PLD-COMP
Rapport de mi-parcours

H4414
A. Belin - A. Nahid - L. Ohl
L. Saos - A. Verrier - I. Zemmouri

27 mars 2019

Table des matières

Table des figures

Un petit mot d'introduction

Ce dossier reprend l'ensemble des points produits et demandés pour le Projet Longue Durée Compilateur, à savoir compiler n très simple programme C consistant d'un main et d'affectation arithmétiques sur des entiers.
Toutefois, qui dit mi-parcours dit aussi « en cours de route ». Il ne sera donc pas étonnant de trouver des aspects grammaticaux, sémantiques, etc, développés en avance par rapport au travail demandé. Ces points particuliers seront décrits en annexes, car n'étant pas au coeur du sujet.

1 Fonctionalités implémentées

Nous allons survoler dans cette section l'ensemble des possibles actuellements réalisable par notre compilateur.

1.1 Ce que le front peut lire

Par « ce que le front peut lire », nous entendons ce que le parser est capable de lire et, en addition, transformer sa lecture en éléments de l'ast. Autrement dit, ce n'est pas que de la grammaire !

Le parser est capable de lire un corps de main contenant une série de définitions, suivi par l'ensemble des instructions arithmétiques et une éventuelle instruction return.
Voici des exemples de corps main tolérés [1] :

```
int main() {
  //...
}
```

```
void main(){
  //...
  return;
}
```

```
void main() {}
```

```
int main() {
  //...
  return 23; //23 ou une expression arithmetique
}
```

Dans ce corps de main, il faut dans un premier temps déclarer et définir les variables qui seront utilisées dans le bloc de code.

1. Pour tout ce qui suivra, les commentaires sont uniquement pour la compréhension de ce qui peut être écrit. En aucun cas le compilateur ne saura les lire. Ils doivent donc être absents des fichiers à compiler

```
int a;
int a=3;
int a,b;
int a=2,b;
int a=3,b=4;
int c,d=a,e;
//idem avec char.
```

Enfin, les expressions arithmétiques suivantes sont opérables :
— L'addition +
— La soustraction -
— La multiplication *
— La division /
— Le *et* bit-à-bit &
— Le *ou* bit-à-bit ||
— Le *xor* bit-à-bit ∧
— L'opérateur unaire de négativation -
— L'opérateur unaire *not* ∼
Ainsi, le programme de mi-parcours est totalement lisible :

```
int main()
{
int a,b,c;
a=17;
b=42;
c = a*a + b*b +1;
return c;
}
```

Bien sûr, il est encore tout à fait possible d'écrire n'importe quoi sémantiquement à ce stade-là. D'où la section suivante.

1.2 La vérification sémantique

Notre vérification sémantique du programme contient les éléments suivants pour le mi-parcours :

Pour le programme complet Nous vérifions qu'il n'existe bien que des fonctions et des variables globales. Chacun sans duplicata. Nous vérifions également la présence de la fonction *main*.

Pour une fonction Nous vérifions récursivement la sémantique de chaque sous-instruction contenue. Nous vérifions également qu'elle ne possède pas de variables nommées similairement à d'autres variables de portée supérieure. Nous observons également si le *return* d'une fonction corresond bien au type prétendu de la fonction.

Pour les expressions arithmétiques Nous vérifions bien que les opérandes ne sont pas de type *Void*, et vérifions que chaque sous-expression d'une expression est elle-même conforme. Enfin, pour ce qui est de l'affectation, nous vérifions systématiquement que l'identifiant d'une variable en l-value réfère bien quelque chose de connue et que la r-value est du même type que la l-value.

1.3 Ce que le back peut écrire

2 Structure du compilateur

2.1 Répartition des packages

Pour synthétiser l'ensemble : un petit schéma.

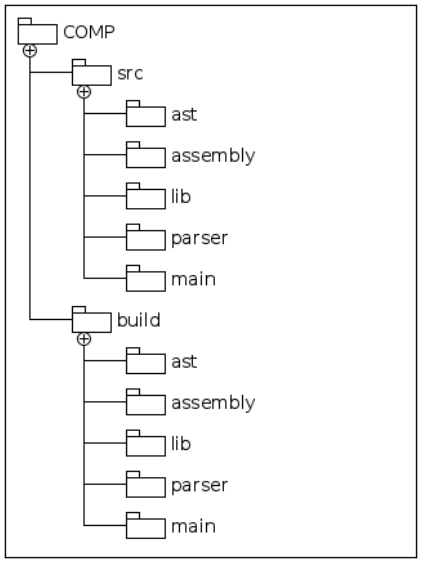

FIGURE 1 – Répartition des packages pour le compilateur

Remarque : le dossier build est similaire afin de conserver un stockage logique des .o lors de la compilation du projet.

ast

Le package ast contient l'ensemble des classes correspondant à ... l'ast ! Nous reviendrons plus en détail en partie 2.2.

assembly

Ce package contient l'ensemble des classes associées à la génération d'assembleur. En plus simple, c'est le dossier contenant le back-end.

lib

Il s'agit d'un petit dossier contenant le code unistd. Comme nous utilisons une fonction [2] de cette bibliothèque, ceux travaillant sur windows avaient besoin d'avoir ce fichier sous la main.

parser

C'est ici que se déroule le travail d'ANTLR4. Dans ce dossier se trouvent le fichier de grammaire, subtilement appelé *expr.g4*, ainsi que l'ensemble des fichiers générés par ANTLR4. À cela s'ajoute notre première part de code : les visiteurs de symboles, situés dans les fichiers, tout aussi subtilement nommés, *visiteur.cpp* et *visiteur.hpp*.

main

Comme souvent, il est un peu seul, et il faut bien le mettre quelque part. Voilà l'endroit où se trouve *main.cpp*.

2.2 Structure de l'ast

Plutôt que de perdre du temps en mot, un diagramme de classe sera plus explicite (page 5).
Si jamais ce diagramme semble indigeste à lire, en voici une version ne gardant que les liens d'héritage (page 8)

2. getopt

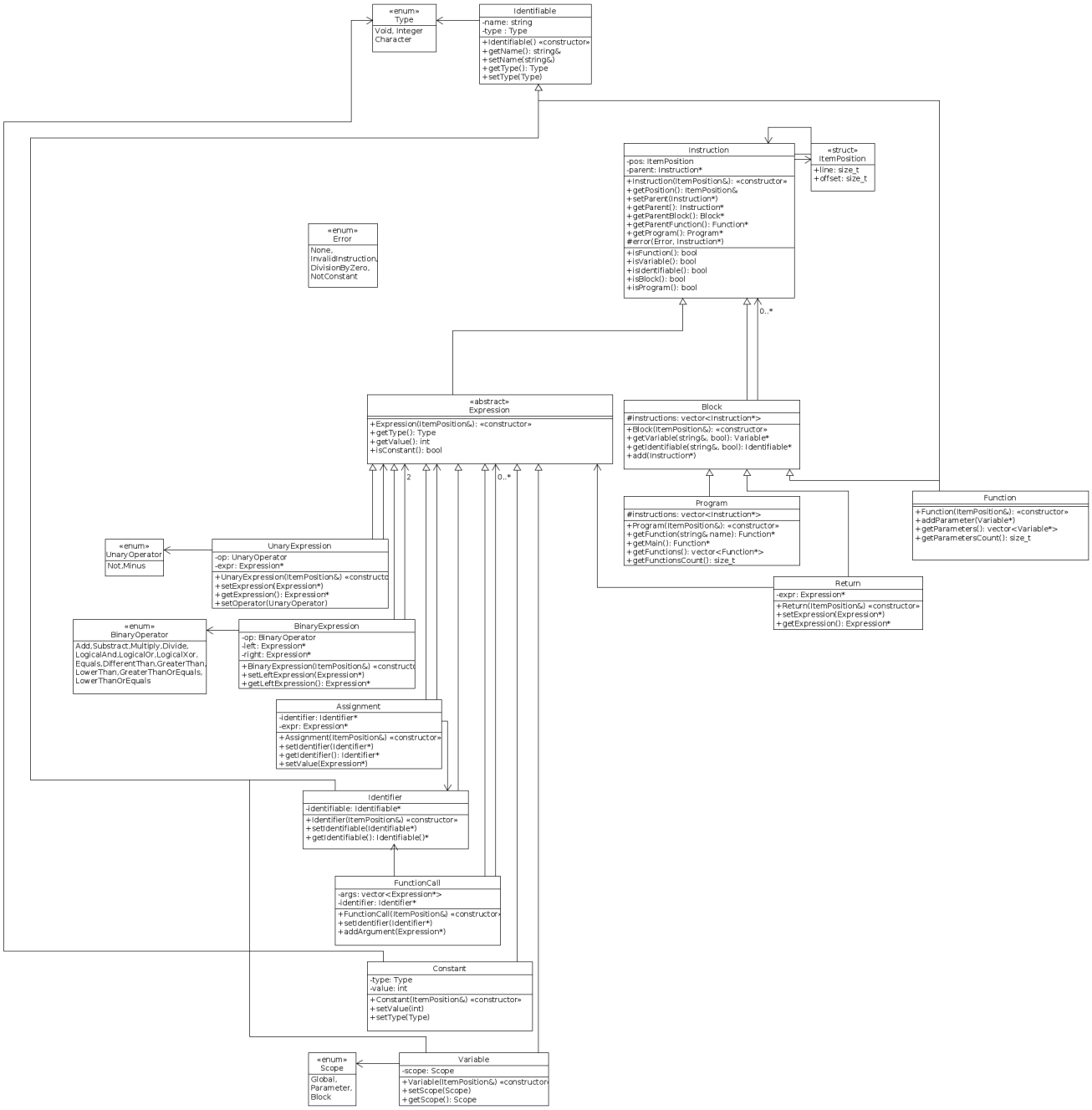

FIGURE 2 – Diagramme de classe complet de l'ast

Pour bien comprendre ce diagramme, nous nous sommes fixés comme paradigme : « tout est instruction ». Il reste ensuite quelques petits points à éclaircir, même si la plupart des éléments portent le même nom que ce soit à quoi il correspondent sémantiquement.

Variables, fonctions et identifiants Les variables et les fonctions sont des éléments que l'on considère *identifiables*. Ils portent un nom qui permet de bien les repérer. Cependant, voir le nom d'une variable n'a pas le même sens selon son contexte. Nous avons donc pris le parti suivant : définir/déclarer une variable revient à ajouter créer un objet *Variable*, tout comme déclarer une fonction crée une *Fonction*. En revanche, si l'on aperçoit le nom d'une variable/fonction en-dehors d'un contexte de déclaration, par exemple affectation ou appel de fonction, alors

on crée un objet *Identifier* qui signifie « On fait appel à un truc donc on ne connait pas trop la nature ». Par exemple :

```
int a, b=5; // a et b sont des Variables
a=b+3; // a et b sont des Identifier, 3 est une constant Integer.
```

```
void main() {//main est une Fonction
}
main();// main est un Identifier
```

3 Compiler le compilateur

3.1 Compiler sur Linux

Normalement, ça ne devrait pas être très compliqué. Il suffit de reprendre les variables localisant ANTLR4 dans le makefile et les adapter à votre installation d'ANTLR, similairement à l'exemple donné sur les machines de l'INSA.

```
ANTLR=path/to/the/antlr4/executable
ANTLRSRC=path/to/the/antlr4/runtime/files
ANTLRRUNTIME=path/to/the/antlr4-runtime.a
```

En fonction de la manière dont vous définissez ANTLRRUNTIME, il faudra probablement aller modifier la définition de la variable LDFLAGS, un peu plus loin.
Il n'y aura ensuite plus qu'à se placer à la racine du projet, exécuter *make* puis utiliser l'exécutable **./compiler**.

3.2 Compiler sur Windows

Il faut installer ANTLR4 adapter pour visual studio, puis lancer un projet visual studio en utilisant le code source fourni. [3]

4 Gestion de projet

Comme brillamment suggéré par le sujet, nous développons en mode AGILE. Comme pour toute méthode AGILE, nous adaptons le mot agile à notre sens. Cela prend la forme suivante : trois groupes de personnes travaillant chacun sur une partie de la chaîne de compilation. Nous ajoutons à chaque itération de nouveaux éléments de grammaire, ensuite interprétés par les visiteurs, mis en forme dans l'ast et ensuite rédigé sous forme assembleur. Voici les groupes :

Officiellement

— L. Ohl et I. Zemmouri : Grammaire, visiteurs, et comptes-rendus.
— L. Saos : Responsable ast.
— A. Belin, A. Nahid et A. Verrier : transcription de l'ast en assembleur.

Nous coordonnons régulièrement les parties afin de résoudre le plus vite les points pouvant être des goulots d'étranglement pour d'autres. Dans les faits, comprendre l'assembleur à pris beaucoup de temps, ce qui explique la grande avance prise par le frontend sur le backend.

Officieusement

— L. Ohl et I. Zemmouri : Grammaire, visiteurs, ragequit antlr et recherche de memes. Scribes lors des RTT.
— L. Saos : Resposable ast, glorieux et officieux chef du bordel ambulant
— A. Belin, A. Nahid et A. Verrier : pauvres victimes sacrifiées à l'assembleur

C'est le zbeul, mais dans la bonne humeur et le rire de voldemort ! [4]

3. Pour être franc, nous n'avons pas vraiment unifié la manière de faire, chacun y allant de son mieux.
4. He, hehe !

Annexe - Ce qui est en cours de route

Grammaire

La grammaire définit de nombreux points qui ne sont pas utilisés pour le moment, i.e. non visités. On y retrouve notamment : les includes, la définition de bloc de code, la construction du type char [5].

L'arithmétique

Vous trouverez de par et d'autres, et notamment dans le fichier *src/ast/operator.hpp* l'ensemble des opérateurs en cours de route. Sont actuellement en cours de développement : le *ou*, le *et*, le *not* logiques.

L'ast

L'ast possède de très nombreuses fonctionnalités encore non exploitées. À titre d'exemple, l'objet block, défini dans *src/ast/block.hpp* permet de décrire un bloc de code de manière générale, et par extension une fonction. Il a été totalement écrit comme s'il devrait être utilisé similairement pur une boucle, une alternative, etc. À l'heure actuelle, il n'est employé que pour l'aspect *fonction*.
D'autres objets sont également en cours de construction : *functionCall*, *ControlStructure*, etc.

5. Ce dernier est réellement visité toutefois

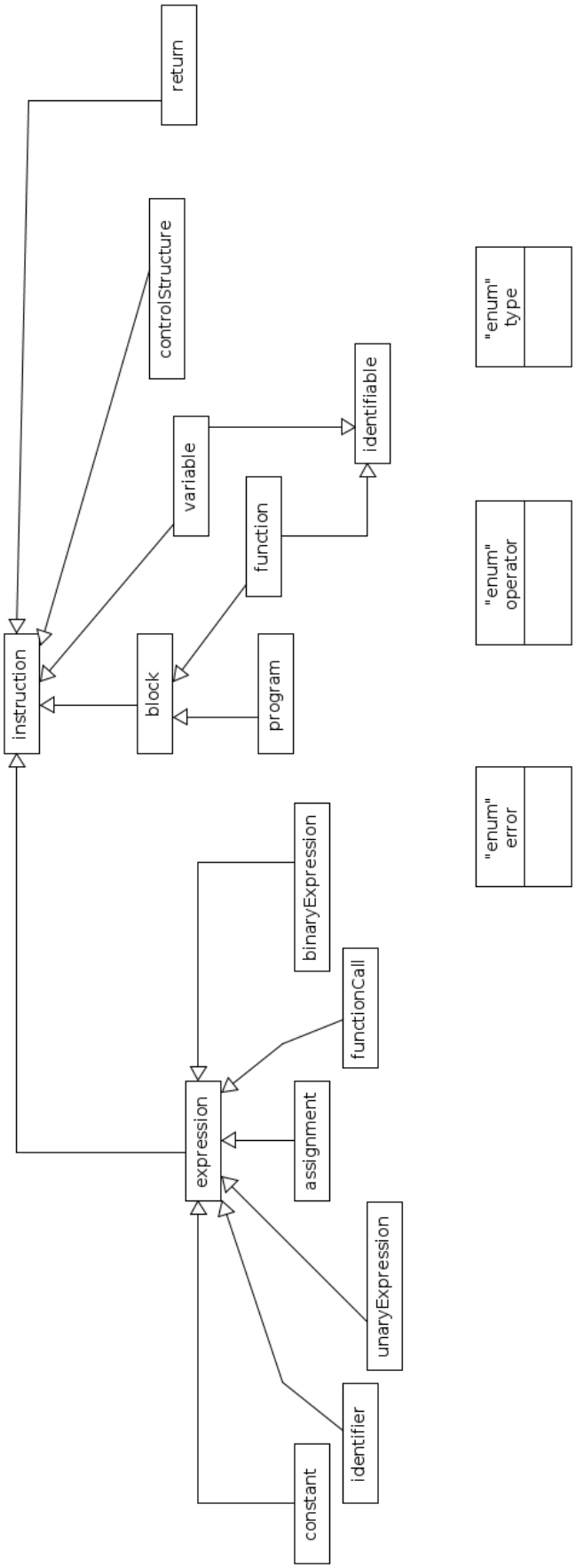

FIGURE 3 – Diagramme de classe simplifié de l'ast