

PLD-COMPILATEUR - RENDU MI PARCOURS

Hexanôme 4244

ALAMI Meryem

AL ZAHABI Hanaa

BELAHCEN Basma

CHELLAOUI Adam

GUILLEVIC Marie

M'BARECK Aichetou

PREVOT Jade

WAQIF Salma

27 mars 2022



Table des matières

1	Description de la grammaire	3
1.1	Les expression régulières utilisées :	3
1.2	Symboles Terminaux	3
1.3	Symboles Non Terminaux et Règles de production	3
1.3.1	prog (axiome de la grammaire)	3
1.3.2	instr	4
1.3.3	declaration	4
1.3.4	variables	4
1.3.5	affectation	4
1.3.6	expression	4
1.3.7	return_stmt	5
1.3.8	value	5
2	Fonctionnalités implémentées	5
3	Navigation dans le code	6
3.1	Code source	6
3.1.1	symbol	6
3.1.2	symbolTable	6
3.1.3	CodeGenVisitor	6
3.2	Tests	6
4	Gestion de projet	7
4.1	Déroulé première partie	7
4.1.1	Première séance	7
4.1.2	Deuxième séance	7
4.1.3	Troisième séance	7
4.1.4	Quatrième séance	7
4.2	Déroulé deuxième partie	7
4.2.1	Première séance	8
4.2.2	Deuxième séance	8
4.2.3	Troisième séance	8
4.2.4	Quatrième séance	8
4.3	Outils utilisés	8

Table des figures

1	Début de la grammaire	3
2	Grammaire : instructions	4
3	Grammaire : déclaration	4
4	Grammaire : variables ajoutées à la déclaration	4
5	Grammaire : affectation d'une variable	4
6	Grammaire : expressions	5
7	Grammaire : <code>return_stmt</code>	5
8	Grammaire : <code>return</code>	5

1 Description de la grammaire

Pour définir la grammaire, on commence par définir les terminaux puis les non-terminaux qui constituent l'alphabet. 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.

1.1 Les expression régulières utilisées :

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

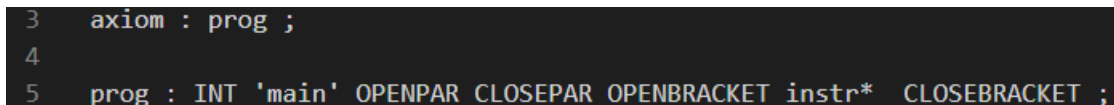
1.2 Symboles Terminaux

```
INT : 'int' ;
CHAR : 'char' ;
OPENPAR : '(' ;
CLOSEPAR : ')' ;
SEMICOLON : ';' ;
OPENBRACKET : '{' ;
CLOSEBRACKET : '}' ;
EQUAL : '=' ;
PLUS : '+' ;
MINUS : '-' ;
MULTIPLY : '*' ;
DIVIDE : '/' ;
OPM: MULTIPLY | DIVIDE;
OPA: PLUS | MINUS;
```

1.3 Symboles Non Terminaux et Règles de production

1.3.1 prog (axiome de la grammaire)

`prog → int 'main' () {instr*}`



```
3  axiom : prog ;
4
5  prog : INT 'main' OPENPAR CLOSEPAR OPENBRACKET instr* CLOSEBRACKET ;
```

FIGURE 1 – Début de la grammaire

1.3.2 instr

instr → declaration
instr → affectation
instr → return_stmt

```
6  instr : declaration #declarationInstr
7      | affectation #affectationInstr
8      | return_stmt #return_stmtInstr ;
```

FIGURE 2 – Grammaire : instructions

1.3.3 declaration

declaration → int variables* VAR ;

```
9  declaration: INT variables* VAR SEMICOLON;
```

FIGURE 3 – Grammaire : déclaration

1.3.4 variables

variables → VAR COMMA

```
10 variables: VAR COMMA;
```

FIGURE 4 – Grammaire : variables ajoutées à la déclaration

1.3.5 affectation

affectation → VAR = expression ;

```
11 affectation: VAR EQUAL expression SEMICOLON;
```

FIGURE 5 – Grammaire : affectation d'une variable

1.3.6 expression

expression → (expression)
expression → expression (MULTIPLY | DIVIDE) expression
expression → expression (PLUS | MINUS) expression

```
expression → VAR  
expression → (MINUS) expression  
expression → CONST
```

```
12 expression: OPENPAR expression CLOSEPAR #par  
13 | expression (MULTIPLY | DIVIDE) expression #multdiv  
14 | expression (PLUS | MINUS) expression #plusminus  
15 | VAR #var  
16 | (MINUS) expression #oppose  
17 | CONST #const;
```

FIGURE 6 – Grammaire : expressions

1.3.7 return_stmt

```
return\_stmt → RETURN expression ;\\
```

```
18 return_stmt : RETURN expression SEMICOLON;
```

FIGURE 7 – Grammaire : *return_sstmt*

1.3.8 value

```
value → CONST\\  
value → expression\\
```

```
19 value : CONST | expression;
```

FIGURE 8 – Grammaire : return

2 Fonctionnalités implémentées

Jusqu'à présent, nous avons implémenté quelques fonctionnalités correspondant à notre grammaire.

Pour les fonctions, nous n'avons implémenté que la fonction main qui ne prend pas d'arguments.

En ce qui concerne les déclarations, nous n'utilisons qu'un seul type : **int**. De plus, notre grammaire nous permet de déclarer en une ligne plusieurs variables les unes à la suite des autres (ex `int a, b;`). En revanche, nous ne mélangeons pas encore les affectations et déclarations. C'est-à-dire que pour l'instant nous ne pouvons pas avoir une expression du type : **int a = 4;**

On peut affecter à une variable une expression qui peut prendre plusieurs formes (une constante, une autre variable etc...).

Les expressions implémentées concernent les additions, soustractions, multiplications et divisions tout en faisant attention à la priorité des opérations. Elles incluent aussi la priorité des parenthèses.

Nous pouvons également effectuer le moins unaire sur une variable (ex : **a = -b**).

Nous avons également inclus la gestion de deux erreurs et d'un warning :

- Si une variable est déclarée 2 fois ou plus : il y a une erreur.
- Si une variable est utilisée sans être déclarée : il y a une erreur.
- Si une variable est déclarée mais jamais utilisée : il y a un warning.

3 Navigation dans le code

3.1 Code source

En plus du code généré dans le dossier generated, notre code se compose en 3 classes différentes.

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 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.

3.2 Tests

Nos tests se divisent en 2 parties : les tests qui génèrent l'assembleur pour des programmes valides et les tests qui génèrent une erreur pour des programmes non valides.

4 Gestion de projet

Pour la gestion de projet, nous avons décidé de travailler par cycle entre chaque séance de cours.

4.1 Déroulé première partie

Pour la première partie du projet, il fallait se concentrer sur la réalisation d'un compilateur minimal qui se compose d'un analyseur syntaxique (parser) et d'un générateur de code. Pour que chacun maîtrise bien les bases, nous avons décidé de travailler en groupe. Etant 9 dans l'hexanôme, nous nous sommes divisés en 2 groupes travaillant en parallèle sur les mêmes tâches.

4.1.1 Première séance

Pendant la première séance, nous avons commencé par installer antlr. Nous avons par la suite procédé à la prise en main du squelette de code fourni.

4.1.2 Deuxième séance

Pendant la deuxième séance, chaque groupe de son côté a travaillé sur les déclarations et affectations de variables. Nous avons donc réussi à l'issue de cette séance à déclarer plusieurs variables sur une même ligne. De plus, nous sommes également parvenus à affecter à chacune des variables déclarées une valeur (une constante ou une autre variable déclarée).

4.1.3 Troisième séance

Pendant la troisième séance, nous avons travaillé sur les expressions arithmétiques ainsi que sur le moins unaire. À l'issue de cette séance, nous avons réussi à réaliser les calculs nécessaires et donc générer le code assembleur pour les expressions arithmétiques de base (additions, soustractions, multiplications et divisions). De plus, nous avons veillé à respecter l'ordre des opérations (la multiplication et la division avant l'addition et la soustraction).

4.1.4 Quatrième séance

Pendant la quatrième séance, nous nous sommes consacrés en priorité à la finition des tâches précédentes et à la rédaction des livrables de mi-parcours. Nous avons aussi entamé la réalisation du compilateur final qui utilise l'IR. Pour pouvoir travailler le plus efficacement possible, nous nous sommes répartis les tâches en binôme ou trinôme.

4.2 Déroulé deuxième partie

Pour la deuxième partie, nous disposons de 4 séances pour réaliser notre compilateur final. La dernière étant dirigée vers les présentations orales, trois séances uniquement seront pleinement consacrées au développement du compilateur. Les tâches seront réparties par binômes ou trinômes. Avant chaque tâche, un binôme qui ne se charge pas du développement devra mettre en place des tests. Vous trouverez ci-dessous un exemple de répartition des tâches que nous essaierons de suivre.

4.2.1 Première séance

- Faire construire l'IR par le front-end.
- Transformer l'IR en assembleur x86 dans le back-end.
- Ajout du type char et de l'inférence de type.
- Mise en place de l'enregistrement d'activation, de l'ABI et des appels de fonction.

4.2.2 Deuxième séance

- Compiler le if ... else.
- Vérifications statiques sur les fonctions.
- Gestion du return n'importe où.
- Compiler les boucles while.

4.2.3 Troisième séance

- Compiler l'affectation à une lvalue quelconque.
- Compiler des tableaux.
- Compiler les appels de fonction ayant plus de 6 arguments.
- Propagation des variables constantes (avec analyse du data-flow).
- Finition des tâches du tableau.
- Préparation de la présentation orale.

4.2.4 Quatrième séance

- Présentation Orale.
- Rédaction des rendus.
- Corrections.

4.3 Outils utilisés

Nous utilisons plusieurs outils pour réaliser le projet :

- Notion pour la gestion de projet.
- Github pour le versionning.
- Visual Studio Code pour le développement.
- Overleaf pour la rédaction des rendus.