



Rapport du projet :
Réalisation d'un compilateur

❖ **Groupe :**

- **Interface graphique :**

Bounar Hibatallah
Zarrouq Soukaina

- **Compilateur-analyse :**

El Hasnaoui Soukayna
Banah Fathiya

- **Compilateur-génération**

Maman Souley Aicha
Sangare Mahamadou

- **Interpréteur**

Boukili Sabah
Ait Malek Yassine

Encadré par : Mr. Kabbaj Adil

Introduction :

Un compilateur est un programme qui traite les instructions écrites dans un langage de programmation donné pour les traduire en langage machine, ou « code », utilisé par le processeur d'un ordinateur. Il a pour rôle de rechercher toutes les erreurs possibles dans un programme source, telles que des fautes d'orthographe, les variables, les types, etc .Le compilateur commence par une analyse lexicale où elle divise le code source en petits morceaux : les tokens (les jetons) et chaque token a une unité lexicale unique de langue ,puis il effectue l'analyse syntaxique de la séquence des tokens donc de toutes les instructions dans le langage de programmation, les unes après les autres pour reconnaître la structure syntaxique de programme ensuite elle attaque l'analyse sémantique qui gère les erreurs du types ou tâches définies telles que les variables locales .Après il assure une modification du code source en code intermédiaire puis une optimisation du code intermédiaire, en rendant le programme plus performant selon son usage enfin une allocation de machine à pile avec la génération de codes et la traduction du code intermédiaire en code cible.

Notre travail est de réaliser un compilateur qui sera basé sur la majorité de ses étapes afin de bien assimiler son fonctionnement.

1. Compilateur-analyse(El Hasnaoui Soukayna – Banah Fathiya)

Le niveau lexical de notre langage est constitué par les catégories lexicales suivantes :

1. Catégories Lexicales :

Elements	Catégories
+	plus
++	incr
-	moins
--	decr
=	affect
==	egal
<	inf
<=	infEgal
<>	diff
>	sup
>=	supEgal
and	et
or	ou
nom_de_variable	ident
{	accoladeOuv
}	accoladeFerm
;	finLigne
[crochetOuv
]	crochetFerm
/	div

*	prod
/*	debCommnt
*/	finCommnt

Mots réservés :

if
else
while
constante
int
float
string
boolean
true
false

Les catégories lexicales sont définies par cette grammaire régulière :

2. Grammaire régulière

:Entier

$T = \{ '+', '-', \text{chiffre} \}$ $N = \{ \text{entier}, N, N1 \}$ $P = \{$
entier $\rightarrow '+' N$;
entier $\rightarrow '-' N$;
entier $\rightarrow \text{chiffre} N1$
; $N \rightarrow \text{chiffre} N1$;
 $N1 \rightarrow \text{chiffre} N1$
; $N1 \rightarrow \emptyset$;
}
 $Z = \text{entier}$

Reel

$T = \{ '+', '-', \text{chiffre} \}$
 $N = \{ \text{Reel}, N, N1, N2, N3 \}$ $P = \{$
Reel $\rightarrow '+' N$;
Reel $\rightarrow '-' N$;
Reel $\rightarrow \text{chiffre} N1$
; $N \rightarrow \text{chiffre} N1$
; $N1 \rightarrow \text{chiffre} N1$

; N1 -> '.' N2 ;
 N2 -> chiffre
 N3 ; N3 ->
 chiffre N3 ; N3 -
 > \emptyset ;

}
 Z = Reel

Ident

T = { '_', chiffre,
 lettre } N = { ident, A
 , B, C } ; ident ->
 lettre A ;
 A -> lettre A ;
 A -> chiffre
 B ; A -> '_' B
 ;
 A -> \emptyset ;
 B - > chiffre
 B ; B - > '_' B
 ;
 B -> lettre C ;
 C -> chiffre
 B ; C -> '_' B
 ;
 C -> lettre C
 ; C -> \emptyset ;
 }
 Z = ident

Chaine

T = { '""', * }

Remarque : * veut dire n'importe quel élément : lettres, chiffres etc...

N = { A, B }
 Chaine -> '""'
 A ; A -> * B ;
 B -> '""' ;
 }
 Z = chaine

En notation EBNF

entier = [+|-] chiffre {chiffre}

lettre = 'A'|'B'|'C'|'D'|'E'|'F'|'G'|'H'|'I'|'J'|'K'|'L'|'M'|'N'|'O'|
'P'|'Q'|'R'|'S'|'T'|'U'|'V'|'W'|'X'|'Y'|'Z'

|'a'|'b'|'c'|'d'|'e'|'f'|'g'|'h'|'i'|'j'|'k'|'l'|'m'|'n'|'o'|'p'|'q'|'r'|'s'|'t'|'u'|'v'|'w'|'x'|'y'|'z'

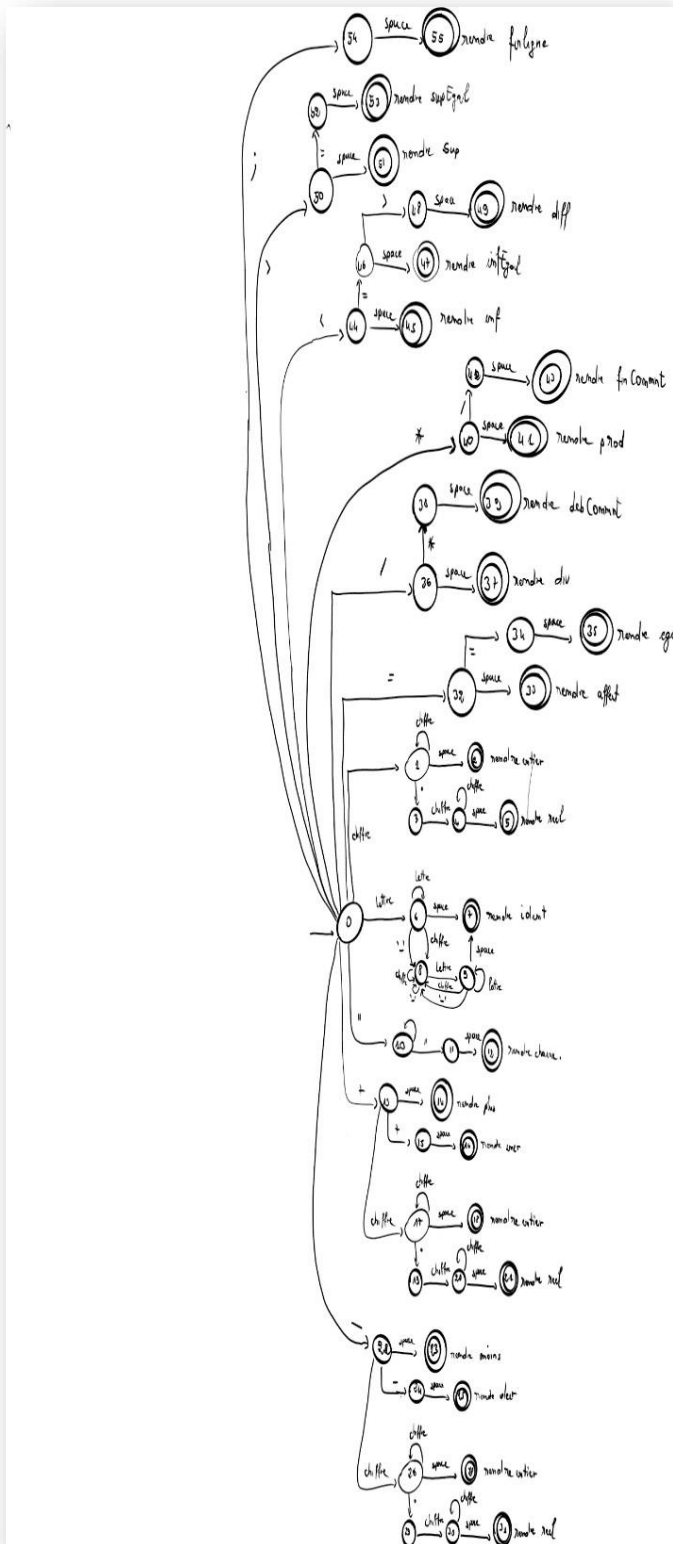
chiffre = 1|2|3|4|5|6|7|8|9

reel = [+|-]chiffre{chiffre} '.' chiffre {chiffre}

ident = lettre { {chiffre| '_' } lettre }

chaine = ''{*}''

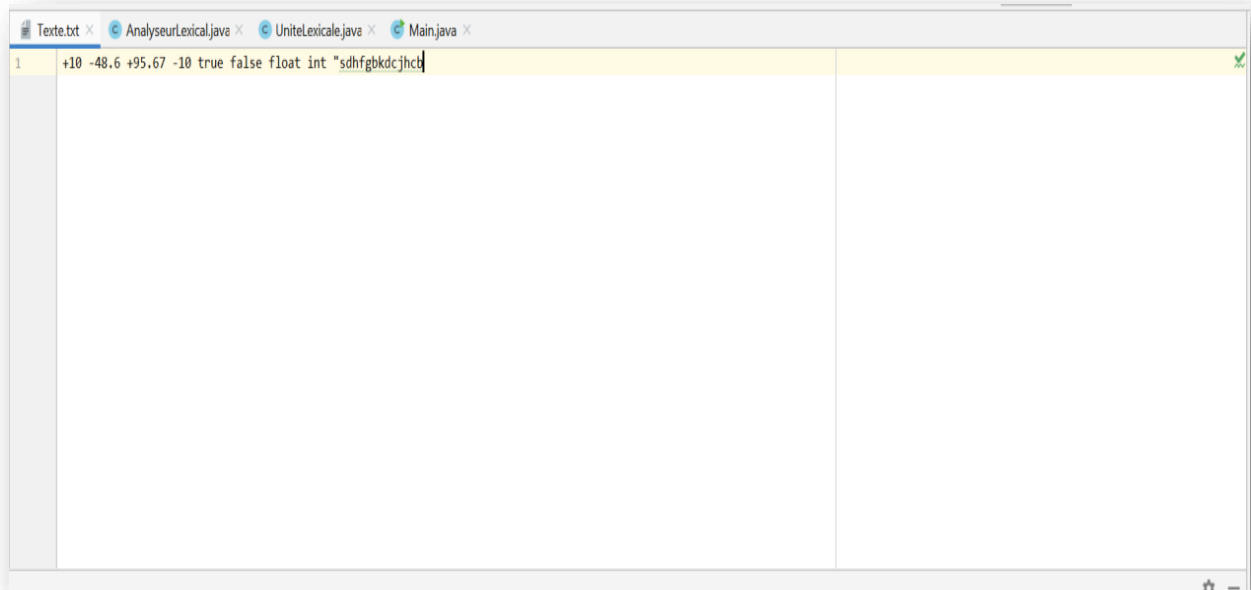
3. Automate globale pour les catégories Lexicales



4. Exemple d'exécution de l'analyseur lexical

Pour le test suivant :

+10 -48.6 +95.67 -10 true false float int "sdhfgbkdcjhcb

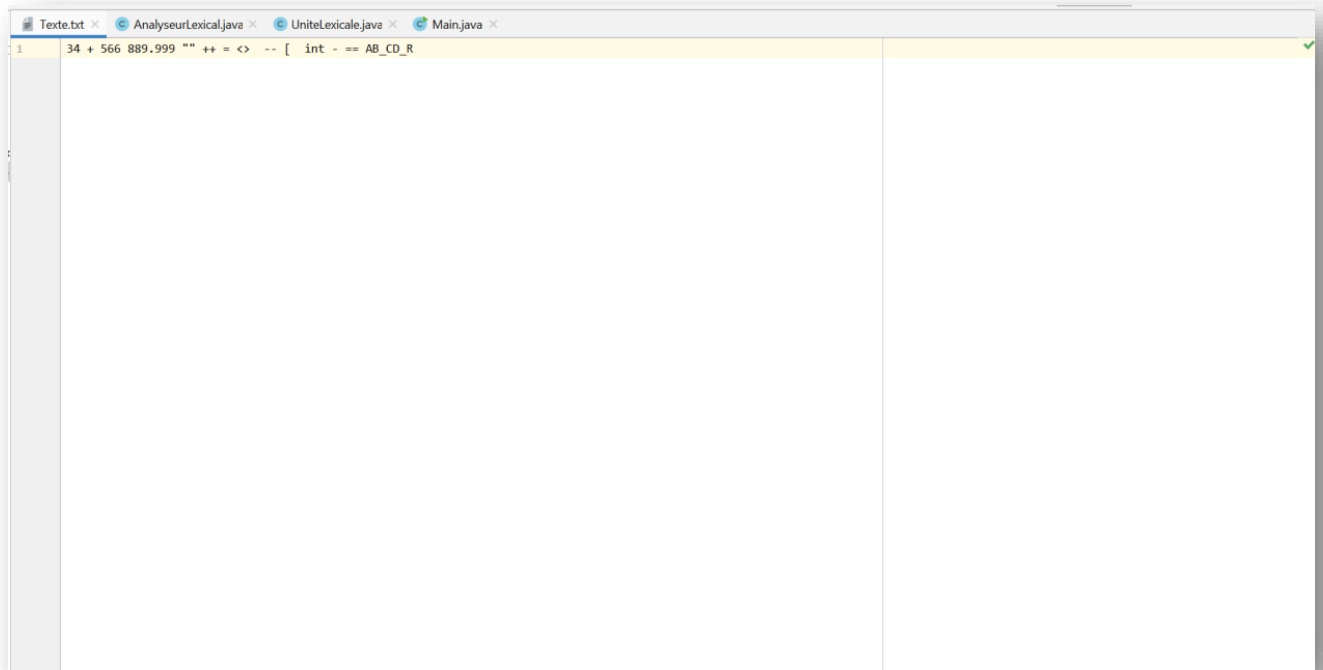


Voici le résultat :



Pour le test suivant :

34 + 566 889.999 "" ++ = <> -- [int - == AB_CD_R



Voici le résultat :



Grammaire Hors Contexte

Le niveau syntaxique de notre langage est constitué par ses unités syntaxiques :

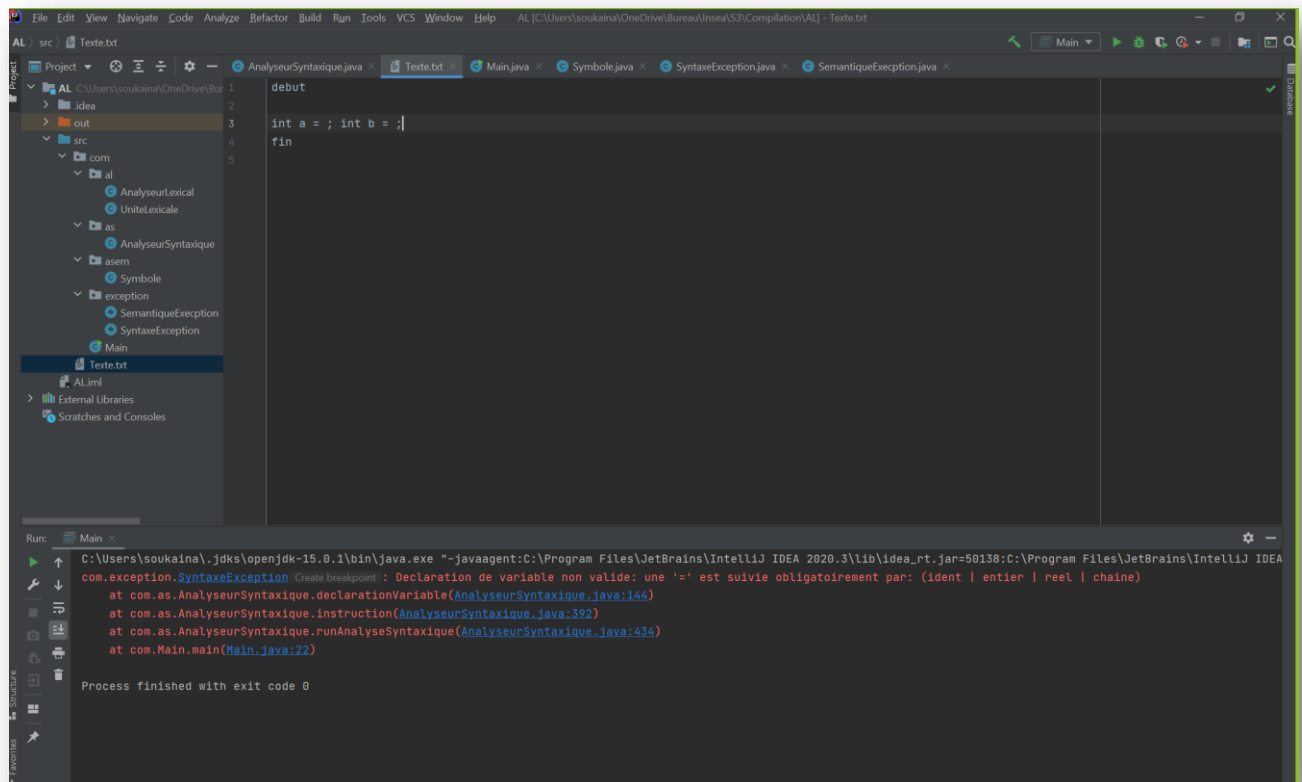
Terme

terme = (ident | entier | reel | chaine)

Declaration de variable

declarationVariable = type ident ['=' terme] { , ident ['=' terme] } ';' ;

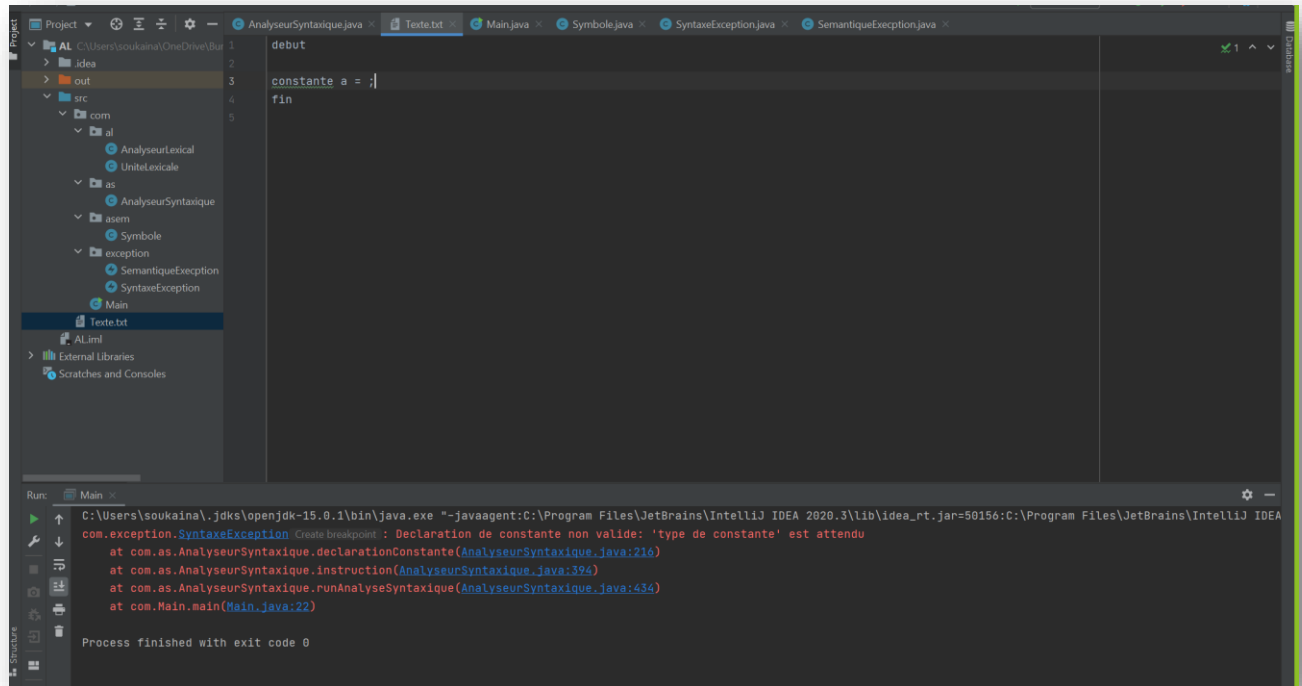
Voici un exemple qui ne vérifie pas cette grammaire :



Declaration de constante

declarationConstante = 'constante' type ident '=' terme ';' ;

Voici un exemple qui ne vérifie pas cette grammaire :



Opérateur

opérateur = ('+' | '*' | '/')

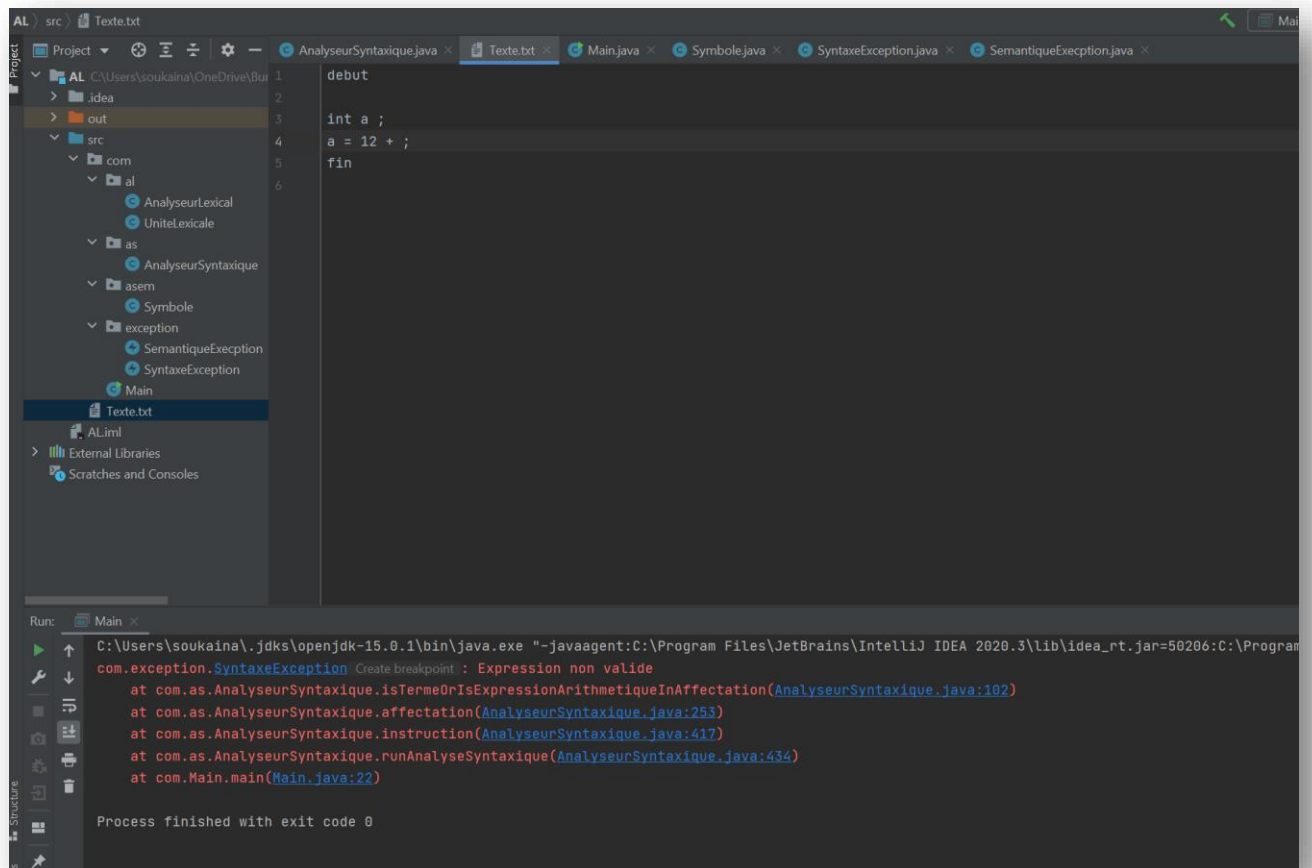
Opérande

opérande = (entier | reel | ident)

Expression Arithmétique

expressionArithmetique = opérande opérateur opérande {opérateur opérande} ';' ;

Voici un exemple qui ne vérifie pas cette grammaire :



Opérateur Conditionnel

opérateurConditionnel = (< | > | <= | >= | <> | ==)

Condition

condition = (expressionArithmetique | terme) opérateurConditionnel

(expressionArithmetique | terme) { (or | and | diff)

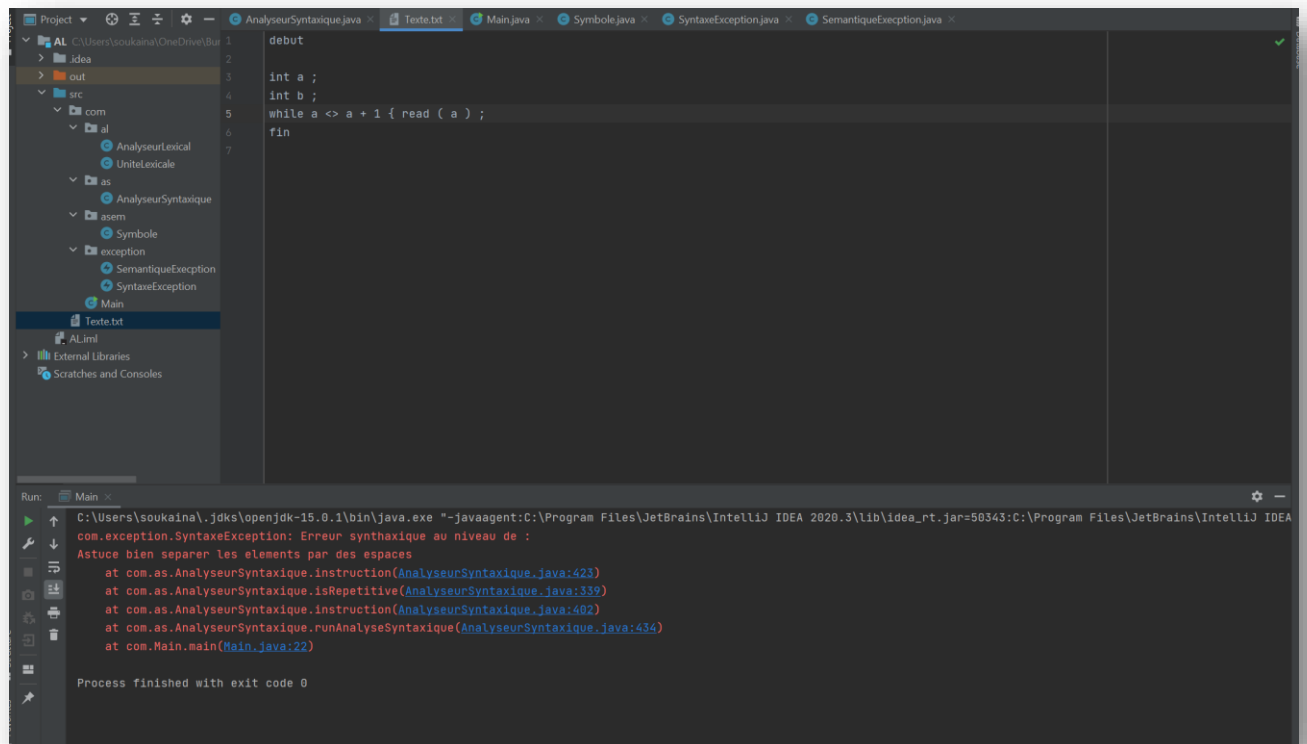
expressionArithmetique | terme) opérateurConditionnel

(expressionArithmetique | terme) }

Répétitive

repetitive = 'while' condition '{' instructions '}'

Voici un exemple qui ne vérifie pas cette grammaire :



```
1  debut
2
3  int a ;
4  int b ;
5  while a <> a + 1 { read ( a ) ;
6
7  fin
```

Run: Main ×

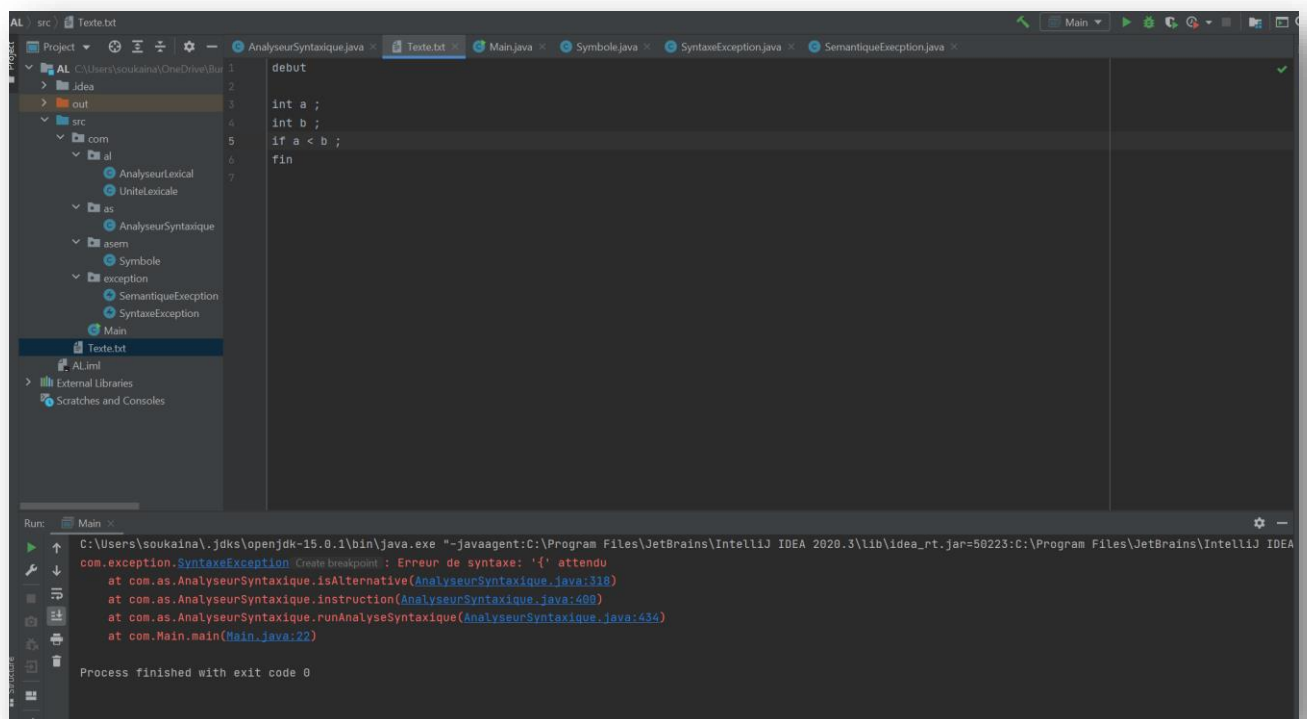
C:\Users\soukaina\.jdk\openjdk-15.0.1\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2020.3\lib\idea_rt.jar=50343:C:\Program Files\JetBrains\IntelliJ IDEA" com.exception.SyntaxException: Erreur synthaxique au niveau de :
Astuce bien separer les elements par des espaces
at com.as.AnalyseurSyntaxique.instruction(AnalyseurSyntaxique.java:423)
at com.as.AnalyseurSyntaxique.isRepetitive(AnalyseurSyntaxique.java:339)
at com.as.AnalyseurSyntaxique.instruction(AnalyseurSyntaxique.java:402)
at com.as.AnalyseurSyntaxique.runAnalyseSyntaxique(AnalyseurSyntaxique.java:434)
at com.Main.main(Main.java:22)

Process finished with exit code 0

Alternative

alternative = 'if' condition '{' instructions '}' ['else' condition '{' instructions '}']

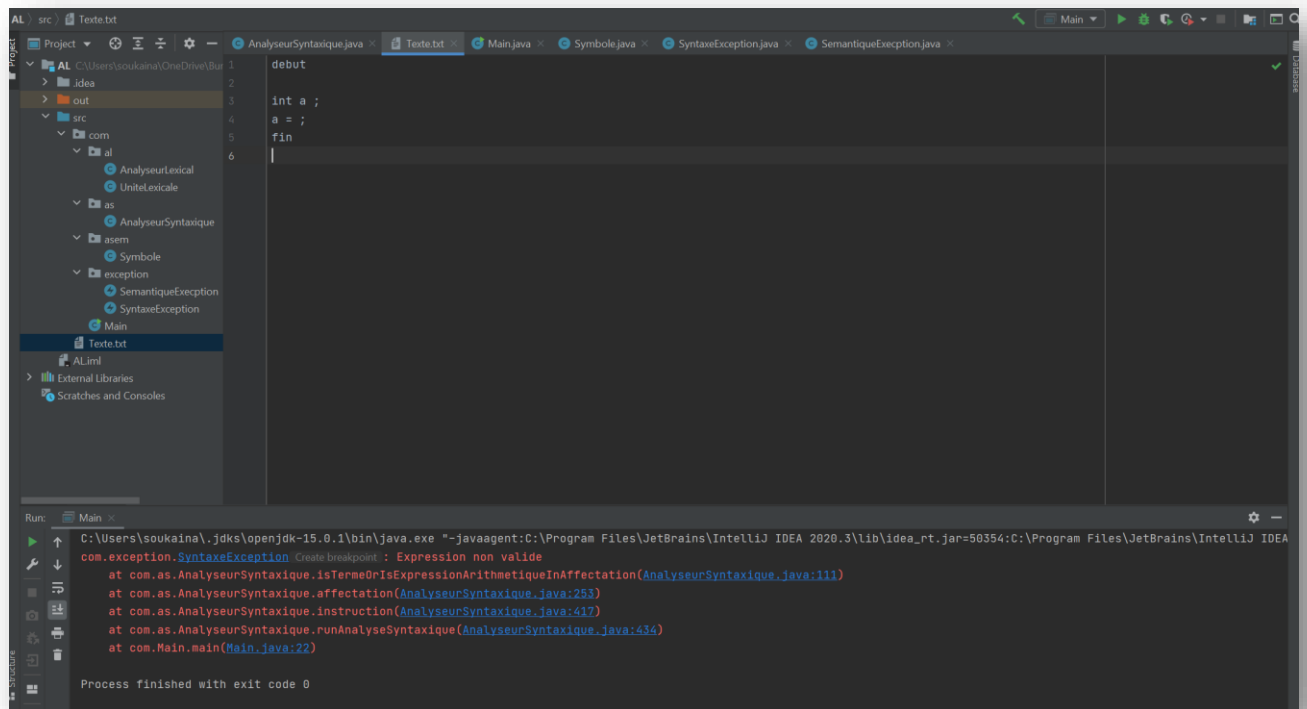
Voici un exemple qui ne vérifie pas cette grammaire :



Affectation

affectation = ident Affect (expressionArithemetique | terme) ';' ;

Voici un exemple qui ne vérifie pas cette grammaire :



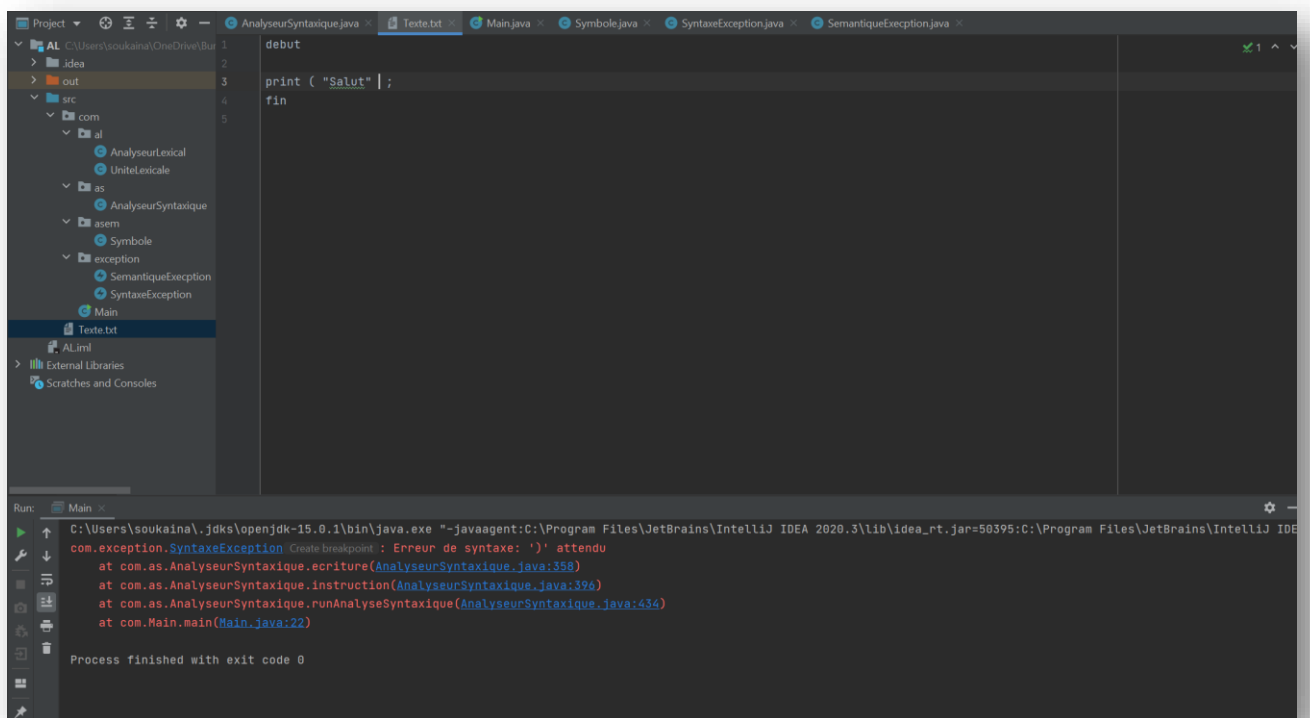
Instruction

instruction = affectation | lecture | ecriture | alternative | repetitive |
expressionArithmetique | declarationVariable | declarationConstante

Ecriture

ecriture = 'print' '(' (operande | chaine | expressionArithmetique) ')' ';' ;

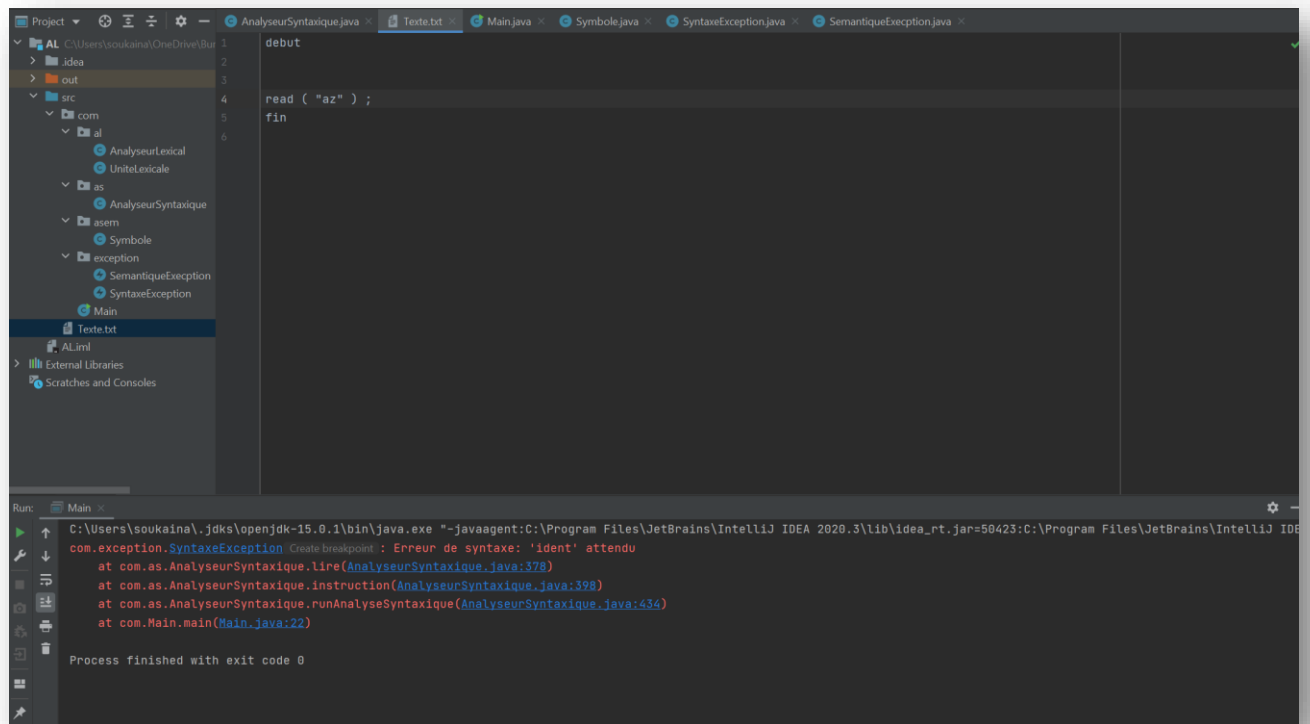
Voici un exemple qui ne vérifie pas cette grammaire :



Lecture

lecture = 'read' '(' ident ')' ';' ;

Voici un exemple qui ne vérifie pas cette grammaire :



Commentaire

commentaire = '/*' chaine '*/'

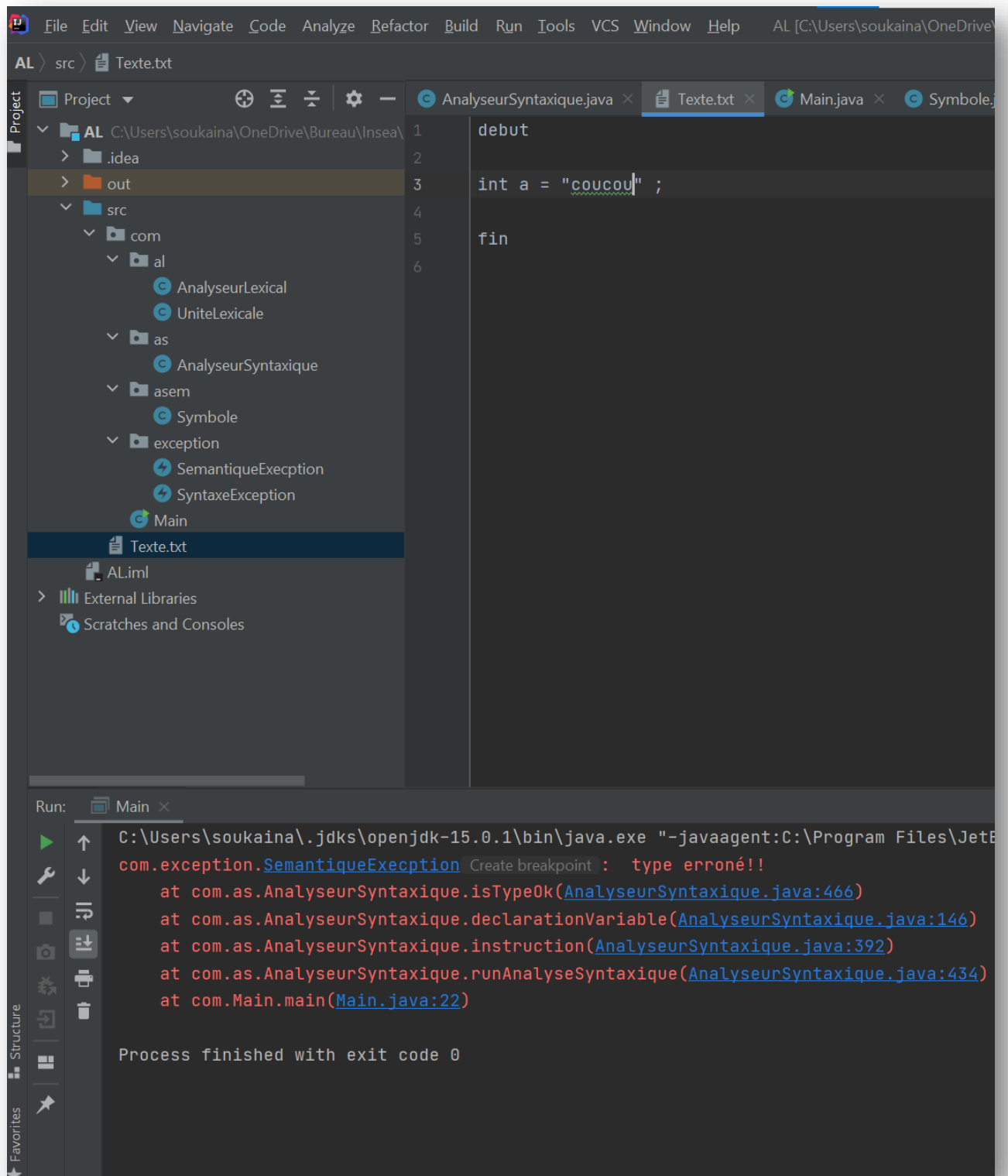
Programme

Program = 'debut' instruction 'fin';

L'analyseur sémantique vérifie ces contraintes :

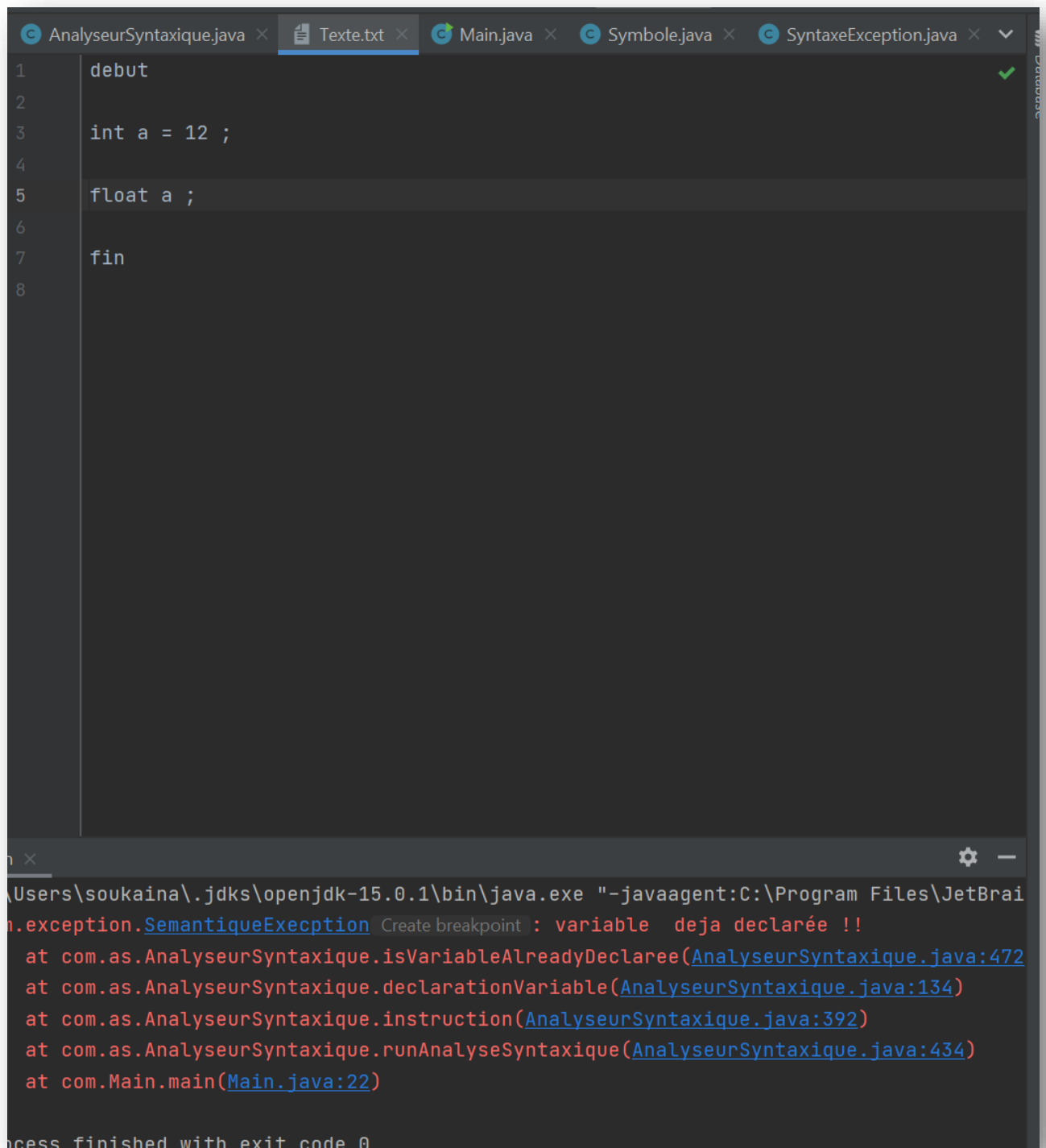
1) On affecte à un chaque type de variable/constante une valeur qui correspond à ce type de variable/constante

Voici un exemple d'affectation qui ne vérifie pas cette contrainte :



2) Une variable ne peut pas être déclarer plusieurs fois

Voici un exemple qui ne vérifie pas cette contrainte :

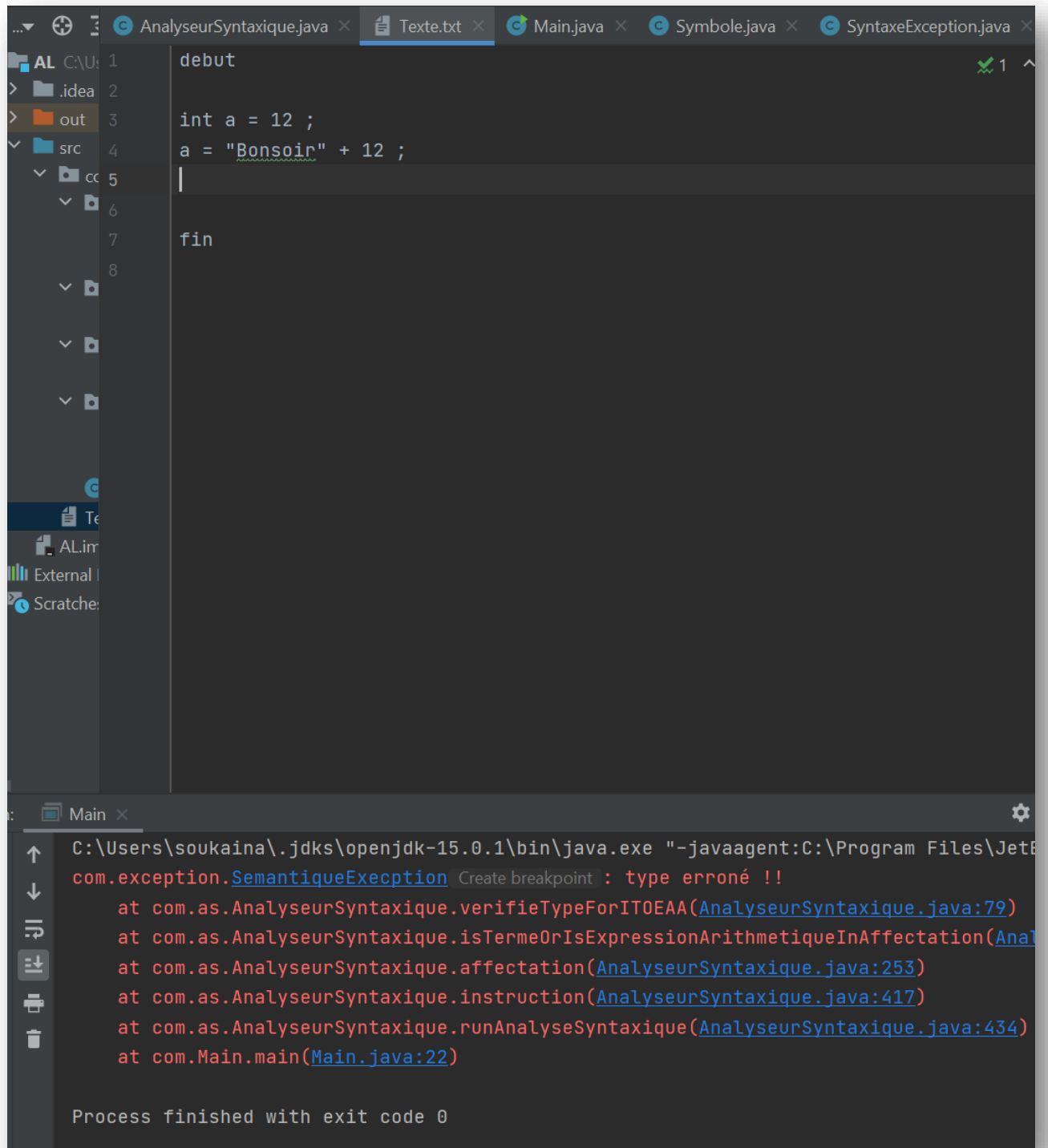


```
1  debut
2
3  int a = 12 ;
4
5  float a ;
6
7  fin
8
```

```
Users\soukaina\.jdk\openjdk-15.0.1\bin\java.exe "-javaagent:C:\Program Files\JetBrai
n.exception.SemantiqueException Create breakpoint : variable deja declarée !!
at com.as.AnalyseurSyntaxique.isVariableAlreadyDeclaree(AnalyseurSyntaxique.java:472)
at com.as.AnalyseurSyntaxique.declarationVariable(AnalyseurSyntaxique.java:134)
at com.as.AnalyseurSyntaxique.instruction(AnalyseurSyntaxique.java:392)
at com.as.AnalyseurSyntaxique.runAnalyseSyntaxique(AnalyseurSyntaxique.java:434)
at com.Main.main(Main.java:22)
Process finished with exit code 0
```

3) Dans une expression arithmétique les opérandes doivent être (entier | réel | ident) :
On utilise le bon type de donnée à la bonne place.

Voici un exemple qui ne vérifie pas cette contrainte :



The screenshot shows an IDE with several tabs: 'AnalyseurSyntaxique.java', 'Texte.txt', 'Main.java', 'Symbole.java', and 'SyntaxeException.java'. The 'Texte.txt' tab is active, displaying the following code:

```
1 debut
2
3 int a = 12 ;
4 a = "Bonsoir" + 12 ;
5
6
7 fin
8
```

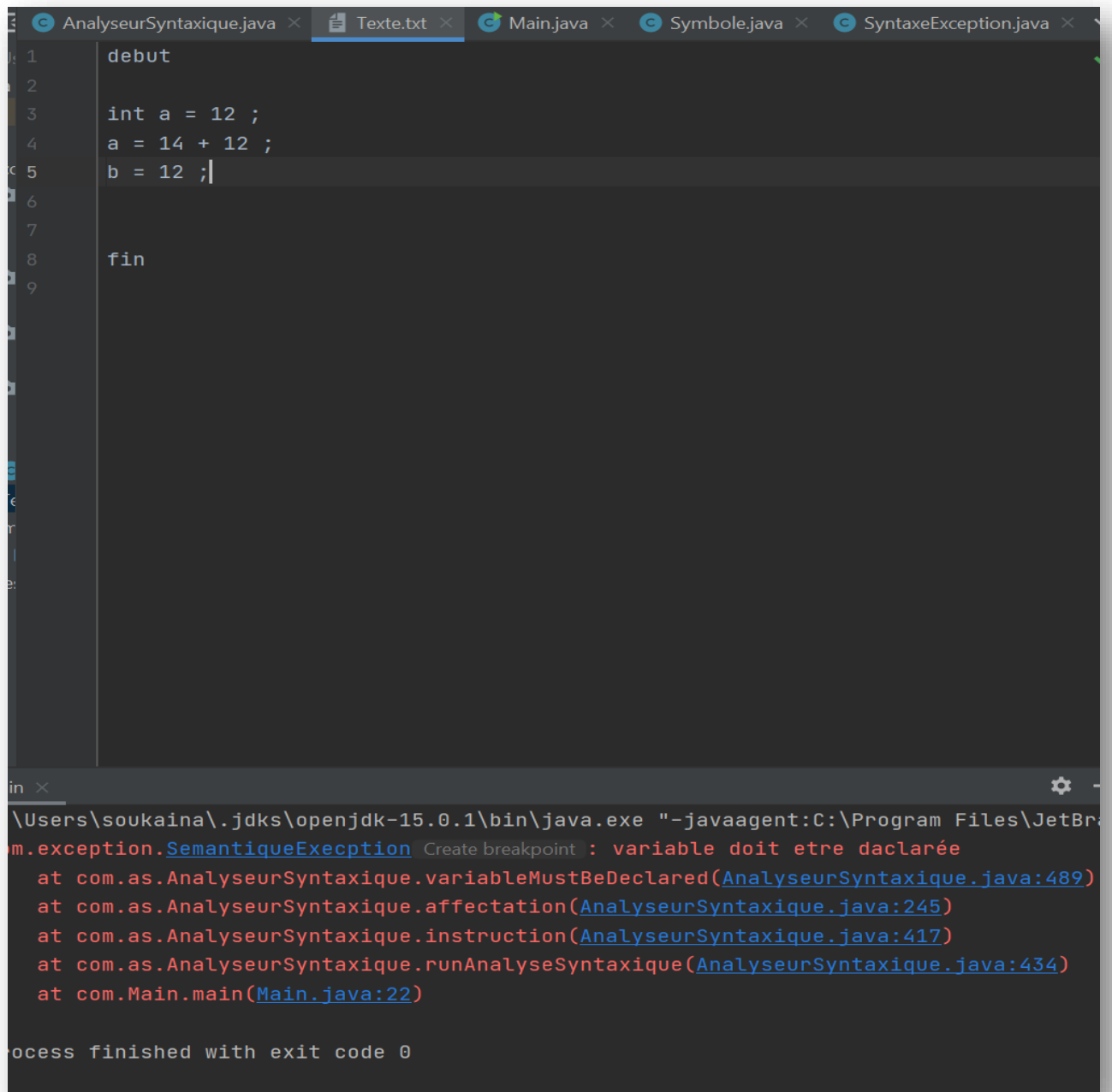
The console window at the bottom shows the execution of 'Main.java' and a stack trace for a `SemantiqueException`:

```
C:\Users\soukaina\.jdk\openjdk-15.0.1\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA\lib\idea_rt.jar=12139:C:\Program Files\JetBrains\IntelliJ IDEA\bin" com.exception.SemantiqueException
com.exception.SemantiqueException: Create breakpoint : type erroné !!
    at com.as.AnalyseurSyntaxique.verifieTypeForITOEAA(AnalyseurSyntaxique.java:79)
    at com.as.AnalyseurSyntaxique.isTermeOrIsExpressionArithmetiqueInAffectation(AnalyseurSyntaxique.java:253)
    at com.as.AnalyseurSyntaxique.affectation(AnalyseurSyntaxique.java:253)
    at com.as.AnalyseurSyntaxique.instruction(AnalyseurSyntaxique.java:417)
    at com.as.AnalyseurSyntaxique.runAnalyseSyntaxique(AnalyseurSyntaxique.java:434)
    at com.Main.main(Main.java:22)

Process finished with exit code 0
```

4) Il faut déclarer une variable avant de l'utiliser

Voici un exemple qui ne vérifie pas cette contrainte :



```
1  debut
2
3  int a = 12 ;
4  a = 14 + 12 ;
5  b = 12 ;|
6
7
8  fin
9
```

Process finished with exit code 0

Exception in thread "main" java.lang.SemantiqueException: variable doit etre déclarée
at com.as.AnalyseurSyntaxique.variableMustBeDeclared(AnalyseurSyntaxique.java:489)
at com.as.AnalyseurSyntaxique.affectation(AnalyseurSyntaxique.java:245)
at com.as.AnalyseurSyntaxique.instruction(AnalyseurSyntaxique.java:417)
at com.as.AnalyseurSyntaxique.runAnalyseSyntaxique(AnalyseurSyntaxique.java:434)
at com.Main.main(Main.java:22)

2. Compilateur-génération (Maman Souley Aicha – Sangare Mahamadou)

Idée générale :

Dans cette partie nous allons essayer de générer Le code cible de notre compilateur. L'idée est la suivante nous allons essayer de parcourir l'analyseur syntaxique et sémantique tout en générant le code intermédiaire. Il faut noter que toute notre génération de code va se faire en respectant les contraintes de notre langage.

➤ **Instruction cible : ic**

Une instruction cible est un élément composé d'un opérateur et d'un opérande. Pour le créer nous allons créer une classe qui aura 2 attributs l'opérateur et l'opérande

Il sera le type qu'on va utiliser pour déclarer le vecteur d'instruction cible

➤ **Vecteur d'instruction cible : vic**

C'est une table composée de plusieurs éléments qui sont le type instruction cible

➤ **Table des variables**

Cette table contient les identifiants des différentes variables de notre langage ainsi que leur valeur leur type leur adresse en mémoire et leur nature. Pour la nature, il s'agit de dire si c'est une constante ou non. À chaque fois qu'une variable est rencontrée dans une instruction nous allons essayer de consulter notre table de variable qui est préalablement elle remplit lors de la déclaration des variables. Donc avant de faire une opération pour une variable nous allons toujours consulter notre table de variable Qui va me retourner soit la valeur soit le type choix de l'adresse de la variable en question ou une erreur la variable n'est pas contenue dans la table.

➤ **Table des étiquettes**

Cette table va me retourner le numéro ou bien l'adresse d'une instruction passée ou précédente selon le contexte Nous allons l'utiliser dans le cas des instructions répétitives

➤ **La fonction de génération** : void **genererInst**('opérateur', 'opérande')

C'est la fonction de génération de code. Elle est élémentaire et ajoute à la table des instructions cibles une entrée pour générer l'instruction en question.

➤ **La génération pour un opérande**

Un opérande peut être un identifiant, un entier, une chaîne ou un réel. Ainsi pour générer l'instruction cible d'un réel, d'un entier ou d'une chaîne il suffit d'appeler la fonction **genererInst**('loadc', 'valeur') alors que Quand il s'agit d'un identifiant il suffit de faire **genererInst**('loadc', 'adresse')

➤ **La génération pour un opérateur**

On dispose de 2 types d'opérateurs. Les opérateurs logiques et les opérateurs de comparaison.

Ainsi pour générer l'instruction cible d'un opérateur logique ou de comparaison on appelle la fonction comme telle `genererInst('Opérateur', '')`

➤ La génération pour une Instruction d'affectation

Pour une affectation nous loadons les opérandes et les opérateurs à gauche. Dans notre cas nous allons utiliser la manière post fixer pour empiler ces derniers. Donc nous allons d'abord empiler(load/ loadc) les opérandes avant les opérateurs. Puis dans un 2e temps nous allons affecter la valeur de l'opération dans l'adresse de la variable qui se trouve à gauche tout en respectant la syntaxe et la sémantique de notre langage. Pour stocker la valeur de l'opération dans l'opérande à gauche appelons la fonction de generation comme telle `genererInst('Store', 'adresse de l'opérande gauche')`

➤ La génération pour une Instruction d'écriture

Lorsque nous faisons face à une instruction d'écriture nous allons appeler la fonction le génération comme tel `genererInst('Writec', 'valeur')` pour une valeur et `genererInst('Opérateur', '')` pour un identifiant.

➤ La génération pour une Instruction de lecture

Dans ce cas nous allons essayer de lire la valeur entrée par l'utilisateur et nous allons stocker cette valeur donne l'adresse de la variable passer en paramètre de la fonction de lecture.

Il suffit donc de faire `genererInst('Store', 'adrese var')`

➤ La génération pour une Instruction répétitive

Nous allons utiliser le Jzero et le jump pour générer une telle instruction.

`genererInst('Jzero', 'e1')` lorsque la condition au sommet de la pile n'est pas vérifiée. En d'autres termes nous allons sauter l'instructions se trouvant dans la boucle si la valeur au sommet de la pile est fausse

`genererInst('Jump', 'adrese var')` nous faisons un saut inconditionnel lorsque la condition est vérifiée. En d'autre terme, retourner dans la boucle.

3. L'interpréteur (BOUKILI Sabah – Ait Malek Yassine)

+ Introduction



Après la réalisation du compilateur générateur, maintenant on est face à la tâche de la réalisation de l'interpréteur. Un interpréteur ne produit pas du code final comme le cas des compilateurs, mais il procède lui-même à l'exécution des instructions du programme source après leur interprétation.

La plupart des interpréteurs sont capables d'exécuter les instructions d'un langage de programmation c'est-à-dire des instructions sous forme de texte. Par exemple, l'interpréteur **php.exe** est capable d'exécuter les instructions du langage PHP et l'interpréteur de **python.exe** est capable d'exécuter les instructions du langage Python. Les autres, plus rares, exécutent les instructions d'un langage machine, c'est-à-dire des instructions codées par des nombres. Les langages machines interprétés sont appelés « **bytecodes** » ou « **langages intermédiaires** ». Par exemple, l'interpréteur **java.exe** est capable d'exécuter les instructions du Java Byte Code.

+ Notre Travail :

- + On va utiliser **une machine à pile**, et comme on a vu au cours, ce type de machines utilisent des piles (stack) d'exécution, et tout se fait par rapport à cette pile. L'automate à pile est une machine abstraite, utilisée en informatique théorique, et plus précisément, en théorie des mesures. Donc on a créé une pile/stack (**screen1**).
- + Et en plus elle dispose d'une mémoire infinie organisée en pile (**screen2 : memory**). Un automate à pile prend en entrée un mot et réalise une série de transition pour chaque lettre de mot, dépendant de la lettre, de l'état, de l'automate, et du sommet de la pile, et peut aussi changer le contenu de la pile. Selon l'état de l'automate et de la pile, à la fin du calcul le mot peut être accepté ou refusé.

```

1 import java.util.ArrayList;
2
3 public class Stack extends ArrayList<String> {
4
5     public String get_str() {
6
7         return this.get( this.size() - 1 );
8
9     }
10
11     public void add_str(String str) {
12
13         this.add(str);
14
15     }
16
17     public String pop_str() {
18
19         String popped_out = this.get_str();
20         this.remove( this.size() - 1 );
21         return popped_out;
22
23     }
24 }
25

```

Screen1 : code de la pile

Screen2 : Memory

```

1 import java.util.HashMap;
2
3 public class Memory extends HashMap <Integer,String>{
4
5     public void add_m(Integer adr,String val) {
6
7         this.put(adr, val);
8
9     }
10
11
12     public String get_m(Integer adr) {
13
14         return this.get(adr);
15
16     }
17 }
18

```


- On a créé aussi un fichier « **operation** » (**Screen3**), qui contient les instructions cibles pour une machine à pile :
 - LOAD** : pour empiler la valeur à l'adresse « @x », ou bien LOADC v pour empiler directement la valeur v ;
 - STORE** : c'est dépiler la valeur qui est associée à l'adresse @x, au sommet de la pile, et la sauvegarder ;
 - ADD, SUB, MUL, DIV, ...Etc.**
 - JUMP** : le saut inconditionnel, pour passer à l'instruction cible prochaine ;
 - Jzero** : le saut conditionnel, si et seulement si la valeur au sommet de la pile est nulle, on peut passer à l'instruction indiquée dans l'adresse @i ;
 - READ, WRITEc, Write...**

```

1 public enum Operation {
2     LOAD, LOADC, STORE,
3     ADD, SUB, MUL, DIV, INC, DEC, NEQUAL,
4     JZERO, JUMP, JNZERO, JUMPNZ, JUMPNLE,
5     AND, OR, XOR, NOT,
6     WRITEC, WRITE, READC, READ,
7     END, ERROR
8 }
9
10
11
12
13

```

Screen3 : Operation

- Et on a créé un fichier aussi pour les instructions, (**screen4**), Les instructions cibles en JAVA, sont représentées par une classe, qui contient deux attributs, un pour l'opérateur, et l'autre pour l'opérande.

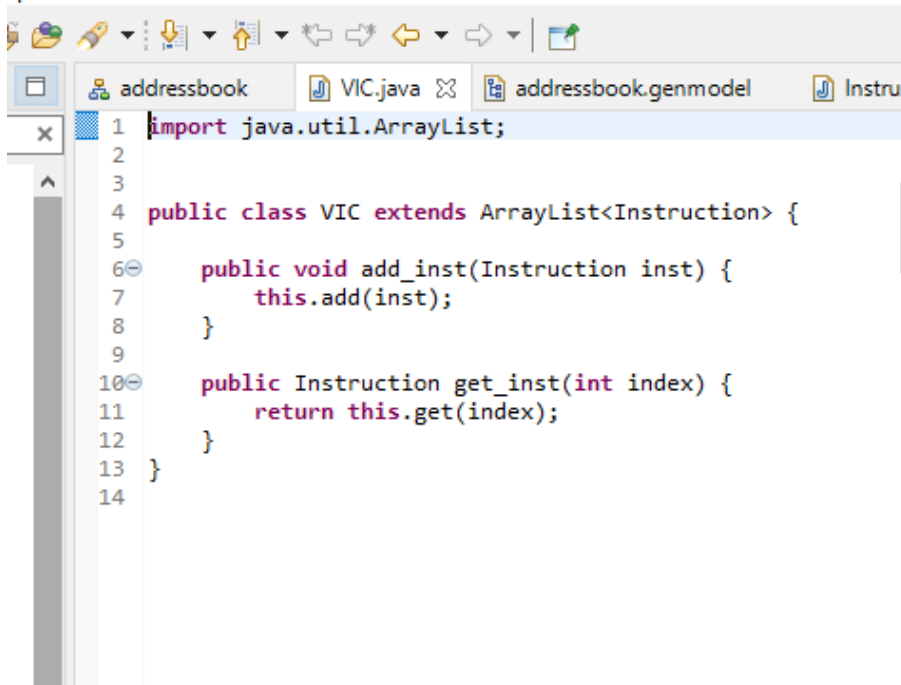
```

1
2 public class Instruction {
3
4     public Operation op;
5     public String opd;
6
7     public Instruction(Operation op, String opd) {
8         super();
9         this.op = op;
10        this.opd = opd;
11    }
12
13    public Instruction(Operation op) {
14        super();
15        this.op = op;
16    }
17
18    public Instruction() {
19        super();
20    }
21
22
23
24
25
26 }
27

```

Screen4 : instruction

- Donc on a besoin aussi où ces instructions vont être stockées, c'est-à-dire le vecteur des instructions cibles : VIC, (**screen5**).

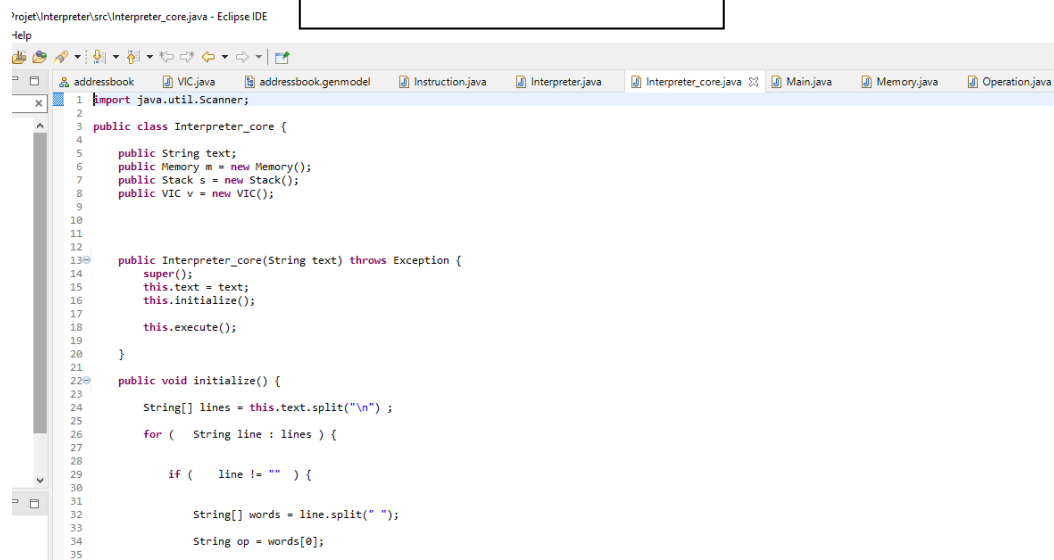


```
1 import java.util.ArrayList;
2
3
4 public class VIC extends ArrayList<Instruction> {
5
6     public void add_inst(Instruction inst) {
7         this.add(inst);
8     }
9
10    public Instruction get_inst(int index) {
11        return this.get(index);
12    }
13 }
14
```

Screen5 : VIC

- Lorsque l'interpréteur va prendre le fichier texte, il va le diviser en lignes et remplir les VIC, avec les instructions cibles, donc il va avoir besoin d'un programme qui fait cette tâche, c'est ce qu'on a réalisé en « interpreter_core.java » (**Screen6**).

Screen6 : interpetor_core



```
1 import java.util.Scanner;
2
3 public class Interpreter_core {
4
5     public String text;
6     public Memory m = new Memory();
7     public Stack s = new Stack();
8     public VIC v = new VIC();
9
10
11
12
13     public Interpreter_core(String text) throws Exception {
14         super();
15         this.text = text;
16         this.initialize();
17         this.execute();
18     }
19
20
21
22     public void initialize() {
23         String[] lines = this.text.split("\n");
24         for (String line : lines) {
25
26             if (line != "") {
27
28                 String[] words = line.split(" ");
29                 String op = words[0];
```

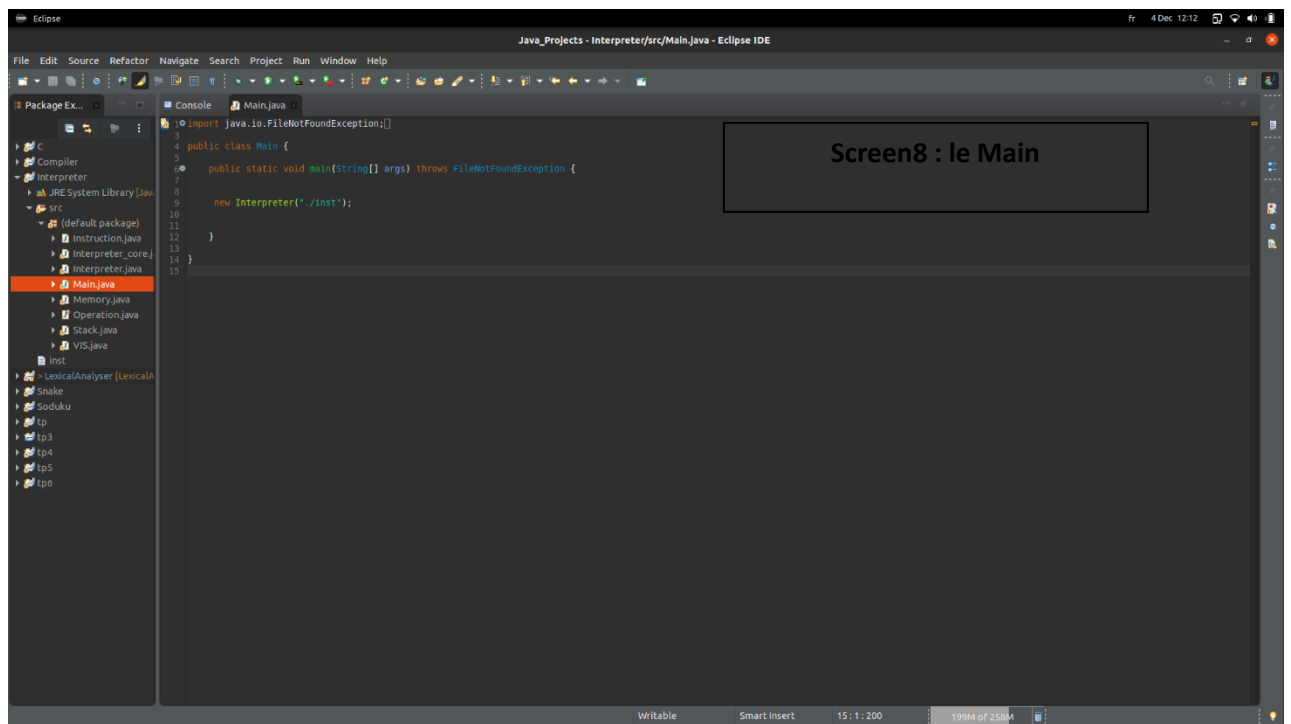
Et l'interpréteur dont on a utilisé un scanner pour prend un fichier txt, et le transformer en string pour le donner à l'interpretor_core (**screen7 : interpreter**).



```
1 import java.io.*;
2
3
4
5 public class Interpreter {
6
7     String File_Path;
8     String script = "";
9     Interpreter_core I;
10
11
12
13 public Interpreter(String file_Path) throws FileNotFoundException {
14     super();
15     this.File_Path = file_Path;
16     this.interpret();
17 }
18
19
20
21
22 public String get_script() throws FileNotFoundException {
23
24     Scanner s = new Scanner (    new DataInputStream (    new FileInputStream (    this.File_Path    )    )
25     String script = "";
26
27     while( s.hasNext() ) {
28
29         this.script += s.nextLine() + "\n" ;
30     }
31
32     return this.script ;
33 }
```

Screen7 : interpreter

Et finalement un Main pour tester notre programme/interpréteur. (**screen8 + 9 +10**).

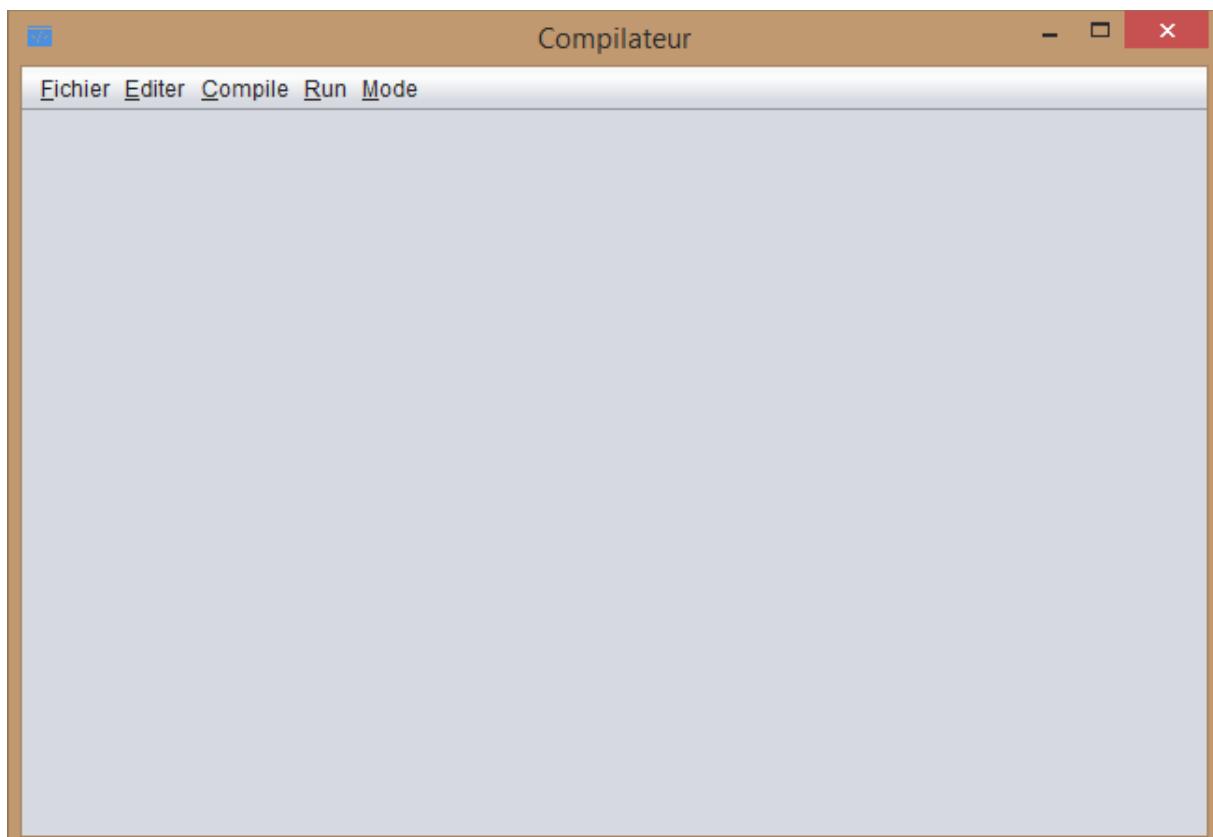


4. L'interface graphique : (Zarrouq Soukaina- Bounar Hibatallah)

L'interface graphique (en anglais GUI pour graphical user interface) désigne la manière dont est présenté un logiciel à l'écran pour l'utilisateur. C'est le langage d'échange entre l'humain et machine qui présente le positionnement des éléments : menus, boutons, fonctionnalités dans la fenêtre.

Notre projet a comme objectif de réaliser un compilateur qui traduit un code source à un autre cible , donc l'interface graphique est indispensable pour pouvoir exécuter toutes les fonctionnalités d'un compilateur et les utiliser aisément.

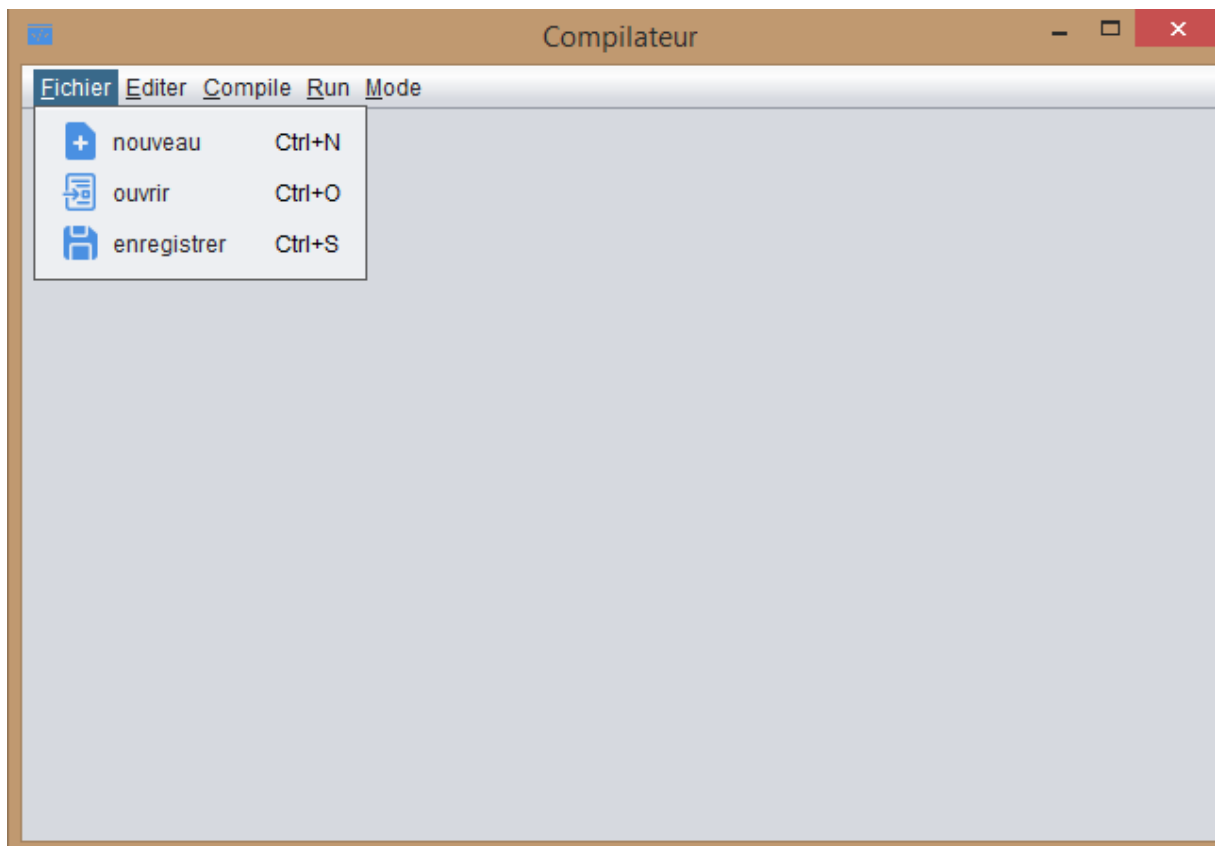
L'interface graphique de notre compilateur se compose d'une barre de menus contenant 5 menus : « Fichier », « Editer », « Compile », « Run » et « Mode ».



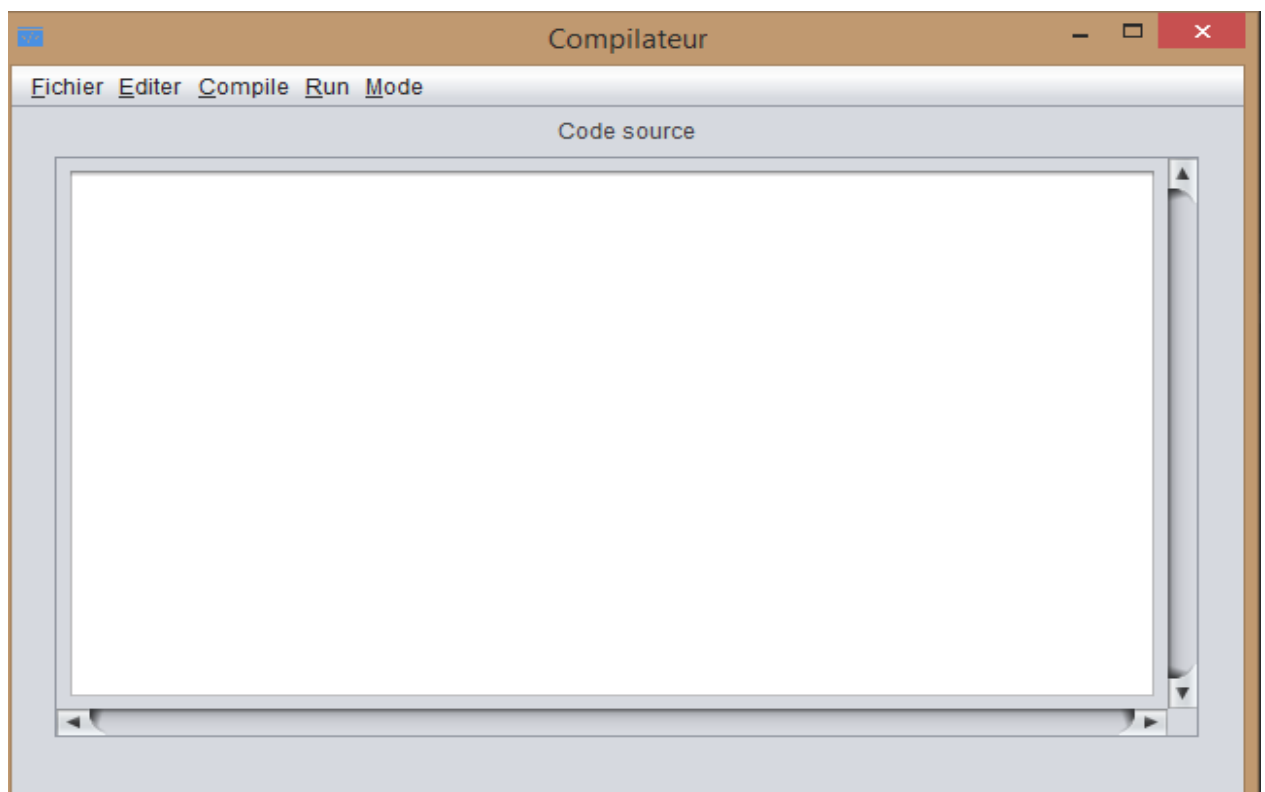
Le menu « Fichier » :

Le premier menu « Fichier » se compose de 3 items , « nouveau » qui permet de générer une zone texte pour saisir le code source , « ouvrir » pour sélectionner un

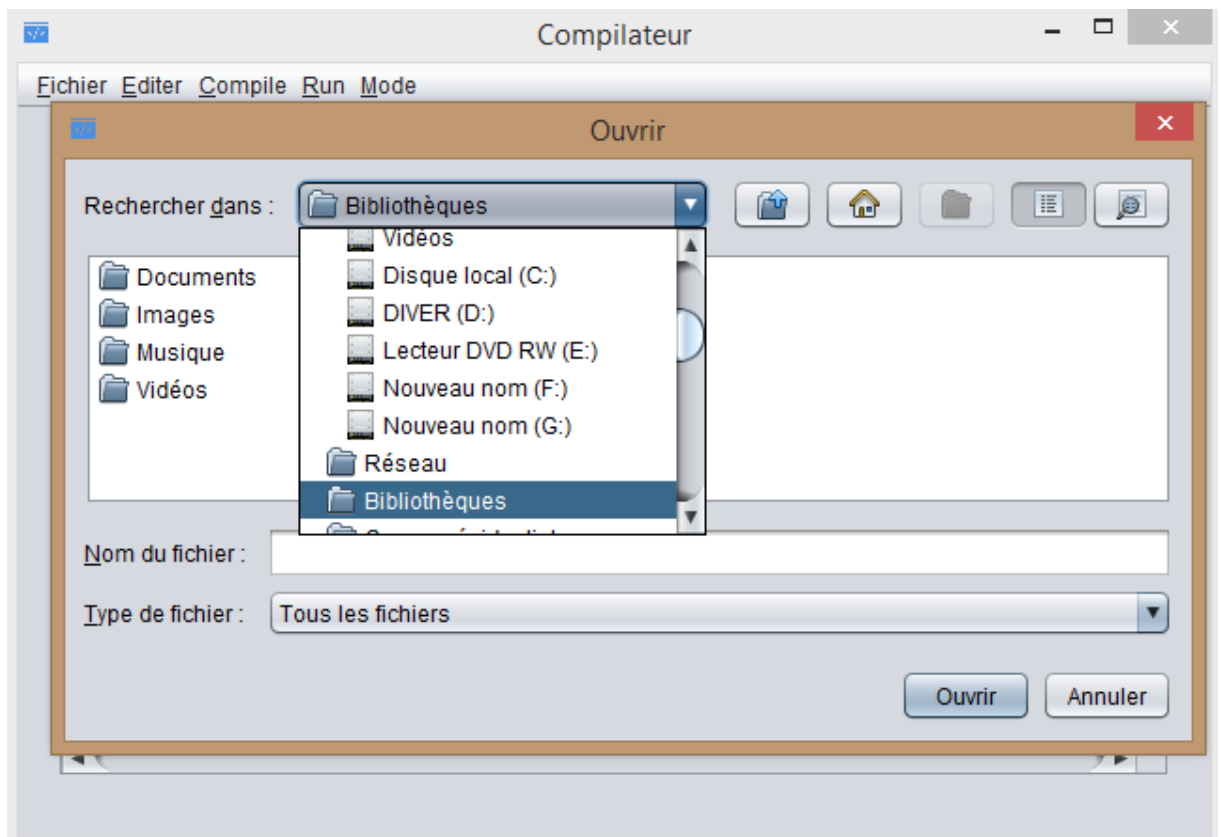
fichier qui contient le code source et qui est déjà sauvegardé dans le pc , et « enregistrer » pour enregistrer le code source .



- Après avoir cliqué sur « nouveau » :

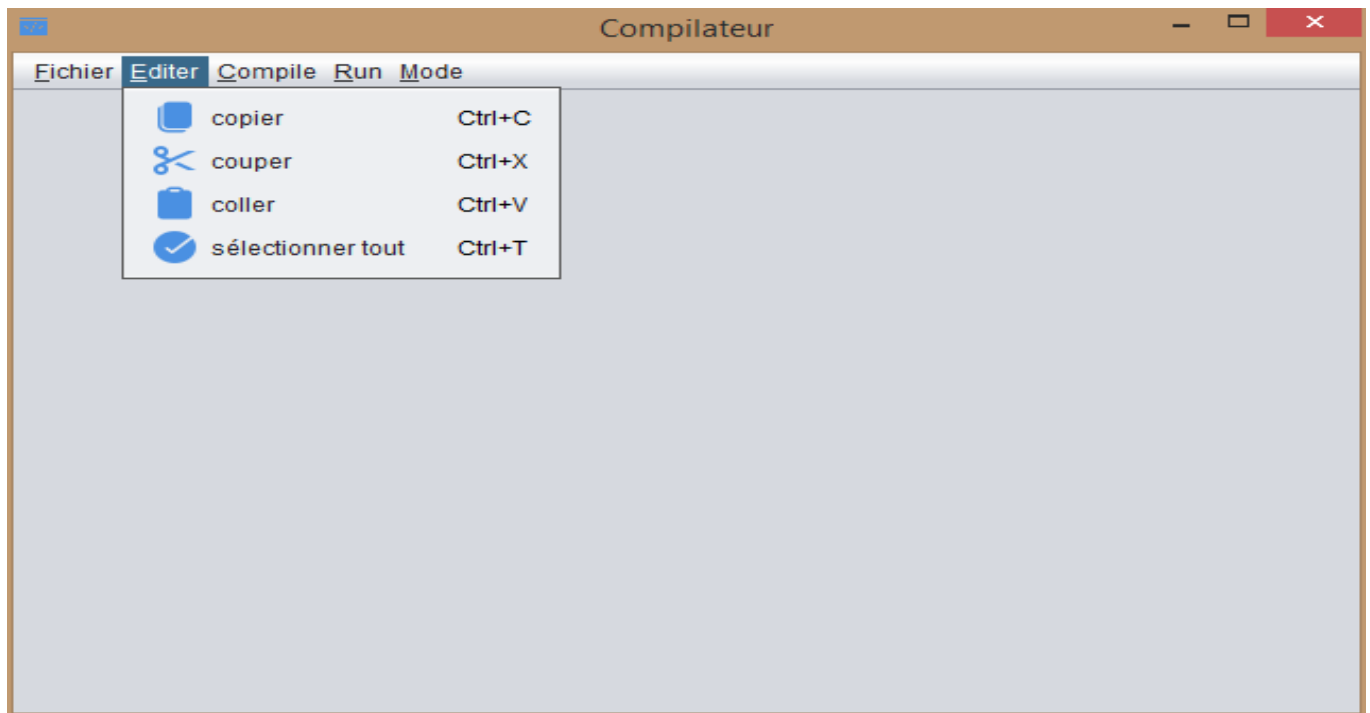


- Après avoir cliqué sur « ouvrir »



Le menu « Editer » :

Le deuxième menu « Editer » contient 4 items : copier qui permet de copier le texte , couper qui permet bien évidemment de le retirer puis sélectionner tout et coller pour avoir le contenu déjà copié.



Pour faciliter la tâche à l'utilisateur il peut directement choisir les différentes fonctionnalités citées précédemment en cliquant sur :

Ctrl+N pour « Nouveau », **Ctrl+O** pour « ouvrir », **Ctrl+S** pour choisir « enregistrer », **Ctrl+C** pour « Copier », **Ctrl+X** pour « Couper », **Ctrl+V** pour « Coller », **Ctrl+T** pour « Sélectionner tous ».

Le menu « Compiler » :

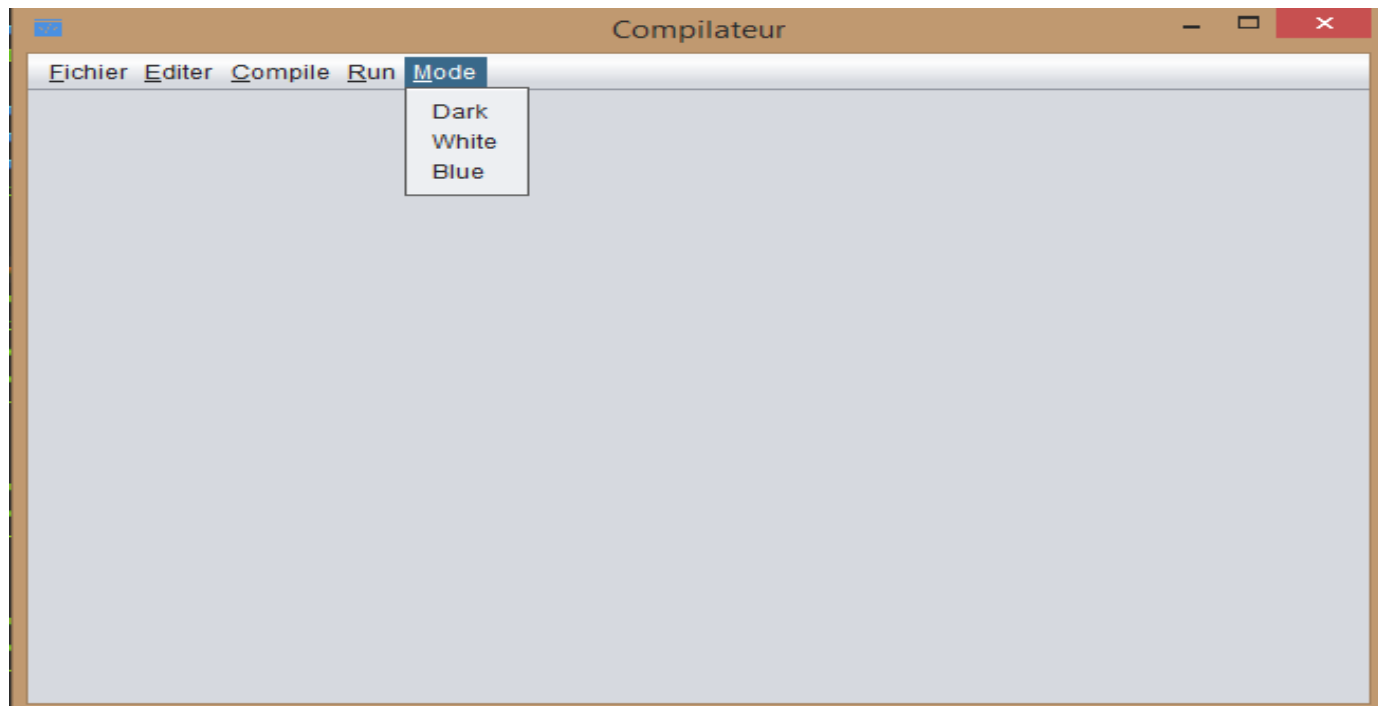
Pour le troisième menu « Compiler », il permet de traduire le code source saisi ou bien sélectionné par l'utilisateur à un code cible qui peut être exécuté par l'interpréteur. Dans ce processus de traduction l'analyseur lexical, l'analyseur syntaxique et l'analyseur sémantique permettent de vérifier si le code source est conforme à la grammaire ou bien s'il contient des erreurs par exemple : les erreurs de syntaxe, utilisation des variables indéfinies.

Le menu « Run » :

Pour le quatrième menu « Run » il permet d'abord de compiler le code source, d'exécuter le code cible obtenu, de vérifier s'il contient des erreurs d'exécution et d'afficher le résultat, cette exécution se fait grâce à l'interpréteur.

Le menu « Mode » :

Pour le cinquième menu « Mode », il permet à l'utilisateur de choisir la couleur de background du compilateur.



Conclusion :

Le travail effectué est résultat d'un travail acharné du groupe mais plein de bon moments . Ce projet nous a permet de bien comprendre le fonctionnement d'un compilateur qu'on utilise toujours en programmation ainsi de bien profiter du travail en groupe en partageant les informations et les techniques utilisées par chacun de nous.