**République algérienne démocratique et populaire**

**Ministère de l'enseignement supérieur et de la recherche scientifique**

Université des sciences et technologies Houari Boumediene

Département d’informatique



**Rapport Projet Compile 2**

Réalisé par :

ADMANE Hocine 171731054926

TEBBANIMohamed Walid 171731088266

N°Binome 7

Sommaire

* Introduction

1- Créations des analyseurs lexicale et syntaxique (Le lexeur et le parseur) :

* Le Lexique de langage COMPIL (Lexeur) :
* La grammaire de langage COMPIL (Le parseur) :

2- Exemple de Code

* Génération d’arbre abstrait (parse tree) :
* Création de table de symbole (TS) :

3- Analyse sémantique

* Double déclaration :
* Variable non déclarée :
* Division sur Zéro :
* Incompatibilité des types :

4- Génération des quadruplets :

* Un autre exemple (instruction Do-While)

5-Code Objet + Optimisation

* Un autre exemple

INTRODUCTION

Ce projet consiste à réaliser un compilateur du langage COMPIL en utilisant l’outil de reconnaissance de ANTLR (ANother Tool for Language Recognition) et le langage JAVA. Dans un premier temps, nous allons montrer comment nous avons défini notre grammaire "Compil.g4" et la création des analyseurs : lexical et Sémantique, ensuit la création des quadruplet et génération de code objet, nous effectuerons aussi quelques tests sur un code du langage COMPIL Une fois les deux analyseurs créés, nous allons définir la structure de notre table de symboles.

Tâches réalisées :

* Tout est réalisé dans ce projet

1- Créations des analyseurs lexicale et syntaxique (Le lexeur et le parseur) :

* Le Lexique de langage COMPIL (Lexeur) :

L’analyse lexicale constitue la première étape dans la compilation. Elle consiste à décomposer un texte en unités ou entités lexicales qui serons ensuite traité dans la partie syntaxique. Nous avons réalisé un analyseur lexical à l’aide de l’outil ANTLR qui génère des TOKENS, le compilateur créé utilise ces TOKENS pour reconnaitre à quoi attendre en suivant (quelle syntaxe va-t-il suivre ANTLR peut aussi utiliser les expressions régulières pour reconnaitre une entité lexicale.

Dans ce projet nous avons créé un analyseur qui peut reconnaitre :

|  |  |  |
| --- | --- | --- |
| Les Mots Clés | Les Operateurs/Séparateurs | Les Expressions régulières (en JAVA) |
| COMPIL : 'compil';  INT : 'intCompil';  FLOAT : 'floatCompil';  STRING : 'stringCompil';  SCAN : 'scancompil' ;  PRINT : 'printcompil' ;  START : 'start' ;  IF : 'if';  THEN : 'then';  ELSE : 'else';  DO: 'do';  WHILE: 'while'; | AFF : '=';  PLUS : '+';  MINUS : '-';  MUL : '\*';  DIV : '/';  SUP : '>';  INF : '<';  SUPE : '>=';  INFE : '<=';  EQ : '==';  DIF : '!=';  PARO : '(';  PARF : ')';  ACODO: '{';  ACODF: '}';  PV : ';';  VER : ','; | ID : [a-zA-Z][a-zA-Z0-9]\*;  INTEGERVAL : '0'|[1-9][0-9]\*;  FLOATVAL : INTEGERVAL('.' INTEGERVAL);  STR : '"' .\*? '"';  WS : [ \n\t]  COMMENTAIRE1 : '/''/'(~[\n])\*  COMMENTAIRE2 : '/\*'(.)\*?'\*/' |

* La grammaire de langage COMPIL (Le parseur) :

Programme (Règle de départ) :

* Programme = ‘compil’ Nom\_de\_programme ‘(‘ ‘)’ ‘{‘ ‘}’ <déclaration des variables> ‘start’ <Liste des instructions>

Partie déclaration :

* <Déclaration des variables> = <Déclaration><List déclaration> | <Déclaration>| e
* <Déclaration> = <Type> <List Variables> ;
* <Type> = ‘intCompil’ | ‘floatCompil’| ‘stringCompil’
* <List Variables> = id ‘,’ <List Variables> | id

Partie instructions :

* <Liste des instructions> = <Instruction> <Liste des instructions>| e
* <Instruction> = <Affectation>|<InstructioIF>|<InstructionDOWHILE>|<Print>|<Scan>

Affectation :

* <Affectation>= id ‘=’ <Expression>’ ;’
* <Expression> = <Expression>[‘+’|’-’]<Expression\_prioritaire>|<Expression\_prioritaire>
* <Expression\_prioritaire> = <Expression\_prioritaire>[‘\*’|’/’]<Suit\_expression> | <Suit\_expression>
* <Suit\_expression> = id | ‘(‘ <Expression> ‘)’ | valeur\_int | valeur\_float | valeur\_String

Instruction IF :

* <InstructioIF> = ‘if’ ‘(’ <Condition> ‘)’ ‘then’ ‘{’ <Liste des instructions> ‘}’ <Else>
* <Else> = ‘{’ <Liste des instructions> ‘}’ | e
* <Condition> = <Expression> <Operateur\_Logique> <Expression>
* <Operateur\_Logique> = ‘>’| ‘>=’ | ‘<’ | ‘<=’ | ‘ !=’ | ‘==’

Instruction Do-While :

* <InstructionDOWHILE> = ‘do’ ‘{’ <Liste des instructions> ‘}’ ‘while’ ‘(’ <Condition> ‘)’ ‘ ;’

Instruction Print :

* <Print> = ‘printCompil’ ‘(’ [valeur\_String | <List\_Ids>] ‘)’ ‘ ;’
* <List\_Ids> = id ‘,’ <List\_Ids>| id

Instruction Scan:

1. <scan> =’Scan’ ‘(’ <List\_Ids> ‘)’ ‘;’

2- Exemple de Code

Voici un exemple de code qu’on va traiter dans un fichier ‘input.txt’  : on va voir l’arbre syntaxique de ce code, ensuite nous créerons la table de symbole, après en va faire une étude sémantique sur un autre exemple sauf qu’il soit erroné (avec les erreur sémantique).

compil prg1 (){

stringCompil th;

intCompil b,a;

floatCompil c;

start

c=1.2+3.4;

scancompil(c);

do{

if(c>2)then{c=2.2+4;}else {c=2.4;}

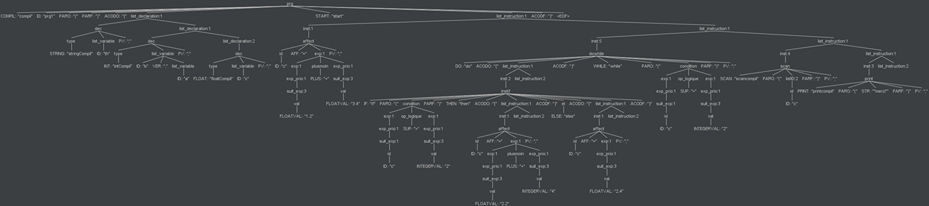
}while(c>2);

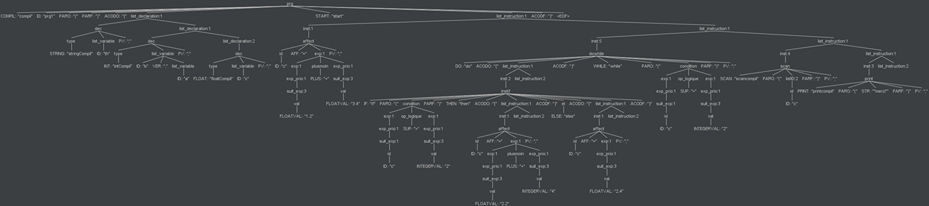
printcompil("merci");

}

* Génération d’arbre abstrait (parse tree) :

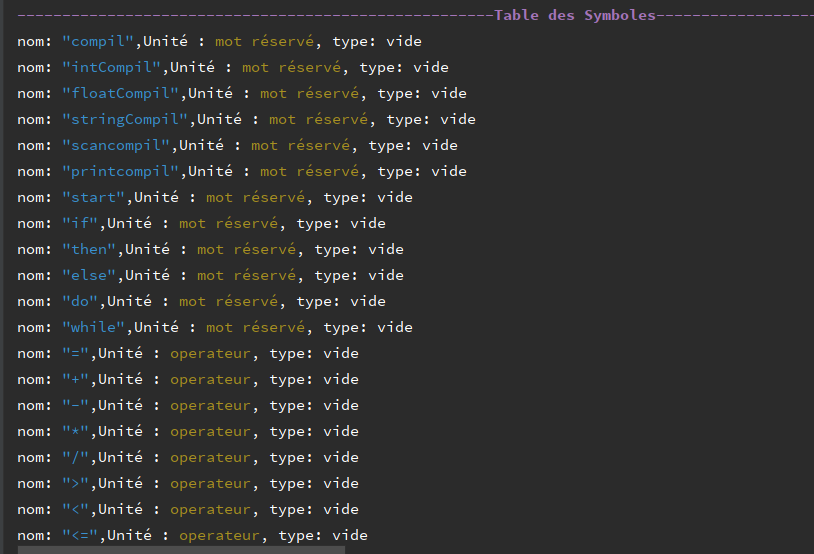
On peut voir que l’arbre ne contient pas une erreur lexical/syntaxique, alors le code est accepté.



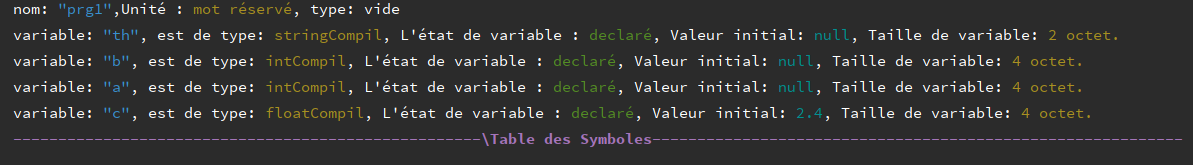


* Création de table de symbole (TS) :

La table de symbole contient au début de les mot clé réservé et Les opérateurs/séparateurs



Lorsque le compilateur déroule le fichier ‘input.txt’ il ajout les variables et il les met ajour leur valeur dans la TS, voici la table de symbole de cet exemple :

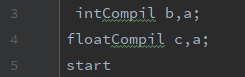


3- Analyse sémantique

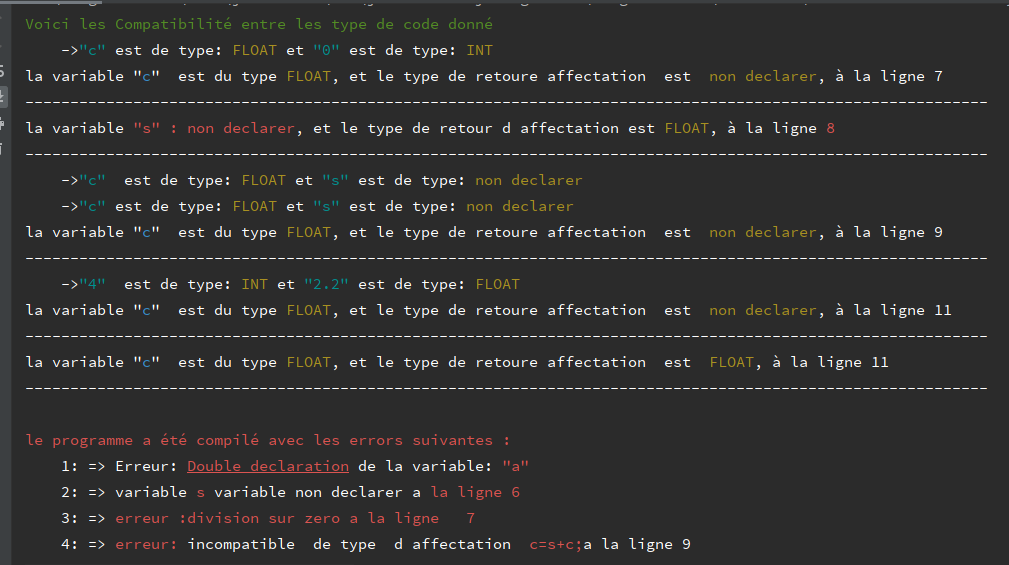
Dans la phase sémantique on a implémenté les contraintes suivantes en utilisant les LISTNESRS qui sont générés par ANTLR :

* Double déclaration :

Voici un exemple de code erroné avec une double déclaration



Voici l’erreur simantique correspandant :

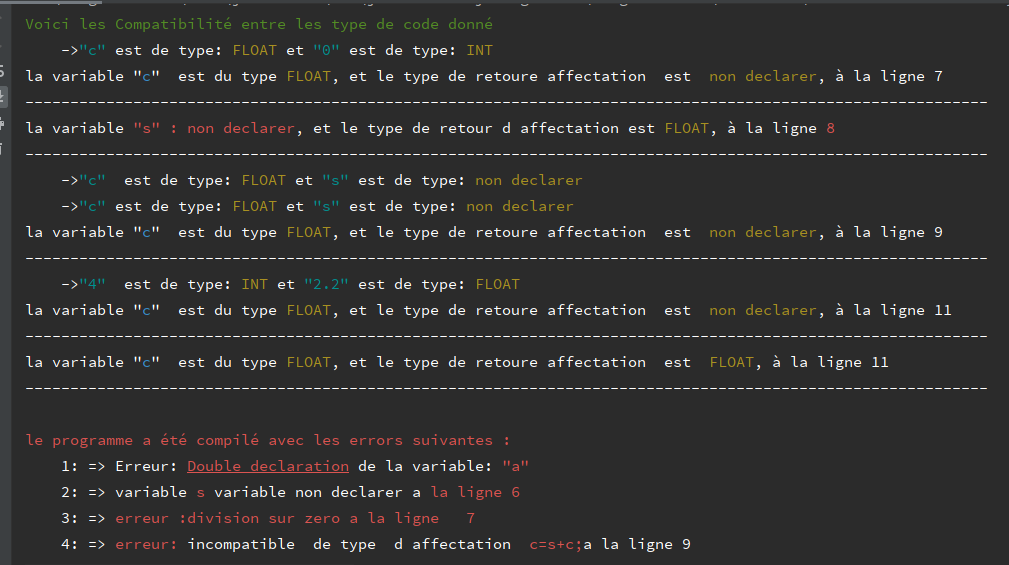


* Variable non déclarée :

Lorsque on utilise une variable dans une opération (affectation, condition, instruction if ..) sans être déclaré dans la partie déclaration, on déclenche une erreur sémantique, voici un exemple : si on prend le même code précédant et en ajout ces deux lignes



Voici l’erreur sémantique correspondant :

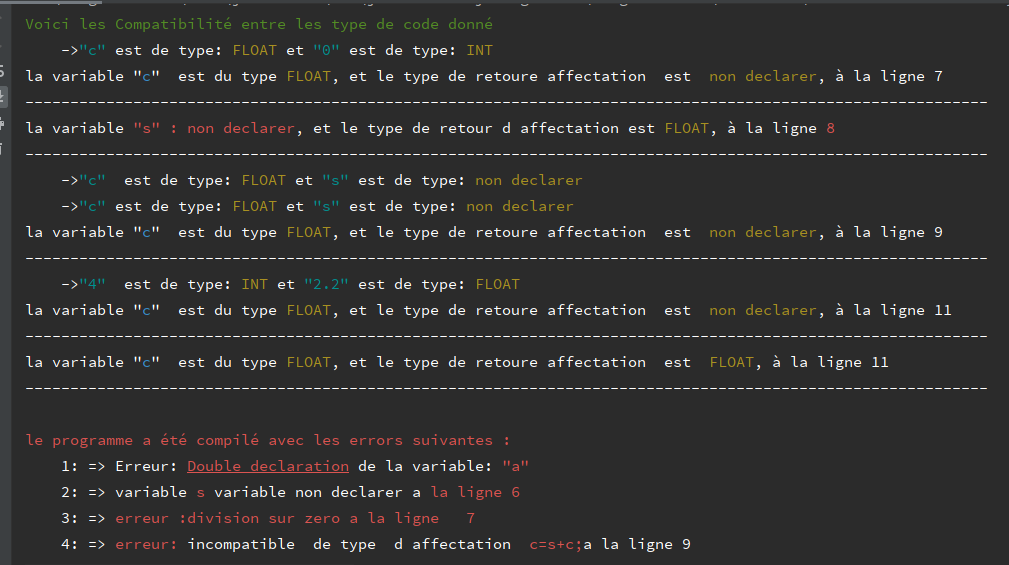


* Division sur Zéro :

La division sur 0 est une erreur (ou exception) sur la plupart des compilateurs parmi le nôtre, voici un exemple : on a ajouté c=c/0

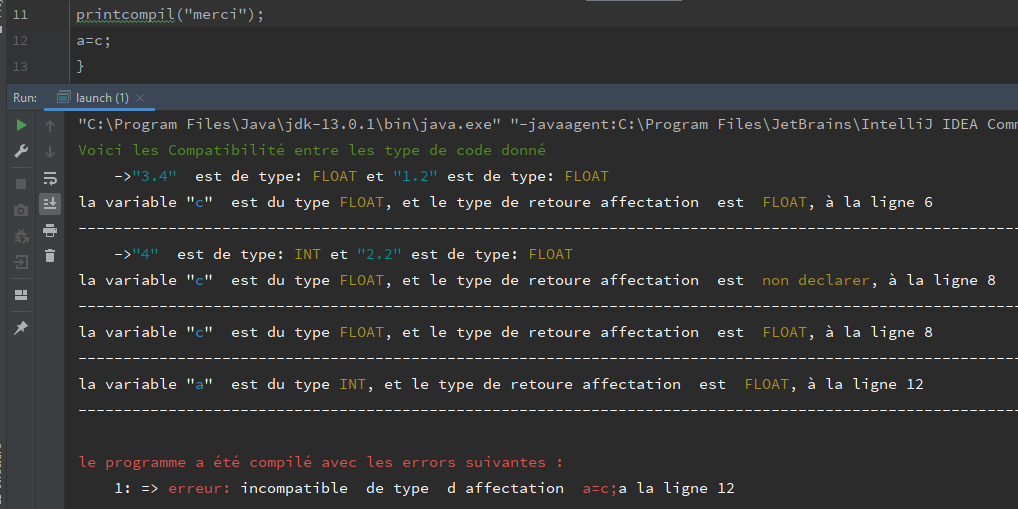


Voici l’erreur sémantique correspondant :

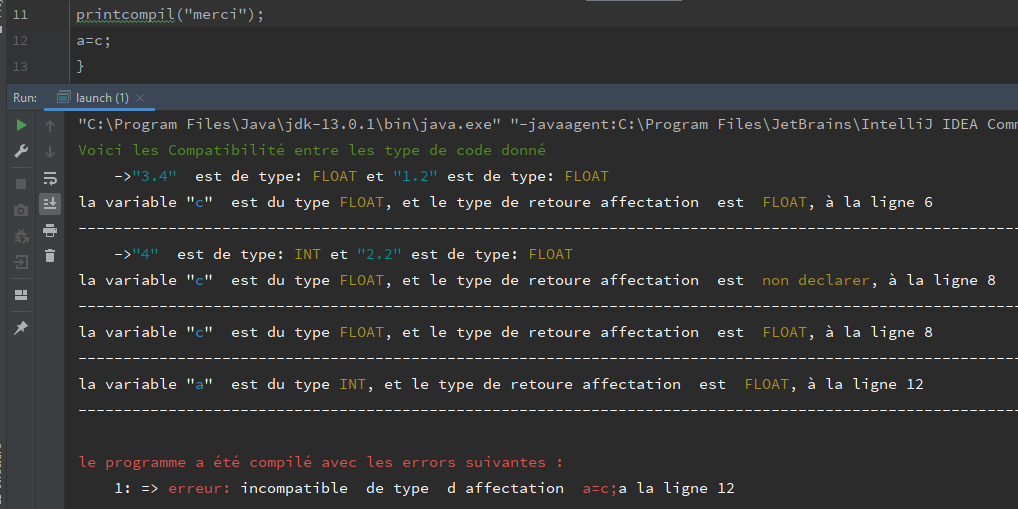


* Incompatibilité des types :

On ne peut pas avoir par exemple un entier qui recoin un type avec virgule flottante, car la partie après la virgule sera ignoré, la plupart des compilateurs le déplanchera comme une erreur, parmi la nôtre, voici un exemple de code précédant (a est un entier et c est un float) :

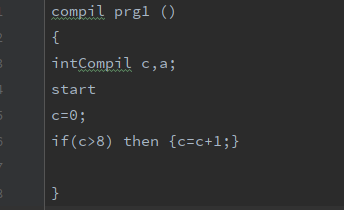


Voici l’erreur sémantique correspondant :

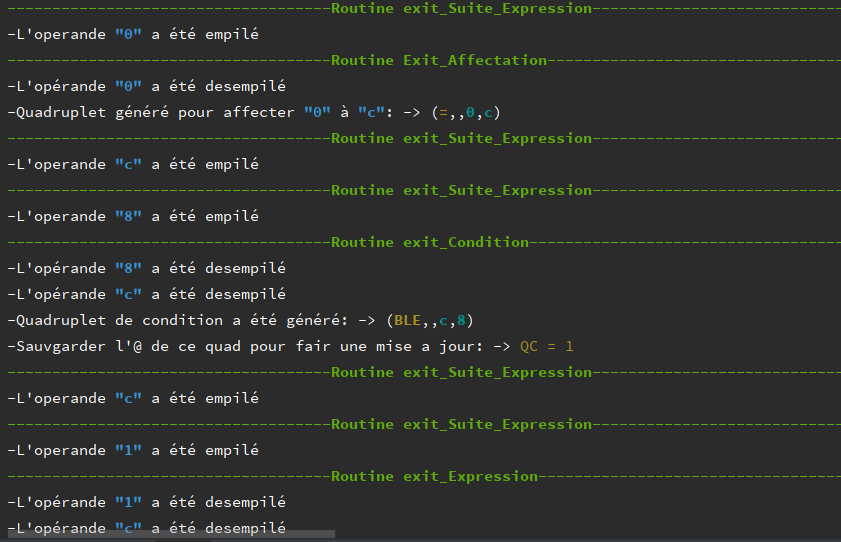


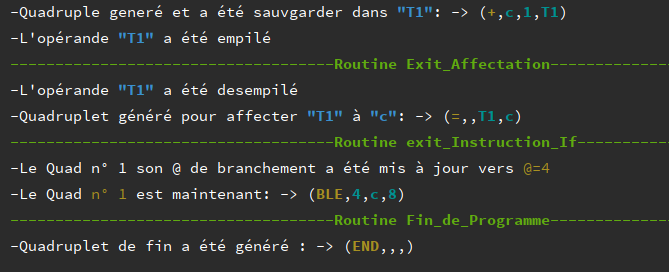
4- Génération des quadruplets :

Les quadruplets ne se génères pas si les code contient les erreurs (syntaxique/sémantique) il faut qu’ils passent l’analyse sémantique pour pouvoir créé les formes intermédiaires du codes. Voici un exemple de code correct qu’à partir on va générer les formes intermédiaire (Les quads) :

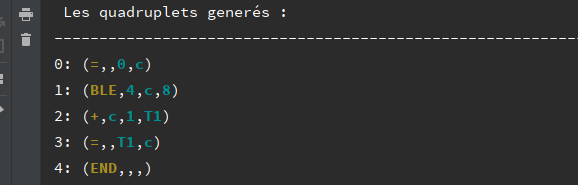


Voici les résultats des routines qui se déclenchent après l’analyse de ce code :



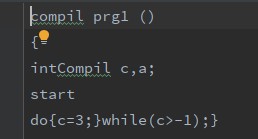


Les Quadruplets générés :

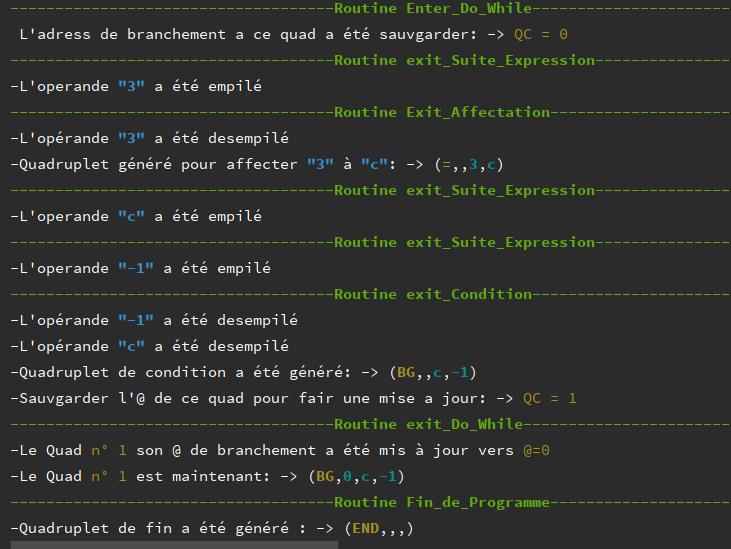


* Un autre exemple (instruction Do-While)

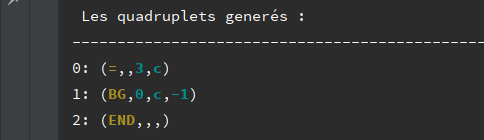
Voici un exemple de code qui contient l’instruction do-while



Les routines qui se déclenchent sont :



Les quads générés :



5- Code Objet + Optimisation

Le code objet se créé après la génération des quadruplets, il utilise la fonction getInAcc pour minimalisé les instructions assembleur, on prend un exemple de code simple suivant

Compil prg1 (){

stringCompil th ;

intCompil b,a ;

floatCompil c ;

Start

c=2+4 ;

}

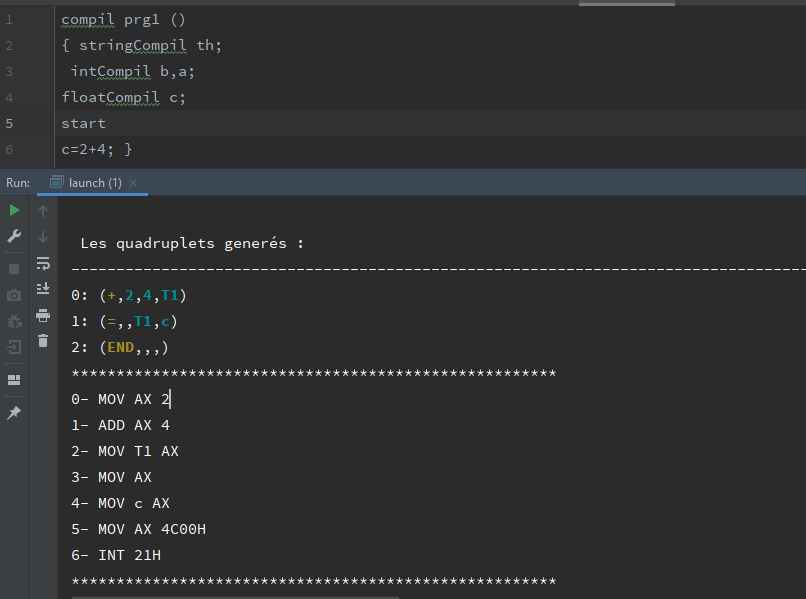
Les quads générés sont :

0 : (+,2,4,T1)

1 : (=, ,T1,c)

2 : (END, , , )

La traduction en code assembleur sera :



* Un autre exemple :

Le code assembleur est généré de manière fausse lorsqu’il rencontre un quadruplet qui contient un branchement (il met l’étiquette de jump à une valeur fausse)

