Projet compilation

Rapport intermédiaire de Projet : Projet compilation 2021

Auteurs :
Coline Chataing
Ahmed Ziani
Laury Thiebaux

Encadrants :
Suzanne Collin
Alexandre Bourbeillon
Sebastien Da Silva

10 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 rapport d'activité traite de la construction de la TDS et des contrôles sémantiques, ce qui constitue la dernière partie de ce projet pour notre groupe.

Chapitre 2

Construction de la Table des Symboles

2.1 Définition et contenu d'une TDS

Une table des symboles est une structure créée au début de l'analyse sémantique, remplie au fur et à mesure de cette dernière. Elle centralise les informations liées aux identificateurs d'un code. Elle contient par exemple pour un identificateur

- le type
- le niveau d'imbrication
- le nom
- le déplacement
- la valeur

2.2 Un exemple simple : "short.c"

Nous utilisons un programme appelé "short.c" pour présenter ce qu'est une table des symboles. Son code est ci-dessous, ainsi qu'une représentation de la table des symboles qui en découle. Les AST sont vite trop fournis pour être imprimés au format papier, nous n'en mettrons donc pas par la suite dans nos exemples, comme nous le faisions dans la partie 2.

```
1   int main(){
2       int b=3;
3       int result;
4       if (b==3){
5
6           int c=1;
7           b=5;
8           result = b+c;
9       }
10      return result;
11  }
12  int fork(){
13  int a;
14  a = 5;
15  if (a==4){
16      a=0;
17  }else {
18
19      int test=25;
20      a=1;
21  }
22  return a;
23  }
```

FIGURE 2.1 – exemple de code Cir-C "short.c"

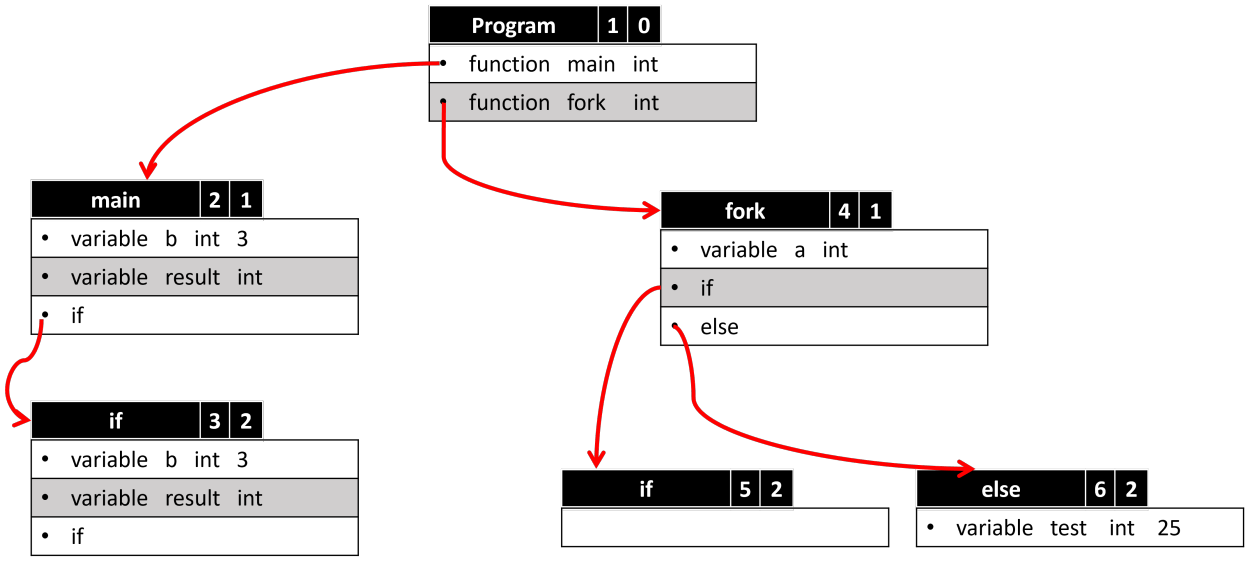

FIGURE 2.2 – Table des symboles de l'exemple short.c

Ci-dessous, la tds de short.c ainsi affichée par le terminal, créée par notre programme.

```
TDS Program | Numéro: 1 | Nv imbrication: 0

-------------------------------------------------
Entry: function  nom: main, type retour: INT
Entry: function  nom: fork, type retour: INT

TDS Fonction main | Numéro: 2 | Nv imbrication: 1

-------------------------------------------------
Entry:  Variable de type: int, nom = b, valeur = 3
Entry:  Variable de type: int, nom = result
Entry: If Then

TDS Then block | Numéro: 3 | Nv imbrication: 2

-------------------------------------------------
Entry:  Variable de type: int, nom = c, valeur = 1

TDS Fonction fork | Numéro: 4 | Nv imbrication: 1

-------------------------------------------------
Entry:  Variable de type: int, nom = a
Entry: If Then Else

TDS Then block | Numéro: 5 | Nv imbrication: 2

-------------------------------------------------
Entry:  Variable de type: int, nom = test, valeur = 25

TDS Then block | Numéro: 5 | Nv imbrication: 2

-------------------------------------------------
Entry:  Variable de type: int, nom = test, valeur = 25
```

FIGURE 2.3 – Table des symboles de l'exemple short.c affichée dans le terminal

2.3 Structure en Java

2.3.1 Diagramme de classes

FIGURE 2.4 – Diagramme de classes du package tds

2.3.2 Détails

La classe **Tds** représente une Table des Symboles, avec comme champs son niveau d'imbrication (**nv_imbrication**, son numéro (**numero**), son nom(**name**), et la liste des (**Ligne**s qu'elle contient (**contenu**).

La classe abstraite **Ligne** représente une ligne de la tds. Nous avons eu besoin d'implémenter différents types de lignes, donc les classes **LigneFonction**, **LigneIf**, **LigneStruct**, **LigneStructParam**, **ligneVariable**, et **LigneWhile** héritent de la classe **Ligne**.

La classe **Parcours** implémente les méthodes de **AstVisitor** (voir rapport d'activité 2), et parcourt l'AST en créant les tds quand elle rencontre un noeud où c'est nécessaire (déclaration de fonction, blocs if et while... etc). Nous utilisons une pile (**Stack<Tds> stack**) lors de ce parcours qui empile les TDS. Le sommet de la pile correspond à la TDS courante. La classe **Parcours** s'occupe aussi des contrôles sémantiques, qui seront explicités dans le prochain chapitre. Nous utilisons une table de Hachage (**HashMap<String, ArrayList<String> >**) pour les contrôles sémantiques liés aux appels de fonction : les clefs sont les noms des fonctions et les valeurs sont les listes des types de paramètres associées. Ainsi, à chaque appel de fonction on vérifie si les paramètres fournis correspondent à ceux dans la table de hachage.

Chapitre 3

Contrôles sémantiques

3.1 Définition de contrôle sémantique

Le contrôle sémantique consiste à effectuer plusieurs vérifications quand au sens du programme, et à renvoyer un message d'erreur lorsqu'un code est incorrect. Les vérifications effectuées peuvent porter sur la validité du type, la portée, la présence d'une initialisation...etc.
Nous rappelons qu'à ce stade, les analyses lexicale et syntaxique ont déjà été effectuées. Il n'y a donc pas de contrôle à faire sur ces points.
Nous rappelons également que les contrôles sémantiques que nous pouvons réaliser lors de la compilation sont les contrôles sémantiques **statiques**, par opposition aux **dynamiques** réalisés à l'exécution.

3.2 Implémentation en Java

Les contrôles sémantiques sont effectués au sein de **Parcours**, et sont stockés dans une **ArrayList<String>** appelée **listerror**. Le fonctionnement est le suivant : lors de la visite, si une erreur est détectée, on ajoute à **listerror** une chaîne qui contient le message d'erreur. Ensuite quand tout a été parcouru, on affiche la liste des erreurs.

L'analyse se poursuit après la détection d'une erreur sémantique. Il ne s'affichera pas de tds, mais une liste de toutes les erreurs à la fin de l'analyse.

3.3 Les erreurs sémantiques traitées

- La variable n'est pas définie (pour une affectation, pour une condition, lors d'une addition, d'une soustraction, d'une multiplication ou d'une division).
- L'appel à une fonction qui n'existe pas.
- Le nombre ou le type des paramètres dans l'appel de fonction n'est pas correct.
- Le type de retour de la fonction n'est pas le bon. Il ne correspond pas à celui dans l'entête de la fonction.
- Une variable du même nom a déjà été déclarée. (pour les int et pour les struct)
- Une fonction du même nom a déjà été déclarée.
- La division par zéro.
- La création de plusieurs structures du même nom. Par exemple struct person{int a} ; et struct person{int b ; int c ;} ;
- Le manque de return. (Ici toutes les fonctions retournent quelque chose : int ou struct)
- Le manque de la fonction main(). Cette fonction a pour type de retour int et n'a pas de paramètre.
- La variable pointée par la flèche -> n'est pas un attribut de la structure.
- Le type de structure utilisé n'exsite pas. Exemple struct person t ; et person n'existe pas.
- L'utilisation d'un sizeof sur une variable de type int. Nous vérifions que le struct sur lequel on appelle le sizeof est déjà déclaré.
- L'appel à une variable qui n'est pas accessible. (La portée des variables)
- L'affectation d'un int à une variable de type struct et inversement.

3.4 Un exemple détaillé : short_false.c

Reprenons notre exemple short.c. Il s'agit d'un code Cir-C correct. Pour tester notre analyse sémantique, nous lui ajoutons à la ligne 10 un appel à une fonction que nous n'avons pas définie.

```
1  int main(){
2      int b=3;
3      int result;
4      if (b==3){
5
6          int c=1;
7          b=5;
8          result = b+c;
9
10         b = fonction_surprise(c);
11
12     }
13     return result;
14 }
15 int fork(){
16 int a;
17 a = 5;
18 if (a==4){
19     a=0;
20 }else {
21
22     int test=25;
23     a=1;
24 }
25 return a;
26 }
```

FIGURE 3.1 – exemple de code Cir-C faux sémantiquement "short_false.c"

Contrairement à short.c, nous n'obtenons plus de tds. Un message d'erreur coloré s'affiche dans notre terminal, en nous expliquant le souci. S'il y a plusieurs erreurs sémantiques alors les erreurs s'affichent les unes à la suite des autres dans l'ordre d'apparition.

```
1 ERROR : Fonction fonction_surprise n'existe pas !
```

FIGURE 3.2 – affichage du message d'erreur

Chapitre 4

Gestion de projet

4.1 Répartition des tâches

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 et la préparation de la soutenance. En revanche, sont incluses les séances de travail à plusieurs.

	Ahmed	Laury	Coline
Conception TDS	4	1	2
Ecriture de la classe TDS	2	2	1
Ecriture des classes Ligne	1	1	1
Ecriture des méthodes pour la TDS	3	3	2
Conception contrôle sémantique	3	3	1
Ecriture des méthodes pour le contrôle sémantique	6	6	1
Ecriture de tests	1	2	4
Rédaction du rapport et du Readme	1	1	5
Total (en heures)	21	19	18

TABLE 4.1 – Répartition de travail

Chapitre 5

Conclusion

5.1 Construction d'un compilateur

Le projet Compilation de 2ème année consiste en la création d'un compilateur, dont les différentes étapes avaient été expliquées en cours de Traduction des Langages. Lors de cette première partie de projet, nous avons donc pu implémenter la partie frontale d'un compilateur Cir-C, ainsi qu'entamer le coeur du compilateur.

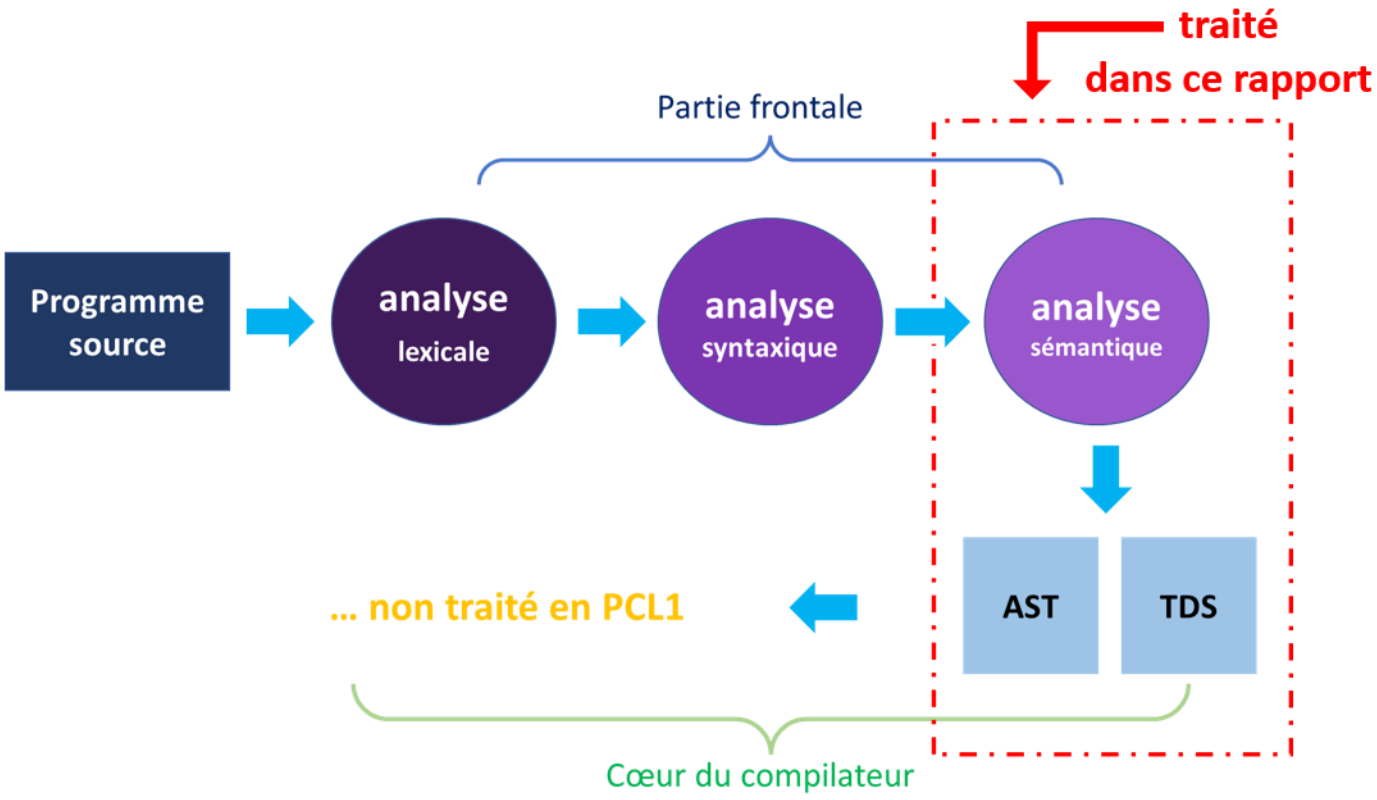

FIGURE 5.1 – Extrait du schéma d'un compilateur, avec les étapes vues en PCL1

Notre groupe n'effectue pas le module PCL2, c'est donc ici que notre projet s'arrête. Même si nous ne livrons pas un compilateur qui génère du code assembleur, nous avons quand même beaucoup appris sur le processus de la compilation, et nous sommes satisfaits de pouvoir pleinement comprendre les étapes qui transforment un programme source sous forme de texte en une liste d'instructions assembleur.

5.2 Annexe

5.2.1 Grammaire finale Cir-C

```
grammar circ;

@header {
package parser;
}

program: decl* EOF;

decl: decl_typ | decl_fct;

decl_vars:
    'int' (IDENT ',')* IDENT ';'                    # Decla
    | 'struct' IDENT ('*' IDENT ',')* '*' IDENT ';' # Struct
    | 'int' IDENT '=' ENTIER ';'                    # DeclaAffect;

decl_typ: 'struct' IDENT '{' liste_decl_vars '}' ';';

decl_fct:
    'int' IDENT '(' liste_param ')' bloc               # IntParam
    | 'struct' IDENT '*' IDENT '(' liste_param ')' bloc # StructParam;

liste_expr: (expr ',')* expr;

param:
    'int' IDENT                # Paramint
    | 'struct' IDENT '*' IDENT  # Paramstruct;

liste_param: (param ',')* param # List | # Vide;

expr:
    ENTIER                              # Entier
    | IDENT                             # Ident
    | IDENT '(' liste_expr ')'          # IdentExprPointeur
    | '!' expr                          # ExclaExpr
    | '-' expr                          # TiretExpr
    | 'sizeof' '(' 'struct' IDENT ')'   # Sizeof
    | '(' expr ')'                      # ParenthExpr
    | expr OPERATEUR expr               # OpExpr
    | expr '->' IDENT                   # Fleche;

instruction:
    expr ';'                                          # ExprSeule
    | 'if' '(' expr ')' instruction                   # IfThen
    | 'if' '(' expr ')' instruction 'else' instruction  # IfThenElse
    | 'while' '(' expr ')' instruction                # While
    | bloc                                            # BlocInstruct
    | affectation                                     # AffectInstruct
    | 'return' expr ';'                               # Return;

affectation: IDENT '=' expr2 ';';

liste_decl_vars: decl_vars*;

liste_instruction: instruction*;

bloc: '{' liste_decl_vars liste_instruction '}';

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

```

```
60  ENTIER : '0' | ('1'..'9') ('0'..'9')* | CARACTERE;
61  IDENT : ('a'..'z'|'A'..'Z')('a'..'z'|'A'..'Z'|'0'..'9'|'_')*;
62  CARACTERE : '\''('!'|'#'|'$'|'%'|'('|')'|';'|'+'|','|'-'|'.'|'&'
63              | ('0'..'9')|':'|';'|'<'|'='|'>'|'?'|'@'|'['|']'|'^'
64              | '_'|'`'|'{'|'}'|'~'|'a'..'z'|'A'..'Z'|'/'|'|')'\''
65              ;
66  WS : ('\n'|' '|'\t'|'\r'|'/*' .*? '*/' | '//' ~[\r\n]*)+ -> skip;
67
68  expr2 : plus;
69
70  plus:  mult (('+'|'-') mult)*;
71
72  mult : expr (('*'|'/') expr)*;
```

FIGURE 5.2 – Notre grammaire, contenue dans le fichier circ.g4