

# Compilation

## Devoir

### Table des matières

I. Introduction .....	2
II. Solution proposée et limitations .....	2
III. Pour tester .....	3
IV. Quelques résultats de tests .....	4
V. Conclusion .....	5

**Baba SOW**

ING1 - Spécialité Informatique

## **I. Introduction :**

Le projet consiste à implémenter un interpréteur du langage **while0**. Ce langage prend en charge des constructions de base telles que les **variables**, les **expressions arithmétiques** et **booléennes**, les **commandes conditionnelles** (if), les **boucles** (while), et les **séquences de commandes**. L'interpréteur utilise Flex pour l'analyse lexicale et Bison pour l'analyse syntaxique.

Les structures de données pour l'arbre syntaxique abstrait (AST) et la gestion des variables sont définies dans les fichiers struct.h et struct.c.

## **II. Solution proposée :**

- **Structures de Données (struct.h et struct.c)**

Les structures de données principales comprennent des nœuds pour l'AST et une liste de variables :

- **NoeudAST** : Représente un nœud de l'AST et peut contenir une valeur, un nom de variable, des nœuds enfants pour des opérations, ou des structures pour les conditions if, les boucles while, et les séquences de commandes.
- **NoeudVariable et VariableList** : Gèrent les variables, chaque variable ayant un nom et une valeur associée.

Les fonctions de création de nœuds (creerNoeudNombre, creerNoeudVariable, etc.) facilitent la construction de l'AST à partir des tokens reconnus par l'analyseur lexical et syntaxique.

- **Analyse Syntaxique (parser.y)**

Le fichier scanner.l définit les règles pour l'analyse lexicale du code source du langage while0 :

- Les **mots-clés** comme **if**, **while**, **then**, **else**, **do**... sont reconnus.
- Les **nombres entiers** sont convertis en nœuds de type **NOEUD\_NOMBRE**.
- Les **identificateurs** sont convertis en nœuds de type **NOEUD\_VARIABLE**.

Flex est utilisé pour générer l'analyseur lexical qui produit des **tokens** pour l'analyse syntaxique.

- **Analyse Syntaxique (parser.y)**

Le fichier parser.y définit la **grammaire** du langage while0 et les règles pour assembler les tokens en structures syntaxiques valides. Les règles de grammaire incluent :

- **program** : La racine représentant une liste de commandes suivie d'un point.
- **command\_list** : Une séquence de commandes séparées par des points-virgules.
- **command** : Peut-être une **affectation**, une **condition if**, une **boucle while**, ou un **bloc de commandes**.

Bison est utilisé pour générer l'analyseur syntaxique qui construit l'AST à partir des tokens fournis par l'analyseur lexical.

### • Évaluation des Expressions et Commandes

Les fonctions d'évaluation (**evaluerExpression** et **evaluerCommande**) parcourent l'AST pour calculer les valeurs des expressions et exécuter les commandes :

**Expressions arithmétiques** : **evaluerExpression** calcule les valeurs des opérations arithmétiques.

**Expressions booléennes** : **evaluerBExpression** évalue les opérations booléennes.

**Commandes** : **evaluerCommande** exécute les commandes, modifiant les valeurs des variables selon les opérations définies dans l'AST.

### • Affichage des Expressions et Commandes

Les fonctions d'affichage (**afficherExpression**, **afficherBExpression**, et **afficherCommande**) convertissent l'AST en une représentation textuelle lisible, utile pour le débogage et la vérification de la structure de l'AST.

### • Limitations

- **Types de données** : Seuls les entiers sont supportés. Pas de gestion des flottants, des chaînes de caractères, ou d'autres types de données.
- **Portée des variables** : Les variables sont globales. Pas de gestion de portée locale.

## III. Pour tester :

Vous pouvez tester le code via les commandes suivantes :

```
flex scanner.l
```

```
bison -d -Wcounterexamples parser.y
```

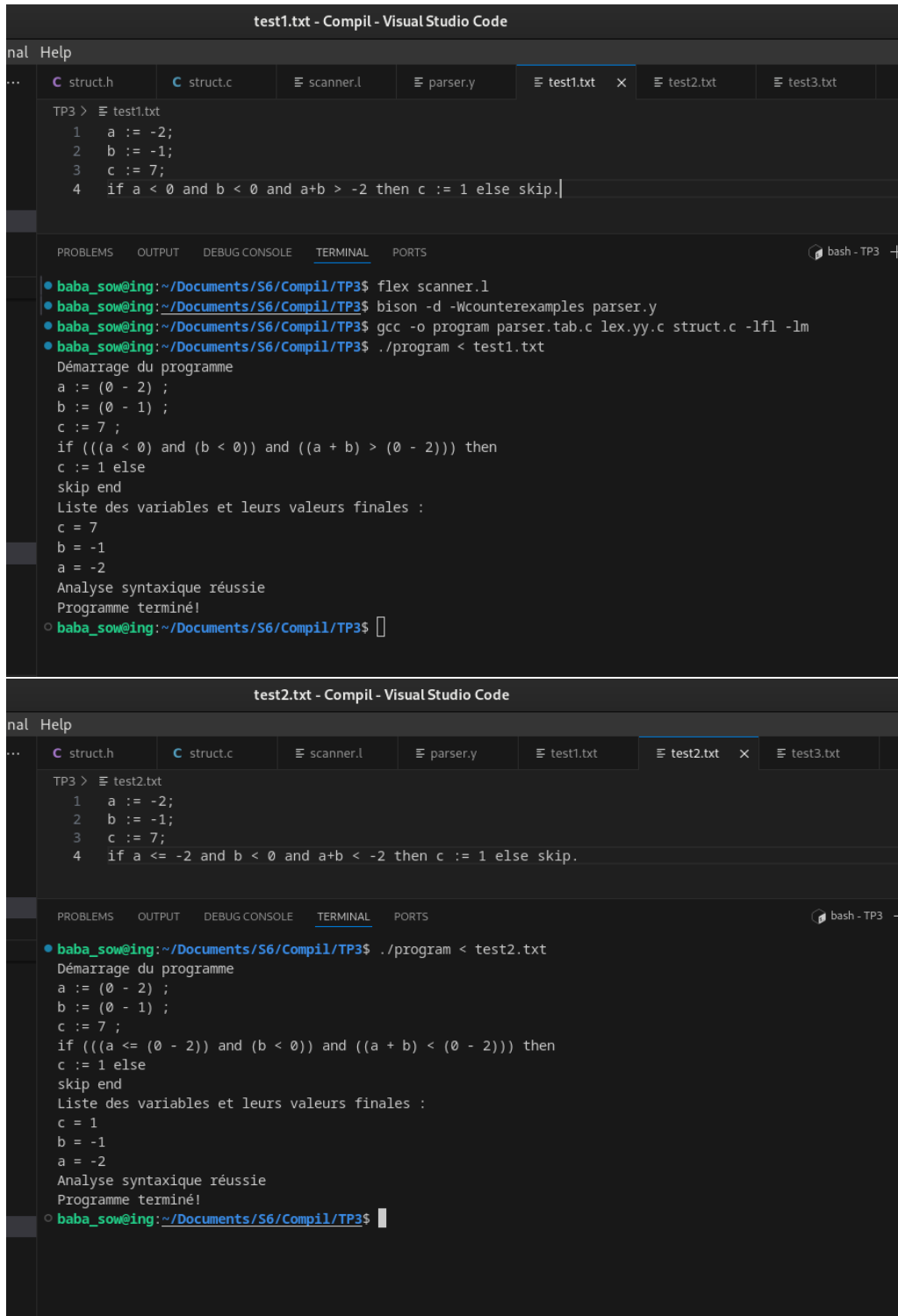
```
gcc -o program parser.tab.c lex.yy.c struct.c -lfl
```

```
./program < test3.txt
```

Vous pouvez tester aussi avec le fichier **test1.txt** ou **test2.txt** ou votre **fichier .txt** qui respecte la syntaxe.

**NB:** le programme doit se terminer par un **“ . ”** à la place de **“ ; ”** (finir la dernière ligne avec un **“ . ”** au lieu de **“ ; ”** pour marquer la fin).

## IV. Quelques résultats de tests

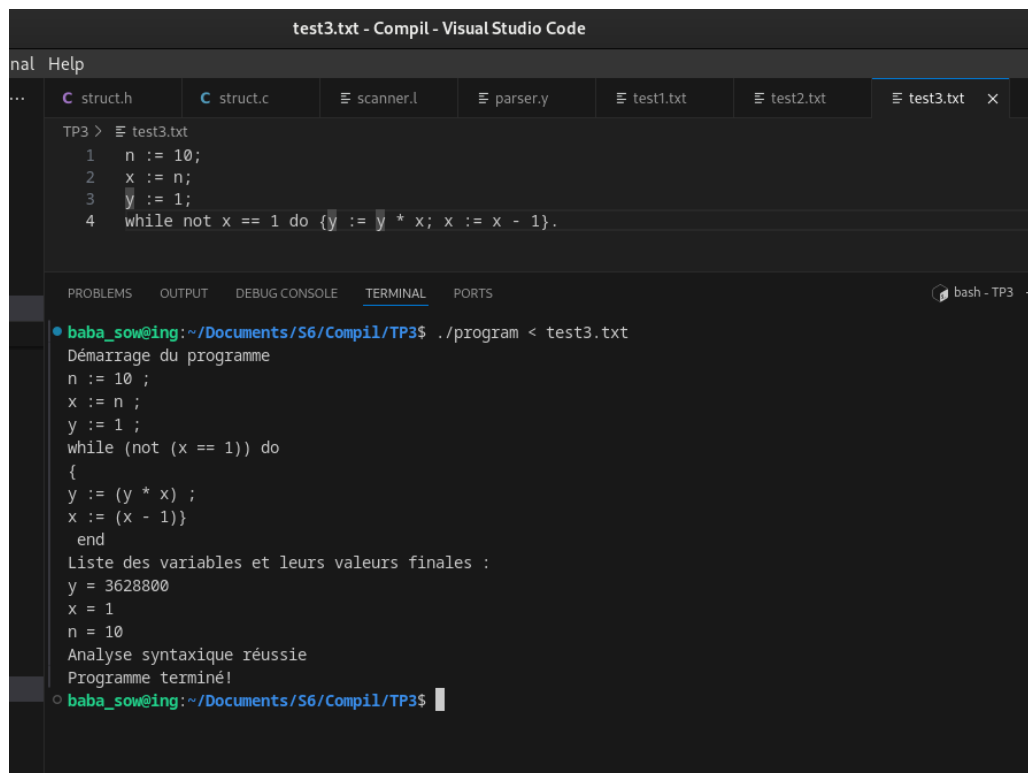


```
test1.txt - Compil - Visual Studio Code
...  C struct.h  C struct.c  scanner.l  parser.y  test1.txt  test2.txt  test3.txt
TP3 > test1.txt
1  a := -2;
2  b := -1;
3  c := 7;
4  if a < 0 and b < 0 and a+b > -2 then c := 1 else skip.

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
bash - TP3
• baba_sow@ing:~/Documents/S6/Compil/TP3$ flex scanner.l
• baba_sow@ing:~/Documents/S6/Compil/TP3$ bison -d -Wcounterexamples parser.y
• baba_sow@ing:~/Documents/S6/Compil/TP3$ gcc -o program parser.tab.c lex.yy.c struct.c -lfl -lm
• baba_sow@ing:~/Documents/S6/Compil/TP3$ ./program < test1.txt
Démarrage du programme
a := (0 - 2) ;
b := (0 - 1) ;
c := 7 ;
if (((a < 0) and (b < 0)) and ((a + b) > (0 - 2))) then
c := 1 else
skip end
Liste des variables et leurs valeurs finales :
c = 7
b = -1
a = -2
Analyse syntaxique réussie
Programme terminé!
• baba_sow@ing:~/Documents/S6/Compil/TP3$

test2.txt - Compil - Visual Studio Code
...  C struct.h  C struct.c  scanner.l  parser.y  test1.txt  test2.txt  test3.txt
TP3 > test2.txt
1  a := -2;
2  b := -1;
3  c := 7;
4  if a <= -2 and b < 0 and a+b < -2 then c := 1 else skip.

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
bash - TP3
• baba_sow@ing:~/Documents/S6/Compil/TP3$ ./program < test2.txt
Démarrage du programme
a := (0 - 2) ;
b := (0 - 1) ;
c := 7 ;
if (((a <= (0 - 2)) and (b < 0)) and ((a + b) < (0 - 2))) then
c := 1 else
skip end
Liste des variables et leurs valeurs finales :
c = 1
b = -1
a = -2
Analyse syntaxique réussie
Programme terminé!
• baba_sow@ing:~/Documents/S6/Compil/TP3$
```



The screenshot shows the Visual Studio Code editor with a file named `test3.txt` open. The file contains the following C code:

```
1  n := 10;  
2  x := n;  
3  y := 1;  
4  while not x == 1 do {y := y * x; x := x - 1};
```

Below the editor, the terminal window shows the output of running the program. The prompt is `baba_sow@ing:~/Documents/S6/Compil/TP3$`. The output is:

```
./program < test3.txt  
Démarrage du programme  
n := 10 ;  
x := n ;  
y := 1 ;  
while (not (x == 1)) do  
{  
y := (y * x) ;  
x := (x - 1)}  
end  
Liste des variables et leurs valeurs finales :  
y = 3628800  
x = 1  
n = 10  
Analyse syntaxique réussie  
Programme terminé!
```

Les deux premiers tests montrent que la solution fonctionne bien avec des conditions (if then else), l'évaluation des expressions arithmétiques, booléennes et des commandes marche bien, et l'analyse syntaxique est bien faite.

Le dernier test (test3 pour calculer factorielle n), montre que la solution fonctionne aussi pour les boucles (while) et la composition de commandes.

## V. Conclusion

Ce projet implémente un interpréteur pour le langage while0, utilisant Flex et Bison pour l'analyse lexicale et syntaxique. Il gère les variables, les expressions arithmétiques et booléennes, les commandes conditionnelles et les boucles. Bien qu'il soit fonctionnel, plusieurs extensions peuvent être envisagées, comme la gestion de nouveaux types de données, l'introduction de la portée locale pour les variables et la gestion des erreurs.