INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

PLD-COMPILATEUR - Documentation pour developpeurs

Hexanôme 4244
ALAMI Meryem
AL ZAHABI Hanaa
BELAHCEN Basma
CHELLAOUI Adam
GREVAUD Paul
GUILLEVIC Marie
M'BARECK Aichetou
PREVOT Jade
WAQIF Salma

10 avril 2022

Table des matières

1 Utilisation du compilateur **3**

2 Description de la grammaire **4**
2.1 Les expression régulières utilisées . . . 4
2.2 Symboles Terminaux . . . 4
2.3 Symboles Non Terminaux et Règles de production . . . 5
2.3.1 prog (axiome de la grammaire) . . . 5
2.3.2 instr . . . 5
2.3.3 declaration . . . 5
2.3.4 type . . . 6
2.3.5 functionCall . . . 6
2.3.6 enddeclaration . . . 6
2.3.7 affectation . . . 6
2.3.8 lvalue . . . 7
2.3.9 expression . . . 7
2.3.10 if_then_else . . . 8
2.3.11 whileloop . . . 8
2.3.12 block . . . 8
2.3.13 blockConditionWhile . . . 8
2.3.14 return-stmt . . . 8

3 Navigation dans le code **9**
3.1 Code source . . . 9
3.1.1 symbol . . . 9
3.1.2 symbolTable . . . 9
3.1.3 functionTable . . . 9
3.1.4 fonction . . . 9
3.1.5 CodeGenVisitor . . . 9
3.1.6 IR . . . 10
3.1.7 IRVisitor . . . 10
3.1.8 ValeurVisitor . . . 10

4 Améliorations **10**

Table des figures

1 Début de la grammaire 5
2 Grammaire : instructions 5
3 Grammaire : déclaration 5
4 Grammaire : type 6
5 Grammaire : functionCall 6
6 Grammaire : enddéclaration 6
7 Grammaire : affectation d'une variable 6
8 Grammaire : lvalue 7
9 Grammaire : expressions 7
10 Grammaire : if_then_else 8
11 Grammaire : while loop 8
12 Grammaire : block 8
13 Grammaire : block Condition While 8
14 Grammaire : return_stmt 8

1 Utilisation du compilateur

Pour utiliser notre compilateur ifcc, vous aurez besoin d'un environnement Linux et un processeur Intel comme nous utilisons le langage assembleur x86.

Il faut également avoir une installation Antlr4 valide sur votre ordinateur qui dépend de l'environnement sur lequel vous travaillez :

- Sur VMWare Horizon avec Linux-2D : vous n'avez pas besoin de faire grand chose, tout est installé. Pour utiliser le compilateur, executer la commande make ou le scipt suivant : ./compiler/runmake-fedora.sh.
- Sur MacOS : utilisez le script ./install-antlr.sh. Pour compiler, utilisez la /commande make dans le dossier /compiler.
- Sur Ubuntu ou WSL/Ubuntu : installez les paquets indiqués dans le début du fichier nommé ./compiler/runmake-ubuntu.sh. Utilisez la commande ./compiler/runmake-ubuntu.sh pour compiler. Attention sous WSL, pour pouvoir utiliser la fonctionnalité d'affichage de l'arbre de dérivation, il faudra installer le paquet GWSL de Microsoft Store.
- Autre distribution Linux : regardez si la distribution offre des paquets pour Antlr4 et Antlr4-runtime, si c'est le cas vous éviterez de tout recompiler. Par exemple sous Fedora, il s'agit des paquets antlr4, antlr4-cpp-runtime et antlr4-cpp-runtime-devel. Sinon lancez le script ./install-antlr.sh.

Si les tests ne sont pas corrects, essayez la commande suivante : "dos2unix ifcc-wrapper.sh". Vous devrez peut-être installer dos2unix avec la commande "sudo apt dos2unix".

Pour les tests, rendez vous dans le dossier /testfiles. Vous pourrez exécuter les tests avec les commandes suivantes :

- Sur VMWare Horizon avec Linux-2D : ./ifcc-test.py testfiles/
- Sur WSL/Ubuntu : python3 ifcc-test.py testfiles/

Nos tests sont répartis en 4 dossiers différents :

- test-code-correct : Ce sont les tests qui génèrent l'assembleur pour des programmes valides.
- test-comportement-compilateur : ce sont les tests dont le comportement est indéfini.
- test-erreurs : ce sont les tests qui produisent des erreurs du fait que le résultat de l'assembleur généré par le compilateur ne correspond pas au résultat de l'assembleur généré par gcc.
- test-pas-implemente : Ce sont des classes qui testent des fonctionnalités que nous n'avons pas implémenté essentiellement par manque de temps.

Nous allons vous présenter la structure de notre code pour que vous ayez une bonne idée de son fonctionnement.

2 Description de la grammaire

Pour définir la grammaire, on commence par définir les terminaux qui constituent l'alphabet puis les non-terminaux . Ces non-terminaux sont formés à partir d'une suite de terminaux et de non-terminaux. Les règles produites par cette grammaire définissent le fonctionnement voulu du compilateur.

2.1 Les expression régulières utilisées

```
RETURN : 'return' ;
CONST : [0-9]+ ;
VAR : [a-zA-Z]+;
CHARACTER : '\''.?'\'';
COMMA : ',';
COMMENT : '/*' .*? '*/' -> skip ;
DIRECTIVE : '#' .*? '\n' -> skip ;
WS    : [ \t\r\n] -> channel(HIDDEN);
```

2.2 Symboles Terminaux

```
INT : 'int'
CHAR : 'char'
OPENPAR : '('
CLOSEPAR : ')'
SEMICOLON : ';'
OPENBRACKET : '{'
CLOSEBRACKET : '}'
EQUAL : '='
PLUS : '+'
MINUS : '-'
MULTIPLY : '*'
DIVIDE : '/'
OPM: MULTIPLY | DIVIDE
OPA: PLUS | MINUS
EXCLA: '!'
AND: '&'
OR: '|'
XOR: '^'
ISEQUAL: '=='
ISDIFFERENT: '!='
GREATER: '>'
SMALLER: '<'
IF: 'if'
ELSE: 'else'
WHILE: 'while'
```

2.3 Symboles Non Terminaux et Règles de production

2.3.1 prog (axiome de la grammaire)

```
prog → int VAR () {instr*}
prog → int VAR (int VAR (COMMA int VAR)*) {instr*}
```

```
3   axiom : prog* ;
4
5
6   prog : INT VAR OPENPAR CLOSEPAR OPENBRACKET instr*  CLOSEBRACKET
7       |  INT VAR OPENPAR INT VAR (COMMA INT VAR)* CLOSEPAR OPENBRACKET instr*  CLOSEBRACKET;
```

FIGURE 1 – Début de la grammaire

2.3.2 instr

```
instr → declaration
instr → functionCall
instr → if_then_else
instr → whileloop
instr → affectation
instr → return_stmt
```

```
9   instr : declaration #declarationInstr
10  | functionCall #functionCallInstr
11  | if_then_else #if_then_elseInstr
12  | whileloop #whileloopInstr
13  | affectation SEMICOLON #affectationInstr
14  | return_stmt #return_stmtInstr ;
```

FIGURE 2 – Grammaire : instructions

2.3.3 declaration

```
declaration → int variables* enddeclarion ;
```

```
16  declaration: type variables* enddeclaration SEMICOLON;
```

FIGURE 3 – Grammaire : déclaration

2.3.4 type

`type → int`
`type → char`

```
18   type: INT #int
19   | CHAR #char;
```

FIGURE 4 – Grammaire : type

2.3.5 functionCall

`functionCall → VAR ((expression COMMA)* expression) ;`

```
21   functionCall: VAR OPENPAR (expression COMMA)* expression CLOSEPAR SEMICOLON;
```

FIGURE 5 – Grammaire : functionCall

2.3.6 enddeclaration

`enddeclaration → lvalue`
`enddeclaration → affectation`

```
26   enddeclaration: lvalue #enddeclvar
27   | affectation #enddeclaffect;
```

FIGURE 6 – Grammaire : enddéclaration

2.3.7 affectation

`affectation → lvalue = expression`

```
29   affectation: lvalue EQUAL expression;
```

FIGURE 7 – Grammaire : affectation d'une variable

2.3.8 lvalue

```
lvalue → VAR
lvalue → VAR [ expression ]
```

```
31  lvalue: VAR #lvalVar
32  | VAR OPENSQBRACKETS expression CLOSESQBRACKETS #lvaltableau;
```

FIGURE 8 – Grammaire : lvalue

2.3.9 expression

```
expression → (expression)
expression → - expression
expression → ! expression
expression → expression (MULTIPLY | DIVIDE) expression
expression → expression (PLUS | MINUS) expression
expression → VAR ((expression COMMA)* expression)
expression → expression & expression
expression → expression ^ expression
expression → expression | expression
expression → expression (GREATER|SMALLER) expression
expression → expression (ISEQUAL|ISDIFFERENT) expression
expression →  VAR [ expression ]
expression → affectation
expression → VAR
expression → CHARACTER
expression → CONST
```

```
34  expression: OPENPAR expression CLOSEPAR #par
35  | MINUS expression #oppose
36  | EXCLA expression #negation
37  | expression (MULTIPLY | DIVIDE) expression #multdiv
38  | expression (PLUS | MINUS) expression #plusminus
39  | VAR OPENPAR (expression COMMA)* expression CLOSEPAR #funcCall
40  | expression AND expression #andlogiq
41  | expression XOR expression #xorlogiq
42  | expression OR expression #orlogiq
43  | expression (GREATER | SMALLER) expression #inequality
44  | expression (ISEQUAL | ISDIFFERENT) expression #equality
45  | VAR OPENSQBRACKETS expression CLOSESQBRACKETS #valTableau
46  | affectation #exprAffectation
47  | VAR #var
48  | CHARACTER #character
49  | CONST #const;
```

FIGURE 9 – Grammaire : expressions

2.3.10 if_then_else

`if_then_else → if (expression) {block} else {block}`

```
51  if_then_else : IF OPENPAR expression CLOSEPAR OPENBRACKET blockthen=block
52                 CLOSEBRACKET ELSE OPENBRACKET blockelse=block CLOSEBRACKET ;
```

FIGURE 10 – Grammaire : if_then_else

2.3.11 whileloop

`whileloop → while ( blockConditionWhile) {block}`

```
54   whileloop : WHILE OPENPAR blockConditionWhile CLOSEPAR OPENBRACKET blockwhile=block CLOSEBRACKET ;
```

FIGURE 11 – Grammaire : while loop

2.3.12 block

`block → instr*`

```
56   block: instr*;
```

FIGURE 12 – Grammaire : block

2.3.13 blockConditionWhile

`blockConditionWhile → expression`

```
58   blockConditionWhile: expression ;
```

FIGURE 13 – Grammaire : block Condition While

2.3.14 return-stmt

`return_stmt → RETURN expression ;`

```
60   return_stmt : RETURN expression SEMICOLON;
```

FIGURE 14 – Grammaire : return_stmt

3 Navigation dans le code

3.1 Code source

En plus du code généré dans le dossier generated, notre code se compose de plusieurs classes (.h et .cpp) différentes.

Nous disposons de deux visiteurs parcourant l'arbre produit avec la grammaire et le code c :

— Un visiteur ValeurVisiteur permettant d'effectuer des calcules sur des constantes. Nous l'utilisons pour l'instant uniquement pour gérer la taille des tableaux (ex int b[5*9])

— Un visiteur IRVisitor permettant de générer les instructions, les blocks et le cfg (classes expliquées par la suite).

Le code se base sur des tables de symboles et de fonctions.

3.1.1 symbol

Cette classe représente un symbole avec son nom, son type, sa position par rapport à rbp, la ligne où il a été déclaré et s'il a été utilisé ou non.

3.1.2 symbolTable

Cette classe permet de représenter la table des symboles et de retourner toutes les informations nécessaires sur les symboles qui y sont stockés.

3.1.3 functionTable

Cette classe prend comme attribut une map et permet de représenter la table des fonctions stockée et de retourner toutes les informations nécessaires sur les fonctions déclarées qui y sont stockés.
On peut y ajouter des fonctions, en retirer ou accéder aux différents attributs.

3.1.4 fonction

Cette classe permet de définir une fonction avec son nom, son type de retour et le nombre d'arguments qu'elle possède. C'est ce qui nous permet lors de la déclaration d'une fonction de voir si celle ci existe déjà ou pas dans la functionTable.

3.1.5 CodeGenVisitor

Cette classe permet de générer le code assembleur en visitant l'arbre généré. Chaque visiteur de non-terminal ira visiter les non-terminaux ou les terminaux contenus dans la partie droite de sa règle et générera le code assembleur associé. (deprecated)

3.1.6 IR

L'implémentation de IR a été faite grâce à la classe IR. Celle ci est composée des 3 classes suivantes :
— IRInstr : C'est la plus petite granularité qui nous permet de faire des opérations comme l'addition (add) ou la soustraction (sub) ou meme l'appel de fonction (call)..
— BasicBlock : C'est un bloc composé d'une suite d'instructions
— CFG : C'est le graphe formé de plusieurs Basic Blocks. On génère un CFG pour chaque fonction.

3.1.7 IRVisitor

Ce visiteur permet de générer les instructions (IRInstr), les blocks (BasicBlock) et le Control Graph Flow (CFG) en visitant l'arbre généré par antlr.

3.1.8 ValeurVisitor

Cette classe sert à explorer chaque symbole non terminal de la grammaire et faire les calculs nécessaires pour retourner le résultat recherché.

4 Améliorations

Par rapport aux améliorations qu'il nous aurait été possible de réaliser pour notre projet, il en est ressorti quelques-unes, dont la liste n'est pas exhaustive. En effet, nous aurions pu ajouter de l'ARM et du MSP430, mais aussi gérer la portée des variables dans les blocs. La propagation des variables et des constantes, ainsi que la gestion des pointeurs, figurent aussi parmi les points dont l'ajout aurait été envisageable. Aussi, nous aurions pu optimiser notre code pour améliorer l'efficacité de notre programme. Vous pouvez regarder la documentation utilisateur où les fonctionnalités non implémentées sont précisées.

Il existe également quelques bugs non réglés dans notre code. Les appels de fonctions récursives peuvent provoquer des segmentation fault du à un overwrite des paramètres appelés. La gestion des char n'est pas optimale, cela est dû à notre implémentation de variables de tailles fixées à 64bits.