上海交大—巴黎高科卓越工程师学院
SJTU-ParisTech Elite Institute of Technology

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

# Théorie des Langages de Programmation

**Henri 516261910008**

# Table des matières

# 1 Introduction

Les codes sont écrits dans VScode (langue C) et exécutés sur linux. J'ai vérifié bien qu'il n'y a ni erreur ni warning pour la compilation. Les variables sont nommées en anglais mais tous les commentaires sont écrits en français. Les fichers **.c* sont les fichers avec *main function* à tester, et les fichiers **.h* contiennent les définitions des fonctions.

```
henri@DESKTOP-DS7DCHE:/mnt/e/CCC/PLT_projet_codes_final$ cat /proc/version
Linux version 4.4.0-18362-Microsoft (Microsoft@Microsoft.com) (gcc version 5.4.0 (GCC) ) #476-Microsoft Fri Nov 01 16:53:
00 PST 2019
```

FIGURE 1 – Environnement de la compilation et l'exécution

Pour les parties *analyse lexicale*, *analyse syntaxiaue* et *analyse sémantique*, je donne un bon exemple et un exemple avec des érreurs pour tester. $(i.e. Dpile.txt, Test-lexical.txt, Test-syntaxique.txt, Test-semantique.txt)$ Après avoir fini ces trois analyses, j'ai réalisé une compilation afin d'obtenir la machine virtuelle. Finalement, selon cette machine virtuelle, j'examine si l'une expression est acceptée ou refusée.

# 2 Diagramme du système

On veut écrire un compilateur d'automate qui lit un fichier contenant la description de l'automate et produit en machine virtuelle un code capable de reconnaître ou pas tout mot qui lui sera fourni. Il permet aussi d'exécuter les machines virtuelles écrites.

L'analyse lexical donne le lexème de l'automate. Puis, l'analyse syntaxique prend ce lexème, examine le syntaxe et donne un arbre syntaxique. Ensuite, l'analyse sémantique prend cet arbre syntaxique, vérifie qu'il a du sens et donne un arbre syntaxique correct. On fait la compilation selon cet arbre et obtient une machine virtuelle. Enfin, on réalise l'exécution selon cette machine virtuelle et vérifie si le mot donné est valid ou invalid.

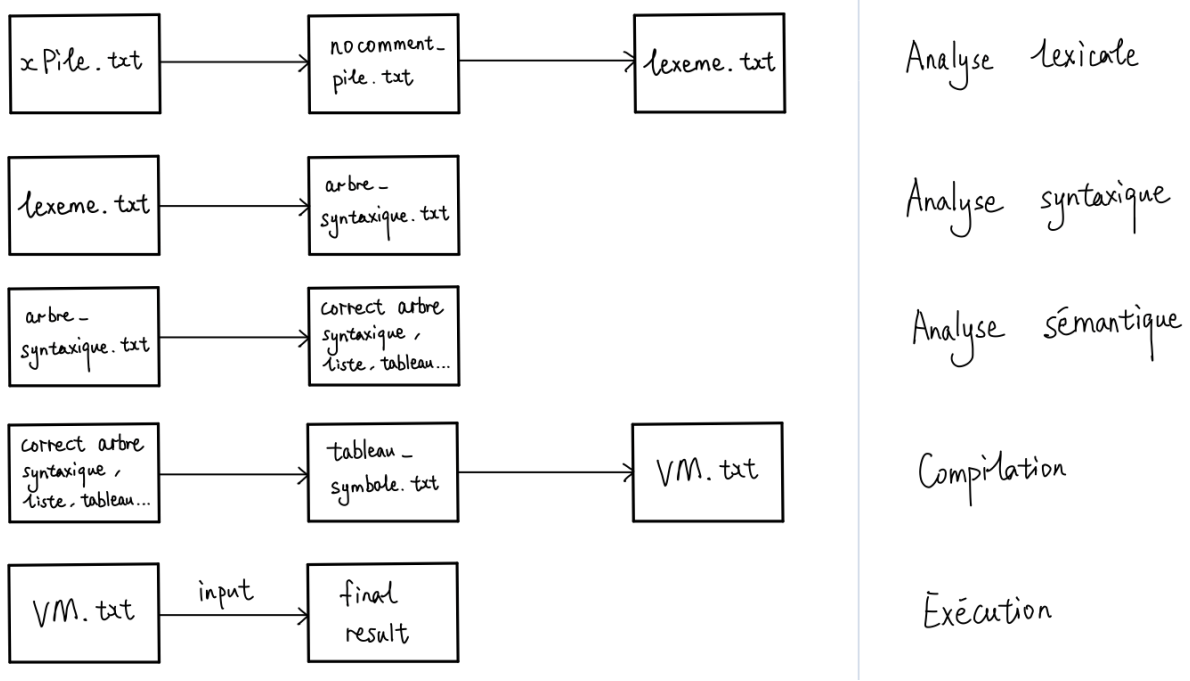


FIGURE 2 – Le diagramme de la programmation

# 3 Description des fonctions

## 3.1 *analyse lexicale*

Dans cette partie, les deux fichiers concernés sont *analyseur-lexical.c* et *analyseur-lexical.h*. Le *.c fichier contient la *main function* et le *.h fichier contient les définitions des fonctions utilisées. Les deux fichiers pour tester sont *Dpile.txt* et *Test-lexical.txt*. Si on détecte les erreurs, le programme s'arrête et donne le type des erreurs.

- Fonction *supprimer-commentaire* a pour but de supprimer les commentaires dans le fichier donné. Par exemple, les paragraphes entre /∗ ... ∗/ ou les lignes apès //... sont les commentaires à enlever. J'ai appliqué un automate avec 5 états pour réalisé cette fonction. L'entrée de cette fonction est le fichier donné, et la sortie est le fichier *nocomment.txt*.

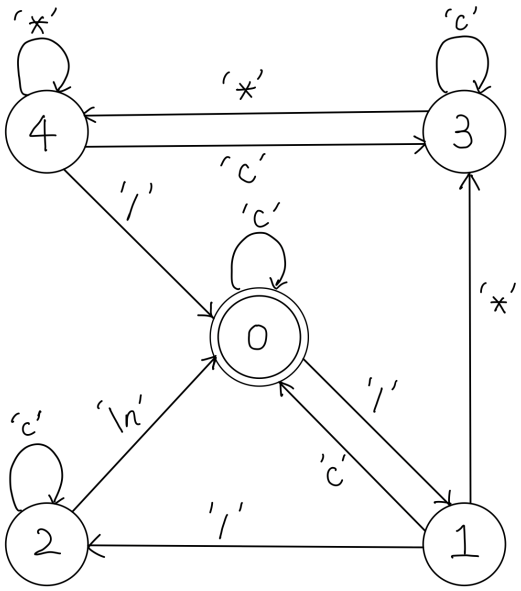


FIGURE 3 – L'automate avec 5 états pour supprimer les commtaires

- Fonction *traiter-file* est définie pour supprimer les espaces et les newlines dans le fichier *nocomment.txt*. La sortie de cette fonction est le fichier *lexeme.txt*.

- Fonction *examiner-caracteres* est utilisée pour vérifier si tous les caractères (lettre par lettre) dans le fichier *lexeme.txt* sont valides (dans le lexème que je définis)) ou pas. Si non, le fichier est lexicalement faux et on arrête le programme.

caractères normaux : 0 ~ 9 a ~ z A ~ Z
symboles normaux : {} () = [] ` " ,
caractères spéciaux: 零 ~ 十 →

FIGURE 4 – Lexème de l'automate

- Fonction *examiner-mots* est dans le but de vérifier si les mots dans le fichier *lexeme.txt* sont des mots clés. C'est à dire que les 5 mots français : **Automate, etats, initial, final, transitions**. Si non, le fichier contient des mots invalids. Alors, le fichier est lexicalement faux et on arrête le programme. Si le fichier *lexeme.txt* passe ces deux fonctions sans avoir d'erreur, alors on obtient le lexème de l'automate donné.

### 3.2 *analyse syntaxique*

Dans cette partie, les deux fichiers concernés sont *analyseur-syntaxique.c* et *analyseur-syntaxique.h*. Le *.c fichier contient la *main function* et le *.h fichier contient les définitions des fonctions utilisées. Les deux fichiers pour tester sont *Dpile.txt* et *Test-syntaxique.txt*. Si on détecte les erreurs, le programme s'arrête et donne le type des erreurs.

- Function *examiner-syntaxique* réalise l'analyse syntaxique en appliquant un automate avec 50 états. Cet automate restreint la formule du fichier de l'automate exactement comme les trois exemples. Par exemple, le premier mot doit être Automate, suivant les deux crochets et le nombre de piles, puis la signe d'égalité, etc. On limite l'écriture du fichier strictement. Tant qu'il y a une erreur du grammaire ou du syntaxe, le fichier est syntaxiquement faux et on arrête le programme.

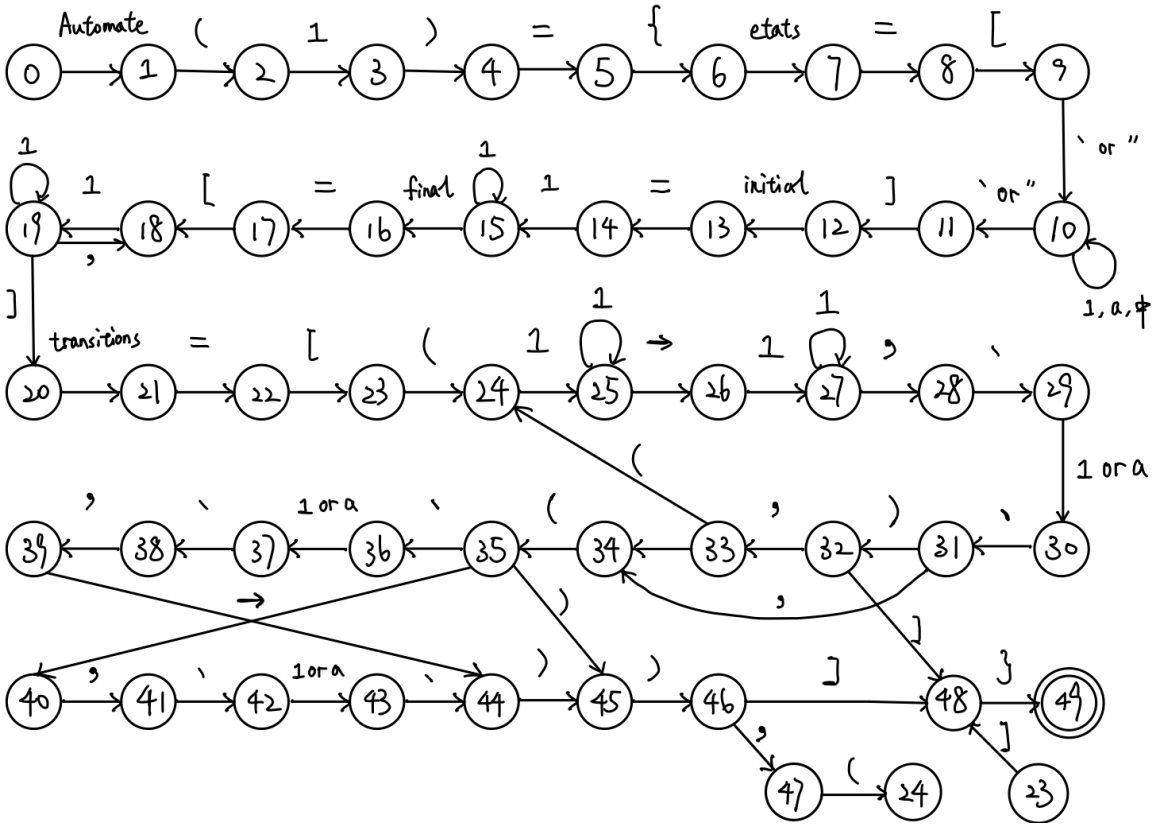


FIGURE 5 – L'automate avec 50 états pour réaliser l'analyse syntaxique

- La grammaire BNF de mon automate est définie comme suivante :
  - ✏ <lettre> : := "a" | ... | "z" | "A" | ... | "Z"
  - ✏ <chiffre> : := "0" | ... | "9"
  - ✏ <naturel> : := <chiffre>{<chiffre>}
  - ✏ <naturels> : := {<naturel>","} <naturel>
  - ✏ <guillemet> : := " ' " | " " "
  - ✏ <chinois> : := "零" | "一" | "二" | "三" | "四" | "五" | "六" | "七" | "八" | "九" | "十"
  - ✏ <spécial> : := <chinois> | "→"
  - ✏ <début> : := "Automate=(" "0"|"1"|"2" ")"
  - ✏ <initial> : := "initial="<naturel>
  - ✏ <mot> : := <lettre> | <lettre><mot>

- ✏ <état> : := <guillemet>(<naturel>|<mot >|<chinois>)<guillemet>
- ✏ <états> : := {<etats>“,”} <etat>
- ✏ <élément> : := <guillemet>(<chiffre>|<lettre>)<guillemet>
- ✏ <push> : := “ (→, ”<élément>“)”
- ✏ <pop> : := “(”<élément>“, →)”
- ✏ <trans> : := <naturel>“→”<naturel>“,”<guillemet><élément><guillemet>
- ✏ <pile> : := “,”( <push> | <pop> | “()” )
- ✏ <transition> : := “(” ( <trans> | <trans><pile> | <trans><pile><pile>) “)”
- ✏ <transitions> : := {<transition>“,”} <transition>
- ✏ <expr> : := ( (“états”|“final”|“transitions”) “=[”<naturels>|<états>|<transitions>“]” ) | <initial>
- ✏ <exprs> : := {<expr>}
- ✏ <automate> : := <début>“{”<exprs>“}”

▶ L'entrée de cette fonction est *lexeme.txt* et la sortie est le fichier *arbre-syntaxique.txt*. Selon ce fichier, on produit le graphe de l'arbre syntaxique par graphviz. Cet arbre montre les informations importantes et la structure de l'automate.

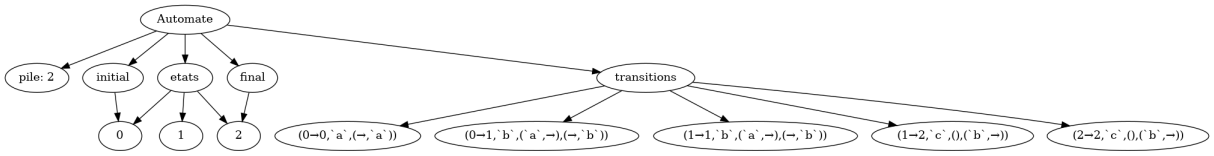


FIGURE 6 – Un exemple de l'arbre syntaxique

### 3.3 *analyse sémantique*

Dans cette partie, les deux fichiers concernés sont *analyseur-semantique.c* et *analyseur-semantique.h*. Le *.c fichier contient la *main function* et le *.h fichier contient les définitions des fonctions utilisées. Les deux fichiers pour tester sont *Dpile.txt* et *Test-semantique.txt*. Si on détecte les erreurs, le programme s'arrête et donne le type des erreurs.

▶ Fonction *examiner-arbre-syntaxique* est la fonction définie pour vérifier que l'arbre syntaxique que l'on a obtenu a du sens. Je vérifie les points suivants :

- ✏ Mots clés sont tous là. (**Automate, etats, initial, final, transitions**)
- ✏ ll n'y pas de répétition dans l'état. (ex. etats = [一, 一, 二, 三])
- ✏ Les numéros des états, de l'initial et du final ne dépassent pas le nombre de l'état moins un.
- ✏ Les numéros des états dans les transitions ne dépassent pas le nombre de l'état moins un.
- ✏ Le nombre des manipulations de piles n'excède pas le nombre des piles défini au début.
- ✏ Les pointers sont utilisé pour enregistrer les données.

▶ J'enregistre l'arbre syntaxique dans les tableaux et les listes afin de les utiliser après. Après avoir fini l'analyse sémantique, on obtient un bon arbre syntaxique pour la compilation. Jusqu'ici, si le fichier donné passe tous les trois analyses sans avoir des erreurs, il est lexicalement, syntaxiquement et sémantiquement correct. Donc, on peut continuer les étapes suivantes : la compilation et l'exécution. Sinon, le programme va s'arrêter en cas des erreurs et donne le type des erreurs.

```
char etats[10][10] = {0};          // etats: 1, aa, bb, 二三
int initial = 0;                   // initial: 1
int final[10] = {0};               // final: 3, 9, 16
int transitions[10][2] = {0};      // transitions: 0 1 | 2 3 | 11 26 | 8 322 |
char caractere[10] = {0};          // caractere: a | b | 1 | 9 |
char pile1[10][2] = {0};           // pile1: + a | - a | + c | - d |
char pile2[10][2] = {0};           // pile2: + d | - e | + a | - b |
int num_etats = 0;                 // nombre d'etat
int num_final = 0;                 // nombre de final
int num_transitions = 0;           // nombre de transitions
int num_pile = 0;                  // nombre de piles
```

FIGURE 7 – Les tableaux et les listes pour enregistrer l'arbre syntaxique correct

### 3.4 *compilation*

Dans cette partie, les cinq fichiers concernés sont *analyseur-lexical.h*, *analyseur-syntaxique.h*, *analyseur-semantique.h*, *compile-automate.h* et *compile-automate.c*. Le *.c fichier contient la *main function* et les *.h fichier contiennent les définitions des fonctions utilisées. Le fichier donné doit passer les trois analyses pour qu'il puisse être compilé.

- Fonction *compilateur* compile l'arbre syntaxique obtenu dans l'analyse sémantique afin d'obtenir la machine virtuelle. La forme de VM est comme suivante :

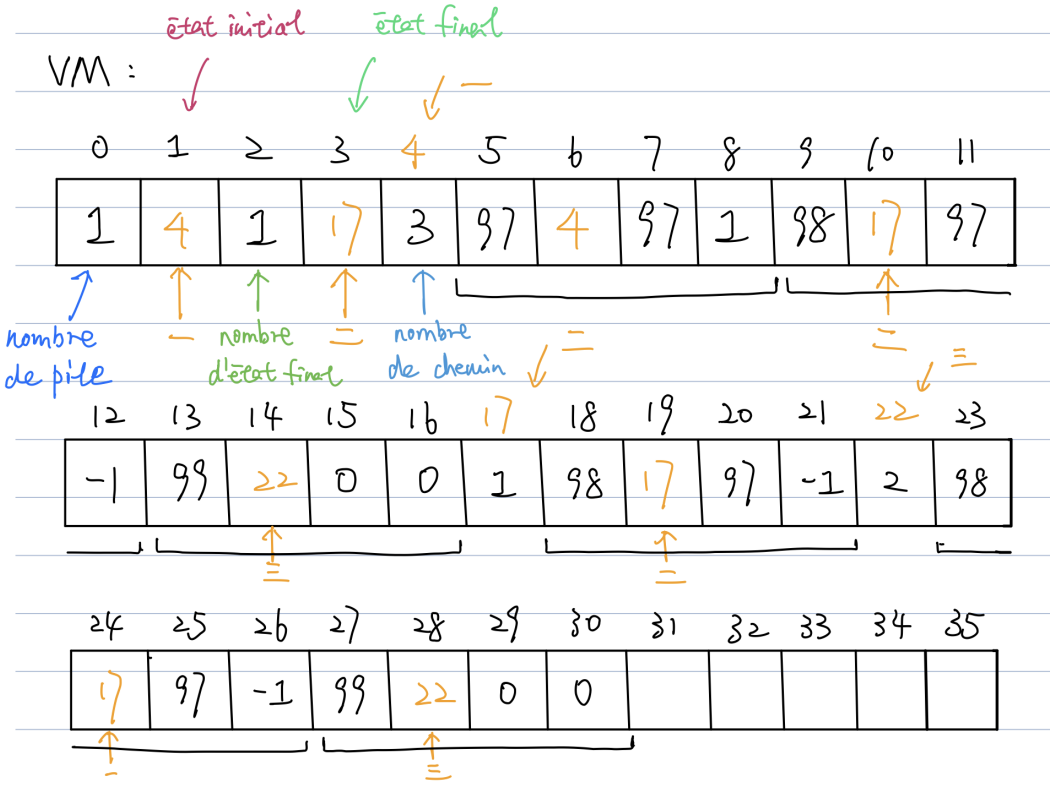


FIGURE 8 – Un exemple de la machine virtuelle avec une pile

- Fonction *enregistrer-VM* a pour but d'enregister la machine virtuelle dans un fichier. Les sortie de la fonction sont deux fichiers : un fichier *tableau-symbole.txt* contenant des informations des états (noms d'état et leurs addresses dans la VM), et l'autre est le fichier *VM.txt*.

### 3.5 *exécution*

Dans cette partie, les six fichiers concernés sont *analyseur-lexical.h*, *analyseur-syntaxique.h*, *analyseur-semantique.h*, *compile-automate.h*, *executeur.h*et *executeur.c*. Le *.c fichier contient la *main function* et les *.h fichier contiennent les définitions des fonctions utilisées. L'exécuteur prendra le fichier VM et sera capable de reconnaître ou pas un mot saisi au clavier. ll y en a deux modes : *mode normal* et *mode debug*. On va voir comment tester dans la partie suivante.

## 4 Test

Dans cette partie, je montre comment "compiler et exécuter" les fichiers que je donnent. Vous pouvez tester avec la même ordre ou comme vous voulez. Je crée trois fichiers avec des erreurs respectivement pour les trois analyses. (*Test-lexical.txt, Test-syntaxique.txt et Test-semantique.txt*)

### 4.1 analyseur-lexical

```
henri@DESKTOP-DS7DCHE:/mnt/e/CCC/PLT_projet_codes_final$ gcc -o analyseur_lexical analyseur_lexical.c
henri@DESKTOP-DS7DCHE:/mnt/e/CCC/PLT_projet_codes_final$ ./analyseur_lexical
Donner le nom de fichier d'entree : Dpile
Automate(2)={etats=[`0`,`1`,`2`]initial=0final=[2]transitions=[(0→0,`a`,(→,`a`)),(0→1,`b`,(`a`,→),(→,`b`)),(1→1,`b`,(`a`,
→),(→,`b`)),(1→2,`c`,(),(`b`,→)),(2→2,`c`,(),(`b`,→))]}
--------------------------------------------------------------------------------------------------------------------------
L'analyse lexicale est finie!
```

FIGURE 9 – Demo : l'analyse lexicale pour Dpile.txt

```
henri@DESKTOP-DS7DCHE:/mnt/e/CCC/PLT_projet_codes_final$ ./analyseur_lexical
Donner le nom de fichier d'entree : Test_lexical
Automathe(0)={Bonjour/etats=["1","2","3","Initial","哈"]initial=3phinal=[0,1,2]@$&transitions=[(3→0,`0`),(3→1,`1`),(3→2,`
2`),(0→1,`1`),(0→2,`2`),(1→0,`0`),(1→2,`2`),(2→0,`0`),(2→1,`1`)]Goodbye}
Caractere inconnu: /
Caractere inconnu: 哈
Caractere inconnu: @
Caractere inconnu: $
Caractere inconnu: &
Le fichier est lexicalement faux! Type: Caractere invalide.
Mot clé inconnu: Automathe
Mot clé inconnu: Bonjour
Mot clé inconnu: phinal
Mot clé inconnu: Goodbye
Le fichier est lexicalement faux! Type: Mot cle invalide.
--------------------------------------------------------------------------------------------------------------------------
L'analyse lexicale est finie!
```

FIGURE 10 – Demo : l'analyse lexicale pour Test-lexical.txt

### 4.2 analyseur-syntaxique

```
henri@DESKTOP-DS7DCHE:/mnt/e/CCC/PLT_projet_codes_final$ gcc -o analyseur_syntaxique analyseur_syntaxique.c
henri@DESKTOP-DS7DCHE:/mnt/e/CCC/PLT_projet_codes_final$ ./analyseur_syntaxique
Donner le nom de fichier d'entree : Dpile
Automate(2)={etats=[`0`,`1`,`2`]initial=0final=[2]transitions=[(0→0,`a`,(→,`a`)),(0→1,`b`,(`a`,→),(→,`b`)),(1→1,`b`,(`a`,
→),(→,`b`)),(1→2,`c`,(),(`b`,→)),(2→2,`c`,(),(`b`,→))]}
--------------------------------------------------------------------------------------------------------------------------
L'analyse lexicale est finie!
--------------------------------------------------------------------------------------------------------------------------
L'analyse syntaxique est finie!
```

FIGURE 11 – Demo : l'analyse syntaxique pour Dpile.txt

```
henri@DESKTOP-DS7DCHE:/mnt/e/CCC/PLT_projet_codes_final$ ./analyseur_syntaxique
Donner le nom de fichier d'entree : Test_syntaxique
Automate(1)={etats=["一","二","三"]initial=0final=[1]transitions=[(0→0,`a`,(→,`a`))),(0→1,`b`,(`a`,→)),(2→1,`b`,,,(`a`,→
)),(0→2,`c`,()),(2→2,`c`),(1→1,`b`,(`a`,→))]}
-------------------------------------------------------------------------------------------------------------------------
L'analyse lexicale est finie!
Incorrect caractere: )
Le fichier est syntaxiquement faux!
-------------------------------------------------------------------------------------------------------------------------
L'analyse syntaxique est finie!
```

FIGURE 12 – Demo : l'analyse syntaxique pour Test-syntaxique.txt

## 4.3 analyseur-semantique

```
henri@DESKTOP-DS7DCHE:/mnt/e/CCC/PLT_projet_codes_final$ gcc -o analyseur_semantique analyseur_semantique.c
henri@DESKTOP-DS7DCHE:/mnt/e/CCC/PLT_projet_codes_final$ ./analyseur_semantique
Donner le nom de fichier d'entree : Dpile
Automate(2)={etats=[`0`,`1`,`2`]initial=0final=[2]transitions=[(0→0,`a`,(→,`a`)),(0→1,`b`,(`a`,→),(→,`b`)),(1→1,`b`,(`a`,
→),(→,`b`)),(1→2,`c`,(),(`b`,→)),(2→2,`c`,(),(`b`,→))]}
-------------------------------------------------------------------------------------------------------------------------
L'analyse lexicale est finie!
-------------------------------------------------------------------------------------------------------------------------
L'analyse syntaxique est finie!
nombre de pile: 2
nombre d'etats: 3
nombre de final: 1
nombre de transitions: 5
etats: 0 1 2
initial: 0
final: 2
transitions: 0 0 | 0 1 | 1 1 | 1 2 | 2 2 |
caractere: a | b | b | c | c |
pile1: + a | - a | - a |     |     |
pile2:     | + b | + b | - b | - b |
-------------------------------------------------------------------------------------------------------------------------
L'analyse semantique est finie!
```

FIGURE 13 – Demo : l'analyse sémantique pour Dpile.txt

```
henri@DESKTOP-DS7DCHE:/mnt/e/CCC/PLT_projet_codes_final$ ./analyseur_semantique
Donner le nom de fichier d'entree : Test_semantique
Automate(2)={etats=[`0`,`二`,`2`]initial=4final=[2,233]transitions=[(0→66,`a`,(→,`1`)),(0→1,`b`,(`a`,→),(→,`b`),(→,`b`),(
→,`b`),(→,`b`)),(1→2,`b`,(`a`,→),(→,`b`)),(1→2,`c`,(),(`b`,→)),(2→2,`c`,(),(`b`,→))]}
-------------------------------------------------------------------------------------------------------------------------
L'analyse lexicale est finie!
-------------------------------------------------------------------------------------------------------------------------
L'analyse syntaxique est finie!
Le fichier est semantiquement faux! Type: Nombre de piles exces.
Le fichier est semantiquement faux! Type: Numero d'initial exces.
Le fichier est semantiquement faux! Type: Numero de final exces.
Le fichier est semantiquement faux! Type: Numero de l'etat exces dans les transitions.
-------------------------------------------------------------------------------------------------------------------------
L'analyse semantique est finie!
```

FIGURE 14 – Demo : l'analyse sémantique pour Test-semantique.txt

## 4.4 compile-automate

```
henri@DESKTOP-DS7DCHE:/mnt/e/CCC/PLT_projet_codes_final$ gcc -o compile_automate compile_automate.c
henri@DESKTOP-DS7DCHE:/mnt/e/CCC/PLT_projet_codes_final$ ./compile_automate
Donner le nom de fichier d'entree : Dpile
Automate(2)={etats=[`0`,`1`,`2`]initial=0final=[2]transitions=[(0→0,`a`,(→,`a`)),(0→1,`b`,(`a`,→),(→,`b`)),(1→1,`b`,(`a`,
→),(→,`b`)),(1→2,`c`,(),(`b`,→)),(2→2,`c`,(),(`b`,→))]}
--------------------------------------------------------------------------------------------------------------------------
L'analyse lexicale est finie!
--------------------------------------------------------------------------------------------------------------------------
L'analyse syntaxique est finie!
nombre de pile: 2
nombre d'etats: 3
nombre de final: 1
nombre de transitions: 5
etats: 0 1 2
initial: 0
final: 2
transitions: 0 0 | 0 1 | 1 1 | 1 2 | 2 2 |
caractere: a | b | b | c | c |
pile1: + a | - a | - a |     |     |
pile2:     | + b | + b | - b | - b |
--------------------------------------------------------------------------------------------------------------------------
L'analyse semantique est finie!
tableau de symbole:  0 4 | 1 17 | 2 30 |
machine virtuelle: 2 4 1 30 2 97 4 97 1 0 0 98 17 97 -1 98 1 2 98 17 97 -1 98 1 99 30 0 0 98 -1 1 99 30 0 0 98 -1
```

FIGURE 15 – Demo : la compilation pour Dpile.txt

```
henri@DESKTOP-DS7DCHE:/mnt/e/CCC/PLT_projet_codes_final$ ./compile_automate
Donner le nom de fichier d'entree : Test_lexical
Automathe(0)={Bonjour/etats=["1","2","3","Initial","哈"]initial=3phinal=[0,1,2]@$&transitions=[(3→0,`0`),(3→1,`1`),(3→2,`
2`),(0→1,`1`),(0→2,`2`),(1→0,`0`),(1→2,`2`),(2→0,`0`),(2→1,`1`)]Goodbye}
Caractere inconnu: /
Caractere inconnu: 哈
Caractere inconnu: @
Caractere inconnu: $
Caractere inconnu: &
Le fichier est lexicalement faux! Type: Caractere invalide.
Mot clé inconnu: Automathe
Mot clé inconnu: Bonjour
Mot clé inconnu: phinal
Mot clé inconnu: Goodbye
Le fichier est lexicalement faux! Type: Mot cle invalide.
--------------------------------------------------------------------------------------------------------------------------
L'analyse lexicale est finie!
Compilation erreur! Type: Faute lexicale.
```

FIGURE 16 – Demo : la compilation pour Test-lexical.txt et l'interruption

## 4.5 executeur

```
henri@DESKTOP-DS7DCHE:/mnt/e/CCC/PLT_projet_codes_final$ ./executeur -debug VM
Donner le nom de fichier d'entree : Dpile
Automate(2)={etats=[`0`,`1`,`2`]initial=0final=[2]transitions=[(0→0,`a`,(→,`a`)),(0→1,`b`,(`a`,→),(→,`b`)),(1→1,`b`,(`a`,
→),(→,`b`)),(1→2,`c`,(),(`b`,→)),(2→2,`c`,(),(`b`,→))]}
-----------------------------------------------------------------------------------------------------------------------
L'analyse lexicale est finie!
-----------------------------------------------------------------------------------------------------------------------
L'analyse syntaxique est finie!
nombre de pile: 2
nombre d'etats: 3
nombre de final: 1
nombre de transitions: 5
etats: 0 1 2
initial: 0
final: 2
transitions: 0 0 | 0 1 | 1 1 | 1 2 | 2 2 |
caractere: a | b | b | c | c |
pile1: + a | - a | - a |     |     |
pile2:     | + b | + b | - b | - b |
-----------------------------------------------------------------------------------------------------------------------
L'analyse semantique est finie!
tableau de symbole:  0 4 | 1 17 | 2 30 |
machine virtuelle: 2 4 1 30 2 97 4 97 1 0 0 98 17 97 -1 98 1 2 98 17 97 -1 98 1 99 30 0 0 98 -1 1 99 30 0 0 98 -1
Donner le mot d'entree : aabbcc
  -> Etat : 0   Pile1 : Vide    Pile2 : Vide
a -> Etat : 0   Pile1 : a       Pile2 : Vide
a -> Etat : 0   Pile1 : aa      Pile2 : Vide
b -> Etat : 1   Pile1 : a       Pile2 : b
b -> Etat : 1   Pile1 : Vide    Pile2 : bb
c -> Etat : 2   Pile1 : Vide    Pile2 : b
c -> Etat : 2   Pile1 : Vide    Pile2 : Vide
Le mot aabbcc est accepte!
```

FIGURE 17 – Demo : l'exécution pour Dpile.txt et un mot accepté dans mode debug

```
henri@DESKTOP-DS7DCHE:/mnt/e/CCC/PLT_projet_codes_final$ ./executeur -debug VM
Donner le nom de fichier d'entree : Dpile
Automate(2)={etats=[`0`,`1`,`2`]initial=0final=[2]transitions=[(0→0,`a`,(→,`a`)),(0→1,`b`,(`a`,→),(→,`b`)),(1→1,`b`,(`a`,
→),(→,`b`)),(1→2,`c`,(),(`b`,→)),(2→2,`c`,(),(`b`,→))]}
-----------------------------------------------------------------------------------------------------------------------
L'analyse lexicale est finie!
-----------------------------------------------------------------------------------------------------------------------
L'analyse syntaxique est finie!
nombre de pile: 2
nombre d'etats: 3
nombre de final: 1
nombre de transitions: 5
etats: 0 1 2
initial: 0
final: 2
transitions: 0 0 | 0 1 | 1 1 | 1 2 | 2 2 |
caractere: a | b | b | c | c |
pile1: + a | - a | - a |     |     |
pile2:     | + b | + b | - b | - b |
-----------------------------------------------------------------------------------------------------------------------
L'analyse semantique est finie!
tableau de symbole:  0 4 | 1 17 | 2 30 |
machine virtuelle: 2 4 1 30 2 97 4 97 1 0 0 98 17 97 -1 98 1 2 98 17 97 -1 98 1 99 30 0 0 98 -1 1 99 30 0 0 98 -1
Donner le mot d'entree : aabbbcc
  -> Etat : 0   Pile1 : Vide    Pile2 : Vide
a -> Etat : 0   Pile1 : a       Pile2 : Vide
a -> Etat : 0   Pile1 : aa      Pile2 : Vide
b -> Etat : 1   Pile1 : a       Pile2 : b
b -> Etat : 1   Pile1 : Vide    Pile2 : bb
b -> Erreur : Pile1 vide!
Le mot aabbbcc est refuse! Pile2 non vide.
```

FIGURE 18 – Demo : l'exécution pour Dpile.txt et un mot refusé dans mode debug

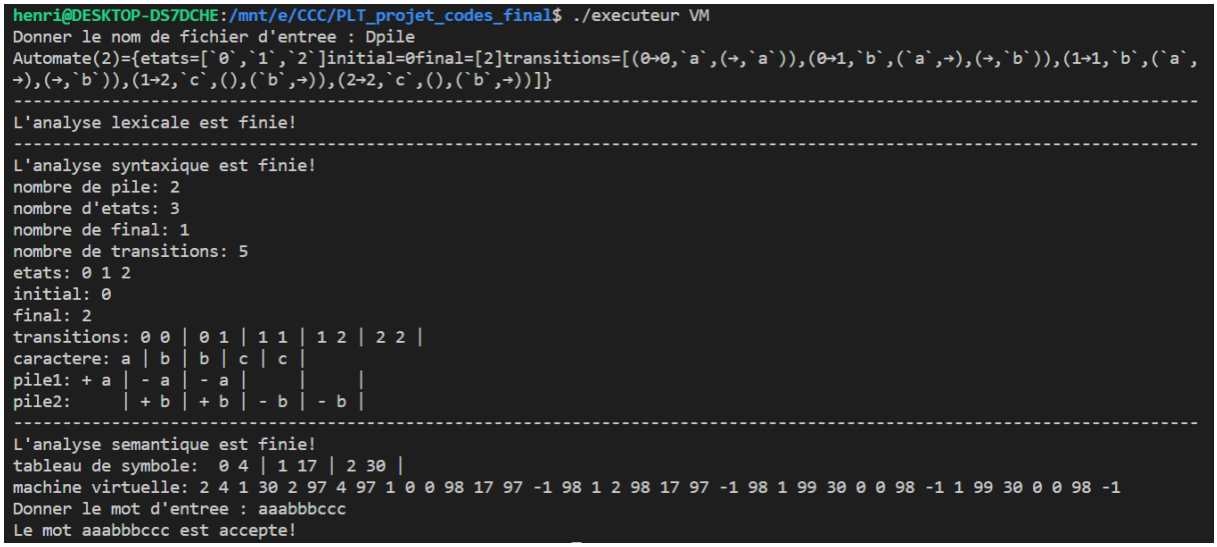


FIGURE 19 – Demo : l'exécution pour Dpile.txt dans mode normal