Rapport : Compilateur d'automate

Part 1: Introduction

Dans ce projet, l'objectif est claire: Écrire un ***compilateur d'automate*** qui peut atteindre les fonctions suivantes:

- Tout d'abord, il lit un fichier contenant la description de l'automate.
- Ensuit, il produit un code capable de reconnaître les mots qui lui sera fourni en machine virtuelle.
- Enfin, il exécute les machines virtuelles écrites.

Afin de réaliser ces fonctions et bien tracer le comportement de l'automate (mode debug) , j'ai effectué des analyses dans les cinq parties suivantes.

1. **Analyse lexicale**
2. **Analyse syntaxique**
3. **Analyse sémantique**
4. **Compilation**
5. **Exécution**.

Partie 2: Analyse lexicale

Méthode d'extraction des lexèmes

Après avoir obtenu un fichier txt d'automate, je dois séparer tous les éléments en lexèmes. je dois donc d'abord définir des différents types de lexèmes. Ici, j'ai divisé les six types suivants:

```
//utilser enum pour numeroser les different types
typedef enum lexeme_t
{
    Mot_clef, // Il comtient 5 mots: Automate etats initial final transitions
    Symbole, // [=()[]{}]+ Tous les symboles sauf les guillemets simples, les
guillemets doubles et les symboles de commentaire.
    Entier, // [Entiers pas entre guillemets
    Chaine, // les caractères simples entre guillemets simples
    Caractere, // les caractères entre guillemets
    Fleche // "→"
} lexeme_t;
```

En parallèle, j'ai construit les structures suivantes pour représenter les différentes caractéristiques du lexème:

- Structure de type et expression de chaque lexème:

  ```
  //structure de type et expression de chaque lexeme
  typedef struct lexeme
  {
      lexeme_t type;
      char * expression;
  }lexeme;
  ```

- Structure d'adresse et contenu(type et expression) de chaque membre:

```
//structure d'adress et contenu(type et expression) de chaque member
typedef struct indice
{
    lexeme* content;
    int* address;
    // int* tail;
}indice;
```

- Structure d'une liste des tous les lexèmes:

```
//structure d'une liste des tous les expressions
typedef struct mot_liste
{
    int length;
    char * liste;
}mot_liste;
```

Avant de faire correspondre le lexème, je dois supprimer les informations "inutiles" dans le texte, c'est-à-dire le commentaire correspondant. Grâce à l'observation, j'ai constaté qu'il existe deux types de commentaires, qui sont marqués par '//' sur une ligne et marqués par '/* */' sur plusieurs lignes. je dois donc supprimer ces deux types de commentaires séparément. Ici, j'ai utilisé la méthode du double pointeur pour supprimer tous les commentaires. L'exemple de code est le suivant :

```
while ((char_cur = fgetc(fp)) && char_cur!= -1){
        // Tout d'abord nous négligeons les commentaires sur une ligne(par '//')
        if (char_tmp == '/'){
            if (char_cur == '/'){
                while ((char_tmp = fgetc(fp)) &&  char_tmp!= -1){
                    if (char_tmp == '\n') break;
                }
                // A la fin du fichier
                if (char_tmp == EOF) {
                    liste_sans_commentaire->liste[count++] = '\n';
                    break;
                }
            }

            // Ensuite nous négligeons les commentaires sur plusieurs lignes(par
'/*...*/')
            else if (char_cur == '*'){
                while ((char_cur = fgetc(fp)) && char_cur != -1){
                    if (char_cur =='/' && char_tmp == '*'){
                        char_tmp = '\n';
                        break;
                    }
                    //mis à jour char_tmp
                    else {
                        char_tmp = char_cur;
                    }
                }
            }
            else{
                 liste_sans_commentaire->liste[count++] = (char)char_tmp;
                 //mis à jour char_tmp
```

```
                char_tmp = char_cur;
            }
        }
        // Pas commentaires
        else{
                liste_sans_commentaire->liste[count++] = (char)char_tmp;
                //mis à jour char_tmp
                char_tmp = char_cur;
            }
    }
```

En même temps, dans le processus de suppression des commentaires, je lis chaque lexème du fichier.txt dans `liste_sans_commentaire`

```
liste_sans_commentaire->liste[count++] = (char)char_tmp;
liste_sans_commentaire->length = count;
```

Après avoir supprimé les commentaires et lu tous les lexèmes, j'ai commencé à réfléchir à la manière de faire correspondre les lexèmes avec leurs types correspondants. Mon idée initiale était de trouver le type correspondant de chaque lexème grâce à une traversée de boucle. Après avoir essayé, il s'est avéré être un peu encombrant. Après avoir revu les leçons enseignées par le professeur, j'ai soudain réalisé que les expressions régulières peuvent être utilisées pour résoudre ce problème:

```
regex_t myregex_mot_clef;
regex_t myregex_symbole;
regex_t myregex_entier;
regex_t myregex_chaine;
regex_t myregex_caractere;
regex_t myregex_fleche;

int num; //Le nombre de lexemes dans le texte

//les expressions régulières
char* mode_mot_clef = "(Automate|etats|initial|final|transition)";
char* mode_entier = "[0-9]+";

char* mode_chaine = "\"[^\".]+\"";
char* mode_caractere = "`[^`.]+`";
char* mode_fleche = "[→]";
```

Ensuite, j'ai utilisé `regular_matching` cette fonction pour obtenir une correspondance d'expression régulière. Au cours du processus, j'ai découvert qu'il y avait un problème, c'est-à-dire que les trois types de parenthèses, les signes égaux et les virgules ne correspondaient pas. Afin de résoudre ce problème, je utilise `estSymbole` cette fonction pour faire correspondre de tels symboles d'une manière parcourant une boucle. La mise en œuvre de fonctions spécifiques peut se référer au code.

Ensuite, j'ai complété l'appariement de tous les lexèmes:

```
// Faire correspondre le lexème par type
    regcomp(&myregex_fleche, mode_fleche, REG_EXTENDED);
    regular_matching(liste_sans_commentaire, myregex_fleche, Fleche, member);

    regcomp(&myregex_mot_clef, mode_mot_clef, REG_EXTENDED);
```

```
    regular_matching(liste_sans_commentaire, myregex_mot_clef, Mot_clef,
member);

    regcomp(&myregex_entier, mode_entier, REG_EXTENDED);
    regular_matching(liste_sans_commentaire, myregex_entier, Entier, member);

    regcomp(&myregex_chaine, mode_chaine, REG_EXTENDED);
    regular_matching(liste_sans_commentaire, myregex_chaine, Chaine, member);

    regcomp(&myregex_caractere, mode_caractere, REG_EXTENDED);
    regular_matching(liste_sans_commentaire, myregex_caractere, Caractere,
member);

    estSymbole(liste_sans_commentaire, member);
```

Après avoir terminé la correspondance et le stockage de tous les lexèmes, j'envisage de sortir les résultats. Je n'ai pas suivi l'ordre de chaque lexème dans le fichier txt dans le processus de correspondance et de stockage, mais je connais l'adresse de chaque lexème. Je complète donc la sortie de chaque lexème dans l'ordre en triant les adresses de la plus petite à la plus grande. Ici, je choisis d'utiliser le tri par sélection.

Après avoir écrit le fichier `Analyse_lexicale.c`, j'ai écrit un fichier de test `test_AL.c` pour tester les résultats de sortie des trois fichiers txt. En fin de compte, j'ai obtenue la résultat correcte.

Résultat du test

Les résultats du test de code sont les suivants:

Pour Zpile.txt:

```
hetianmo@hetianmo-VirtualBox:~/plt/projet$ gcc test_AL.c -o test_AL
hetianmo@hetianmo-VirtualBox:~/plt/projet$ ./test_AL
Ficher Nom :
Zpile.txt
Il n'exist pas des errors!

1: type: Mot_clef, l'expression: Automate
2: type: Symbole, l'expression: (
3: type: Entier, l'expression: 0
4: type: Symbole, l'expression: )
5: type: Symbole, l'expression: =
6: type: Symbole, l'expression: {
7: type: Mot_clef, l'expression: etats
8: type: Symbole, l'expression: =
9: type: Symbole, l'expression: [
10: type: Chaine, l'expression: "1"
11: type: Symbole, l'expression: ,
12: type: Chaine, l'expression: "2"
13: type: Symbole, l'expression: ,
14: type: Chaine, l'expression: "3"
15: type: Symbole, l'expression: ,
16: type: Chaine, l'expression: "Init"
17: type: Symbole, l'expression: ]
18: type: Mot_clef, l'expression: initial
19: type: Symbole, l'expression: =
20: type: Entier, l'expression: 3
21: type: Mot_clef, l'expression: final
22: type: Symbole, l'expression: =
23: type: Symbole, l'expression: [
24: type: Entier, l'expression: 0
25: type: Symbole, l'expression: ,
26: type: Entier, l'expression: 1
27: type: Symbole, l'expression: ,
28: type: Entier, l'expression: 2
29: type: Symbole, l'expression: ]
30: type: Mot_clef, l'expression: transition
31: type: Symbole, l'expression: =
32: type: Symbole, l'expression: [
33: type: Symbole, l'expression: (
34: type: Entier, l'expression: 3
35: type: Flech, l'expression: →
36: type: Entier, l'expression: 0
37: type: Symbole, l'expression: ,
38: type: Caractere, l'expression: `0`
39: type: Symbole, l'expression: )
40: type: Symbole, l'expression: ,
41: type: Symbole, l'expression: (
42: type: Entier, l'expression: 3
```

Figure 1: résultats des test de Zpile.txt (partiel)

Pour Upile.txt:

```
hetianmo@hetianmo-VirtualBox:~/plt/projet$ ./test_AL
Ficher Nom :
Upile.txt
Il n'exist pas des errors!

1: type: Mot_clef, l'expression: Automate
2: type: Symbole, l'expression: (
3: type: Entier, l'expression: 1
4: type: Symbole, l'expression: )
5: type: Symbole, l'expression: =
6: type: Symbole, l'expression: {
7: type: Mot_clef, l'expression: etats
8: type: Symbole, l'expression: =
9: type: Symbole, l'expression: [
10: type: Chaine, l'expression: "一"
11: type: Symbole, l'expression: ,
12: type: Chaine, l'expression: "二"
13: type: Symbole, l'expression: ,
14: type: Chaine, l'expression: "三"
15: type: Symbole, l'expression: ]
16: type: Mot_clef, l'expression: initial
17: type: Symbole, l'expression: =
18: type: Entier, l'expression: 0
19: type: Mot_clef, l'expression: final
20: type: Symbole, l'expression: =
21: type: Symbole, l'expression: [
22: type: Entier, l'expression: 1
23: type: Symbole, l'expression: ]
24: type: Mot_clef, l'expression: transition
25: type: Symbole, l'expression: =
26: type: Symbole, l'expression: [
27: type: Symbole, l'expression: (
28: type: Entier, l'expression: 0
29: type: Flech, l'expression: →
30: type: Entier, l'expression: 0
31: type: Symbole, l'expression: ,
32: type: Caractere, l'expression: `a`
33: type: Symbole, l'expression: ,
34: type: Symbole, l'expression: (
35: type: Flech, l'expression: →
36: type: Symbole, l'expression: ,
37: type: Caractere, l'expression: `a`
38: type: Symbole, l'expression: )
39: type: Symbole, l'expression: )
40: type: Symbole, l'expression: ,
41: type: Symbole, l'expression: (
42: type: Entier, l'expression: 0
43: type: Flech, l'expression: →
```

Figure 2: résultats des test de Upile.txt (partiel)

Pour Dpile.txt:

```
hetianmo@hetianmo-VirtualBox:~/plt/projet$ ./test_AL
Ficher Nom :
Dpile.txt
Il n'exist pas des errors!

1: type: Mot_clef, l'expression: Automate
2: type: Symbole, l'expression: (
3: type: Entier, l'expression: 2
4: type: Symbole, l'expression: )
5: type: Symbole, l'expression: =
6: type: Symbole, l'expression: {
7: type: Mot_clef, l'expression: etats
8: type: Symbole, l'expression: =
9: type: Symbole, l'expression: [
10: type: Caractere, l'expression: `0`
11: type: Symbole, l'expression: ,
12: type: Caractere, l'expression: `1`
13: type: Symbole, l'expression: ,
14: type: Caractere, l'expression: `2`
15: type: Symbole, l'expression: ]
16: type: Mot_clef, l'expression: initial
17: type: Symbole, l'expression: =
18: type: Entier, l'expression: 0
19: type: Mot_clef, l'expression: final
20: type: Symbole, l'expression: =
21: type: Symbole, l'expression: [
22: type: Entier, l'expression: 2
23: type: Symbole, l'expression: ]
24: type: Mot_clef, l'expression: transition
25: type: Symbole, l'expression: =
26: type: Symbole, l'expression: [
27: type: Symbole, l'expression: (
28: type: Entier, l'expression: 0
29: type: Flech, l'expression: →
30: type: Entier, l'expression: 0
31: type: Symbole, l'expression: ,
32: type: Caractere, l'expression: `a`
33: type: Symbole, l'expression: ,
34: type: Symbole, l'expression: (
35: type: Flech, l'expression: →
36: type: Symbole, l'expression: ,
37: type: Caractere, l'expression: `a`
38: type: Symbole, l'expression: )
39: type: Symbole, l'expression: )
40: type: Symbole, l'expression: ,
41: type: Symbole, l'expression: (
42: type: Entier, l'expression: 0
```

Partie 3: Analyse syntaxique

Méthode de réalisation

Dans cette partie, je dois vérifier que le résultat obtenu après analyse lexicale est syntaxiquement correct. Par exemple, je dois juger si la position des crochets et des signes égal est correcte. En même temps je dois aussi vérifier que les opérateurs ont le bon nombres d'arguments. Dans ce processus, j'enregistre les informations clés de l'automate:

- le nombre de pile
- le nombre d'états , inclus initial et final
- les états
- l'information de chaque transition (par exemple l'info pour empiler et dépiler)

Mon idée générale est de parcourir les résultats de la première partie du lexique, et de déterminer quelle opération doit être effectuée pour chaque lexème qui est bouclé. De cette façon, la collecte d'informations sur tous les mots-clés peut être réalisée.

Tout d'abord, j'ai créé deux structures pour représenter toutes les informations dont nous avons besoin pour créer un automate. La deuxième structure représente les informations privées de chaque transition:

```
typedef struct trans_info
{
    int source; // la source de lexeme dans une trasition
    int destination; // la destination de lexeme dans une transition
    char emission; // l'emission du lexeme
    int em_ou_de[10]; // empiler:1 depiler: -1
    char pile[10]; // la pile contient des lexemes qui empilent

} trans_info;

typedef struct syntax_info
{
    char etats[20][20];
    int etat_nombre; // l'automate avoir plusieurs etats
    int initial; // il existe seulement un etat initial
    int final[15]; // l'automate peut avoir plusieurs etats final
    int final_nombre;
    int trans_info_nombre; // le nomber de transitions
    int pile_nombre; // nombre de pile
    trans_info* trans[50];
} syntax_info;
```

Ensuite, j'ai crée une pile afin de pouvoir juger si les parenthèses sont syntaxiquement correctes. Dans le même temps, je dois également simuler le processus d'empilement et d'éclatement du lexème au cours de chaque transition. Ici, j'utilise directement les fonctions de création de pile et de empiler et dépiler de la pile dont le professeur a parlé en classe

```
typedef struct pile
{
    char (*pop)(void);
    void (*push)(char);
    int (*vide)(void);
} pile;
pile Mapile;
```

```
char pop_aux(void)
{
    if (hauteur == 0)
    {
        printf("Pile vide !\n");
        exit(1);
    }
    else
        return(data[--hauteur]);
};
void push_aux(char c)
{
    if (hauteur == 50)
    {
        printf("Pile pleine !\n");
        exit(2);
    }
    else
        data[hauteur++] = c;
};
int vide_aux(void)
{
    return(hauteur == 0);
};
```

Ensuite, je définis `lexeme_address` ce paramètre et le mets à jour constamment pour obtenir une lecture continue du lexème en `res`. Chaque fois que je lis un mot-clé, j'analyse les informations de chaque mot-clé. Ici, j'ai construit 5 fonctions pour analyser les cinq mots-clés séparément. Par exemple la fonction ci-dessous:

```
// analyser mot clef "initial"
void analyse_motclef_initial(lexeme* res)
{
        // examine '='
    lexeme_address += 1;
    is_egale(1, res);

    // noter l'info d'initial
    lexeme_address += 1;
    syntax_res->initial = is_entier(res);
    lexeme_address += 1;
    return;
}
```

Une fois qu'il y a une erreur de syntaxe, j'utilise une fonction `is_error` pour signaler l'erreur.

De plus, j'ai également construit des fonctions pour juger des symboles, et j'ai appelé ces fonctions dans la fonction d'analyse.

Par exemple:

```
// Si c'est un caractere
char is_caractere(lexeme* res)
{


    if (res[lexeme_address].type != Caractere)
        is_error(res);

    return res[lexeme_address].expression[1];

}
```

Le jugement de transition est le plus compliqué et le plus critique. Parce que chaque transition peut avoir plusieurs informations empiler et dépiler, ou elle peut ne pas utiliser la pile. J'ai donc construit une fonction `search_trans_info` pour compléter la collecte de chaque information de transition.

Enfin, après avoir collecté et stocké les informations de chaque mot-clé, je sors les informations et j'obtiens le résultat.

Problèmes rencontrés

Dan ce processus j'ai rencontré principalement trois problèmes:

1. Tout d'abord, comment dois-je stocker toutes les informations sur les mots clés? Au début, j'ai essayé d'utiliser une liste chaînée ou un tas pour stocker des informations, mais j'ai rencontré beaucoup d'erreurs lors de la compilation et la mise en œuvre était lourde. Enfin, après avoir discuté avec mes camarades de classe, j'ai décidé de construire deux structures pour représenter clairement le stockage de chaque information.
2. Initialement, j'ai complété la référence du paramètre `lexeme_address` en passant le paramètre dans la fonction, mais dans le processus, la valeur du paramètre n'est pas mise à jour, ce qui peut être dû au passage de la valeur. Pour résoudre ce problème, j'ai choisi de modifier la variable globale directement dans chaque fonction pour mettre à jour la valeur du paramètre.
3. À l'origine, je n'ai pas passé le paramètre `res` en tant que paramètre formel dans chaque fonction, mais une erreur de **segmentation fault** s'est produite lors de la compilation finale. C'est ce qui me dérange le plus. Enfin, après avoir essayé plusieurs fois, j'ai découvert que je devais passer des paramètres dans chaque fonction. Je pense que la raison possible est que s'il n'est pas transmis, l'accès au tableau sera hors limites.

Résultat du test

Exécution de l'instruction:

```
hetianmo@hetianmo-VirtualBox:~/plt/projet$ gcc test_ASy.c -o test_ASy
hetianmo@hetianmo-VirtualBox:~/plt/projet$ ./test_ASy
```

Figure 4: Exécution de l'instruction

Pour Zpile.txt:

```
-----Syntax Resultat-----
pile_nombre = 0
etat_nombre = 4
états = "1" "2" "3" "Init"
initial = 3
final_nombre = 3
final états = 0 1 2

Transition 1:
source = 3
destination = 0
emission = 0


Transition 2:
source = 3
destination = 1
emission = 1


Transition 3:
source = 3
destination = 2
emission = 2


Transition 4:
source = 0
destination = 1
emission = 1


Transition 5:
source = 0
destination = 2
emission = 2


Transition 6:
source = 1
destination = 0
emission = 0


Transition 7:
source = 1
destination = 2
emission = 2
```

Figure 5: résultats des test de Zpile.txt (partiel)

Pour Upile.txt:

```
-----Syntax Resultat-----
pile_nombre = 1
etat_nombre = 3
états = "一" "二" "三"
initial = 0
final_nombre = 1
final états = 1

Transition 1:
source = 0
destination = 0
emission = a
Pile info: 1 : a

Transition 2:
source = 0
destination = 1
emission = b
Pile info: -1 : a

Transition 3:
source = 2
destination = 1
emission = b
Pile info: -1 : a

Transition 4:
source = 0
destination = 2
emission = c


Transition 5:
source = 2
destination = 2
emission = c


Transition 6:
source = 1
destination = 1
emission = b
Pile info: -1 : a
```

Figure 6: résultats des test de Upile.txt (partiel)

Pour Dpile.txt:

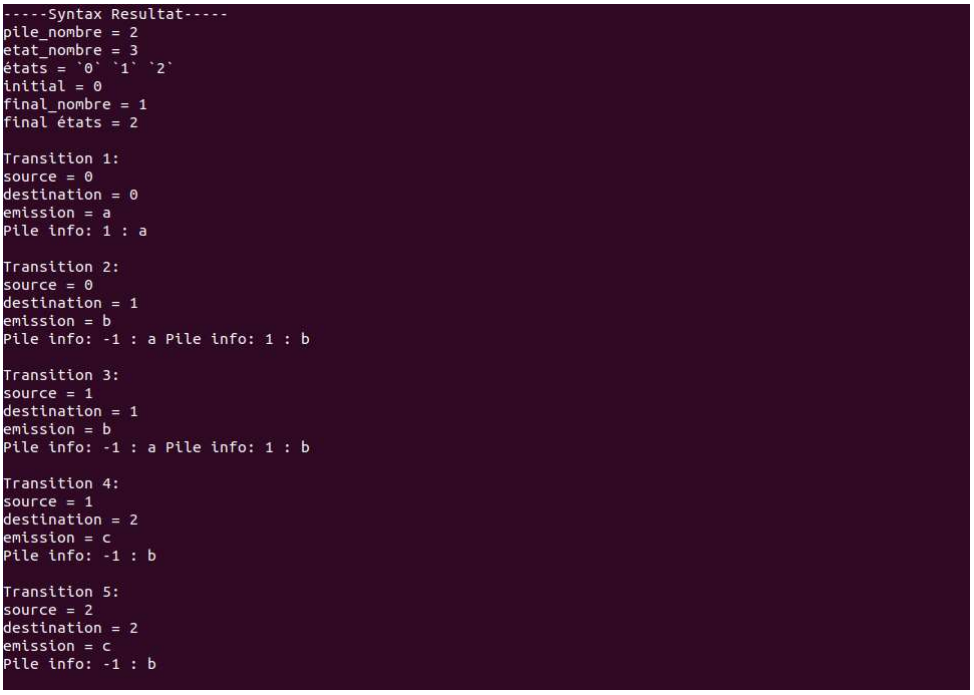


Figure 6: résultats des test de Dpile.txt (partiel)

Partie 4: Analyse Sémantique

Méthode de réalisation

Pour l'analyse sémantique , je fais quelques vérifications sur le résultat de l'analyse syntaxique:

- vérification que l'automate est bien déterministe ;
- vérification que les transitions ont des syntaxes corrects avec le nombre de piles ;
- vérification que les nombres des états dans les transitions sont corrects ;
- vérification que les nombres des piles sont corrects

Il faut noter ici que je n'ai pas vérifié si le type de chaque lexème est correct, **car le type de lexème a été vérifié à l'avance lors de l'analyse syntaxique !** Nous n'avons donc pas besoin de porter de vérification sur le type lors de l'analyse sémantique.

Par exemple, pour vérifier les nombres des piles sont corrects, j'ai construit une fonction `examine_pile_nombre()`:

```
// examiner le nombre des piles
void examine_pile_nombre()
{
    for (int j = 0; j < syntax_res->trans_info_nombre; j++)
    {
        int count = -1;
        for (int k = 0; k < 10; k++)
        {
            // noter les actions sur la pile
            if (syntax_res->trans[j]->em_ou_de[k] != 0)
            {
                //printf("\n%d", syntax_res->trans[j]->em_ou_de[k]);
                count++;
            }
        }
        //printf("\n%d", count);
        if (count > syntax_res->pile_nombre)
        {
```

```
            printf("\nSémantique erreur : le nombre de la pile de transition :
%d\n" , count);
            printf("le nombre de pile totals : %d\n", syntax_res->pile_nombre);
            printf("le nombre de la pile de transition > le nombre de pile
totals.\n");
            exit(1);
        }
    }
}
```

D'autres fonctions de vérification peuvent se référer au code complet. Parce que j'ai fait assez de travail dans l'analyse syntaxique, je n'ai pas de problèmes insolubles dans l'analyse sémantique.

Résultat du test

Ici, nous utilisons Upile.txt comme exemple pour tester.

Exécution de l'instruction:

```
hetianmo@hetianmo-VirtualBox:~/plt/projet$ gcc test_ASe.c -o test_ASe
hetianmo@hetianmo-VirtualBox:~/plt/projet$ ./test_ASe
```

Le résultat montre que le fichier n'a pas d'erreurs sémantiques:

```
D'après analyse syntaxique, il n'existe pas des erreurs sémantiques
hetianmo@hetianmo-VirtualBox:~/plt/projet$
```

Ensuite, nous modifions Upile.txt pour vérifier que notre fonction peut être réalisée.

- vérification que l'automate est bien déterministe

  J'ai changé le texte:

  ```
  transitions=[(0→ 0, `a`,(→, `a`)),
               (0 → 1,`a`, (`a`,→)),
  ```

  Le résultat:

  ```
  hetianmo@hetianmo-VirtualBox:~/plt/projet$ gcc test_ASe.c -o test_ASe
  hetianmo@hetianmo-VirtualBox:~/plt/projet$ ./test_ASe
  Ficher Nom :
  Upile.txt

  Semantique erreur : Non deterministe, voir les deux transitions:
  transition 1 : 0 -> 0, par a

  transition 2 : 0 -> 1, par a
  ```

- vérification que les transitions ont des syntaxes corrects avec le nombre de piles

  J'ai changé le texte:

  ```
  /* Cet automate a une pile, il permet de programmer toutes les grammaires BNF et
  de simuler la reconnaissance des langages algébriques*/
  Automate(0) ={
  /* Cet automate a une pile
  ```

  Le résultat:

```
hetianmo@hetianmo-VirtualBox:~/plt/projet$ gcc test_ASe.c -o test_ASe
hetianmo@hetianmo-VirtualBox:~/plt/projet$ ./test_ASe
Ficher Nom :
Upile.txt

Sémantique erreur : le value d'état 4 est plus grand que le nombre d'états 3
```

- vérification que les nombres des états dans les transitions sont corrects

  J'ai changé le texte:

```
        etats =["一","二","三"] // Les noms peuvent être différents de numéros
// Chaque état est repéré par son numéro dans la liste etats
        initial= 0
// final est une liste, même s'il n'y a qu'un état final
        final =[1, 2, 3, 4]
```

  Le résultat:

```
hetianmo@hetianmo-VirtualBox:~/plt/projet$ gcc test_ASe.c -o test_ASe
hetianmo@hetianmo-VirtualBox:~/plt/projet$ ./test_ASe
Ficher Nom :
Upile.txt

Sémantique erreur : le nombre d'état final 4 est plus grand que le nombre d'états 3
```

- vérification que les nombres des piles sont corrects:

  J'ai changé le texte:

```
 a^n.c^p.b^n, où n>0 */
        etats =["一","二","三"] // Les noms peuvent être différents de numéros
// Chaque état est repéré par son numéro dans la liste etats
        initial= 0
// final est une liste, même s'il n'y a qu'un état final
        final =[1]
        transitions=[(4→ 0, `a`,(→, `a`)),
```

  Le résultat:

```
hetianmo@hetianmo-VirtualBox:~/plt/projet$ gcc test_ASe.c -o test_ASe
hetianmo@hetianmo-VirtualBox:~/plt/projet$ ./test_ASe
Ficher Nom :
Upile.txt

Semantique erreur : le nombre de la pile de transition : 1
le nombre de pile totals : 0
le nombre de la pile de transition > le nombre de pile totals.
```

  Les résultats des tests montrent que nous pouvons détecter toutes nos erreurs sémantiques spécifiées. Nous avons donc terminé le travail d'analyse sémantique.

Partie 5: Compilation

Méthode de réalisation

Dans la partie compilation, je souhaite envisager de stocker les résultats obtenus dans les trois premières parties dans deux fichiers, le fichier TS.txt et le fichier VM. Les règles de stockage des informations de chaque état dans la VM ont été expliquées en détail dans le rapport, et l'enseignant a également donné des exemples clairs en classe.

Je dois donc décider comment stocker l'adresse et les autres informations correspondant à chaque état du tableau `vm`. J'ai d'abord déterminé la longueur requise du tableau `vm` et le nombre de transitions de chaque état:

```
int vm[100];
```

```
int etat_vm_len[20];
int etat_trans_nom[20];
int etat_adresse[20];
int vm_length = 0;

void vm_trans(int cur_trans)
{
    int source = syntax_res->trans[cur_trans]->source;
    // Mettre à jour la longueur de chaque état (2 est longueur de source et
destination)
    etat_vm_len[source] += 2 * syntax_res->pile_nombre + 2;
    // Mettre à jour le nombre de transitions pour chaque état
    etat_trans_nom[source] += 1;
}
```

Ensuite, je le mappe à chaque état en itérant sur la source de chaque transition. Après que chaque transition correspond à état, je remplisse le tableau `vm` avec les informations de transition de chaque état à tour de rôle.

```
if (etat == temp->source)
        {
            vm[index] = temp->emission;
            index += 1;

            vm[index] = etat_adresse[temp->destination];
            index += 1;

            for (int i = 0; i < syntax_res->pile_nombre; i++)
            {
                vm[index] = temp->pile[i];
                index += 1;
                vm[index] = temp->em_ou_de[i];
                index += 1;

            }
        }
```

Enfin, j'imprime les résultats et les stocke dans deux fichiers, TS.txt et VM, respectivement.

```
    // stocker la table des symboles dans TS.txt
    FILE* ts = fopen("TS.txt", "w");
    for (int i = 0; i < syntax_res->etat_nombre; i++)
        fprintf(ts, "le Nom d'état = %s, l'adresse = %d\n", syntax_res-
>etats[i], etat_adresse[i]);
    fclose(ts);

    // Stocker vm
    FILE* vm_file = fopen("VM", "wb");
    fwrite(vm, sizeof(int), vm_length, vm_file);
    fclose(vm_file);
```

Problèmes rencontrés

Le principal problème que j'ai rencontré dans cette partie était que la transition n'était pas correctement mappée sur chaque état au début, mais était directement stockée de manière séquentielle. Cela entraîne une incompatibilité entre le contenu et l'adresse stockés par la machine virtuelle. J'ai résolu le problème après avoir demandé conseil à mes camarades de classe et obtenu le bon résultat.

Résultat du test

Exécution de l'instruction:

```
hetianmo@hetianmo-VirtualBox:~/plt/projet$ gcc test_compile.c -o test_compile
hetianmo@hetianmo-VirtualBox:~/plt/projet$ ./test_compile
```

Pour Zpile.txt:

```
Tableau VM:
0 21 3 6 11 16 2 49 11 50 16 2 48 6 50 16 2 48 6 49 11 3 48 6 49 11 50 16

le nombre de pile = 0
l'adresse initiale = 21
le nombre des états finals = 3
l'adresse des états finals = 6  11  16

état 0:
le nom d'état = "1", l'adresse de source = 6
le nombre de transition = 2
déclencher 1, l'adresse de destination = 11
déclencher 2, l'adresse de destination = 16

état 1:
le nom d'état = "2", l'adresse de source = 11
le nombre de transition = 2
déclencher 0, l'adresse de destination = 6
déclencher 2, l'adresse de destination = 16

état 2:
le nom d'état = "3", l'adresse de source = 16
le nombre de transition = 2
déclencher 0, l'adresse de destination = 6
déclencher 1, l'adresse de destination = 11

état 3:
le nom d'état = "Init", l'adresse de source = 21
le nombre de transition = 3
déclencher 0, l'adresse de destination = 6
déclencher 1, l'adresse de destination = 11
déclencher 2, l'adresse de destination = 16
```

Pour Upile.txt:

```
Tableau VM:
1 4 1 17 3 97 4 97 1 98 17 97 -1 99 22 0 0 1 98 17 97 -1 2 98 17 97 -1 99 22 0 0

le nombre de pile = 1
l'adresse initiale = 4
le nombre des états finals = 1
l'adresse des états finals = 17

état 0:
le nom d'état = "一", l'adresse de source = 4
le nombre de transition = 3
déclencher a, l'adresse de destination = 4  Pile 1 : empiler a
déclencher b, l'adresse de destination = 17  Pile 1 : dépiler a
déclencher c, l'adresse de destination = 22

état 1:
le nom d'état = "二", l'adresse de source = 17
le nombre de transition = 1
déclencher b, l'adresse de destination = 17  Pile 1 : dépiler a

état 2:
le nom d'état = "三", l'adresse de source = 22
le nombre de transition = 2
déclencher b, l'adresse de destination = 17  Pile 1 : dépiler a
déclencher c, l'adresse de destination = 22
```

Pour Dpile.txt:

```
Tableau VM:
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

le nombre de pile = 2
l'adresse initiale = 4
le nombre des états finals = 1
l'adresse des états finals = 30

état 0:
le nom d'état = `0`, l'adresse de source = 4
le nombre de transition = 2
déclencher a, l'adresse de destination = 4  Pile 1 : empiler a
déclencher b, l'adresse de destination = 17  Pile 1 : dépiler a   Pile 2 : empiler b

état 1:
le nom d'état = `1`, l'adresse de source = 17
le nombre de transition = 2
déclencher b, l'adresse de destination = 17  Pile 1 : dépiler a   Pile 2 : empiler b
déclencher c, l'adresse de destination = 30  Pile 2 : dépiler b

état 2:
le nom d'état = `2`, l'adresse de source = 30
le nombre de transition = 1
déclencher c, l'adresse de destination = 30  Pile 2 : dépiler b
```

Après vérification, j'ai trouvé que tous les résultats des tests étaient corrects. Jusqu'à présent, j'ai terminé le travail de compilation d'automate.

Partie 6: Exécution

Méthode de réalisation

La dernière partie de l'exécution consiste à entrer une chaîne de chaînes dans le compilateur que nous avons écrit. Ce processus nécessite l'utilisation de deux fichiers que j'ai générés dans la quatrième partie, à savoir le fichier TS.txt et le fichier VM.

1. Tout d'abord, je dois juger la commande d'entrée. J'ai appris de projet qu'il existe deux modes d'exécution, **./runtime VM** et **./runtime debug VM**, où **./runtime debug VM** montre l'ensemble du processus de debug. J'ai défini deux modes dans main fonction pour l'exécution.

```
    if (argc == 2)
    {
        // non debug mode
        debug = 0;
        if (strcmp(argv[1], "VM") != 0) {
            printf("Erreur dans le paramètre d'entrée!\n");
            exit(2);
        }
    }
    else if (argc == 3)
    {
        // debug mode
        debug = 1;
        if (strcmp(argv[1], "-debug") != 0 || strcmp(argv[2], "VM") != 0) {
            printf("Erreur dans le paramètre d'entrée!\n");
            exit(2);
        }
    }
```

2. Ensuite, nous utilisons la fonction `fscanf` pour extraire les informations sur l'automatisation qui ont été obtenues à partir des fichiers VM et TS.txt.

```
void analyse_vm()
{
    FILE* vm_fichier = fopen("VM", "r");
```

```
    vm_length = 0;
    size_t len = 10;
    while (len == 10) {
        len = fread(vm + vm_length, sizeof(int), 10, vm_fichier);
        vm_length += len;
    }
    fclose(vm_fichier);

    pile_nombre = vm[0];
    initial_etat_adresse = vm[1];

    final_nombre = vm[2];

    for (int i = 0; i < final_nombre; i++)
    {
        final_etat_adresse[i] = vm[i + 3];
    }

}
```

3. Je stocke ces informations dans une nouvelle structure définie. Dans le même temps, nous perfectionnerons la pile créée lors de l'analyse syntaxique, pour indiquer clairement la position de chaque élément dans la pile.

```
// Définir la structure état pour stocker les informations extraites des
fichiers VM et TS.txt
typedef struct etat_info
{
    int etat_adresse[10];
    char etat_nom[20][20];
} etat_info;

// Definir la nouvelle pile pour realiser les operations sur different
adresses des elements des piles
typedef struct pile
{
    void (*push)(int*, char*, char);
    char (*pop)(int*, char*);
    int (*vide)(int);
    int hauteur;
    char data[100];
} pile;
```

4. Ensuite, nous commençons à faire le cycle de traverser chaque caractère de la chaîne d'entrée. Pour chaque caractère, nous déterminons s'il y a une transition appropriée pour la correspondre et que la commande dans la pile est correcte au cours de ce processus. Lorsqu'un caractère est jugé, nous mettons à jour l'adresse de la machine virtuelle pour effectuer le caractère suivant. Une fois que tous les caractères sont complètement traversés, nous vérifions que la pile est vide. S'il est vide, cette chaîne est refusé, si la pile est vide, indiquant que cette chaîne est acceptée. Un processus de mise en œuvre détaillé peut faire référence au code et aux commentaires :

```
// Si la pile est vide finalement, le mot est accepté!
    if (is_Vide() == true)
        printf("Le mot %s est accepté ! \n", s);
    else {
        printf("Le mot %s est refusé ! ", s);
        printf("Piles non vides \n");
    }
```

Problèmes rencontrés

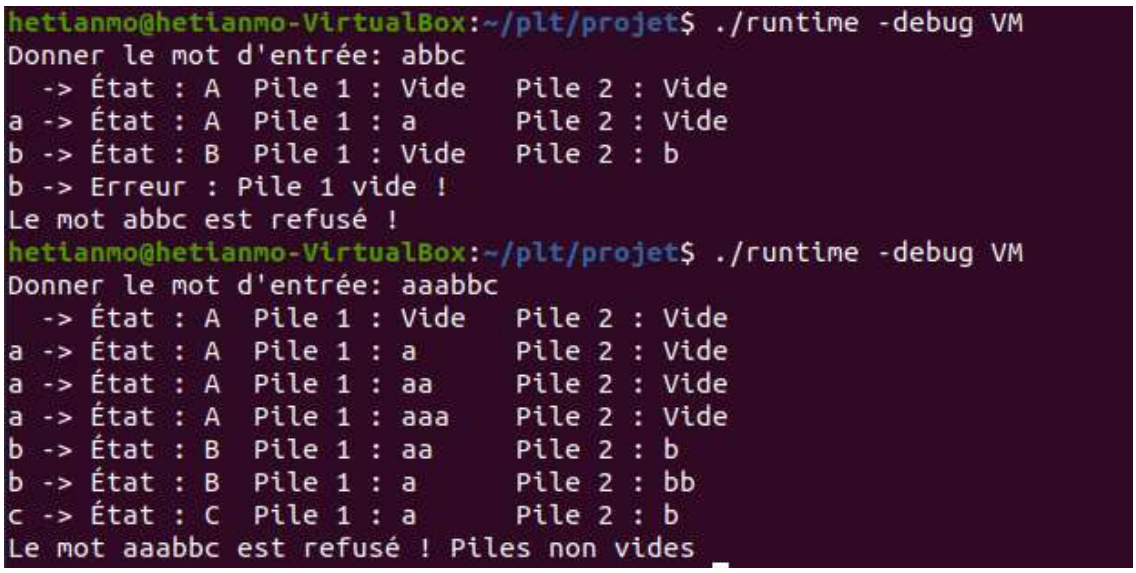
1. Lorsque vous rencontrez la première question est lorsque je lis le fichier DPILE.TXT. Les guillemets simples dans le fichier donné ne peuvent pas correspondre, nous changeons donc les guillemets simples en guillemets doubles pour être cohérents avec les autres fichiers:

```
4 /* Cet automate a deux piles
5 // Il correspond à l'automate reconnaissant le langage
6 // a^n.b^n.c^n où n>0 (cours 10, page 18) */
7         etats = ["A","B","C"]
8         // On pourrait mettre aussi [`a`, `b`,`c`]
9 // Chaque état est repéré par son numéro dans la liste etats
0         initial = 0
```

2. Initialement, j'ai utilisé la pile créée précédemment, mais je ne peux pas représenter l'emplacement des éléments de la pile, ce qui apporte beaucoup de problèmes, alors je compte encore et construire une pile.

Résultat du test

Voici utiliser le Dpile.txt comme exemple de test:



Les résultats des tests prouvent que notre compilateur réussit!

Partie 7 : Conclusion

Résumez notre travail, nous avons effectué une analyse lexicale, une analyse de la syntaxe, une analyse sémantique, une compilation du compilateur et une mise en œuvre finale, nous avons reçu les résultats corrects, réalisez pleinement les objectifs du projet.