



# PLD-Compilateur-documentation

23.03.2021

---

H4222

CHEN Gong

FENG Xinyu

GUO Jiadong

HE Muye

ZHOU Shihang

ZONG Yuxuan

## Fonctionnalités implémentées

### Déclaration et définition des variables de type int

Dans notre compilateur, on permet des utilisateurs de faire les déclarations unique ou multiple:

```
int a,b,c;  
int d;
```

De plus, on peut faire les définition des variables en utilisant une constante, un variable ou une expression arithmétique composée des variables et constantes relié avec les multiplications, additions, soustractions et parenthèses:

```
int a = 2;  
int b = a;  
int c = a * a + b * 2 - 1;
```

Malheureusement, on ne peut pas combiner les déclarations et la définitions dans une même phrase.

```
int a = 2, b, c = a; // Error! mismatched input ',' expecting ';'
int a,b = 2;         // Error! extraneous input '=' expecting {';',',','}'
```

En plus, les variables peuvent composer uniquement des lettres [a-z] et [A-Z] et [0-9] qui ne doivent pas commencer par [0-9].

```
int _a = 5;          //Token recognition error at: '_'
```

### Affectation des valeurs à un variable

On peut affecter les valeurs aux variables déjà définies ou déjà déclarées. Mais il n'est pas possible de mettre plusieurs passages d'affectation dans une même phrase.

```
int b = 2;  
int a;  
a = b;           // Affecter un variable vers un variable  
a = 4;           // Affecter une constante vers un variable  
a = (a + 6) * 3 - 9; // Affecter une expression vers un variable  
a = b = 5;       // mismatched input '=' expecting ';' 
```

## Commentaires

Dans notre programme, on peut commenter sous la forme `/*...*/`

```
int a;  
a = 12; /* Affectation de la variable a*/
```

## Arithmétiques

Jusqu'à maintenant, nous avons implémenté les parenthèses, l'addition, la soustraction, et la multiplication pour l'affectation d'une expression arithmétique à une variable.

Selon les priorités de calcul, le traitement des parenthèses est devant tous, suivi par une combinaison de la multiplication et de la division (pas implémentée). La soustraction et l'addition sont de même niveau, qui se situe à la fin. Les calculs de même priorité doivent être traités dans une seule syntaxe de antlr 4 afin d'éviter les erreurs.

Une expression arithmétique correcte peut être calculée correctement, et une erreur syntaxique va arrêter la compilation.

```
int a, b, c;  
a = 5;  
b = 47;  
c = a * (a + 3) - 5 * (5 + (5 * (4 + (1)))) + b * a;
```

## Structure de codes

Notre projet garde la même structure que le squelette. Après avoir configuré l'environnement de antlr4, on va implémenter les règles pour notre compilateur.

- Compiler
  - antlr4-runtime
  - antlr4-generated
  - ifcc.g4
  - main.cpp
  - **visitor.h**
  - **checkVisitor.h**
  - .....
- tests
  - tests
    - Init
      - test\_prog.c
      - test\_arith.c
      - .....

Nous développons le projet sous principe "test driven": pour développer chaque syntaxe, on commence par implanter une syntaxe primitive, et puis écrire un test correct(correct par gcc) et un test faux(faux par gcc), pendant lesquels on modifie et valide les règles, et décide des traitements pour certains cas complexes, par exemple un mélange de définition et déclaration dans une même phrase, qui est considéré comme faux par notre compilateur.

Ainsi, les règles lexicales sont également définies dans le ifcc.g4, par exemple la règle de nommage d'une variable.

Ensuite les règles sont définies dans le fichier ifcc.g4 qui sera traité par antlr 4 en produisant antlr4-generated.

En même temps, on va écrire les commandes assembleur correspondantes dans les visiteurs pour générer le fichier .s quand ils parcourent les codes à compiler.

Particulièrement, le checkVisitor.h va d'abord parcourir le code pour valider la syntaxe afin d'éviter les erreurs comme double déclaration. Après, le visitor.h peut générer le fichier .s selon les codes. Chaque méthode dans le visitor.h correspond a une syntaxe définie dans le fichier ifcc.g4, indiquant les commandes assembleurs.

Par exemple, la règle de déclaration dans ifcc.g4

```
expr : 'int' VAR dec* ';'          # déclaration
```

correspond a l'implémentation visitDeclaration dans le visitor

```
virtual antlrcpp::Any visitDeclaration(ifccParser::DeclarationContext *ctx)
override {
    string var_name(ctx->VAR()->getText());
    variables[var_name] = 1;
    visitChildren(ctx);
    return 0;
}
```

La syntaxe sera vérifiée par le test test\_dec\_succ.c

```
int main(){
    int a;
    int b, c, d;
    return a;
}
```

## Gestion de projet

Notre gestion de projet est composée de deux sprints dont le premier correspond au mi-parcours et le deuxième correspond au livrable final. Grâce à l'Antlr 4, la quantité de codes est réduite. Nous avons donc pu nous concentrer sur les règles à implémenter sur l'Antlr et les visiteurs qui parcourent l'arbre de syntaxe abstraite.

## Configuration de l'environnement

Vu que nous focalisons notamment sur langage assembleur qui dépend du système d'exploitation(Windows, Mac OS, Linux), nous avons donc décidé d'utiliser le docker avec l'image fournie qui est relativement légère et qui nous permet de travailler avec notre propre ordinateur portable.

Nous avons choisi GitHub pour la gestion de versions de codes et Google Drive pour la rédaction de documentation.

## Prise en main de codes

Avant la première séance, chacun d'entre nous s'est informé sur l'utilisation de l'Antlr 4 et exécuter le programme selon les fichiers README.md dans les codes de base afin que nous puissions commencer le projet plus rapidement.


Le Chef de projet organise la première réunion qui nous permet de nous mettre au même niveau à la première séance.

Ensuite, nous nous séparons en deux groupes de trois pour découvrir et discuter les résultats en suivant 4.1 et 4.2.

## Discussions & Recherches des solutions

Après la première séance, nous avons pu connaître la base de projet. Nous avons donc commencé par la conception des règles d'Antlr qui génère le lexer et le parser. Ensuite, deux membres implémentent les tests pour trouver les bugs que nous n'avons pas anticipés, deux membres implémentent les visiteurs qui parcourent l'arbre de syntaxe abstraite, deux membres travaillent sur la grammaire et l'implémentation de langage assembleur. Nous avons débuté avec la définition des variables, puis la déclaration et l'affectation(constantes ou variables), ensuite les arithmétiques(+, -, \*, /).

## Gestion d'exceptions



Pour la quatrième séance, nous avons commencé à gérer les exceptions selon les tests pour afficher les erreurs ou alertes. Par exemple, l'utilisation de variable sans déclarer fera planter le programme et affichera une erreur sur le terminal.

## Rédaction de documentation

Nous avons divisé cette documentation en trois parties et chaque partie est rédigée par un membre de l'équipe pour la quatrième séance.