Projet compilation

Rapport intermédiaire de Projet : Projet compilation 2021

Auteurs :
Coline Chataing
Ahmed Ziani
Laury Thiebaux

Encadrants :
Suzanne Collin
Alexandre Bourbeillon
Sebastien Da Silva

12 janvier 2022

Table des matières

Chapitre 1

Introduction

Dans le cadre du projet de compilation, nous effectuons les premières étapes de création d'un compilateur pour le langage CIR-C, qui est un fragement du langage C. Pour réaliser ce compilateur, nous avons dû d'abord écrire une grammaire reconnaissant ce langage pour ensuite créer les méthodes de construction de des AST ainsi que la table des symboles et les contrôles sémantiques nécessaires. Ce compilateur devra également signaler par un message explicite les erreurs syntaxiques, sémantiques et lexicales rencontrées.

Chapitre 2

Construction des AST

2.0.1 Règles et labels

Depuis le rendu intermédiaire, nous avons appliqué des modifications minimes à notre grammaire, mais surtout utilisé des labels pour différencier les alternatives à une même règle.

2.0.2 Arbres Abstraits obtenus à la partie 1

Après avoir défini la grammaire, ANTLR nous a permis d'obtenir les arbres syntaxiques de nos exemples. Cependant, ils contiennent une multitude de noeud inutiles, et nous voudrions pouvoir obtenir des arbres ne contenant que les informations nécessaires.

Un AST est un arbre dont les nœuds internes sont marqués par des opérateurs et dont les feuilles (ou nœuds externes) représentent les opérandes de ces opérateurs. Autrement dit, généralement, une feuille est une variable ou une constante.

Un AST diffère d'un arbre d'analyse par l'omission des nœuds et des branches qui n'affectent pas la sémantique d'un programme. Un exemple classique est l'omission des parenthèses de groupement puisque, dans un AST, le groupement des opérandes est explicité par la structure de l'arbre.

2.0.3 Les Visiteurs et GraphVizVisitor

Pour la construction de l'AST, nous avons opté pour le Design Pattern Visitor ; son but est de pouvoir facilement parcourir une structure d'objets complexes.
Le principe est assez simple : chaque élément de notre structure (les noeuds de notre arbre) peut accepter un visiteur et doit appliquer le visiteur à chaque noeud fils. Charge au visiteur de faire ce qu'il veut à chaque étape.

Pour implémenter ce Design Pattern, il nous faut tout d'abord définir une interface Ast où nous allons mentionner toutes les classes représentant notre structure complexe.
Pour représenter notre arbre graphiquement, nous utilisons encore un visiteur ; GraphVizVisitor, nous avons défini une interface AstVisitor qui permet de visiter les noeuds de type Ast. De plus, l'interface Ast implémente déjà la méthode accept sur ce visiteur, qui génère un fichier .dot valide représentant l'AST passé en argument et l'enregistre dans le fichier out/tree.dot. Nous utilisons également la librairie GraphViz : Cette librairie permet de simplement représenter des diagrammes en boite (diagramme UML, graphe de dépendance...).

Pour visualiser le fichier, on exécute simplement la commande : dot -Tsvg ./out/tree.dot -o ./out/tree.svg. Cela crée un fichier svg qu'on peut visualiser avec n'importe quelle visionneuse d'images.

2.0.3.1 Diagramme de classe

FIGURE 2.1 – Diagramme de classes du package ast

2.0.4 Un exemple détaillé de création de noeuds : déclaration de variables

```
10  decl :
11      decl_typ
12      | decl_fct
13      | decl_vars     ;
14
15  decl_vars :
16       'int' (IDENT',')* IDENT ';'                          #Decla
17      | 'struct' IDENT IDENT* ('(' '*' IDENT ')'',')* ';' #Struct
18      | 'int' IDENT '=' ENTIER';'                          #DeclaAffect ;
19
```

FIGURE 2.2 – Un extrait de notre grammaire

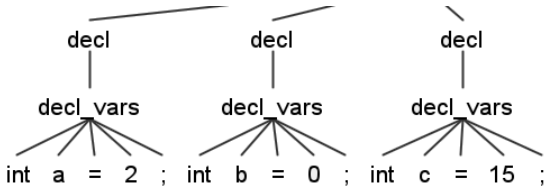

FIGURE 2.3 – l'arbre syntaxique pour les déclarations de variables

On observe tout d'abord que l'enchainement du noeud **decl** et **decl_vars** est inutile. En CIR-C, on ne manipule que des entiers, donc la feuille **int** est superflue. Enfin, les feuilles = et **;** sont aussi inutiles.

```
36      @Override
37      public Ast visitDecl(circParser.DeclContext ctx) {
38          return ctx.getChild(0).accept(this);
39      }
40
```

FIGURE 2.4 – méthode visitDecl

Commmentaire sur **visitDecl** : Le noeud decl est supperflu, on utilise donc directement la méthode **accept**, et on ne crée pas de classe **Decl**.

```
@Override
public Ast visitDeclaAffect(circParser.DeclaAffectContext ctx) {
    String identString = ctx.getChild(1).toString();
    String entierString = ctx.getChild(3).toString();

    Ident ident = new Ident(identString);
    Entier entier = new Entier(entierString);
    return new DeclaAffect(ident,entier);
}
```

FIGURE 2.5 – méthode visitDeclaAffect

Commentaire sur **visitDeclaAffect** : on commence par récupérer le terminal **IDENT** et **ENTIER**, qui sont en position 1 et 3 dans la règle **DeclaAffect** de circ.g4. On utilise la méthode .toString pour les transformer en chaîne de charactères.

```java
package ast;

public class Ident implements Ast {

    public <T> T accept(AstVisitor<T> visitor){
        return visitor.visit(this);
    }

    public String name;

    public Ident(String name){
        this.name = name;
    }
}

```

FIGURE 2.6 – classe Ident

```java
package ast;

public class Entier implements Ast {

    public <T> T accept(AstVisitor<T> visitor){
        return visitor.visit(this);
    }

    public String num;

    public Entier(String num){
        this.num = num;
    }
}
```

FIGURE 2.7 – classe Entier

Commentaire sur les classes **Ident** et **Entier** : Ces classes contiennent des champs nom et num qui contiennent respectivement le nom de l'identificateur et sa valeur numérique.

Commentaire sur la classe **DeclaAffect** : elle contient un champ ident et un champ entier qui seront les feuilles de notre AST.

2.0.5 GraphViz

```java
public String visit(DeclaAffect declaff){
        String nodeIdentifier = this.nextState();
        this.addNode(nodeIdentifier, "declaraffect");

        String a = declaff.ident.accept(this);
        String b = declaff.entier.accept(this);

        this.addTransition(nodeIdentifier, a);
        this.addTransition(nodeIdentifier, b);

        return nodeIdentifier;
    }
```

FIGURE 2.9 – une méthode visit de GraphViz

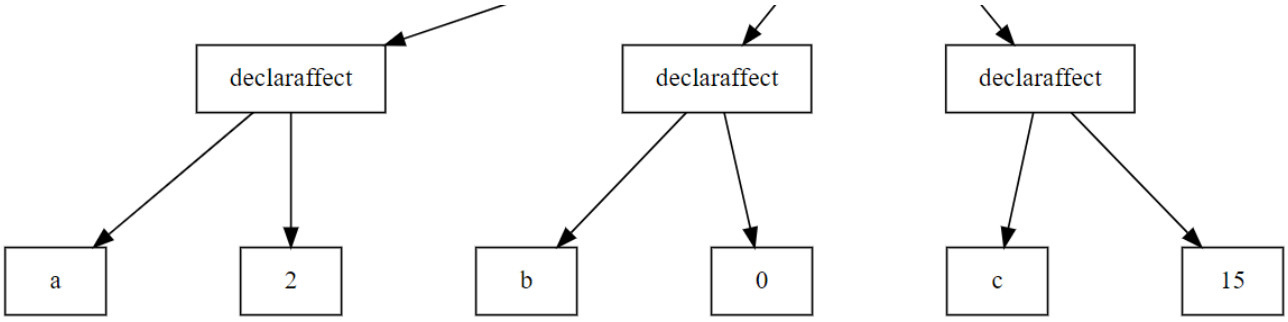

FIGURE 2.10 une méthode visit de GraphViz

Chapitre 3

Exemples d'AST

3.1 Code circ good.c

```c
int a = 2;
int b = 0;
int c = 15;

/* test de commentaire */

//test de commentaire

int main (){
if (a) {
    b=1+2+3*5+9;
}
else {
    b=2;
}
}
```

FIGURE 3.1 – Exemple de code reconnu

3.2 Arbre Abstrait good.c de la partie 1

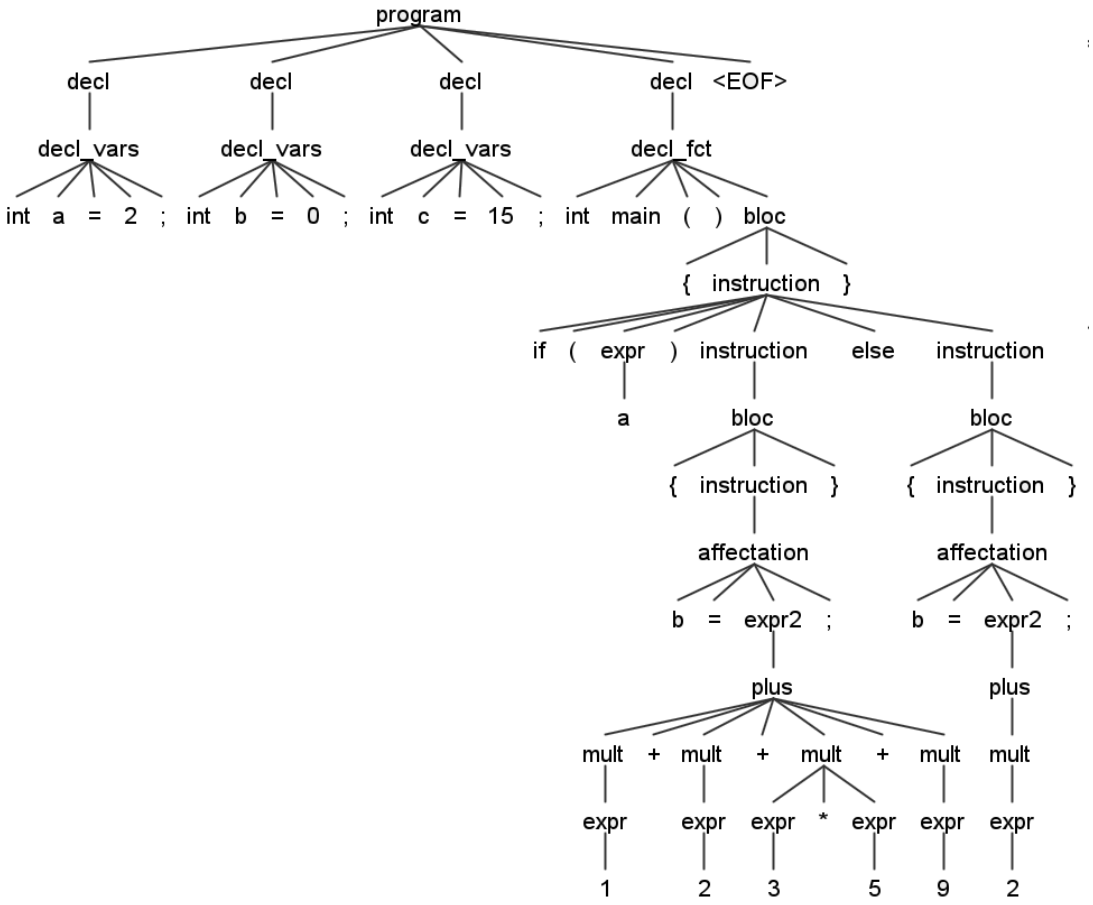

FIGURE 3.2 – Exemple de code reconnu

3.3 AST good.c

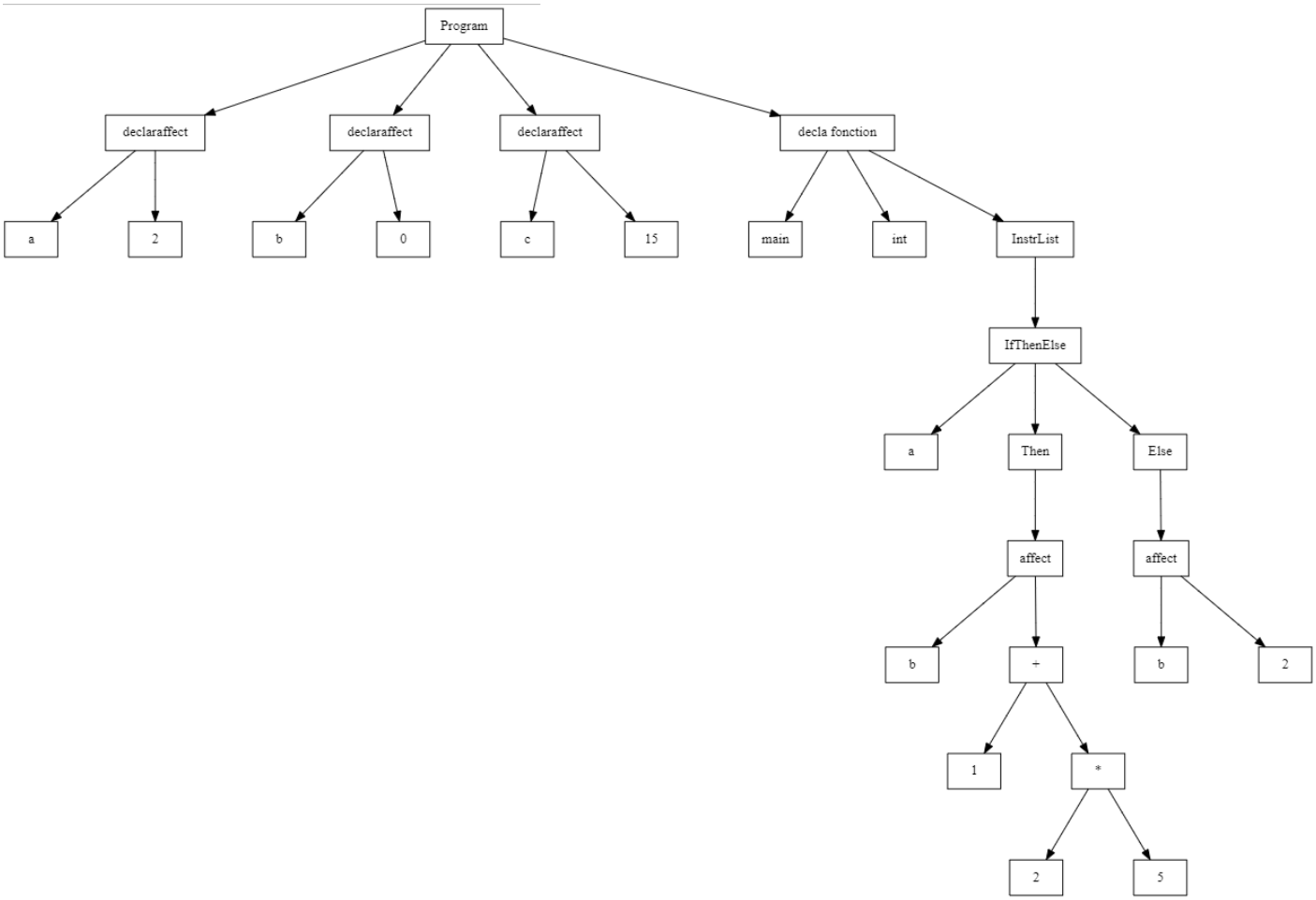

FIGURE 3.3 – Exemple de code reconnu

Chapitre 4

Gestion de projet

4.1 Répartition des tâches

La majorité du travail effectué pour ce rendu consistait en la rédaction des méthodes d'AstCreator et des classes implémentant AST, ainsi que les méthodes de GraphVizVisitor. Nous nous sommes répartis équitablement les méthodes et avont fait à chaque fois celle d'AstCreator et celle du GraphVizVisitor correspondant.

Méthodes	**Développeur**
visitProgram	Laury
visitDecl	Laury
visitExprSeule	Laury
visitIfThen	Laury
visitIfThenElse	Laury
visitWhile	Laury
visitBlocInstruct	Laury
visitAffectInstruct	Laury
visitReturn	Laury
visitAffectation	Laury
visitListe_decl_vars	Coline
visitListe_instruction	Coline
visitBloc	Coline
visitExpr2	Coline
visitPlus	Laury

visitMult	Laury
visitDecla	Coline
visitStruct	Coline
visitDeclaAffect	Coline
visitDecl_typ	Coline
visitIntParam	Coline
visitStructParam	Coline
visitStructParam	Coline
visitListe_param	Coline
visitEntier	Ahmed
visitTiretExpr	Ahmed
visitTiretExprExpr	Ahmed
visitSizeof	Ahmed
visitSizeofExpr	Ahmed
visitParenthExpr	Ahmed
visitParenthExprExpr	Ahmed
visitOperateur	Ahmed
visitOpExpr	Ahmed
visitOpExprExpr	Ahmed
visitListeExpr	Ahmed
visitIntNode	Ahmed
visitIdentExprPointeur	Ahmed
visitIdentExprPointeurExpr	Ahmed
visitExclaExpr	Ahmed
visitExclaExprExpr	Ahmed
visitFleche	Ahmed
visitFlecheExpr	Ahmed

4.1.1 Liste des tâches et temps

Ce tableau résume le temps passé par les différents membres du projet sur les différentes tâches. Ne sont pas incluses les réunions, et les séances de TP.

	Ahmed	Laury	Coline
Corrections de la grammaire	2	2	1
Ecriture des méthodes et classes	4	5	6
Ecriture GraphVizVisitor	3	5	2
Codes test	2	2	2
Rédaction du rapport LaTeXet gestion de projet	1	1	4
Total (en heures)	12	15	15

TABLE 4.1 – Répartition de travail

4.2 Annexe

4.2.1 Grammaire CIR-C

```
1
2  grammar circ;
3
4  @header{
5  package parser;
6  }
7
8  program :
9      decl* EOF;
10
11 decl :
12     decl_typ
13     | decl_fct
14     | decl_vars     ;
15
16 decl_vars :
17       'int' (IDENT',')* IDENT ';'                        #Decla
18     | 'struct' IDENT IDENT* ('(' '*' IDENT ')'',')* ';' #Struct
19     | 'int' IDENT '=' ENTIER';'                          #DeclaAffect ;
20
21 decl_typ :
22     'struct' IDENT '{' liste_decl_vars '}' ';';
23
24 decl_fct :
25     'int' IDENT '(' liste_param ')' bloc    #IntParam
26     | 'struct' IDENT '*' IDENT '(' liste_param ')' bloc    #StructParam ;
27
28 liste_param :
29     param*;
30
31 liste_expr : (expr',')* ;
32
33 param :
34     'int' IDENT                          #Paramint
35     | 'struct' IDENT  '*' IDENT          #Paramstruct;
36
37 expr :  ENTIER expr1                              #EntierExpr
38         |IDENT expr1                               #IdentExpr
39         | IDENT '(' liste_expr ')' expr1           #IdentExprPointeurExpr
40         | '!' expr expr1                           #ExclaExprExpr
41         | '-' expr expr1                           #TiretExprExpr
42         | 'sizeof' '(' 'struct' IDENT ')' expr1 #SizeofExpr
43         | '(' expr ')' expr1                       #ParenthExprExpr
44         | ENTIER                                   #Entier
45         | IDENT                                    #Ident
46         | IDENT '(' liste_expr ')'                 #IdentExprPointeur
47         | '!' expr                                 #ExclaExpr
48         | '-' expr                                 #TiretExpr
49         | 'sizeof' '(' 'struct' IDENT ')'          #Sizeof
50         | '(' expr ')'                             #ParenthExpr ;
51
52 expr1 : '->' IDENT expr1    #FlecheExpr
53         |   '->' IDENT         #Fleche
54         | OPERATEUR expr expr1     #OpExprExpr
55         | OPERATEUR expr    #OpExpr
56         ;
57
58 instruction :
59      expr ';' #ExprSeule
60     | 'if' '(' expr ')' instruction #IfThen
61     | 'if' '(' expr ')' instruction 'else' instruction #IfThenElse
62     | 'while' '(' expr ')' instruction #While
63     | bloc  #BlocInstruct
64     | affectation #AffectInstruct
65     | 'return' expr ';'    #Return ;
66
67 affectation : IDENT '=' expr2 ';' ;
68
69 liste_decl_vars :
70     decl_vars*;
71
72 liste_instruction :
```

```
    instruction*;

bloc :
    '{' liste_decl_vars liste_instruction '}';

OPERATEUR : '=' | '==' | '!=' | '<' | '<=' | '>' | '>=' | '+' | '-' | '*' | '/' | '&&' | '||';

ENTIER : '0' | ('1'..'9') ('0'..'9')* | CARACTERE;
IDENT : ('a'..'z'|'A'..'Z')('a'..'z'|'A'..'Z'|'0'..'9'|'_')*;
CARACTERE : '\''('!'|'#'|'$'|'%'|'('|')'|';'|'+'|','|'-'|'.'|'&'
            | ('0'..'9')|':'|';'|'<'|'='|'>'|'?'|'@'|'['|']'|'^'
            | '_'|'`'|'{'|'}'|'~'|'a'..'z'|'A'..'Z'|'/'|'|')'\''
            ;
WS : ('\n'|' '|'\t'|'\r'|'/*' .*? '*/' | '//' ~[\r\n]*)+ -> skip;

expr2 : plus;

plus:  mult (('+'|'-') mult)*;

mult : expr (('*'|'/') expr)*;
```