



REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

UNIVERSITE DE SCIENCE ET DE TECHNOLOGIE HOUARI BOUMEDIENE

Département d'informatique

Compilation

Projet Small Java

Réalisé par:

**BOUNOUH NESRINE
SAHNOUNE NASSIMA**

**Année universitaire:
2019-2020**

Table de matière

Introduction

Chapitre1: L'environnement de travail et La grammaire Lexer et Parser

Introduction	1
L'environnement de travail	1
Grammaire	1
Conclusion	5

Chapitre2: Les erreurs sémantiques

Introduction	6
Les importations	6
Les variables	8
Variable doublement déclarée	9
La longueur d'un identificateur	10
Une variable non déclarée	11
Une variable non initialisée	13
Les expressions arithmétiques	15
Incompatibilité de type	15
Importation du bon package	29
Le type String_SJ	30
Les I/O	31
L'importation du bon packages	31
La compatibilité entre le format et le type	32
Initialiser la variable en question	33
Les conditions	33

Chapitre 3: Les quadruplets

Introduction	37
Structure de données utilisées	37
Les quadruplets des déclarations	37

Table de matière

Les quadruplets des affectations	38
Les quadruplets des expressions arithmetiques	39
Les quadruplets des I/O	39
Les quadruplets des conditions	39
Conclusion	40
Chapitre4: Le code objet	
Introduction	41
Les declarations	41
La partie code	42
L'affectation	42
Les expressions arithmetiques	43
Les E/S	45
Les conditions	45

Table des figures

<i>Illustration 1: Les erreurs sémantiques des importations</i>	8
<i>Illustration 2: La longueur de l'identificateur</i>	11
<i>Illustration 3: Routine pour les variables non déclaré</i>	12
<i>Illustration 4: Routine pour les variables non initialisée</i>	14
<i>Illustration 5: Incompatibilité de type</i>	29
<i>Illustration 6: importation de package pour les E/S et opérations arithmétiques</i>	30
<i>Illustration 7: importer le bon package pour les I/O</i>	32
<i>Illustration 8: incompatibilité de type dans les I/O</i>	33
<i>Illustration 9: incompatibilité de type dans les conditions</i>	35
<i>Illustration 10: Les déclarations en assembleurs</i>	42
<i>Illustration 11: Les affectations en assembleurs</i>	43
<i>Illustration 12: Les expressions arithmétiques en assembleurs</i>	45

Introduction

De nos jours, Apprendre à créer un compilateur est une obligation pour un programmeur. Avec l'évolution de l'IA et surtout avec l'apparition des jeux vidéo, il nécessaire de trouver de meilleurs outils pour faciliter la tâche de création d'un compilateur.

Il y a quelques années il existait l'outil flex et bison mais il apparait très compliqué à manipuler d'ou l'outil ANTLR est apparu. Qui est un outil très puissant, facile à utiliser.

Durant ce projet Small_JAVA, nous allons créer un compilateur pour le langage Small_JAVA dont nous allons expliquer chaque phase en détails.

Ce projet est constitué de quatre chapitres, commençant par l'environnement de travail puis la grammaire. Ensuite les erreurs sémantiques et les quadruplets. Enfin le code objet.

Introduction:

ANTLR nous offre un outil puissant qui est le Lexer et Parser. Ces deux outils permettent de générer une grammaire descendante, Les entités lexicales sont écrites en majuscule et les règles sont écrites en miniscule. Puisque il s'agit d'une grammaire descendante, les règles sont définis avant les entités.

Tout au long de notre projet small_java, nous allons utiliser la grammaire suivante; Dont nous avons respecter toutes les contraintes mentionnées dans l'énoncé.

L'environnement de travail:

- Système d'exploitation: Windows10
- Logiciel de programmation: IntelliJ

Grammaire:

```
grammar grammarOfProject;
```

```
//rules
```

```
prog: (|imports) (|modificateur) CLASS CLASSNAME '{' declaration MAIN '{' insts '}' '}';
```

```
imports : IMPORT libraryName ';' imports | IMPORT libraryName ';;
```

```
libraryName: JAVAIO | JAVALONG;
```

```
modificateur: PUBLIC | PRIVATE;
```

```
//declaration part
```

```
declaration: (decvar declaration) | decvar;
```

```
decvar: type variablelist ';;
```

```
type: INT | FLOAT | STRING;
```

```
variablelist: (IDF ',' variablelist) | IDF;
```

```
//instructions part
```

```
insts: (inst ';' insts) | inst ';' | siinsts insts | siinsts;
```

Chapitre 1 : La grammaire Lexer et Parser

inst : assignment | input | output;

//assignment

assignment: identifier ':=' arithmetic_expression ;

identifieur : IDF;

//arithmetic expression

arithmetic_expression: '(' arithmetic_expression ')'

 | arithmetic_expression DIV arithmetic_expression

 | arithmetic_expression MUL arithmetic_expression

 | arithmetic_expression PLUS arithmetic_expression

 | arithmetic_expression MINUS arithmetic_expression

 | val

 | identifier

 | "\" TEXT \"";

val: VALUEOFFLOAT | VALUEOFINTEGER | VALUEOFSTRING;

//input

input: IN '(' "\" format "\" ',' identifier ')';

format: INTFORMAT | FLOATFORMAT | STRINGFORMAT;

//output

output: OUT '(' "\" format "\" ',' identifier')'

 | OUT '(' "\" IDF format "\" ',' identifier')';

//conditions

Chapitre 1 : La grammaire Lexer et Parser

```
siinsts: SI '(' condition ')' alors '{' insts '}'  
      | SI '(' condition ')' alors '{' insts '}' sinon;  
alors: ALORS;  
sinon: SINON '{' insts '}';  
condition: logic | arithmetic;  
logic: NOT logic  
      | logic AND logic  
      | logic OR logic  
      | '(' condition ')';  
arithmetic: arithmetic_expression GREATER arithmetic_expression  
          | arithmetic_expression LESS arithmetic_expression  
          | arithmetic_expression GREATEROREQUAL arithmetic_expression  
          | arithmetic_expression LESSOREQUAL arithmetic_expression  
          | arithmetic_expression EQUAL arithmetic_expression  
          | arithmetic_expression DIFFERENT arithmetic_expression;  
  
//declare key words  
PUBLIC: 'public';  
PRIVATE: 'protected';  
CLASS: 'class_SJ';  
MAIN: 'main_SJ';  
IMPORT: 'import';  
  
//type  
INT: 'int_SJ';  
FLOAT: 'float_SJ';
```


Chapitre 1 : La grammaire Lexer et Parser

STRING: 'string_SJ';

//imports

JAVAIO: 'Small_Java.io';

JAVALONG: 'Small_Java.lang';

//arithmetic operator

PLUS: '+';

MUL: '*';

DIV: '/';

MINUS: '-';

//login operator

GREATER: '>';

LESS: '<';

EQUAL: '=';

DIFFERENT: '!=';

GREATEROREQUAL: '>=';

LESSOREQUAL: '<=';

AND: '&';

OR: '|';

NOT: '!';

//format

INTFORMAT: '%d';

FLOATFORMAT: '%f';

Chapitre 1 : La grammaire Lexer et Parser

```
STRINGFORMAT: '%s';

//input/output expression
OUT: 'OUT_SJ';
IN: 'IN_SJ';

// condition
SI: 'si';
ALORS: 'alors';
SINON: 'sinon';

//texte
IDF: [a-z][a-z0-9]*; //identificateur ne dépasse pas 10 lettre
CLASSNAME: [A-Z][A-Za-z0-9]*;
TEXT : [a-zA-Z]+;
//val idf
VALUEOFSTRING: ""(~["]"\\")*"";
VALUEOFFLOAT: [+]?[0-9]+'.[0-9]+ ;
VALUEOFINTEGER: '0'|[+]?[1-9][0-9]* ;

//skip some values
WHITESPACE : [ \n\t\r] -> skip;
```

Conclusion:

La grammaire précédente nous a permis de vérifier la validité de code coté lexicale et syntaxique Car c'est au Lexer et Parser le valider.

Chapitre 2: Les erreurs sémantiques

Introduction:

Avec le Lexer et Parser, nous avons généré une grammaire correcte lexicalement et syntaxiquement. Car ANTLR va automatiquement générer une erreur si le code introduit par le programmeur ne correspond pas exactement à notre grammaire. Mais il ne permet pas de vérifier la sémantique du code, et ça à notre compilateur `grammarOfProject` de gérer ce genre d'erreur.

1.1. Les importations :

Il arrive souvent pour un développeur d'importer un package deux fois, ce qui cause un souci au niveau de la zone mémoire. Afin de pouvoir optimiser le code au maximum, nous avons pensé à avertir le développeur au lieu de le pénaliser en déclenchant une erreur.

- Explication détaillée:

Le langage développé permet d'importer deux types de packages :

- ❖ `Small_Java.io` : un package importé lors des opérations d'entrée sortie.
- ❖ `Small_Java.lang` : un package importé lors des opérations arithmétiques.

La syntaxe utilisée pour importer un package est : `import Small_Java.io` ou `import Small_Java.lang`. Il peut importer uniquement le premier package ou le deuxième ou alors les deux en même temps.

Afin de pouvoir réaliser cette opération d'import, nous avons rédigé cette partie dans la grammaire :

```
imports    :    IMPORT    libraryName    ';'    imports    |    IMPORT    libraryName    ';;  
libraryName: JAVAIO | JAVALONG;
```

`Imports` est un non terminal, qui définit la règle générale de l'importation. D'abord le mot clé `import` puis le nom de la bibliothèque à importer suivie par un `' ; '` qui marque la fin de l'instruction. Pour exprimer le nombre de fois que le développeur peut importer un package donné, nous avons distingué deux cas, soit `import libraryName ' ; '` qui signifie un seul import ou alors répétition avec une récursivité.

`libraryName` est non terminal qui désigne le nom de package précédemment définie.

Voilà, ce qui concerne la partie grammaire qui permet de traiter les importations maintenant, sans doute, nous devons expliquer la partie routine.

Chapitre 2: Les erreurs sémantiques

Pour la partie routine, ANTLR nous propose une interface `grammarOfProjectBaseListner.java` qui contient deux méthodes différentes pour chaque nœud, `enter` qui est déclenché lorsque on entre au nœud, et la méthode `exit` lors de la sortie. Dans notre projet, nous utilisons uniquement la méthode `exit`.

Dans cette partie, nous bénéficions de la méthode :

```
@Override public void exitLibraryName(grammarOfProjectParser.ArithmeticContext ctx) { }
```

Afin de vérifier, si nous avons déjà importer ce package ou non, nous utilisons une `HashMap`. Grâce à sa propriété pas de répétition pour les clés, nous pouvons vérifier avec la méthode `containsKey` si le package est déjà importer ou non, si c'est le cas, nous allons déclencher un warning, sinon nous allons l'ajouter puisque c'est sa première apparition.

Le corps dans notre méthode devient :

```
/** check if there is double import or not*/
```

```
@Override public void exitLibraryName(grammarOfProjectParser.LibraryNameContext ctx) {  
    String libraryName = ctx.getText();  
    if(imports.containsKey(libraryName)){  
        /** error double import*/  
        warnings.add("nes/w1: Double import\n"+ libraryName+" is imported multiple time");  
    }else{  
        imports.put(libraryName,libraryName);  
    }  
}
```

Test :

```
import Small_Java.io;  
import Small_Java.lang;  
import Small_Java.lang;  
import Small_Java.lang;
```

Chapitre 2: Les erreurs sémantiques

```
protected class_SJ Test{  
    float_SJ c,d,k;  
    int_SJ a,e,f;  
    string_SJ b,m,g,l;  
    main_SJ{  
        f := 5 ;  
    }  
}
```

Résultat :

```
program compiled without errors!  
errors: 0  
warnings: 2  
nes/w1: Double import  
Small_Java.lang is imported multiple time  
nes/w1: Double import  
Small_Java.lang is imported multiple time
```

Illustration 1: Les erreurs sémantiques des importations

1.2. Les variables :

Les problèmes posés par les variables sont :

- ❖ Une variable utilisée mais non déclarée.
- ❖ Une variable doublement déclarée.
- ❖ Une variable non initialisée.
- ❖ Le nom d'une variable supérieur à 10.

Commençons à résoudre les problèmes un par un.

Chapitre 2: Les erreurs sémantiques

Une variable doublement déclarée :

A la fin des déclarations de toutes les variables on procède à la vérification des variables doublement déclarées par cette partie de grammaire:

declaration: (decvar declaration) | decvar;

decvar: type variablelist ';';

type: INT | FLOAT | STRING;

variablelist: (IDF ' ' variablelist) | IDF;

Le nom decvar permet de déclarer plusieurs variables avec son type. A la sortie de ce nœud decvar, nous allons avoir la liste de nos variables dans variablelist. Ce qui va nous permettre avec un simple parcourt de détecter si la variable est déjà déclarée ou non.

La fonction suivante qui permet de déclencher la routine qui fait la vérification précédente.

```
@Override public void exitDecvar(grammarOfProjectParser.DecvarContext ctx) {  
    /** retrieve the type*/  
    int type ;  
  
    if(ctx.type().getText().equals("int_SJ")){  
        type = 0;  
    }else if(ctx.type().getText().equals("float_SJ")){  
        type = 1;  
    }else{  
        type = 2;  
    }  
  
    grammarOfProjectParser.VariablelistContext variablelist = ctx.variablelist();  
  
    for( ; ; variablelist = variablelist.variablelist()){
```

Chapitre 2: Les erreurs sémantiques

```
String nameOfVariable = variablelist.getChild(0).getText();
```

```
if(ts.containsToken(nameOfVariable)) {  
    errors.add("nes/e1: Double declaration of variable of\n" + nameOfVariable );
```

```
}else {  
    /** check the size of idf*/  
    if(nameOfVariable.length()>10){  
        errors.add("nes/e2: IDF too long\n"+nameOfVariable+" longer than 10");  
    }else{  
        ts.addToken(new TokenClass(nameOfVariable,DECLARED,type));  
    }  
  
    }  
    if(variablelist.variablelist() == null) return;  
}  
}
```

La longueur d'un identificateur :

Dans la même fonction, nous pouvons vérifier la longueur d'une variable en utilisant la méthode java lenght().

Test :

```
import Small_Java.io;  
import Small_Java.lang;  
protected class _SJ Test{  
    float _SJ c,d,kghhggfffttyjjhhggfdcv;  
    int _SJ a,e,f,d;
```

Chapitre 2: Les erreurs sémantiques

```
main_SJ{  
    f := 5;  
}  
}
```

Résultat :

```
program compiled with the following errors  
nes/e2: IDF too long  
kghhggfffttyjjhhggfdcv longer than 10  
nes/e1: Double declaration of variable of  
d
```

Illustration 2: La longueur de l'identificateur

Une variable non déclarée :

Pour dire qu'une variable n'est pas déclarée, il faudra bien qu'elle soit utilisée. Elle pourra être utilisée dans plusieurs cas : une affectation, une opération de lecture/écriture, les expressions arithmétiques, ... etc. Et pour cela, nous avons défini un nœud identifier qui fait référence à une variable. La vérification de la déclaration ou non d'une variable utilisée se fait par cette partie de grammaire:

identifieur : IDF;

Et voici la partie de la routine :

```
/** check if an idf is declared or not  
 * if it's not declared i generate the error but also declare it to not generate  
 * the error more than once  
 * @param ctx  
 */  
  
@Override public void exitIdentifier(grammarOfProjectParser.IdentifierContext ctx) {  
    String isDeclared = ctx.IDF().getText();  
    if(! ts.containsToken(isDeclared)) {
```


Chapitre 2: Les erreurs sémantiques

```
errors.add("nes/e4 missing declaration\n" + isDeclared + " is used but not declared ");

/** the type of the token is not important because will have an error */

//TODO check if type == -1

ts.addToken(new TokenClass(isDeclared,1,-1));

}

}
```

Test :

```
import Small_Java.io;

import Small_Java.lang;

protected class_SJ Test{

    float_SJ c,d;

    int_SJ a,e;

    string_SJ b,m,g,l;

    main_SJ{

        f := 5;

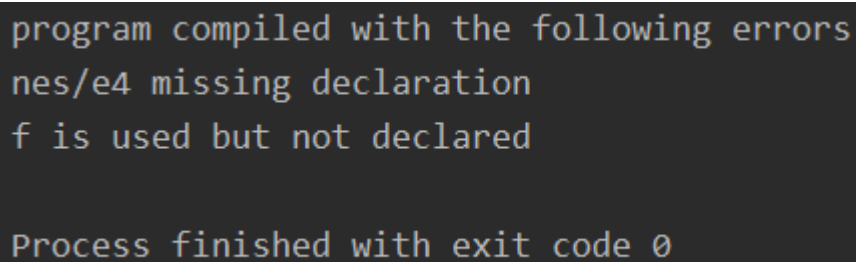
        a:= (f + 45);

        e := 15;

    }

}
```

Résultat :

A screenshot of a terminal window with a dark background and light-colored text. It displays the output of a compilation process. The text shows that the program was compiled with two semantic errors: 'nes/e4 missing declaration' and 'f is used but not declared'. The process finished with an exit code of 0.

```
program compiled with the following errors
nes/e4 missing declaration
f is used but not declared

Process finished with exit code 0
```

Illustration 3: Routine pour les variables non déclaré

Chapitre 2: Les erreurs sémantiques

Bien entendu, l'erreur ne doit pas être générée plusieurs fois, comme solution est la déclarée mais la question qui se pose qu'elle type devons nous lui affecter. Tel que cité avant, nous avons trois type : String_SJ, float_SJ, int_SJ. Nous avons constaté que nous aurons besoin d'un quatrième type celui de l'erreur. Car, notre programme ne s'arrête pas à la première erreur et ne répète pas la même erreur, donc certainement pour les variables non déclarées il doit avoir un type qui est -1 qui désigne type erroné. Nous revenons à cette remarque un peu plus tard, lorsque nous commençons le traitement des expressions arithmétiques.

Une variable non initialisée :

Une variable non initialisée cause un problème dans les affectations et les expressions arithmétiques car, lors de l'exécution, nous allons récupérer les valeurs mais si comme par hasard, une des valeurs manques, nous aurons un souci lors de l'exécution. Donc il est vraiment nécessaire de Contrôler que chaque variable utilisée est déjà initialisée.

La partie de la grammaire :

```
arithmetic_expression: '(' arithmetic_expression ')'
                        | arithmetic_expression DIV arithmetic_expression
                        | arithmetic_expression MUL arithmetic_expression
                        | arithmetic_expression PLUS arithmetic_expression
                        | arithmetic_expression MINUS arithmetic_expression
                        | val
                        | identifier
                        | "\" TEXT \"";
```

//output

```
output: OUT '(' "\" format "\" ',' identifier')
        | OUT '(' "\" IDF format "\" ',' identifier');
```

Dans les deux cas précédents, nous pouvons avoir le problème d'initialisation. Donc, ce n'est pas un problème général aux identificateurs, mais uniquement lorsque nous avons besoin de leurs valeurs.

La partie de la routine :

```
if(ts.getToken(ctx.identifieur().getText()).getInitialized() == 0){
    errors.add("nes/e5: missing initialization\n " + ctx.identifieur().getText()+" is not initialized");
}
```

Chapitre 2: Les erreurs sémantiques

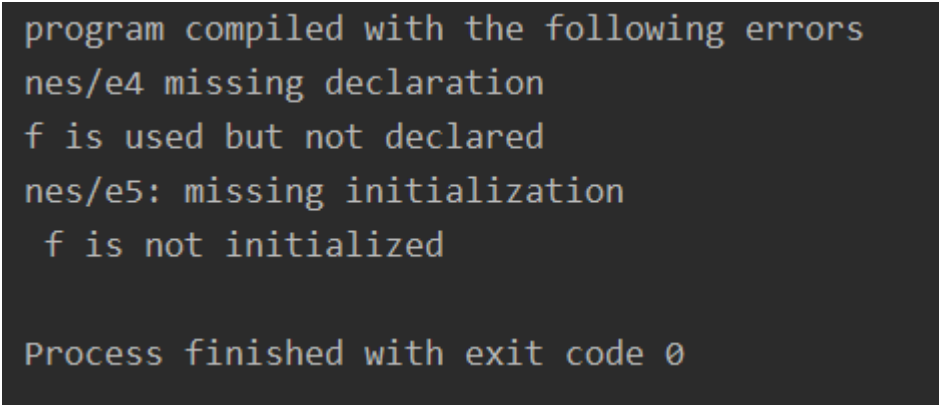
Pour celà, nous avons juste besoin de vérifier si la variable est initialisée ou non en utilisant la table des symboles. Qui contient pour chaque variable un champs nommé initialisation qui a 1 si la variable est initialisée ou alors 0 sinon.

Test :

```
import Small_Java.io;
import Small_Java.lang;
protected class_SJ Test{
    float_SJ c,d;
    int_SJ a,e;
    string_SJ b,m,g,l;
    main_SJ{
        a:= (f + 45);
        e := 15;
        OUT_SJ('salam %f',f);

    }
}
```

Résultat :



```
program compiled with the following errors
nes/e4 missing declaration
f is used but not declared
nes/e5: missing initialization
f is not initialized

Process finished with exit code 0
```

Illustration 4: Routine pour les variables non initialisée

1.3. Les expressions arithmétiques :

Les routines déclencher par les expressions arithmétiques :

Chapitre 2: Les erreurs sémantiques

- ❖ L'incompatibilité de type.
- ❖ L'importation du bon package.
- ❖ Récupération des valeurs des expressions et des types.

L'incompatibilité de type :

Dans l'incompatibilité de type, on trouve deux cas :

- L'affectation :

Dans ce cas les types suivants sont acceptés :

- ✓ Int_SJ = Int _SJ.
- ✓ Int_SJ = type erroné : ce cas est accepté, car comme déjà cité, notre programme ne s'arrête pas suite à une seule erreur. Mais collecte l'ensemble des erreurs et les affichent à la fois. Lorsqu'on recherche le type d'une variable on le fait soit par héritage soit par table de symbole. Dans le cas d'une table de symbole la valeur sera nulle, si on n'ajoute pas le type erroné et le compilateur s'arrête suite à une erreur. Donc le type erroné signifie qu'il va exister une erreur quel que soit x mais existe-il une autre erreur. L'incompatibilité de type ne doit pas être générée car c'est nous qu'il avons provoqué.
- ✓ Float_SJ = int_SJ.
- ✓ Float_SJ = float_SJ.
- ✓ Float_SJ = type erroné.
- ✓ String_SJ = String_SJ.
- ✓ String_SJ = Float_SJ.
- ✓ String_SJ = type erroné.

Tous les autres types sont erronés.

Pour vérifier la compatibilité de type nous avons écrit une fonction :

```
private static boolean checkTypeCompatibleForAssignment(int t1, int t2){
```

```
    /** i accept this following assignment type
```

```
    * int = int
```

```
    * float = float
```

```
    * float = int
```

Chapitre 2: Les erreurs sémantiques

```
* string = string
* string = int
* string = float
* except this everything is refused
*/

if(t1 == 0 && t2 == 0) return true;
if(t1 == 1 && t2 == 1) return true;
if(t1 == 1 && t2 == 0) return true;
if(t1 == 2 && t2 == 2) return true;
if(t1 == 2 && t2 == 1) return true;
if(t1 == 2 && t2 == 0) return true;

/** -1 is always accepted because it generates an error*/
if(t1 == 0 && t2 == -1) return true;
if(t1 == 1 && t2 == -1) return true;
if(t1 == -1 && t2 == 0) return true;
if(t1 == 2 && t2 == -1) return true;
if(t1 == -1 && t2 == 2) return true;
if(t1 == -1 && t2 == 1) return true;

return false;
}
```

Maintenant, pour pouvoir faire cette vérification, nous devons récupérer les types des variables et expressions. Par exemple, si nous avons $f := (a+b)+c+e$ nous devons connaître le type de l'expression : $(a+b)+c+e$ et le vérifier avec f . pour cela, nous utilisons le principe de l'héritage. A chaque fois qu'on rencontre un idf, on recherche son type dans la table des symboles et on le sauvegarde dans une HashMap types avec la clé ctx qui est le idf lui-même et la valeur (0|1|2|-1) tout dépend de son type. La même chose pour une valeur sauf que son type est récupérée selon le nœud non nul soit VALUEOFFLOAT | VALUEOFINTEGER | VALUEOFSTRING, avec la clé la valeur et son type. Dans le cas des expressions, le type est calculé avec la méthode suivante :

```
private static ErrorType getResultingType(int t1, int t2){
```

Chapitre 2: Les erreurs sémantiques

```
ErrorType errortype = null;

if(t1 == 0 && t2 == 0){
    errortype = new ErrorType(INTSJ,INTSJ);
    return errortype;
}

if(t1 == 0 && t2 == 1){
    errortype = new ErrorType(FLOATSJ,FLOATSJ);
    return errortype;
}

if(t1 == 1 && t2 == 0){ Chapitre 1 : La grammaire Lexer et Parser
    errortype = new ErrorType(FLOATSJ,FLOATSJ);
    return errortype;
}

if(t1 == 1 && t2 == 1){
    errortype = new ErrorType(FLOATSJ,FLOATSJ);
    return errortype;
}

if (t1 == 2 && t2 == 2) {
    errortype = new ErrorType(STRINGSJ,STRINGSJ);
    return errortype;
}

if (t2 == 2 && t1 == 2) {
    errortype = new ErrorType(STRINGSJ,STRINGSJ);
    return errortype;
}

if (t2 == 2 && t1 == 1) {
    errortype = new ErrorType(STRINGSJ,STRINGSJ);
```

Chapitre 2: Les erreurs sémantiques

```
    return errortype;
}

if (t2 == 2 && t1 == 0) {
    errortype = new ErrorType(STRINGSJ,STRINGSJ);
    return errortype;
}

if (t2 == 0 && t1 == 2) {
    errortype = new ErrorType(STRINGSJ,STRINGSJ);
    return errortype;
}

if (t2 == 1 && t1 == 2) {
    errortype = new ErrorType(STRINGSJ,STRINGSJ);
    return errortype;
}

/** -1 with any type is the type because it'll generate an error*/
if(t1 == 0 && t2 == -1){
    errortype = new ErrorType(INTSJ,-1);
    return errortype;
}

if(t1 == 1 && t2 == -1){
    errortype = new ErrorType(FLOATSJ,-1);
    return errortype;
}

if(t1 == 2 && t2 == -1){
    errortype = new ErrorType(STRINGSJ,-1);
    return errortype;
}
```

Chapitre 2: Les erreurs sémantiques

```
}  
  
if(t1 == -1 && t2 == 0){  
    errortype = new ErrorType(INTSJ,-1);  
    return errortype;  
}  
  
if(t1 == -1 && t2 == 1){  
    errortype = new ErrorType(FLOATSJ,-1);  
    return errortype;  
}  
  
if(t1 == -1 && t2 == 2){  
    errortype = new ErrorType(FLOATSJ,-1);  
    return errortype;  
}  
  
return new ErrorType(STRINGSJ,-1);  
  
}
```

Et il est sauvegardé avec le ctx qui désigne l'expression elle-même et le type calculé comme valeur dans le HASHMAP types.

Comme ça, nous allons obtenir le type de l'expression de l'affectation et nous vérifions est ce que ce type est compatible ou non avec une simple instruction :

```
if(!  
checkTypeCompatibleForAssignment(ts.getToken(ctx.identifieur().getText()).getType(),types.get(ctx.ari  
thmetic_expression())))  
  
    errors.add("nes/e3: incompatible type in assignment\n Hints: \n you can't affect to a integer  
neither a string nor a float\n");
```

- Entre les operateurs de l'expression arithmétique:

Chapitre 2: Les erreurs sémantiques

sans vous rendre compte, cette partie est déjà expliquée lorsque nous avons créé la fonction `getResultingType`. Le type de cette fonction est `ErrorType` qui est une classe de deux attributs `resultType` et `errorType`.

Si `errorType == -1`

Alors une incompatibilité de type, mais l'expression reçoit le type erroné `-1` au lieu de `null` puisque le compilateur continue son exécution.

Sinon pas d'erreur l'expression reçoit le `resultType`.

Voilà ce qui concerne la partie incompatibilité de type.

La grammaire dans les expressions arithmétiques :

```
arithmetic: arithmetic_expression GREATER arithmetic_expression
| arithmetic_expression LESS arithmetic_expression
| arithmetic_expression GREATEROREQUAL arithmetic_expression
| arithmetic_expression LESSOREQUAL arithmetic_expression
| arithmetic_expression EQUAL arithmetic_expression
| arithmetic_expression DIFFERENT arithmetic_expression;
```

La routine:

- Cas d'affectation:

```
@Override public void exitAssignment(grammarOfProjectParser.AssignmentContext ctx) {
    /** check for compatible type*/
    if(!
checkTypeCompatibleForAssignment(ts.getToken(ctx.identifier().getText()).getType(),types.get(ctx.ari
thmetic_expression()))
        errors.add("nes/e3: incompatible type in assignment\n Hints: \n you can't affect to a integer
neither a string nor a float\n");

    types.clear();

    ts.getToken(ctx.identifier().getText()).setValue(values.get(ctx.arithmetic_expression()));

    values.clear();
}
```

Chapitre 2: Les erreurs sémantiques

```
/** init variable*/
```

```
ts.getToken(ctx.identifieur().getText()).setInitialized(1);
```

```
}
```

- Cas entre opérateur:

```
@Override public void  
exitArithmetic_expression(grammarOfProjectParser.Arithmetic_expressionContext ctx) {  
    //System.out.println(ctx.getText());  
    if(ctx.arithmetic_expression().size() == 1){  
        String[] sep = ctx.getText().split("\\(|\\)");  
        if(!sep[1].equals("")){  
            ParserRuleContext save =null;  
            for (Map.Entry<ParserRuleContext, Integer> entry : types.entrySet()) {  
                if(entry.getKey().getText().equals(sep[1])){  
                    save = entry.getKey();  
                }  
            }  
            types.put(ctx,types.get(save));  
            values.put(ctx,values.get(save));  
        }  
    }  
  
    if(ctx.identifieur() != null){  
        types.put(ctx, ts.getToken(ctx.identifieur().getText()).getType());
```

Chapitre 2: Les erreurs sémantiques

```
values.put(ctx,ts.getToken(ctx.identifieur().getText()).getValue());

s.push(ctx);

}

if(ctx.val() != null){
    types.put(ctx, types.get(ctx.val()));
    values.put(ctx,values.get(ctx.val()));
    s.push(ctx);
}

if(ctx.PLUS() != null){
    if(! imports.containsKey("Small_Java.lang")) {
        errors.add("nes/e6 missing import\n add import Small_java.lang to your package list");
        imports.put("Small_Java.lang", "Small_Java.lang");
    }

    ParserRuleContext second = (ParserRuleContext) s.pop();
    ParserRuleContext first = (ParserRuleContext) s.pop();
    if(checkTypeCompatible(types.get(first),types.get(second))){
        ErrorType resultingType = getResultingType(types.get(first),types.get(second));
        types.put(ctx, resultingType.getResultType());
        if(resultingType.getErrorType() != -1){
            String param1 = values.get(first);
            String param2 = values.get(second);
            if(resultingType.getResultType() == 0){
                if(param1 != null && param2 != null){
                    int resultValue = getresultingValueOfInteger(param1,param2,"+");
                    values.put(ctx,String.valueOf(resultValue));
                }
            }
        }
    }
}
```

Chapitre 2: Les erreurs sémantiques

```
    }else if(resultingType.getResultType() == 1){
        if(param1 != null && param2 != null){
            float resultValue = getResultingValueOfFloat(param1,param2,"+");
            values.put(ctx,String.valueOf(resultValue));
        }else{
            System.out.println("253");

        }
    }else{
        if(param1 != null && param2 != null){
            String resultValue = getResultingValueOfString(param1,param2,"+");
            values.put(ctx,resultValue);

        }

    }

    }else{
        values.put(ctx,"-1");
    }
    }else{
        errors.add("249: incompatible type");
        types.put(ctx, -1);
    }
    s.push(ctx);
}
if(ctx.DIV() != null){
```

Chapitre 2: Les erreurs sémantiques

```
if(! imports.containsKey("Small_Java.lang")) {
    errors.add("nes/e6 missing import\n add import Small_java.lang to your package list");
    imports.put("Small_Java.lang", "Small_Java.lang");
}

ParserRuleContext second = (ParserRuleContext) s.pop();
ParserRuleContext first = (ParserRuleContext) s.pop();

/** devision by zero*/
if(values.get(second) != null && values.get(second).equals("0")){
    errors.add("265: division by zero");
}

if(checkTypeCompatible(types.get(first),types.get(second))){
    ErrorType resultingType = getResultingType(types.get(first),types.get(second));
    types.put(ctx, resultingType.getResultType());
    if(resultingType.getErrorType() != -1){
        String param1 = values.get(first);
        String param2 = values.get(second);
        if(resultingType.getResultType() == 0){
            if(param1 != null && param2 != null){
                int resultValue = getresultingValueOfInteger(param1,param2,"/");
                values.put(ctx,String.valueOf(resultValue));
            }
        }
        }else if(resultingType.getResultType() == 1){
            if(param1 != null && param2 != null){
                float resultValue = getresultingValueOfFloat(param1,param2,"/");
                values.put(ctx,String.valueOf(resultValue));
            }
        }
```

Chapitre 2: Les erreurs sémantiques

```
    }else{
        errors.add("292: operator not allowed to string");

    }
}
}else{
    values.put(ctx, "-1");
}
}else{
    errors.add("249: incompatible type");
    types.put(ctx, -1);
}
s.push(ctx);
}

if(ctx.MUL() != null){
    if(! imports.containsKey("Small_Java.lang")) {
        errors.add("nes/e6 missing import\n add import Small_java.lang to your package list");
        imports.put("Small_Java.lang", "Small_Java.lang");
    }
    ParserRuleContext parent = (ParserRuleContext) s.pop();
    ParserRuleContext second = (ParserRuleContext) s.pop();
    ParserRuleContext first = (ParserRuleContext) s.pop();
    if(checkTypeCompatible(types.get(first),types.get(second))){
        ErrorType resultingType = getResultingType(types.get(first),types.get(second));
        types.put(ctx, resultingType.getResultType());
        if(resultingType.getErrorType() != -1){
            String param1 = values.get(first);
            String param2 = values.get(second);
```

Chapitre 2: Les erreurs sémantiques

```
if(resultingType.getResultType() == 0){
    if(param1 != null && param2 != null){
        int resultValue = getResultingValueOfInteger(param1,param2,"*");
        values.put(ctx,String.valueOf(resultValue));
    }
}else if(resultingType.getResultType() == 1){
    if(param1 != null && param2 != null){
        float resultValue = getResultingValueOfFloat(param1,param2,"*");
        values.put(ctx,String.valueOf(resultValue));
    }
}else{
    errors.add("326 : operation not allowed to string");
}
}else{
    values.put(ctx,"-1");
}
}else{
    errors.add("249: incompatible type");
    types.put(ctx, -1);
}
s.push(parent);
}
if(ctx.MINUS() != null){
    if(! imports.containsKey("Small_Java.lang")) {
        errors.add("nes/e6 missing import\n add import Small_java.lang to your package list");
        imports.put("Small_Java.lang", "Small_Java.lang");
    }
}
```

Chapitre 2: Les erreurs sémantiques

```
ParserRuleContext second = (ParserRuleContext) s.pop();
ParserRuleContext first = (ParserRuleContext) s.pop();
if(checkTypeCompatible(types.get(first),types.get(second))) {
    ErrorType resultingType = getResultingType(types.get(first),types.get(second));
    types.put(ctx, resultingType.getResultType());
    if(resultingType.getErrorType() != -1){
        String param1 = values.get(first);
        String param2 = values.get(second);
        if(resultingType.getResultType() == 0){
            if(param1 != null && param2 != null){
                int resultValue = getResultingValueOfInteger(param1,param2,"-");
                values.put(ctx,String.valueOf(resultValue));
            }
        }else if(resultingType.getResultType() == 1){
            if(param1 != null && param2 != null){
                float resultValue = getResultingValueOfFloat(param1,param2,"-");
                values.put(ctx,String.valueOf(resultValue));
            }
        }else{
            errors.add("360: operation not allowed to string");
        }
    }else{
        values.put(ctx,"-1");
    }
}else{
    errors.add("249: incompatible type");
    types.put(ctx, -1);
}
```


Chapitre 2: Les erreurs sémantiques

```
}  
s.push(ctx);  
}
```

```
}
```

La compatibilité de type se fait entre 2 opérandes, pour cela nous utilisons la pile `s` pour sauvegarder les opérandes, une fois un opérateur est rencontré, nous dépilerons deux opérandes.

Pour la récupération des valeurs, ça se fait avec la même logique sauf qu'on utilise une autre `HashMap` qui est `values`.

Test :

```
import Small_Java.io;  
  
import Small_Java.lang;  
  
protected class_SJ Test{  
    float_SJ c,d,f;  
    int_SJ a,e;  
    string_SJ b,m,g,l;  
    main_SJ{  
        a:= (b + 45);  
        e := 15;  
  
    }  
}
```

Résultat :

Dans cet exemple, nous essayons d'affecter à un `int_SJ` `String_SJ` dans nous devons avoir une erreur.

Chapitre 2: Les erreurs sémantiques

```
program compiled with the following errors
nes/e3: incompatible type in assignment
Hints:
you can't affect to a integer neither a string nor a float

Process finished with exit code 0
```

Illustration 5: Incompatibilité de type

L'importation du bon package :

Dans les expressions arithmétiques, nous devons vérifier que le développeur à bien importer le package `Small_Java.lang`. Et cela lorsque, le compilateur rencontre les opérations arithmétiques que cette vérification aura lieu. Cette partie de routine existe dans le code précédent.

Test :

```
import Small_Java.io;

protected class_SJ Test{

    float_SJ c,d,f;

    int_SJ a,e;

    string_SJ b,m,g,l;

    main_SJ{

        f := 15.23;

        c:= (f + 45);

    }

}
```

Résultat :

Chapitre 2: Les erreurs sémantiques

```
program compiled with the following errors
nes/e6 missing import
  add import Small_java.lang to your package list

Process finished with exit code 0
```

Illustration 6: importation de package pour les E/S et opérations arithmétiques

Le type String SJ :

Avec le type String_SJ, les opérations -, * et / ne sont pas permises et c'est une tâche déjà vérifiée dans le code précédent. Si la fonction getResultingType renvoie 2 donc si le compilateur rencontre un -, * ou / génère une erreur.

Test :

```
import Small_Java.io;

import Small_Java.lang;

protected class_SJ Test{
    string_SJ b,m,g,l;
    main_SJ{
        b := "hi"/"hello";

    }
}
```

Résultat :

```
program compiled with the following errors
292: operator not allowed to string

Process finished with exit code 0
```

Voilà ce qui concerne les expressions arithmétiques partie routine.

Chapitre 2: Les erreurs sémantiques

1.4. Les I/O :

La grammaire :

input: IN '(' '\" format '\" ',' identifier ');

format: INTFORMAT | FLOATFORMAT | STRINGFORMAT;

output: OUT '(' '\" format '\" ',' identifier')'

| OUT '(' '\" IDF format '\" ',' identifier)';

Pour les inputs nous avons traité les cas suivants :

- L'importation du bon package.
- La compatibilité entre le format et le type.
- Initialiser la variable en question.

L'importation du bon package :

Selon l'énoncé, lors d'une E/S le package Small_Java.io doit être importer, et pour cela, à la sortie de la règle input, nous allons vérifier si le package existe dans la HashMap imports. S'il n'existe pas, on l'ajoute au HashMap errors et au imports pour éviter les doubles erreurs.

Notons qu'il s'agit d'une erreur et non pas un warning, car ce package est important pour avoir un programme correct.

Test :

```
import Small_Java.lang;

protected class_SJ Test{

    string_SJ b,m,g,l;

    int_SJ a;

    main_SJ{

        b := "hi"+"hello";

        IN_SJ('%s',b);

        IN_SJ('%d',a);

    }

}
```

Chapitre 2: Les erreurs sémantiques

Résultat :

```
program compiled with the following errors
nes/e6 missing import
  add import Small_Java.io to your package list

Process finished with exit code 0
```

Illustration 7: importer le bon package pour les I/O

La compatibilité entre le format et le type :

C'est une tâche essentielle vu que nous avons trois types différents, chacun avec son format. Pour réaliser cette vérification, à la sortie de la règle input/output nous testons avec la condition suivante :

```
if(format.equals("%d") && type != 0 && type != -1)
    errors.add("nes/e7 incompatible format\nHints:\nyou are assigning %d to "+type);
if(format.equals("%f") && type != 1 && type != -1)
    errors.add("nes/e7 incompatible format\nHints:\nyou are assigning %f to "+type);
if(format.equals("%s") && type != 2 && type != -1)
    errors.add("nes/e7 incompatible format\nHints:\nyou are assigning %s to "+type);
```

Vous pouvez constater que nous avons ajouté `type != -1` car, comme déjà dit, le type erroné est accepter quel que soit le type à comparer avec.

Test :

```
import Small_Java.lang;
import Small_Java.io;
protected class_SJ Test{
    string_SJ b,m,g,l;
    int_SJ a;
    main_SJ{
        IN_SJ('%s',a);
        IN_SJ('%d',b);
```

```
}
```

```
}
```

Résultat :

```
program compiled with the following errors
nes/e7 incompatible format
Hints:
you are assigning %s to 0
nes/e7 incompatible format
Hints:
you are assigning %d to 2

Process finished with exit code 0
```

Illustration 8: incompatibilité de type dans les I/O

Initialiser la variable en question :

L'initialisation concerne uniquement les inputs, car nous désirons à un certain moment savoir s'il y a une division par zéro ou non, afin de déclencher cette erreur, nous devons calculer les valeurs des variables. Comme un input est une variable inconnue avant l'exécution, alors nous allons l'initialiser à 1 pour pouvoir déterminer les erreurs au maximum dans le programme donné à analyser.

1.5. Les conditions :

Dans les conditions, nous avons deux expressions en question, qui doivent être compatible et de type différent que le string. Pour faire, à la sortie de la règle arithmetic, nous allons appeler la fonction `checkTypeCompatibleForCondition` qui retourne une valeur booléenne.

La grammaire :

siinsts: SI '(' condition ')' ALORS '{' insts '}'

| SI '(' condition ')' ALORS '{' insts '}' sinon;

sinon: SINON '{' insts '}';

condition: logic | arithmetic;

logic: NOT logic

Chapitre 2: Les erreurs sémantiques

|logic AND logic

| logic OR logic

| '(' condition '');

arithmetic: arithmetic_expression GREATER arithmetic_expression

| arithmetic_expression LESS arithmetic_expression

| arithmetic_expression GREATEROREQUAL arithmetic_expression

| arithmetic_expression LESSOREQUAL arithmetic_expression

| arithmetic_expression EQUAL arithmetic_expression

| arithmetic_expression DIFFERENT arithmetic_expression;

La fonction :

```
private static boolean checkTypeCompatibleForCondition(int t1, int t2){  
    if(t1 == 0 && t2 == 0) return true;  
    if(t1 == 1 && t2 == 0) return true;  
    if(t1 == 0 && t2 == 1) return true;  
    if(t1 == 1 && t2 == 1) return true;  
    if(t1 == -1 && t2 == 0) return true;  
    if(t1 == -1 && t2 == 1) return true;  
    if(t1 == 0 && t2 == -1) return true;  
    if(t1 == 1 && t2 == -1) return true;  
  
    return false;  
}
```

Test :

```
import Small_Java.lang;  
import Small_Java.io;  
protected class_SJ Test{  
    string_SJ b,m,g,l;
```

Chapitre 2: Les erreurs sémantiques

```
int_SJ a,d,e,f;

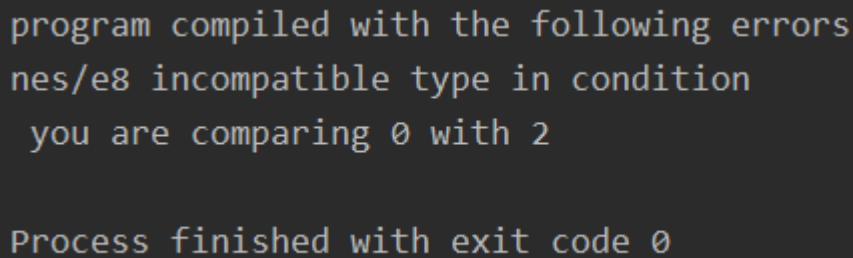
main_SJ{

    a := 5;
    d := 6;
    g := "hello";
    si( ( a + d ) > g)
    alors{
        IN_SJ('%s',b);
        IN_SJ('%d',a);
    }

}

}
```

Résultat :



```
program compiled with the following errors
nes/e8 incompatible type in condition
you are comparing 0 with 2

Process finished with exit code 0
```

Illustration 9: incompatibilité de type dans les conditions

Conclusion:

Nous arrivons à la fin du chapitre : En résumé, voilà l'ensemble des routines traitées :

- Double importation d'un package.
- Variable non initialisée.
- Variable non déclarée.
- Le nom d'un identifiant très long.

Chapitre 2: Les erreurs sémantiques

- Incompatibilité de type dans l'affectation.
- Incompatibilité de type entre les opérateurs.
- Package non importé.
- Devisions par zéro.
- Les opérations non permises dans les chaines de caractères.
- Incompatibilité de type dans les conditions.
- Récupérations des valeurs.

A la sortie de routine, nous aurons un programme correcte lexicalement, syntaxiquement et sémantiquement. Une table de symbole remplis. Voilà tout est prêt pour le prochain chapitre: les quadruplets.

Chapitre 3: Les quadruplées;

Introduction:

L'étape de construction des quadruplets ne peut être effectuée qu'après avoir un programme correct sémantiquement. Car des erreurs sémantiques signifient fin de la compilation avec échecs. Dans le cas contraire, on commence à construire nos quadruplets.

Le but principale de cette phase est de transformer le code fournis par le programmeur en un autre code plus proche du langage machine. Ce qu'on appelle les quadruplets.

1. Structure des données utilisées :

Afin de pouvoir réaliser cette phase, nous avons eu besoin d'une :

- LinkedList quad : qui a pour but de sauvegarder l'ensemble des quadruplets fournis par notre compilateur.
- stack : une pile pour sauvegarder les opérandes et les temporaires des expressions arithmétiques.
- stackCondition : une pile pour sauvegarder les sous conditions des expressions booléennes.

2. Les quadruplets des déclarations :

Dans le code objet, il va exister une partie déclaration. Afin de pouvoir générer cette partie, il faudra garder trace dans les quadruplets. Le résultat fournit par la fonction exitDecVar nous permettra de détecter toute les variables déclarées.

Le format général du quadruplet :

(« DEC », nom de la variable, le type de la variable déclaré, « ») ;

La fonction exitDecVar :

```
@Override public void exitDecvar(grammarOfProjectParser.DecvarContext ctx){  
    grammarOfProjectParser.VariablelistContext variablelist = ctx.variablelist();  
    /** loop variablelist
```

Chapitre 3: Les quadruplées;

** for each variable, i generate DEC*/*

```
for( ; ; variablelist = variablelist.variablelist()){  
    String nameOfVariable = variablelist.getChild(0).getText();  
  
    quad.addQuad("DEC",nameOfVariable,this.routines.getTs().getToken(nameOfVariable).getValu  
e(),"");  
  
    if(variablelist.variablelist() == null) return;  
}  
}
```

3. Les quadruplets des affectations :

Une affectation est sous la forme `var := var | val; .` La partie de la grammaire qui nous permet de construire le quadruplet des affectations est :

assignment: identifi er ':=' arithmetic_expression ;

A la sortie de la fonction pr  d  finie, nous g  n  rons le quadruplet qui aura la forme suivante :

(« := », temporaire de l'expressions arithmetique, « », « »,identifi er) ;

Le temporaire qui existe dans la forme g  n  rale est extrait    partir d'une pile. Cette pile est remplie lors de la rencontre des expressions arithm  tiques. Cette partie sera expliqu   prochainement.

La fonction exitAssignment :

```
@Override public void exitAssignment(grammarOfProjectParser.AssignmentContext ctx){  
    String tmp = stack.removeLast();  
    quad.addQuad(":= ",tmp,"",ctx.identifi er().getText());  
  
}
```

Chapitre 3: Les quadruplées;

4. Les quadruples des expressions arithmétiques :

Le principe est le même que celui des routines, sauf que au lieu de générer des erreurs, nous allons générer des quadruplets. A chaque fois qu'on rencontre une valeur ou une variable, on empile. Un fois une opération arithmétique est rencontré, on dépile deux fois et on génère le quadruplet adéquat. Et à la fin, on rempile le temporaire jusqu'à obtenir tous les quadruplets d'une expression. Le dernier temporaire sera utilisé dans l'affectation.

5. Les quadruplets des I/O :

Pour les quadruplets des entrées, on enregistre juste qu'il s'agit d'une lecture et le nom de la variable concerner.

Pour les quadruplets des sorties, on enregistre qu'il s'agit d'une écriture de la variable et sa valeur.

6. Les quadruplets des conditions :

Dans les conditions nous avons deux types : les conditions simples et les conditions composées. Pour les conditions simples, ça englobe les opérations arithmétiques logiques tel que $>$, $<$, $=$, ...etc. contrairement aux variables composées, elles englobent les opérations logiques tel que $\&$, $|$ et le not.

Dans le cas des opérations arithmétiques simples, il suffit juste de dépiler le temporaire à droite et le temporaire à gauche et générer le quadruplet des opérations en respectant les différents cas :

- BG : \leq
- BGE : $<$
- BL : \geq
- BLE : $>$
- BE : \neq
- BNE : $==$

Mais pour les opérateurs logiques, on aura une expressions logique simple à droite et une autre à gauche, dans le cas de :

- $\&$: Pour faire un saut vers le sinon, il faut que les deux conditions soient fausses. Donc il faudra vérifier la première condition, si elle est fausse c'est suffisant pour faire un saut vers le sinon, sinon nous devons vérifier également les deuxièmes conditions.

Chapitre 3: Les quadruplées;

- `| :` contrairement ou `&`, il suffit que la première condition soit vrai pour faire un saut vers le alors sinon il faudra vérifier la deuxième condition si elle est fausse il faudra faire un saut vers le sinon.

Afin de pouvoir faire le saut, il faudra sauvegarder les indices, et nous utilisons une pile pour cela.

Conclusion:

Avec les conditions, nous arrivons à la fin de ce chapitre. Il nous reste qu'à générer le code projet pour déclarer la fin de la compilation.

Chapitre 4: Le code objet

Introduction:

Le code assembleur est divisé en deux parties : La partie déclaration et la partie code. Dans ce qui suit nous allons les présenter.

1. Les déclarations :

Pour les déclarations, nous avons opté pour l'utilisation de ce jeu d'instruction :

- Si il s'agit du `int_SJ` ou `string_SJ` alors en assembleur on alloue la taille `DD`.
- Sinon on alloue la taille `DW`.

Test :

```
import Small_Java.lang;
import Small_Java.io;
protected class_SJ Test{
    string_SJ b,m,g,l;
    int_SJ a,d,e,f;
    main_SJ{

        a := 5;
        d := 6;

    }
}
```

Résultat :

Chapitre 4: Le code objet

```
b DD ?  
m DD ?  
g DD ?  
l DD ?  
a DD ?  
d DD ?  
e DD ?  
f DD ?
```

*Illustration 10: Les
déclarations en
assembleurs*

2. La partie code :

Les affectations :

Afin de réaliser le code objet de l'affectation, nous effectuons un load pour la valeur puis un store dans la variable.

Test :

```
import Small_Java.lang;  
import Small_Java.io;  
protected class_SJ Test{  
    string_SJ b,m,g,l;  
    int_SJ a,d,e,f;  
    main_SJ{  
  
        a := 5;  
        d := 6;  
        e := (5+a)/d;
```

Chapitre 4: Le code objet

}

}

Résultat :

```
CODE SEGMENT
LOAD 5
STORE a
LOAD 6
STORE d
LOAD 5
ADD a
DIV d
LOAD T1
STORE e
STORE T1
LOAD a
ADD d
STORE T2
LOAD 6
ADD 5
LOAD 5
STORE d
CODE ENDS
```

Illustration 11: Les affectations en assembleurs

Les expressions arithmétiques :

Pour les expressions arithmétiques, nous avons utilisé une fonctions déjà vu en cours, getInAcc. Le jeu d'instructions est le suivant :

- **LOAD M => (M)->Acc.**

Chapitre 4: Le code objet

- $\text{STORE } M \Rightarrow (\text{Acc}) \rightarrow M.$
- $\text{ADD } M \Rightarrow (\text{Acc}) + M \rightarrow \text{Acc}.$
- $\text{SUB } M \Rightarrow (\text{Acc}) - M \rightarrow \text{Acc}.$
- $\text{MULT } M \Rightarrow (\text{Acc}) * M \rightarrow \text{Acc}.$
- $\text{DIV } M \Rightarrow (\text{Acc}) / M \rightarrow \text{Acc}.$

Test:

```
import Small_Java.lang;
import Small_Java.io;
protected class_SJ Test{
    string_SJ b,m,g,l;
    int_SJ a,d,e,f;
    main_SJ{

        a := 5;
        d := 6;
        e := (5+a)/d;
    }
}
```

Résultat:

Chapitre 4: Le code objet

```
CODE SEGMENT
LOAD 5
STORE a
LOAD 6
STORE d
LOAD 5
ADD a
DIV d
LOAD T1
STORE e
STORE T1
LOAD a
ADD d
STORE T2
LOAD 6
ADD 5
LOAD 5
STORE d
CODE ENDS
```

Illustration 12: Les expressions arithmétiques en assembleurs

Les entrées sorties:

Le jeu d'instructions des entrées est IN port la variable désirée. Pour la sortie OUT variable port.

Les conditions :

Le jeu d'instruction :

- **CMP cond1 cond2:** elle est utilisé pour comparer entre deux conditions.

Les instructions de saut vers une étiquette:

Chapitre 4: Le code objet

- BGE etiq
- BG etiq
- BLE etiq
- BL etiq
- BNE etiq
- BE etiq

Les étiquettes peuvent être dans notre cas soit Alors, Sinon ou bien un numéro qui indique aller à condition.

Exemples d'exécutions

Ensemble des erreurs générer:

Test:

```
protected class_SJ Test{
    string_SJ b,m,g,g,l;
    int_SJ d,e,f,c;
    float_SJ elfhghjfhrrjk,w;
    main_SJ{
        a := 5 + 2 ;
        d := f + m;
        c := "25"*"12";
        e := 12.25;
        f := 0;

        si( ( a > "salam" ) | ( 31 >= 25 ) & ( 5 > 20 ) | ( 26 != 13 ) )
        alors{
            e := a * d + ( a + d ) / f;
            OUT_SJ('hello %f',a);

        }
        sinon{
            IN_SJ('%s',b);
        }
    }
}
```

Exemples d'exécutions

```
}  
}
```

Résultat:

nes/e1: Double declaration of variable of g
nes/e2: IDF too long elfhghjfhjrjk longer than 10
nes/e4 missing declaration a is used but not declared
nes/e6 missing import add import Small_java.lang to your package list
nes/e5: missing initialization f is not initialized
nes/e5: missing initialization m is not initialized
nes/e3: incompatible type in assignment d:=f+m
nes/e8: operator * not allowed to string in "25"*"12"
nes/e3: incompatible type in assignment c:="25"*"12"
nes/e3: incompatible type in assignment e:=12.25
nes/e8 incompatible type in condition you are comparing -1 with 2 in a>"salam"
nes/e8 division by zero in (a+d)/f
nes/e6 missing import add import Small_Java.io to your package list

Les conditions avec et:

Test:

```
import Small_Java.lang;  
import Small_Java.io;  
protected class _SJ Test{  
    string _SJ b,m,g,l;
```

Exemples d'exécutions

```
int_SJ a,d,e,f,c;

main_SJ{

    a := 5 + 2 ;

    d := 10;

    c := 5;

    e := 15;


    si( ( a > d ) & ( 31 >= 25 ) & ( 5 > 20 ) )
    alors{
        e := a * d / ( a + d );
        OUT_SJ('hello %d',a);

    }
    sinon{
        IN_SJ('%s',b);
    }
}
}
```

Résultat:

program compiled without errors!

errors: 0

warnings: 0

Exemples d'executions

***** SHOW TABLE OF SYMBOLE *****

b string_SJ declared 1

m string_SJ declared null

g string_SJ declared null

l string_SJ declared null

a int_SJ declared 7

d int_SJ declared 10

e int_SJ declared 0

f int_SJ declared null

c int_SJ declared 5

***** END SHOW TABLE OF SYMBOLE *****

*** quad ***

0-quad(DATA SEGMENT,,);

1-quad(DEC,b,string_SJ,);

2-quad(DEC,m,string_SJ,);

3-quad(DEC,g,string_SJ,);

4-quad(DEC,l,string_SJ,);

5-quad(DEC,a,int_SJ,);

6-quad(DEC,d,int_SJ,);

7-quad(DEC,e,int_SJ,);

8-quad(DEC,f,int_SJ,);

9-quad(DEC,c,int_SJ,);

10-quad(DATA ENDS,,);

11-quad(CODE SEGMENT,,);

12-quad(+,5,2,T0);

13-quad(:=,T0,,a);

Exemples d'exécutions

```
14-quad(:=,10,,d);
15-quad(:=,5,,c);
16-quad(:=,15,,e);
17-quad(BLE,27,a,d);
18-quad(BL,27,31,25);
19-quad(BLE,27,5,20);
20-quad(alors,,);
21-quad(+,a,d,T1);
22-quad(/,d,T1,T2);
23-quad(*,a,T2,T3);
24-quad(:=,T3,,e);
25-quad(WRITE,a,7,);
26-quad(BR,29,,);
27-quad(else,,);
28-quad(READ,b,,);
29-quad(END,,);
*** end of quads ***

**** Object code ****

DATA SEGMENT

b DD ?
m DD ?
g DD ?
l DD ?
a DD ?
d DD ?
e DD ?
```


Exemples d'exécutions

f DD ?

c DD ?

DATA ENDS

CODE SEGMENT

LOAD 5

ADD 2

STORE a

LOAD 10

STORE d

LOAD 5

STORE c

LOAD 15

STORE e

CMP a d

BLE else

CMP 31 25

BL else

CMP 5 20

BLE else

alors:

LOAD a

ADD d

STORE T1

LOAD d

DIV T1

MULT T2

Exemples d'exécutions

STORE e

OUT a port

else:

IN port b

CODE ENDS

**** End of Object code ****

Les conditions avec ou

```
import Small_Java.lang;
```

```
    import Small_Java.io;
```

```
    protected class_SJ Test{
```

```
        string_SJ b,m,g,l;
```

```
        int_SJ a,d,e,f,c;
```

```
        main_SJ{
```

```
            a := 5 + 2 ;
```

```
            d := 10;
```

```
            c := 5;
```

```
            e := 15;
```

```
            si( ( a > d ) | ( 31 >= 25 ) | ( 5 > 20 ) )
```

```
            alors{
```

```
                e := a * d / ( a + d );
```

```
                OUT_SJ('hello %d',a);
```

```
            }
```

Exemples d'exécutions

```
sinon{  
    IN_SJ('%s',b);  
}  
  
}  
  
}
```

Resultat:

program compiled without errors!

errors: 0

warnings: 0

***** SHOW TABLE OF SYMBOLE *****

b string_SJ declared 1

m string_SJ declared null

g string_SJ declared null

l string_SJ declared null

a int_SJ declared 7

d int_SJ declared 10

e int_SJ declared 0

f int_SJ declared null

c int_SJ declared 5

***** END SHOW TABLE OF SYMBOLE *****

*** quad ***

0-quad(DATA SEGMENT,,);

Exemples d'exécutions

```
1-quad(DEC,b,string_SJ,);
2-quad(DEC,m,string_SJ,);
3-quad(DEC,g,string_SJ,);
4-quad(DEC,l,string_SJ,);
5-quad(DEC,a,int_SJ,);
6-quad(DEC,d,int_SJ,);
7-quad(DEC,e,int_SJ,);
8-quad(DEC,f,int_SJ,);
9-quad(DEC,c,int_SJ,);
10-quad(DATA ENDS,,);
11-quad(CODE SEGMENT,,);
12-quad(+,5,2,T0);
13-quad(:=,T0,,a);
14-quad(:=,10,,d);
15-quad(:=,5,,c);
16-quad(:=,15,,e);
17-quad(BG,20,a,d);
18-quad(BGE,20,31,25);
19-quad(BLE,27,5,20);
20-quad(alors,,);
21-quad(+,a,d,T1);
22-quad(/,d,T1,T2);
23-quad(*,a,T2,T3);
24-quad(:=,T3,,e);
25-quad(WRITE,a,7,);
26-quad(BR,29,,);
```

Exemples d'exécutions

27-quad(else,,);

28-quad(READ,b,,);

29-quad(END,,);

*** end of quads ***

**** Object code ****

DATA SEGMENT

b DD ?

m DD ?

g DD ?

l DD ?

a DD ?

d DD ?

e DD ?

f DD ?

c DD ?

DATA ENDS

CODE SEGMENT

LOAD 5

ADD 2

STORE a

LOAD 10

STORE d

LOAD 5

STORE c

LOAD 15

STORE e

Exemples d'exécutions

CMP a d

BG alors

CMP 31 25

BGE alors

CMP 5 20

BLE else

alors:

LOAD a

ADD d

STORE T1

LOAD d

DIV T1

MULT T2

STORE e

OUT a port

else:

IN port b

CODE ENDS

**** End of Object code ****

Les conditions simples:

Test:

```
import Small_Java.lang;
```

```
import Small_Java.io;
```

```
protected class_SJ Test{
```

```
    string_SJ b,m,g,l;
```

Exemples d'exécutions

```
int_SJ a,d,e,f,c;
```

```
main_SJ{
```

```
    a := 5 + 2 ;
```

```
    d := 10;
```

```
    c := 5;
```

```
    e := 15;
```

```
    si( ( a > ( d + e ) ) )
```

```
    alors{
```

```
        e := a * d / ( a + d );
```

```
        OUT_SJ('hello %d',a);
```

```
    }
```

```
    sinon{
```

```
        IN_SJ('%s',b);
```

```
    }
```

```
}
```

```
}
```

Résultat:

program compiled without errors!

errors: 0

warnings: 0

***** SHOW TABLE OF SYMBOLE *****

Exemples d'exécutions

b string_SJ declared 1

m string_SJ declared null

g string_SJ declared null

l string_SJ declared null

a int_SJ declared 7

d int_SJ declared 10

e int_SJ declared 0

f int_SJ declared null

c int_SJ declared 5

***** END SHOW TABLE OF SYMBOLE *****

*** quad ***

0-quad(DATA SEGMENT,,);

1-quad(DEC,b,string_SJ,);

2-quad(DEC,m,string_SJ,);

3-quad(DEC,g,string_SJ,);

4-quad(DEC,l,string_SJ,);

5-quad(DEC,a,int_SJ,);

6-quad(DEC,d,int_SJ,);

7-quad(DEC,e,int_SJ,);

8-quad(DEC,f,int_SJ,);

9-quad(DEC,c,int_SJ,);

10-quad(DATA ENDS,,);

11-quad(CODE SEGMENT,,);

12-quad(+,5,2,T0);

13-quad(:=,T0,,a);

14-quad(:=,10,,d);

Exemples d'exécutions

15-quad(:=,5,,c);

16-quad(:=,15,,e);

17-quad(+,d,e,T1);

18-quad(BLE,26,a,T1);

19-quad(alors,,,);

20-quad(+,a,d,T2);

21-quad(/,d,T2,T3);

22-quad(*,a,T3,T4);

23-quad(:=,T4,,e);

24-quad(WRITE,a,7,);

25-quad(BR,28,,);

26-quad(else,,,);

27-quad(READ,b,,);

28-quad(END,,,);

*** end of quads ***

***** Object code *****

DATA SEGMENT

b DD ?

m DD ?

g DD ?

l DD ?

a DD ?

d DD ?

e DD ?

f DD ?

c DD ?

Exemples d'exécutions

DATA ENDS

CODE SEGMENT

LOAD 5

ADD 2

STORE a

LOAD 10

STORE d

LOAD 5

STORE c

LOAD 15

STORE e

LOAD d

ADD e

CMP a T1

BLE else

alors:

STORE T1

LOAD a

ADD d

STORE T2

LOAD d

DIV T2

MULT T3

STORE e

OUT a port

else:

Exemples d'executions

IN port b

CODE ENDS

**** End of Object code ****