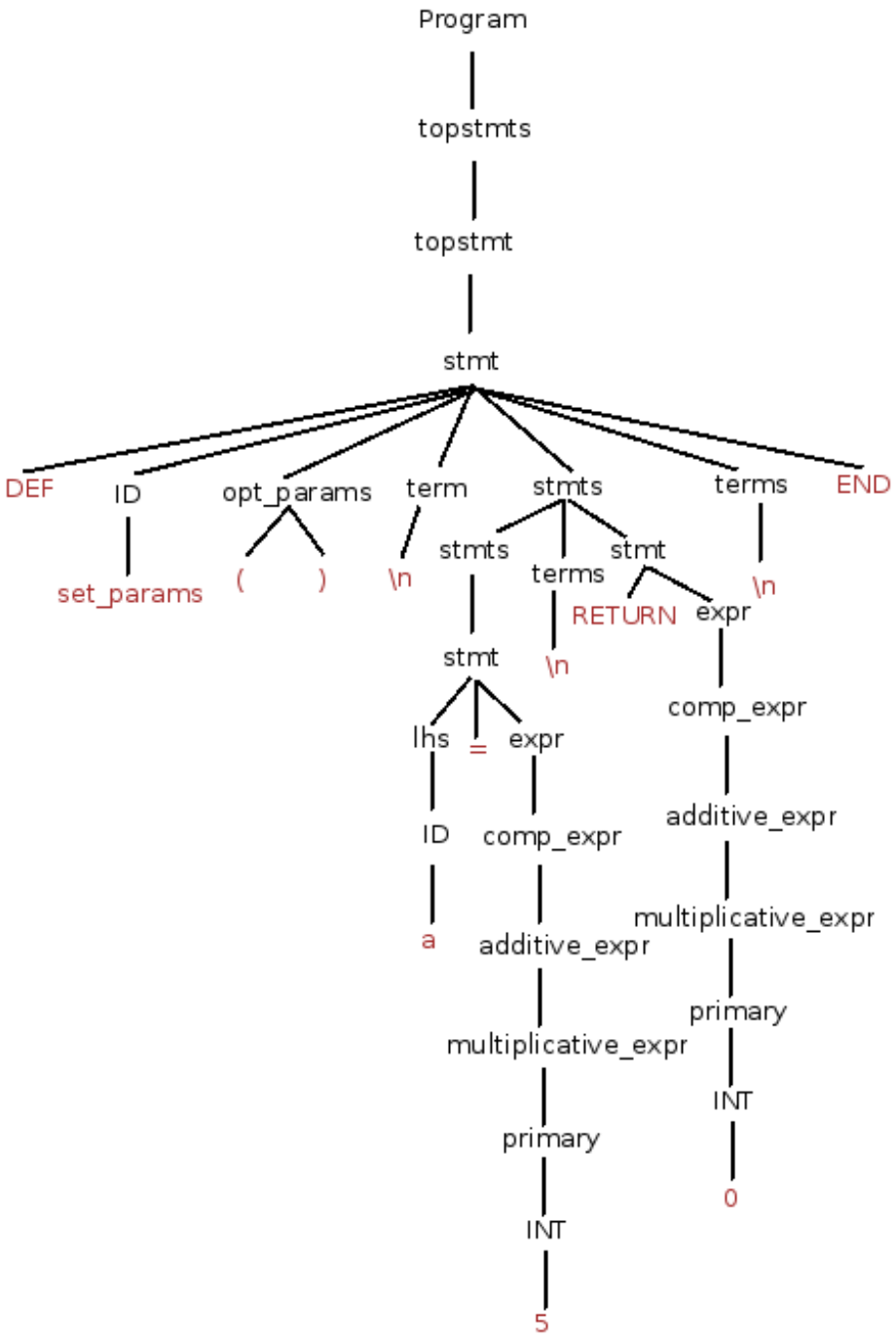

Figure 1 – arbre de dérivation

1.2 Les modifications apportées sur la grammaire

Au niveau des conditionnelles avec *if* :

Nous avons ajouté les lignes suivantes pour pouvoir traiter les conditionnelles sans *"then"* :

stmt → IF expr stmts terms END
|IF expr stmts terms ELSE stmts terms END

Au niveau des conditionnelles avec *unless* :

Nous avons ajouté les lignes suivantes pour implémenter les conditionnelles avec *"unless"* (fonctionnalité facultative) :

stmt → UNLESS expr THEN stmts terms END
| UNLESS expr THEN stmts terms ELSE stmts terms END
| UNLESS expr stmts terms END
| UNLESS expr stmts terms ELSE stmts terms END

2 Conception

2.1 Les symboles

Structure

Une table de symboles doit stocker un nombre important de noms de variables et d'informations qui leurs sont reliées. Elle n'a pas donc de taille fixe. De plus, il faut pouvoir y accéder de manière rapide.

Un symbole est caractérisé par son type, sa valeur (son contenu) et la table dans laquelle il se trouve. Celui-ci peut être soit un identificateur, soit un opérateur, soit une valeur (entier, réel ou chaîne de caractères). Pour cela, nous avons commencé par définir un type énuméré pour désigner le type du symbole de la manière suivante :

```
enum NodeEnum {TYPE_CONTENT, TYPE_INDEX, TYPE_OP};
```

où :
- *TYPE_ CONTENT* pour une valeur (exemple : 5, 16.3 etc...)
- *TYPE_ INDEX* pour un identificateur (exemple : id, @x, etc...)
- *TYPE_ OP* pour un opérateur (exemple : if, while, then, end, +, ×, etc...)

Pour le contenu d'une variable, nous avons défini un type énuméré *content* vu que le type du contenu peut être soit **int**, **float**, **string** ou **boolean** :

```
union content {
int e;
float f;
char* s;
int b;
}
```

La dernière variable b désigne le contenu d'un booléen : elle prend 1 si c'est vrai, 0 sinon.

Un symbole est donc représenté sous la forme d'une structure de la manière suivante :

```
struct Node {
NodeEnum type;
int valuetype;
Content content;
int index;
OpNode op;
};
```

avec :
- *valuetype* est égal à :
  - 0 pour *int*

  - 1 pour *float*
  - 2 pour *string*
  - 3 pour *boolean*
  - -1 pour *undefined*
- *index* est le numéro du symbole dans la table. Cela permet de le retrouver facilemnt.
- *op* est la structure de la table des symboles.

Actions sur les symboles

Les actions réalisées sur les symboles ont pour but de créer, initialiser et afficher un symbole :

- NewNodeInt (int) → Node* : cette fonction crée un nouveau symbole ayant une valeur de type entier. De la même manière, nous avons implémenté les fonctions *NewNodeFloat, NewNodeBoolean, NewNodeString* et *NewNodeVide*.
- NodeFree (Node*) → void
- NodePrint(Node*) → void
- opr_node (int type, char opr, Node*a, Node* b) → Node* : effectue l'opération *opr* entre les deux symboles a et b.

2.2 Les expressions

Une expression est définie comme suit :

```
struct OpNode {
int name;
int num;
Node* node[1];
}
```

avec :
- *name* le nom de l'identificateur au début de l'expression.
- *num* le nombre de noeuds dans l'expression.
- *node* le pointeur sur le premier noeud dans l'expression.

Exemple :
Pour l'expression if (x == 5) || (y < 8) then z = 5.5 end, nous avons le schéma suivant :

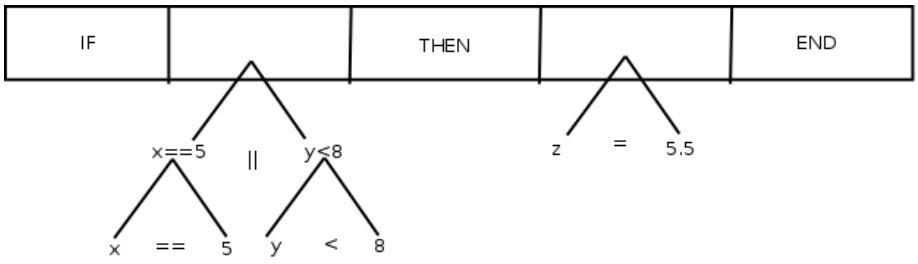

Figure 2 – exemple

3 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.

3.1 Table des symboles

Pour les opérateurs $+$, $\times$ et *cmp*, voici les tables donnant pour chaque type d'opérande, le type du résultat :

+	I	F	S	B
I	I	F		
F	F	F		
S			S	
B				

*	I	F	S	B
I	I	F		
F	F	F		
S				
B				

cmp	I	F	S	B
I	B	B		
F	B	B		
S			B	
B				

3.2 Table des expressions

Chaque expression est constituée d'un *noeud* résultat, un *opérateur*, un *noeud* gauche et un *noeud* droit.

4 Production du code

La génération du code intermédiaire du compilateur LLVM se fait sous forme de fichier texte. Le code assembleur LLVM est structuré en fonctions et utilise des variables locales préfixées par % et des variables globales préfixées par @. Les types utilisés sont **i32** pour un entier et **float** pour un réel.

4.1 Traitement des instructions arithmétiques et logiques

Toute construction d'expression du langage est associée à une expression sémantique où un registre est créé pour contenir le résultat de l'évaluation d'une sous-expression.

Tout d'abord, nous avons détecté le type de données d'entrée :
- Pour *int*, nous avons utilisé **i32**.
- Pour *float*, nous avons utilisé **float**.
- Pour *boolean*, nous avons utilisé **i1**.

Pour les opérations arithmétiques, nous commençons par distinguer le type des variables :
- Pour un entier, nous utilisons **add, sub, mul, div**.
- Pour un réel, nous utilisons **fadd, fsub, fmul, fdiv**.

Exemple :
2+3 donne %r0 = add i32 2,i32 3 ;
5.3 + 2.1 donne %r0 = fadd float 5.3, float 2.1 ;

Pour les opérations logiques, nous avons utilisé **icmp** pour la comparaison des entiers et **fcmp** pour les réels. La syntaxe générale des opérations logiques s'écrit comme suit :
<résultat> = <icmp ou fcmp> <opération> <type : i32 ou float> <op1>, <op2>
Le résulat est un *boolean*, donc de type **i1**.

Exemple :
%r0 = icmp eq i32 3, 2 ;

4.2 Traitement des conditionnelles et des boucles *for* et *while*

Pour les conditionnelles, nous avons utilisé **select** :
<resultat> = select i1 <condition>, <type> <val1>, <type> <val2>

Si *condition* = vrai, *résultat* prend la valeur *val1*, sinon il prend la valeur *val2*.

Exemple :
%r1 = select i1 %r1, i32 %r2, i32 %r3 ;

Pour les boucles *for* et *while*, nous faisons un appel récursif aux conditionnelles et appelons *select* plusieurs fois.

5 Tests

5.1 Test des instructions arithmétiques et logiques

Code Source

```
6+9
(9+8)*(7-3)
@a = 3
@b = 4
@a + @b
```

Code Cible

```
@str = constant[7 x i8] c"=> %d\0A\00"
declare i32 @printf(i8*, ...)

define i32 @main(){
%r1 = add i32 6, 9
call i32 (i8*, ...)*@printf(i8* getelementptr ([7 x i8]* @str, i32 0, i32 0), i32 %r1)
%r2 = sub i32 7, 3
%r3 = add i32 9, 8
%r4 = mul i32 17, 4
call i32 (i8*, ...)*@printf(i8* getelementptr ([7 x i8]* @str, i32 0, i32 0), i32 %r4)
%r5 = add i32 3, 4
call i32 (i8*, ...)*@printf(i8* getelementptr ([7 x i8]* @str, i32 0, i32 0), i32 %r5)
ret i32 0;
}
```

Resultat

```
=> 15
=> 68
=> 7
```

5.2 Test des conditionnelles

Code Source

```
if ( 5 < 7 ) then return 8
else return 9
end

if ( 5 > 7 ) then return 8
else return 9
end
```

Code Cible

```
@str = constant[7 x i8] c"=> %d\0A\00"
declare i32 @printf(i8*, ...)

define i32 @main(){
%ptr0 = alloca i32
store i32 8, i32* %ptr0
%ptr1 = alloca i32
store i32 9, i32* %ptr1
%r1 = icmp ult i32 5, 7
%r2 = load i32* %ptr0
%r3 = load i32* %ptr1
%r4 = select i1 %r1,i32 %r2,i32 %r3
call i32 (i8*, ...)*@printf(i8* getelementptr ([7 x i8]* @str, i32 0, i32 0), i32 %r4)
%ptr2 = alloca i32
store i32 8, i32* %ptr2
%ptr3 = alloca i32
store i32 9, i32* %ptr3
%r5 = icmp ugt i32 5, 7
%r6 = load i32* %ptr2
%r7 = load i32* %ptr3
%r8 = select i1 %r5,i32 %r6,i32 %r7
call i32 (i8*, ...)*@printf(i8* getelementptr ([7 x i8]* @str, i32 0, i32 0), i32 %r8)
ret i32 0;
}
```

Resultat

```
=> 8
=> 9
```

Conclusion

Au cours de ce projet, nous avons pu mettre en oeuvre les connaissances acquises au cours des séances de cours et de TD et les concepts élémentaires de compilation de langage de programmation.

Nous nous sommes familiarisées avec des outils d'analyse lexicale tel LEX et d'analyse syntaxique tel YACC.