République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université Aboubekr BELKAÏD - TLEMCEN
Faculté des Sciences
Département d'Informatique

TITRE

POLYCOPIÉ DE TRAVAUX PRATIQUES COMPILATION

Adressé aux étudiants niveau : Licences (L 3)
Ingénieur (ING 3)

Domaine : Compilation
Filière : Licence - Ingénieur

Etabli Par :

- Mr. MERZOUG Mohammed
- Mr. ETCHIALI Abdelhak

Année 2024-2025

AVANT-PROPOS

Ce polycopié de travaux pratiques, enrichi de rappels de cours, s'adresse aux étudiants de troisième année ingénieur et licence en informatique, ainsi qu'à toutes les filières dont le programme inclut le module de Compilation.

Il propose un ensemble structuré de rappels sur le langage C, en particulier sur la gestion des fichiers, l'utilisation des structures de données telles que les piles et les files, ainsi que des exercices pratiques portant sur la programmation des automates et l'analyse lexicale et syntaxique.

Les travaux pratiques ont pour objectif d'initier les étudiants aux différentes étapes du processus de compilation. Ils leur permettent de manipuler les notions fondamentales à travers des implémentations concrètes en langage C : gestion d'automates finis, conception d'analyseurs lexicaux, construction d'analyseurs syntaxiques, etc.

Ce support a pour ambition de fournir aux étudiants les outils nécessaires à la compréhension et à la mise en œuvre des mécanismes internes d'un compilateur, tout en consolidant leurs acquis en programmation.

Nous remercions chaleureusement les membres du laboratoire pédagogique d'informatique de la Faculté des Sciences pour leur engagement et leur soutien constant dans l'élaboration de ce document.

SOMMAIRE

****

****

TP N°1 : Rappel sur la programmation en C

1- OBJECTIF DU TP

- L'objectif de ce chapitre est de rappeler quelques notions fondamentales de la programmation en langage **C**. Pour écrire et tester vos programmes **C**, vous pouvez utiliser un éditeur ou un environnement de développement intégré (IDE) tel que **Code::Blocks**.
- Le langage **C** est un langage impératif et procédural largement utilisé pour la programmation système, le développement embarqué et l'enseignement. Un programme **C** se compose généralement d'un ou plusieurs fichiers source (**.c**), éventuellement de fichiers d'en-tête (**.h**), et doit être compilé pour produire un exécutable.

2- ÉTAPES POUR ECRIRE ET EXECUTER LE PREMIER PROGRAMME C AVEC CODE::BLOCKS

2.1- Installation :

- **Sous Windows :** télécharger Code::Blocks (version avec MinGW incluse si vous n'avez pas de compilateur) depuis le site officiel et installer.
- **Sous Linux :** installer codeblocks et build-essential (ou gcc) via le gestionnaire de paquets.

2.2- Créer un nouveau projet :

- Lancer Code::Blocks puis dans le menu, cliquer sur : Fichier → Nouveau → Projet → Choisir "Console application" → Langage C → Nom et emplacement du projet.

2.3- Ajouter un fichier source

- Dans l'onglet Projects, descendez jusqu'à la section Sources, puis ajoutez un nouveau fichier source (.c) ou modifiez le fichier ***main.c*** créé automatiquement.

2.4- Écrire le programme

- Taper le programme suivant :

```c
#include<stdio.h>

int main(void){
    int nombre;
    printf("\n Entrer Un Nombre Entier SVP:");
    scanf("%d",&nombre);
    printf("\n Vous Avez Tapez le Nombre Entier:%d \n",nombre);
    return 0;
}
```

2.5- Compiler et exécuter

- Cliquer sur « **Build** » ou « **Build and Run** ». L'IDE appellera le compilateur et affichera les erreurs/avertissements dans la console de **build**. S'il n'y a aucune erreur, le programme sera exécuté automatiquement et affichera le résultat à l'écran.

```
Entrer Un Nombre Entier SVP:-93

Vous Avez Tapez le Nombre Entier:-93

Process returned 0 (0x0)   execution time : 8.505 s
Press any key to continue.
```

Exercice 1 : Écrire un programme en langage C permettant de simuler une calculatrice. Le programme doit demander à l'utilisateur deux nombres entiers ainsi que le type d'opération à effectuer (A : Addition, S : Soustraction, M : Multiplication, D : Division), puis afficher le résultat de l'opération choisie.

Solution exercice 1

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

int main()
{
    int a,b;
    double result=0;
    char choix;
    printf ("\n\t****************** Calculatrice ************************\n");
    printf("\n\tA :Addition \n");
    printf("\n\tS :Soustraction \n");
    printf("\n\tM :Multiplication \n");
    printf("\n\tD : Division \n");
    printf("\n\t************************************************************\n");

    printf("\n\tNombre A = ");
    scanf("%d",&a);
    printf("\n\tNombre B = ");
    scanf("%d",&b);

    /***Cette fonction pour vider le buffer **/
        int K;
        do {
            K= getchar();
        } while (K != EOF && K != '\n');
    /*****************************************/
```

```
29
30      printf("\nEntez Votre Choix : ");
31      scanf("%c",&choix);
32
33      switch(choix)
34      {
35          case 'A':printf("\n\t%d + %d = %d\n",a,b,a+b);
36                   break;
37
38          case 'S': printf("\n\t%d - %d = %d\n",a,b,a-b);
39                    break;
40
41          case 'M':printf("\n\t%d * %d = %d\n",a,b,a*b);
42                   break;
43
44          case 'D': if (b!=0){
45                      result = a/b;
46                      printf("\n\t%d / %d = %.2lf\n",a,b,result);
47                    }
48                    else
49                      printf("\n\tImpossible de diviser sur un nombre null \n");
50                    break;
51          default : printf("\n\tErreur!!!\n");
52                    break;
53      }
54
55      return 0;
56  }
```

3- LES POINTEURS

- Un pointeur est une variable, il est destiné à contenir l'adresse d'une variable. Pour différencier un pointeur d'une variable ordinaire, on fait précéder son nom du signe '*****' lors de sa déclaration.
- La déclaration **Type *P** déclare un pointeur **P** qui peut recevoir des adresses de variables du type **Type**. L'opérateur **&** désigne l'adresse d'une variable : **&Prix** fournit l'adresse de la variable Prix.
- L'opérateur ***** désigne le contenu d'une adresse : ***adr** désigne le contenu de l'adresse référencée par le pointeur **adr**.

Exercice 2 : Ecrire une fonction échange permettant d'échanger les valeurs de deux nombres entiers a et b, appeler cette fonction dans un programme C pour échanger les valeurs de deux entiers saisis au clavier.

Réaliser deux versions de la fonction :

a- Une version utilisant le passage par valeur.

b- Une version utilisant le passage par adresse.

Solution exercice 2

a- Une version utilisant le passage par valeur :

```
void permuter_passage_valeur(int X, int Y){
    int Z;

    printf("\n\tAfichage Passage par Valeur  1******** A = %d      B = %d\n",X,Y);

    Z = X;
    X = Y;
    Y = Z;

    printf("\n\tAfichage Passage par Valeur  2******** A = %d      B = %d\n",X,Y);
}
```

- ***Appel de la fonction dans main :***

```
printf("\n\t====================PASSAGE PAR VALEUR======================\n\n");
printf("\n\t Afichage programme principal 1******** A = %d      B = %d\n",A,B);
permuter_passage_valeur(A, B);
printf("\n\tAfichage programme principal 2******** A = %d      B = %d\n",A,B);
printf("\n\t============================================================\n\n");
```

b- Une version utilisant le passage par adresse :

```
void permuter_passage_adresse(int *X, int *Y){
    int Z;

    printf("\n\tAfichage Passage par Adresse 1******** A = %d      B = %d\n",*X,*Y);

    Z = *X;
    *X = *Y;
    *Y = Z;

    printf("\n\tAfichage Passage par Adresse 2******** A = %d      B = %d\n",*X,*Y);
}
```

- ***Appel de la fonction dans main :***

```
printf("\n\t====================PASSAGE PAR ADRESSE=====================\n\n");
/**********  passage par Adresse *******/
printf("\n\tAfichage programme principal 3******** A = %d      B = %d\n",A,B);
permuter_passage_adresse(&A, &B);
printf("\n\tAfichage programme principal 4******** A = %d      B = %d\n",A,B);
printf("\n\t============================================================\n\n");
```

4- POINTEURS ET TABLEAUX :

- Le nom d'un tableau est considéré comme un pointeur sur son premier élément.
- Les écritures suivantes sont équivalentes :
    - `&tab[0]et tab;`
    - `tab[0] et *tab;`
    - `tab[i] et *(tab+1)`

Exercice 03 : Ecrire un programme C permettant de rechercher la plus grande valeur dans un tableau de dix nombres réels.

Solution exercice 3

```c
#include <stdio.h>
/**========================== LECTURE D UN TABLEAU ============================== **/
void Lire_Tab(double t[],int n){
    int i;
    for(i=0;i<n;i++){
        printf("Tab[%d] = ",i);
        scanf("%lf",&t[i]);
    }
}

/**========================== AFFICHAGE D UN TABLEAU ============================== **/
void Afficher_Tab(double t[],int n){
    int i;
    printf("\n");
    for(i=0;i<n;i++)
        printf(" %.2lf  ",t[i]);
    printf("\n");
}

/**========================== Maximum D UN TABLEAU ============================== **/
double Max_Val_Tab(double t[],int n){
    int j;
    double max=t[0];

    for(j=0;j<n;j++){
        if(max<t[j])
            max=t[j];
    }
    return max;
}
```

```c
/**========================== Programme Principal ============================== **/
int main()
{
    double tab[10],max;
    printf("\n\tEntrez les 10 valeurs réels du tableau:\n");
    Lire_Tab(tab,10);
    printf("\n\tLes Elements du tableau sont:\n");
    Afficher_Tab(tab,10);
    printf("la plus grand valeur est : %.2lf\n",Max_Val_Tab(tab,10));
    return 0;
}
```

5- LES CHAINES DE CARACTERES :

- En C, on représente les chaînes de caractères par un tableau de caractères, dont le dernier est un caractère de code nul (\0). Ainsi une chaîne composée de n éléments sera en fait un tableau de n+1 éléments de type char.
- Une constante caractère est identifiée par les guillemets " (double quotte).

Exemple :

```
char message[5]="mail" :
```
message est un tableau de 5 caractères (\0 compris).

```
puts (message);
```

- On peut également initialiser un pointeur avec une chaîne de caractères :

```
char *ptrch = "mail";
```

A-Fonctionsd'entrées/sortiespourleschaînes(stdio.h):

- `gets(char*) :` lecture d'une chaîne sur stdin.
- `puts(char*) : affiche`, sur stdout, la chaîne de caractères puis positionne le curseur en début de ligne suivante.

- Dans **conio.h,** on trouve les fonctions de base de gestion de la console :
  - `putch(char)`: Affiche sur l'écran (stdout) le caractère fourni en argument, cette fonction rend le caractère affiché ou EOF en cas d'erreur.
  - `getch(void)`: Attend le prochain appui sur le clavier, et rend le caractère qui a été saisi. L'appui sur une touche se fait sans écho.
  - `getche(void)`: Idem getch mais avec écho.
  - `getchar(void) :` fonctionne comme getche, mais utilise le même tampon que scanf.

B-Fonctionsutilesàlamanipulationdechaînes(string.h):

- `intstrlen(chaîne) :` donne la longueur de la chaîne (\0 non compris).
- `char *strcpy(char *dest,char *src) :` recopie la source dans la destination, rend un pointeur sur la destination.
- `char*strncpy(char*destination,char*source,intmax) :` idem que strcpy mais s'arrête au \0 ou au max caractères lus.
- `char*strcat(char*dest,char*src) :` concatène la source à la suite de la destination, rend un pointeur sur la destination.
- `char *strncat(char *destination,char *source,int longmax) :` idem que **strcat** mais s'arrête au \0 ou au max caractères lus.
- `int strcmp(char *str1,char*str2)`rend 0 si str1==str2, <0 si str1<str2, >0 si

str1>str2. Idem strncmp.

C-Fonctions de conversions entre scalaires et chaînes (`stdlib.h`) :

-`int atoi(char *s)` : traduit la chaîne en entier (s'arrête au premier caractère impossible, 0 si erreur dès le premier caractère) (voir aussi atol et atof).

D-Fonctions limitées aux caractères (`ctype.h`):

-`intisdigit(intc)` : rend un entier non nul si c'est un chiffre ('0' à '9'), 0 sinon.

- De même : `isalpha(intc)` : de A à Z et a à z, mais pas les accents.

-`isalnum(intc)` (`isalpha|isdigit`, `islower(minuscule)`, `isupper`,`isspace` (blanc, tab, return...) , `isxdigit` (0 à 9, A à F, a à f) ...

-`inttoupper` (int c) : rendAàZsicestaàz,rendcsinon.(voiraussi`tolower(intc)`);

E-Fonctionsdelagestiondynamiquedemémoire(`alloc.h`)

- `void *malloc(int taille)` : réserve une zone mémoire contiguë de taille octets, et retourne un pointeur sur le début du bloc réservé. Retourne le pointeur NULL en cas d'erreur (en général car pas assez de mémoire).

-`void*calloc (int nb,int taille)` : équivalent à `malloc(nb*taille)`.

-`void free (void *pointeur)` : libère la place réservée auparavant par `malloc` ou `calloc`. Pointeur est l'adresse retournée lors de l'allocation.

Exercice 04 :

Ecrire une procédure *Affiche_Chaine (char* chaine)* qui prend en paramètre une chaîne de caractères et l'affiche en insérant un espace entre chacun de ses caractères.

Ecrire la fonction main dans laquelle il faut lire au clavier une chaîne de caractères et l'affichez en utilisant A*ffiche_Chaine*.

Solution exercice 4

```
char* Affiche_Chaine_1(char* chaine){
    char tmp[10];
    char tmp2[10];
    char *espace=" ";
    int i=0,j=i+1,k=j+1, fin=strlen(chaine)*2, cpt=0;

    while(cpt<fin){
        strcpy(tmp,(chaine+j));
        strcpy(chaine+j,espace);
        strcpy(chaine+k, tmp);
        i+=2;
        j=i+1,k=j+1;
        cpt++;
    }
    return chaine;
}
```

```c
void Affiche_Chaine_2(char* chaine){
    int i,n=strlen(chaine);
    char chaine_avec_espaces[200];

    for(i=0;(i<n)||(chaine[i]!='\0');i++){
        chaine_avec_espaces[i*2]=chaine[i];
        chaine_avec_espaces[i*2+1]=' ';
    }
    chaine_avec_espaces[i*2]='\0';

    printf("\n\tChaine de Caractères avec Espaces 2 = %s\n", chaine_avec_espaces);
}
```

Exercice 05 :

Ecrire une fonction *Nombre_Voyelles (chaine)* permettant de retourner le nombre des lettres voyelles dans *chaine.*

Solution exercice 5

```c
int Char_Est_Voyelle(char c){
    if(c=='a' || c=='e' || c=='y' || c=='u' || c=='i' || c=='o') return 1;
    return 0;
}

int Nombre_Voyelles(char chaine[]){
    int i,n=strlen(chaine);
    int nbr_voyelle=0;
    for(i=0;(i<n)&&(chaine[i]!='\0');i++)
        if (Char_Est_Voyelle(tolower(chaine[i])))
            nbr_voyelle++;
    return nbr_voyelle;
}
```

Exercice 06 : VERIFICATION DE LOGIN & MOT DE PASSE

Après être employé à une boite de développement, votre chef vous demande de créer un programme de vérification du login / mot de passe. Ce programme sera utilisé seulement pour vérifier le *login / mot de passe* lors de la première création d'un compte dans un site web (comme Gmail, Yahoo, etc). Ainsi ce programme doit vérifier :

- Que le *login* commence par une lettre alphabétique ;
- Le *login* doit contenir que des caractères de nature alpha-numérique ;
- Le *mot de passe* doit contenir au moins un caractère majuscule ;
- La taille du *mot de passe* doit être supérieure à 10 ;
- *Le mot de passe doit être différent du login ;*

Par exemple :

- le login "*2med*" n'est pas valide car ce login commence par un chiffre, par contre "*med2*" est valide.

- Le mot de passe "*mdpkst2018*" n'est pas valide car celui-ci ne contient pas un caractère majuscule.

Question : Ecrire un programme qui permet de lire le login & mot de passe et de vérifier les conditions précédentes.

Solution exercice 6

```
int Verification_Login_Mot_de_Passe(char* login, char* mdp){

    // si le 1er caractere n'est pas alpaha, retourne false
    if (!isalpha(login[0])){
            printf("\n LOGIN INCORRECT : CARACTERE 1 NOT ALPHA ");
            return 0;
    }

    // si l'un des caractere n'est pas alphanum, retourne false
    int i ;
    int n= strlen(login);
    for(i = 0; i < n; i ++)
       if (!isalnum(login[i])){
            printf("\n LOGIN INCORRECT 2");
            return 0;
        }

    // si aucun caractere dans le mdp n'est majuscule, retourne false
    n=strlen(mdp);
    int nbr_majuscule=0;
    for( i = 0 ; i < n; i ++){
        if(isupper(mdp[i]))
            nbr_majuscule++;
    }
    if(nbr_majuscule==0){
        printf("\n MDP  INCORRECT 3");
        return 0;
    }

    // si la langueur du mdp est inferieur à 10, retourne false
    if(strlen(mdp)<10){
            printf("\n MDP INCORRECT 4");
            return 0;
    }

    // si mdp == login retourn false
    if(strcmp(mdp,login)== 0){
            printf("\n LOGIN MDP INCORRECT 5");
            return 0;
    }
    return 1;
}
```

TP N°2 : Rappel sur les Listes Chaînées – les Piles – les Files – les Fichiers

1- LES LISTES CHAINEES

1.1- *Structure d'une liste chaînée :*

```
typedef struct Mot_De_La_Phrase {
    char   Mon_mot[20];
    struct Mot_De_La_Phrase* Adresse_Mot_suivant;
} Mot;
```

1.2- Insertion au début de la liste chaînée :

```
Mot* Inserer_au_debut(Mot* les_mots_de_laphrase,char* mot){
        // creer un nouvel element
        Mot* premier_mot=malloc(sizeof(Mot));
        // copier la chaine de caractere
        strcpy(premier_mot->Mon_mot,mot);
        // affecter le pointeur du suivant au debut de la liste principale
        premier_mot->Adresse_Mot_suivant=les_mots_de_laphrase;
        // retourner l'@ du 1er element
        return premier_mot;
}
```

1.3- Insertion à la fin de la liste chaînée :

```
Mot* Inserer_a_la_fin(Mot* les_mots_de_laphrase, char* mot){
        // si la liste chainee est vdie
        if(les_mots_de_laphrase==NULL){
                // cration de la liste / reservation de memoire pour le 1er element
                les_mots_de_laphrase=malloc(sizeof(Mot));
                // c'est le 1er et dernier element (donc le prochain element est NULL)
                les_mots_de_laphrase->Adresse_Mot_suivant=NULL;
                // copier le mot
                strcpy(les_mots_de_laphrase->Mon_mot,mot);
                // retourner l'addresse du 1er element
                return les_mots_de_laphrase;
        }
        // si la liste chainee n'est pas vide affecter le 1er element a un pointeur temporaire
        Mot* dernier_mot=les_mots_de_laphrase;
        // tanque le pointeur suivant n'es pas NULL (tanqu'on est pas arrivé au dernier
        while(dernier_mot->Adresse_Mot_suivant!=NULL){
                // passer a l'element suivant
                dernier_mot=dernier_mot->Adresse_Mot_suivant;
        }
        // creer un nouvel element a la fin de la liste
        dernier_mot->Adresse_Mot_suivant=malloc(sizeof(Mot));
        // copier le mot
        strcpy(dernier_mot->Adresse_Mot_suivant->Mon_mot,mot);
        // c'est le 1er et dernier element (donc le prochain element est NULL)
        dernier_mot->Adresse_Mot_suivant->Adresse_Mot_suivant=NULL;
        // retourner l'addresse du 1er element
        return les_mots_de_laphrase;
}
```

2- LES PILES

1.1- *Structure d'une pile :*

```c
// Structure de la cellule pour stocker une ligne
typedef struct cellule {
    char ligne[500];
    struct cellule* next;
} Element;

// Structure de la pile
typedef struct pile {
    Element* sommet;
} Pile;

// Initialise une pile vide
Pile* initialiserPile(Pile* p) {
    p = malloc(sizeof(Pile));
    if (p == NULL) {
        fprintf(stderr, "Mémoire insuffisante\n");
        exit(1);
    }
    p->sommet = NULL;
    return p;
}
```

1.2- *Empiler une chaine de caractères :*

```c
// Empile une ligne dans la pile
Pile* Empiler(Pile* p, char* ligne) {
    Element* nv = malloc(sizeof(Element));
    if (nv == NULL) {
        fprintf(stderr, "Mémoire insuffisante\n");
        exit(1);
    }
    strncpy(nv->ligne, ligne, sizeof(nv->ligne));
    nv->next = p->sommet;
    p->sommet = nv;
    return p;
}
```

1.3- *Dépiler une chaine de caractères :*

```c
Pile* Depiler(Pile*p){
    if(p->sommet==NULL)
        return p;
    Element *e=p->sommet;
    p->sommet=p->sommet->next;
    free(e);
    return p;
}
```

1.4- Afficher le contenu d'une pile :

```c
// Affiche le contenu de la pile (ligne par ligne)
void AfficherPile(Pile* p) {
    Element* ep = p->sommet;
    while (ep != NULL) {
        printf("%s\n", ep->ligne);
        ep = ep->next;
    }
    printf("\n");
}
```

3- LES FILES

1.1- Structure d'une file :

```c
// Structure de la cellule pour stocker un mot
typedef struct cellule {
    char mot[100];
    struct cellule* next;
} Element;

// Structure de la file
typedef struct file {
    Element* premier;
    Element* dernier;
} File;

// Initialise une file vide
File* initialiserFile(File* f) {
    f = malloc(sizeof(File));
    if (f == NULL) {
        fprintf(stderr, "Mémoire insuffisante\n");
        exit(1);
    }
    f->premier = f->dernier = NULL;
    return f;
}
```

1.2- Enfiler une chaine de caractères :

```c
// Enfile un mot dans la file
File* Enfiler(File* f, char* mot) {
    Element* nv = malloc(sizeof(Element));
    if (nv == NULL) {
        fprintf(stderr, "Mémoire insuffisante\n");
        exit(1);
    }
    strncpy(nv->mot, mot, sizeof(nv->mot));
    nv->next = NULL;
    if (f->dernier == NULL) {
        f->premier = f->dernier = nv;
    } else {
        f->dernier->next = nv;
        f->dernier = nv;
    }
    return f;
}
```

1.3- Dépiler une chaine de caractères :

```
// Défile un mot de la file
File* Defiler(File* f) {
    if (f->premier == NULL) {
        fprintf(stderr, "La file est vide, impossible de défiler\n");
        return f;
    }
    Element* e = f->premier;
    f->premier = f->premier->next;
    free(e);
    if (f->premier == NULL) {
        f->dernier = NULL;
    }
    return f;
}
```

1.4- Afficher le contenu d'une file :

```
// Affiche le contenu de la file
void AfficherFile(File* f) {
    Element* ep = f->premier;
    while (ep != NULL) {
        printf("%s\n", ep->mot);
        ep = ep->next;
    }
    printf("\n");
}
```

4- LES FICHIERS

```
/** Declaration variable de type FILE **/
FILE *file1,*file2;

/** Declaration variables chaine de caractères **/
char chaine[100]  = {0};
char chaine1[100]  = {0};
char chaine2[100]  = {0};
char c;
char ligne[100]  = {0};

/** Declaration compteur de Type Entier **/
int i=0;
```

```c
/** 1- Ecriture dans un ficher  **/
    /** Créer et Ouvrir un nouveau fichier Fichier1.txt en mode Ecriture (w)**/
    file1 = fopen("Fichier1.txt","w");

    if(file1 == NULL){
        printf("Erreur");
        return 0;
    }

    printf("Entrer une chaine: ");

    /** Lire la chaine à partir de l'écran **/
    fgets(chaine,100,stdin);

    /** Ecrire la chaine saisie dans le fichier Fichier1.txt **/
    fprintf(file1,"%s",chaine);

    /** Afficher la chaine saisie sur l'écran aussi **/
    fprintf(stdout,"%s",chaine);

    /** Fermer le Fichier Fichier1.txt **/
    fclose(file1);


/** 2- Ecriture sur un fichier à partir d'un fichier caractere par caractere **/
    /** Ouvrir le Fichier existant Fichier1.txt en mode Lecture (r) **/
    file1 = fopen("Fichier1.txt","r");

    /** Créer et Ouvrir un nouveau fichier Fichier2.txt en mode Ecriture (w)**/
    file2 = fopen("Fichier2.txt","w");

    if((file1 == NULL)||(file2 == NULL)){
        printf("Erreur d'Ouverture Fichier");
        return 0;
    }

    printf("\nEcriture Caractère par caractère de Fichier1 vers Fichier2 \n");

    /** Copier le contenu de Fichier1.txt vers Fichier2.txt caractère par caractère  **/
    while ((c = fgetc(file1)) != EOF)
        fputc(c,file2);

    /** Fermer les  Fichiers :  Fichier1.txt et Fichier2.txt **/
    fclose(file1);
    fclose(file2);
```

```c
/** 3-A Ecriture sur un fichier à partir d'un fichier Ligne par Ligne   **/
    /** Ouvrir le Fichier existant Texte.txt en mode Lecture (r) **/
    file1 = fopen("Texte.txt","r");

    /** Créer et Ouvrir un nouveau fichier Texte2.txt en mode Ecriture (w)**/
    file2 = fopen("Texte2.txt","w");

    if((file1 == NULL)||(file2 == NULL)){
        printf("Erreur d'Ouverture Fichier");
        return 0;
    }

    printf("\nEcriture sur Fichier Ligne par Ligne fputs\n");

    /** Copier le contenu de Texte1.txt vers Texte2.txt Ligne par Ligne  **/
    while( fgets (ligne,50, file1)!=NULL )
        fputs(ligne,file2);

    /** Fermer les  Fichiers :  Texte1.txt et Texte2.txt **/
    fclose(file1);
    fclose(file2);


/** 3-B Ecriture sur un fichier à partir d'un fichier Ligne par Ligne   **/
    /** Ouvrir le Fichier existant Texte.txt en mode Lecture (r) **/
    file1 = fopen("Texte.txt","r");

    /** Créer et Ouvrir un nouveau fichier Texte3.txt en mode Ecriture (w)**/
    file2 = fopen("Texte3.txt","w");

    if((file1 == NULL)||(file2 == NULL)){
        printf("Erreur d'Ouverture Fichier");
        return 0;
    }
    printf("\nEcriture sur Fichier Ligne par Ligne fprintf\n");

    /** Copier le contenu de Texte1.txt vers Texte3.txt Ligne par Ligne  **/
    while( fgets (ligne,50, file1)!=NULL )
        fprintf(file2,"%s",ligne);

    /** Fermer les  Fichiers :  Texte1.txt et Texte3.txt **/
    fclose(file1);
    fclose(file2);
```

```c
/** 4- Lecture d'un fichier Ligne par Ligne et Affichage sur Ecran **/
   /** Ouvrir le Fichier existant Texte.txt en mode Lecture (r) **/
   file1 = fopen("Texte.txt","r");

    if(file1 == NULL){
        printf("Erreur");
        return 0;
    }

    printf("\nLecture Ligne par Ligne et Affichage sur Ecran \n");

    /** Copier le contenu de chaque ligne du fichier Texte.txt dans la variable ligne (50 caractères Max)    **/
    while( fgets (ligne,50, file1)!=NULL ) {
        /**  Affichage sur Ecran d'une seul ligne: 3 méthodes **/
        puts(ligne);
        /** ou bien **/
        //fprintf(stdout,"%s",ligne);
        /** ou bien **/
        //printf("%s",ligne);
    }

    /** Fermer le Fchier Texte.txt**/
    fclose(file1);
```

```c
/** 5- Lecture d'un fichier Carectere par Caractere et Affichage sur Ecran **/
    /** Ouvrir le Fichier existant Texte.txt en mode Lecture (r) **/
    file1 = fopen("Texte.txt","r");

     if(file1 == NULL){
         printf("Erreur");
         return 0;
     }

     printf("\nLecture Caractere par Caractere et Affichage sur Ecran \n");

     /** Copier chaque caractère du fichier Texte.txt dans la variable c **/
     while ((c = fgetc(file1)) != EOF){
         /**Affichage sur Ecran caracère par caractère: 4 méthodes **/
         putc(c,stdout);
         /** ou bien **/
         putchar(c);
         /** ou bien **/
         fprintf(stdout,"%c",c);
         /** ou bien **/
        printf("%c",c);

     }

     /** Fermer le Fchier Texte.txt**/
     fclose(file1);
```

Exercice 1 : découpage d'une phrase

Ecrire un programme qui permet de découper une phrase et d'enregistrer les mots de la phrase dans une ***Liste Chaînée (ou une File).***

Les mots de la phrase peuvent être séparés par des espaces ou par des séparateurs (**, ; . ! ? -**)

Indice :

- Créer une fonction qui teste si un caractère appartient à la liste des caractères séparateurs pour découper une phrase.
- Utiliser les fonctions nécessaires précédentes.

Solution de l'exercice 1

```
int EstCharSeparateur(char c){
    switch(c){
                case ';':
                case ',':
                case '.':
                case ' ':
                case '!':
                case '?':
                case '\n':
                case '-':
                case '_':
                case '\t': return 1;
                break;
              }
    return 0;
}
```

```
Mot* DecouperPhrase(char* phrase){

    Mot* Liste_MOTS=NULL;

    int n = strlen(phrase);
    int i;

    // chaine de caractere comporaire pour assembler un mot en cours de lecture
    char tmp[20];
    int indx=0;

    for(i=0 ; i<n && phrase[i]!='\0' ;i++){

        if(EstCharSeparateur(phrase[i])==0){ // si le caractere n'est pas un separateur
                                             // copier le caractere en cours dans le tompon
            tmp[indx]=phrase[i];
            // assume que le prochain caractere sera le dernier
            tmp[indx+1]='\0';
            // incrementer lindice du tompon
            indx++;
        }

        else{
            // si le tompon n'est pas vide
            if(indx!=0){
                // Ajouter le dernier mot dans la liste des mots
                Liste_MOTS=Inserer_a_la_fin(Liste_MOTS,tmp);
                // vider le tompon
                tmp[0]='\0';
                indx=0;
            }

        }
    }

    if(indx!=0){ // is le tompon n'est pas vide (car le dernier caractere n'est pas un separateur)
                 // ajouter le dernier mot dans la liste
                 Liste_MOTS=Inserer_a_la_fin(Liste_MOTS,tmp);
    }
    return Liste_MOTS;
}
```

Exercice 2 : FILE & PILE ET MANIPULATION DE FICHIER

Ecrire un programme qui permet de :

- Créer et ouvrir un fichier texte, nommé ***« etudiant.txt »***, en mode écriture puis insérer le texte suivant :

```
Je suis un(e) étudiant(e)

En 3 Année Licence Informatique

Module Compilation

Département d'Informatique

Faculté des Sciences

Université Abou Bekr Belkaid Tlemcen

Année universitaire 2025-2026
```

- Créer une pile **P** puis :
  - Ouvrir le fichier ***« etudiant.txt »*** en mode lecture
  - Lire le contenu du fichier ***« etudiant.txt »*** ligne par ligne.
  - Empiler chaque ligne du fichier dans la pile **P**.
  - Afficher la pile **P** :
    - Dépiler chaque ligne affichée de la pile **P** pour avoir une pile vide à la fin.
- Créer une file **F** puis :
  - Ouvrir le fichier ***« etudiant.txt »*** en mode lecture
  - Lire le contenu du fichier ***« etudiant.txt »*** ligne par ligne.
  - Découper chaque ligne en mot.
  - Enfiler chaque mot dans la file **F**.
  - Afficher la file **F** :
    - Défiler chaque mot affiché de la file **F** pour avoir une file vide à la fin.

Solution de l'exercice 2

- Créer et ouvrir un fichier texte, nommé ***« etudiant.txt »***, en mode écriture puis insérer le texte suivant :

```c
/** Remplir Fichier Ligne par Ligne **/
void Ecriture_Fichier_Ligne_par_Ligne(char* Nom_Fichier){

        FILE* fichier =NULL;
    /** Créer et Ouvrir un nouveau fichier en mode Ecriture (w)**/
        fichier = fopen(Nom_Fichier,"w");
        if(fichier == NULL){
                fprintf(stderr, "Erreur lors de l'Ouverture du Fichier\n");
                exit(1);
        }
    /** Ecrire dans le Fichier  **/
        fprintf(fichier, "Je suis un(e) étudiant(e)\n");
        fprintf(fichier, "En 3 Année Licence Informatique\n");
        fprintf(fichier, "Module Compilation\n");
        fprintf(fichier, "Département d'informatique\n");
        fprintf(fichier, "Faculté des Sciences\n");
        fprintf(fichier, "Université Abou Bekr Belkaid Tlemcen\n");
        fprintf(fichier, "Année universitaire 2025-2026\n");
    /** Fermer le Fichier **/
        fclose(fichier);
}
```

- Créer une pile **P** puis :

```c
/** Structure de la cellule pour stocker une ligne **/
typedef struct cellule {
    char ligne[500];
    struct cellule* next;
} Element;

/** Structure de la pile **/
typedef struct pile {
    Element* sommet;
} Pile;

/**  Initialise une pile vide **/
Pile* initialiserPile(Pile* p) {
    p = malloc(sizeof(Pile));
    if (p == NULL) {
        fprintf(stderr, "Mémoire insuffisante\n");
        exit(1);
    }
    p->sommet = NULL;
    return p;
}
```

  - Ouvrir le fichier ***« etudiant.txt »*** en mode lecture.
  - Lire le contenu du fichier ***« etudiant.txt »*** ligne par ligne.
  - Empiler chaque ligne du fichier dans la pile **P**.

```c
/** Lire Fichier Ligne par Ligne **/
Pile* Lire_Fichier_Ligne_par_Ligne(char* Nom_Fichier, Pile* P){

        FILE* fichier =NULL;
    /** on suppose qu'une ligne ne contient pas plus de 500 char **/
        char* ligne= malloc(sizeof(char)*500);

    /** Créer et Ouvrir un nouveau fichier en mode Ecriture (w)**/
        fichier = fopen(Nom_Fichier,"r");
        if(fichier == NULL){
                fprintf(stderr, "Erreur lors de l'Ouverture du Fichier\n");
                exit(1);
        }
    /** Empiler chaque Ligne lu dans le fichier **/
        while( fgets (ligne,500, fichier)!=NULL ) {
            /**  Empiler la Ligne **/
            P = Empiler(P, ligne);
        }
    /** Fermer le Fichier **/
        fclose(fichier);
        return P;
}
```

```c
/** Empile une ligne dans la pile **/
Pile* Empiler(Pile* p, char* ligne) {
    Element* nv = malloc(sizeof(Element));
    if (nv == NULL) {
        fprintf(stderr, "Mémoire insuffisante\n");
        exit(1);
    }
    strncpy(nv->ligne, ligne, sizeof(nv->ligne));
    nv->next = p->sommet;
    p->sommet = nv;
    return p;
}
```

- Afficher la pile **P** :
    - Dépiler chaque ligne affichée de la pile **P** pour avoir une pile vide à la fin.

```c
/** Affiche le contenu de la pile (ligne par ligne) **/
void AfficherPile(Pile* p) {
    while (p->sommet != NULL) {
        printf("%s\n", p->sommet->ligne);
        p = Depiler(p);
    }
    printf("\n");
}
```

```c
/** Empile une ligne dans la pile **/
Pile* Depiler(Pile*p){
    if(p->sommet==NULL)
        return p;
    Element *e=p->sommet;
    p->sommet=p->sommet->next;
    free(e);
    return p;
}
```

- Créer une file **F** puis :

```c
/** Structure de la cellule pour stocker un mot **/
typedef struct cellule {
    char mot[100];
    struct cellule* next;
} Element;

/** Structure de la file **/
typedef struct file {
    Element* premier;
    Element* dernier;
} File;

/** Initialise une file vide **/
File* initialiserFile(File* f) {
    f = malloc(sizeof(File));
    if (f == NULL) {
        fprintf(stderr, "Mémoire insuffisante\n");
        exit(1);
    }
    f->premier = f->dernier = NULL;
    return f;
}
```

  - Ouvrir le fichier ***« etudiant.txt »*** en mode lecture.
  - Lire le contenu du fichier ***« etudiant.txt »*** ligne par ligne.
  - Découper chaque ligne en mot.
  - Enfiler chaque mot dans la file **F**.

```c
/** Lire Fichier Ligne par Ligne **/
File* Lire_Fichier_Ligne_par_Ligne(char* Nom_Fichier, File* F){

        FILE* fichier =NULL;
    /** on suppose qu'une ligne ne contient pas plus de 500 char **/
        char* ligne= malloc(sizeof(char)*500);

    /** Créer et Ouvrir un nouveau fichier en mode Ecriture (w)**/
        fichier = fopen(Nom_Fichier,"r");
        if(fichier == NULL){
                fprintf(stderr, "Erreur lors de l'Ouverture du Fichier\n");
                exit(1);
        }
    /** Empiler chaque Ligne lu dans le fichier **/
        while( fgets (ligne,500, fichier)!=NULL ) {
            /**  Empiler la Ligne **/
            F = Decouper_Mot(F, ligne);
        }
    /** Fermer le Fichier **/
        fclose(fichier);
        return F;
}
```

```c
/** Découper la Ligne  en mot **/
File* Decouper_Mot(File* F, char* ligne){
    int n = strlen(ligne);
    int i;
    char tmp[20];
    int indx=0;
    for(i=0 ; i<n && ligne[i]!='\0' ;i++){
        if(EstCharSeparateur(ligne[i])==0){
            tmp[indx]=ligne[i];
            tmp[indx+1]='\0';
            indx++;
        }
        else{
            if(indx!=0){
                F =Enfiler(F,tmp);
                tmp[0]='\0';
                indx=0;
            }
        }
    }
    if(indx!=0)
        F=Enfiler(F,tmp);
    return F;
}
```

```c
/** Enfile un mot dans la file **/
File* Enfiler(File* f, char* mot) {
    Element* nv = malloc(sizeof(Element));
    if (nv == NULL) {
        fprintf(stderr, "Mémoire insuffisante\n");
        exit(1);
    }
    strncpy(nv->mot, mot, sizeof(nv->mot));
    nv->next = NULL;
    if (f->dernier == NULL) {
        f->premier = f->dernier = nv;
    } else {
        f->dernier->next = nv;
        f->dernier = nv;
    }
    return f;
}
```

- Afficher la file **F** :
    - Défiler chaque mot affiché de la file **F** pour avoir une file vide à la fin.

```c
/** Affiche le contenu de la file **/
void AfficherFile(File* f) {
    while (f->premier != NULL) {
        printf("%s\n", f->premier->mot);
        f=Defiler(f);
    }
    printf("\n");
}
```

```c
/** Défile un mot de la file **/
File* Defiler(File* f) {
    if (f->premier == NULL) {
        fprintf(stderr, "La file est vide, impossible de défiler\n");
        return f;
    }
    Element* e = f->premier;
    f->premier = f->premier->next;
    free(e);
    if (f->premier == NULL) {
        f->dernier = NULL;
    }
    return f;
}
```

TP N°3 : Les Automates et Les Expressions Régulières

1- Objectif :

L'objectif de ce TP est de comprendre, modéliser et implémenter des automates à états finis, afin d'apprendre à représenter des langages réguliers et à simuler leur fonctionnement à travers des programmes.

2- Rappel sur les automates et les expressions régulières :

Un **automate à états finis** est un modèle mathématique utilisé pour représenter le **comportement d'un système séquentiel**. Il est constitué d'un nombre fini d'états et permet de reconnaître des **langages réguliers**, en traitant une **entrée caractère par caractère**.

Un automate est un 5-uplet A= (Q, Σ, δ, q0, F) A = (Q, \Sigma, \delta, q_0, F) A=(Q, Σ, δ, q0, F)

où :

- Q : ensemble fini d'états,
- Σ : alphabet (ensemble de symboles d'entrée),
- δ : fonction de transition $Q \times \Sigma \rightarrow Q$,
- $q_0 \in Q$: état initial,
- $F \subseteq Q$: ensemble des états finaux (ou acceptants).

Dans le **module de compilation**, les automates sont utilisés dans la **phase d'analyse lexicale** pour reconnaître les **tokens** (mots-clés, identificateurs, nombres, etc.) dans un programme source.

Une **expression régulière** est une **formule mathématique** qui décrit un **ensemble de mots** formés avec un alphabet donné. Elle permet d'écrire de manière compacte les **règles de formation des lexèmes** (comme les mots-clés ou identifiants d'un langage).

En compilation :

Elle sert à **définir les motifs** des mots du langage (ex. : un identifiant commence par une lettre, suivi de lettres ou chiffres).

3- Exemples Automate :

Soit **X ={0,1,2,….,9}** un alphabet et **L** un langage qui est l'ensemble des nombres multiples de **5**

(L={0,5,10,...etc**}).**

Expression régulière **:**

r : :=((1|2|3|4|6|7|8|9)*(0|5))$^+$

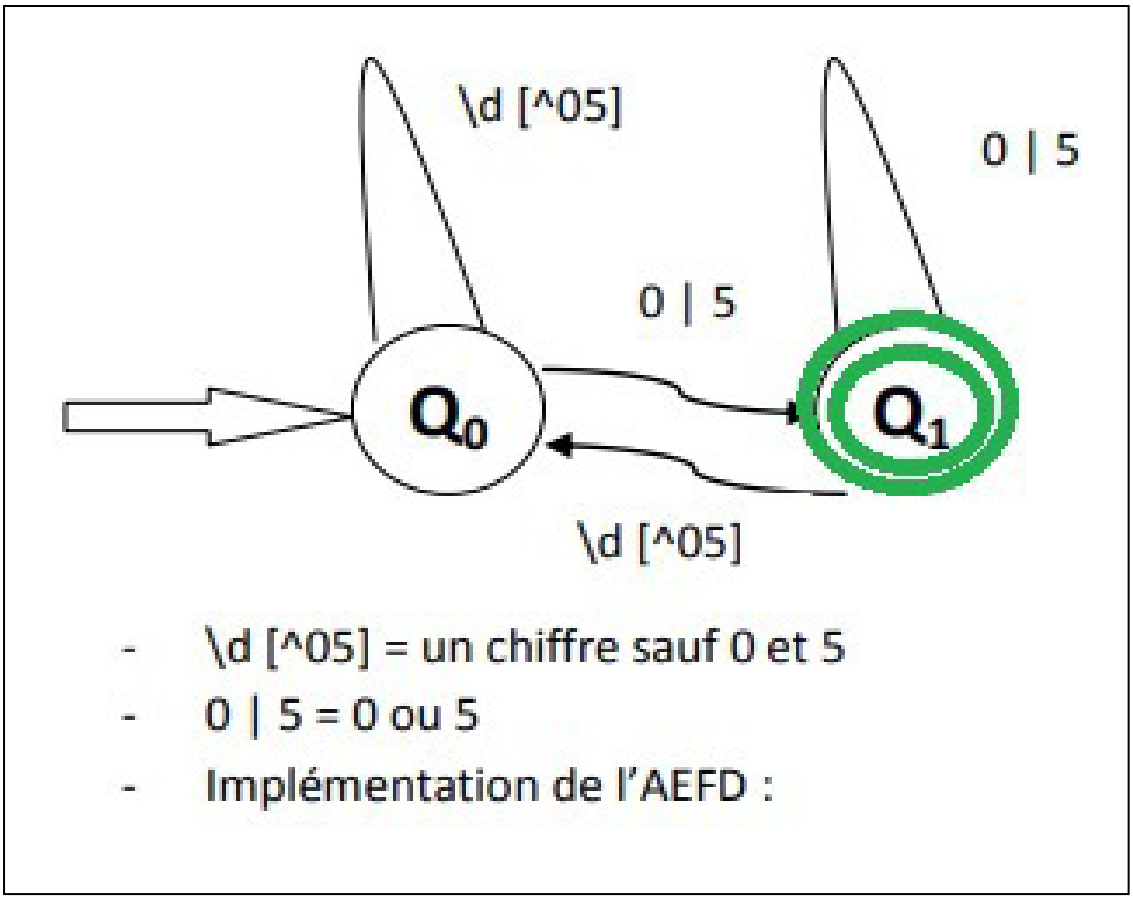


Image 3.1- AEF pour les multiples de 5

Implémentation :

```c
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#define Q0  0
#define Q1  1
int AEF_Multiple_5(char* nbr){
    int i, n=strlen(nbr);
    int etat_automate=0, etat_finale=1;
    /** parcourir la chaine de caractères **/
    for(i=0; (i<n) && (nbr[i]!='\0'); i++)
        switch(etat_automate){
                    /** si le chiffre courant == 0 ou 5 le nombre devient multiple de 5 **/
            case Q0:    if(nbr[i]=='0'||nbr[i]=='5')
                            etat_automate=1;
                        /** sinon le nombre n'est pas un multiple de 5 (a cet etap) **/
                        else if(isdigit(nbr[i]))
                            etat_automate= 0;
                        /** si le caractere n'est pas un chiffre, ce n'est pas un nombre et ce n'est pas multiple de 5 **/
                        else
                            return 0;
                        break;
                        /** si le chiffre courant == 0 ou 5 le nombre reste multiple de 5 **/

            case Q1:    if(nbr[i]=='0'||nbr[i]=='5')
                            etat_automate=1;
                        /** sinon le nombre n'est pas un multiple de 5 **/
                        else if(isdigit(nbr[i]))
                            etat_automate= 0;
                        /** si le caractère n'est pas un chiffre, ce n'est pas un nombre et ce n'est pas multiple de 5 (a cet étape) **/
                        else
                            return 0;
                        break;
        }
    /** Teste si l'etat courant apres le parcour de la chaine de caractere est un etat final (Q1 est l'etat final) **/
    return (etat_automate==Q1);
}
```

```c
void main(){

    int i;
    char tmp [33];
    printf ("\n**************** Les Multiples de 5 de 0 => 100 ***************\n\n",i);
    for (i=0; i<=100; i++){
        if (AEF_Multiple_5(itoa(i,tmp,10))==1)
            printf ("\t%d est un Multiple de 5\n",i);
    }
    return 0;
}
```

Exercice 1 :

- Soit **X={0,1}** un alphabet et **L1** le langage formé des suites binaires qui se terminent par la séquence **110**.

  L1= {110,1110, 10110, 11110, ...}.

- Soit **X={a,b,c,...,z, /,<,>}** un alphabet et **L2** l'ensemble des balises **XML** (ouvrantes, fermantes et vide).

  L2= {</>, <a>, , <abc>, </abc>, ….}.

- Soit **X={0,1,2,....,9,+,-,*,/}** un alphabet et **L3** le langage des formules arithmétiques pour les nombres entiers où les opérandes ne sont pas signés. Un nombre entier est accepté par le langage.

  L3= {9, 90, 4+53, 300*57-12+80, …. }.

- Soit **X ={0,1,2,....,9}** un alphabet et **L4** un langage qui reconnait les nombres pairs.

  L4={0,2,4,6,8, ……....**}**

Questions :

1- Donnez les AEFD (*automates à états finis déterministes*) qui reconnaissent les chaînes de L1, L2, L3 et L4.

2- Donnez l'expression régulière qui définit chaque langage.

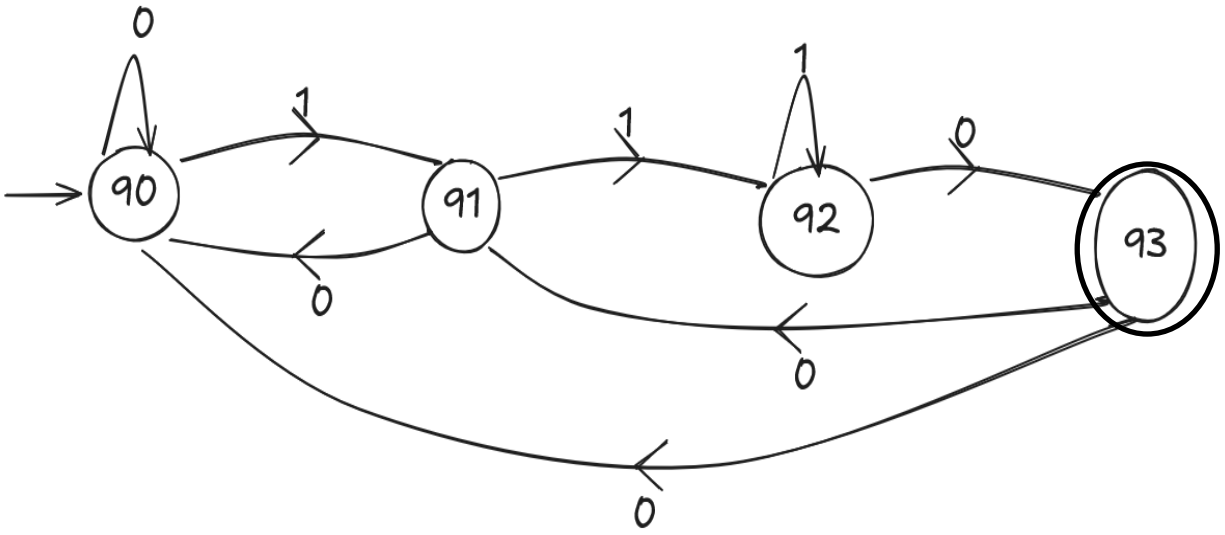
3- Traduisez chaque AEFD (L1, L2, L3, L4) en un programme.

4- En utilisant l'automate 4 des nombres pairs, affichez les nombres pairs et impairs compris entre 20 et 55.

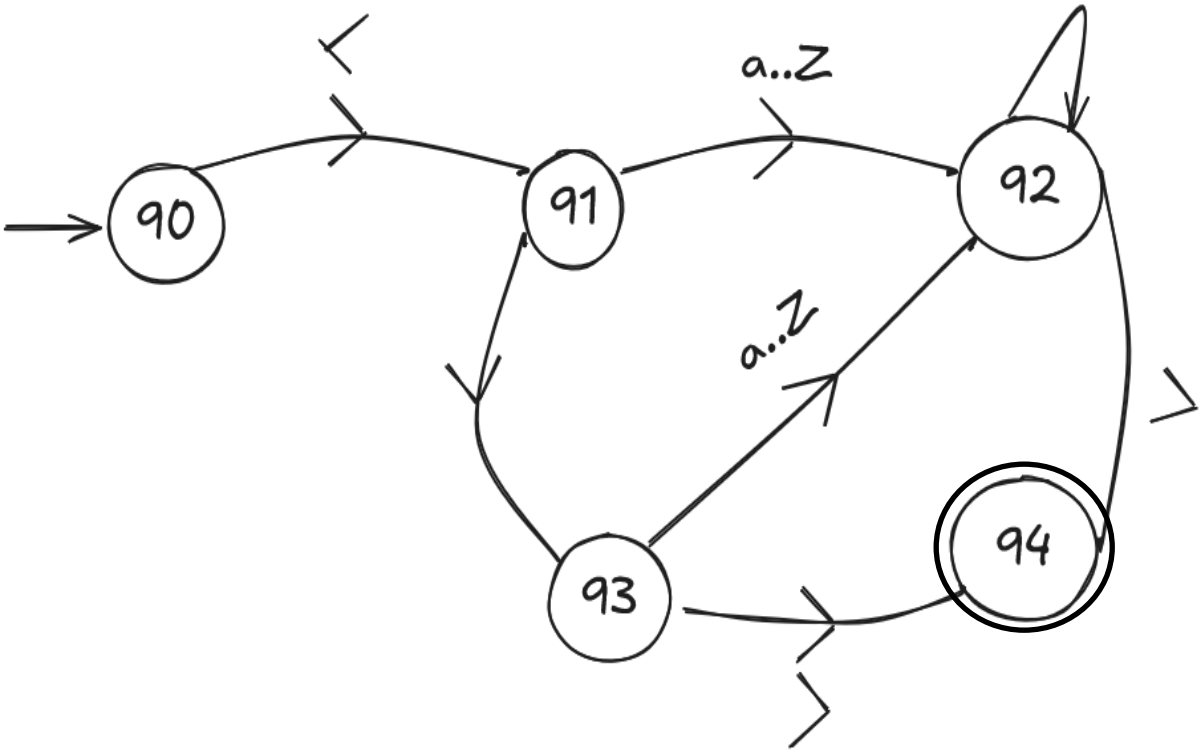
Solution de l'exercice 1

1- Donnez les AEFD (*automates à états finis déterministes*) qui reconnaissent les chaînes de L1, L2, L3 et L4 :

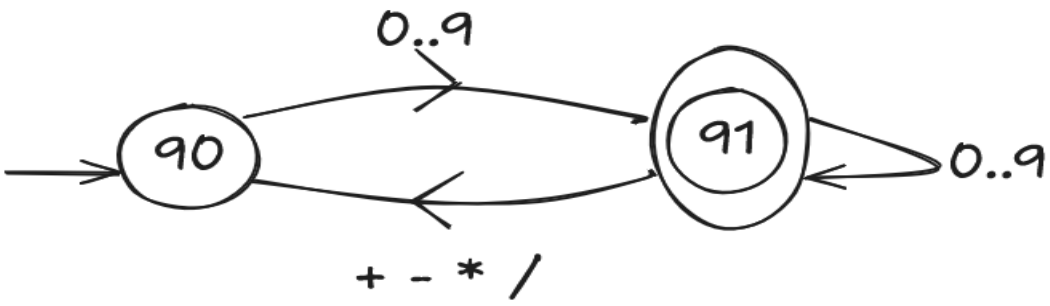
a- Automate L1 :



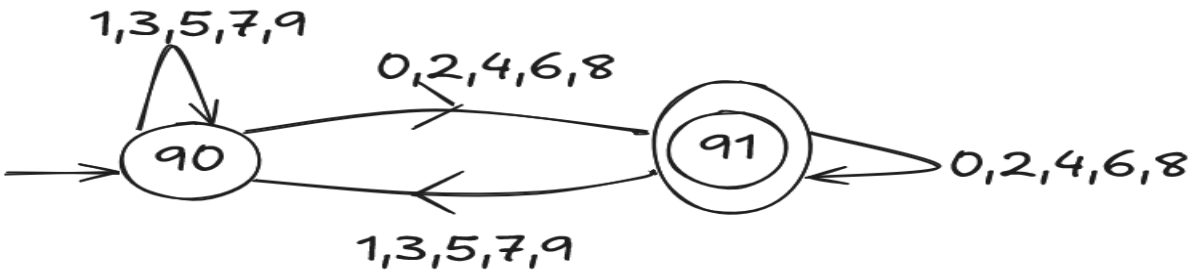
b- Automate L2 :



c- Automate L3 :



d- Automate L4 :



2- Traduisez chaque AEFD (L1, L2, L3, L4) en un programme.

a- Implémentation & expression régulière L1 :

```
                        /**  Fonction Automate Suite Binaire
                    Expression reguliere r::= (([10]*1[1]+)+0 **/
int Automate_Suite_110(char* NB){
    int i,n=strlen(NB);
    int etat=0;
    for(i=0;(i<n)&&(NB[i]!='\0');i++)
         switch(etat){
           case 0: if(NB[i]=='0') etat=0;
                    else if(NB[i]=='1') etat=1;
                        else return 0;
                   break;
           case 1: if(NB[i]=='0') etat=0;
                    else if(NB[i]=='1') etat=2;
                        else return 0;
                   break;
           case 2: if(NB[i]=='0') etat=3;
                      else if(NB[i]=='1') etat=2;
                        else return 0;
                   break;
           case 3: if(NB[i]=='0') etat=0;
                   else if(NB[i]=='1') etat=1;
                     else return 0;
                   break;
       }

     return etat;
}
```

b- Implémentation & expression régulière L2 :

```
                    /**         Fonction Automate Balise XML sans balise vide
                              Expression reguliere r::= <[\]?[\w]+>          **/

int Automate_Balise_XML(char* Element_Balise){
    int i,n=strlen(Element_Balise);
    int etat=0;
    for(i=0;(i<n)&&(Element_Balise[i]!='\0');i++)
        switch(etat){
            case 0: if(Element_Balise[i]=='<')
                        etat=1;
                    else
                        return 0;
                    break;
            case 1: if(isalpha(Element_Balise[i]))
                        etat=2;
                        else if(Element_Balise[i]=='/')
                            etat=3;
                              else
                                return 0;
                    break;
            case 2: if(isalpha(Element_Balise[i]))
                        etat=2;
                        else if(Element_Balise[i]=='>')
                            etat=4;
                              else
                                return 0;
                    break;

            case 3: if(isalpha(Element_Balise[i]))
                        etat=2;
                    else if(Element_Balise[i]=='>')
                            etat=4;
                    else
                        return 0;
                    break;


            }
    return etat;
}
```

c- Implémentation & expression régulière L3 :

```
int est_Operateur_Arithmetique(char c){
    switch(c){
                case '+':
                case '-':
                case '*':
                case '/':
                case '%': return 1;
                          break;
                }
    return 0;
}
```

```
/**     Fonction Automate Expression Arithmetique avec des opérandes non signés
                Expression reguliere r::= [\d]([\d]*([+-/*%][\d]+)*)    **/
int Automate_Exp_Arith(char* exp){
    int i,n=strlen(exp);
    int etat=0;
    for(i=0;(i<n)&&(exp[i]!='\0');i++)
        switch(etat){
            case 0: if(isdigit(exp[i]))
                        etat=1;
                    else
                        return 0;
                    break;
            case 1: if(isdigit(exp[i]))
                            etat=1;
                        else if(est_Operateur_Arithmetique(exp[i]))
                            etat=0;
                         else
                            return 0;
                    break;
        }
    return etat;
}
```

d- Implémentation & expression régulière L4 :

```
                        /** Fonction Automate Nombre Pair
                         r : :=( (1|3|5|7|9)*(0|2|4|6|8) )+ **/
int AEF_Pair(char* nbr){

    int i, n=strlen(nbr);
    int etat_automate=0, etat_finale=1;

    /** parcourir la chaine de caractères **/
    for(i=0; (i<n) && (nbr[i]!='\0'); i++)
        switch(etat_automate){
                    /** si le chiffre courant == 0,2,4,6,8   nombre paire **/
            case 0:    if(nbr[i]=='0'||nbr[i]=='2'||nbr[i]=='4'||nbr[i]=='6'||nbr[i]=='8')
                            etat_automate=1;
                       /** sinon impaire **/
                       else if(isdigit(nbr[i]))
                            etat_automate= 0;
                       /** si le caractere n'est pas un chiffre, ce n'est pas un nombre **/
                       else
                            return 0;
                       break;
                       /** si le chiffre courant == 0,2,4,6,8 **/
            case 1:    if(nbr[i]=='0'||nbr[i]=='2'||nbr[i]=='4'||nbr[i]=='6'||nbr[i]=='8')
                            etat_automate=1;
                       /** sinon le nombre n'est pas un nombre pair **/
                       else if(isdigit(nbr[i]))
                            etat_automate= 0;
                       /** si le caractère n'est pas un chiffre, ce n'est pas un nombre (a cet étape) **/
                       else
                            return 0;
                       break;
        }
    return etat_automate;
}
```

Exercice 2 :

- Soit **X= {a, b, c, ...z,0…9, /, *}** un alphabet et **L** est le langage qui reconnait les commentaires multilignes dans le langage C.
- Exemple : **/***

Commentaire ***Ligne 1

*******Ligne2 ********

Ligne3

***/**

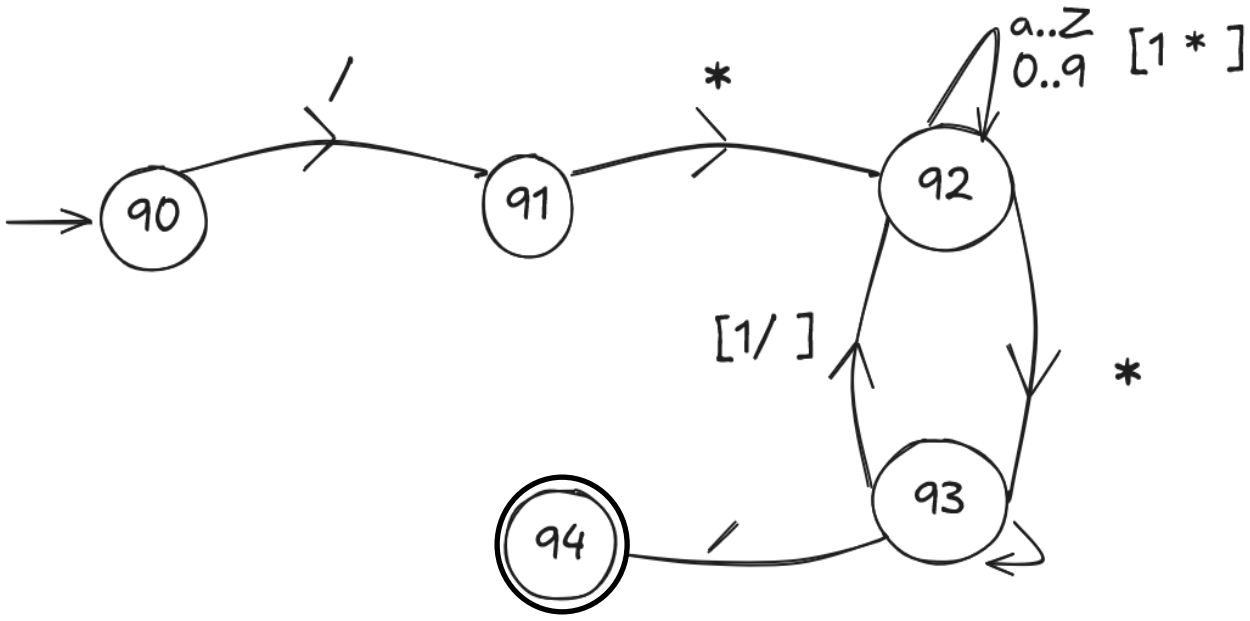
Questions :

1- Donnez l'AEFD qui reconnait le langage L.

2- Implémentez l'AEFD correspondant en lisant le commentaire C multiligne à partir d'un fichier.

<u>Solution de l'exercice 2</u>

1- Donnez l'AEFD qui reconnait le langage L :



2- Implémentation :

```
#define Q0  0
#define Q1  1
#define Q2  2
#define Q3  3
#define Q4  4

int AEF_Commentaire_ML_C(char* nbr){
int i, n=strlen(nbr);
int etat=Q0;
    for(i=0;i<n;i++){
        switch(etat){
            case Q0:    if(isspace(nbr[i]))
                            etat=Q0;
                        else if(nbr[i]=='/')
                            etat=Q1;
                        else
                            return 0;
                        break;
            case Q1:    if(nbr[i]=='*')
                            etat=Q2;
                        else
                            return 0;
                        break;

            case Q2:    if(nbr[i]=='*')
                            etat=Q3;
                        else
                            etat=Q2;
                        break;
            case Q3:    if(nbr[i]=='*')
                            etat=Q3;
                        else if(nbr[i]=='/')
                            etat=Q4;
                        else
                            etat=Q2;
                        break;
            case Q4:    if(isspace(nbr[i]))
                            etat=Q4;
                        else
                            return 0;
                        break;
        }
    }
    return(etat==Q4);
}
```

EXERCICES SUPPLEMENTAIRES

Exercice 1 :

Q1- Définissez l'alphabet, le langage et l'expression régulière qui permet de reconnaitre un nombre entier ou réel signé et non signé.

- Par exemple, l'automate doit reconnaitre :
  - Les nombres entiers : 135, +135, -135
  - Les nombres réels : 14.7509, +14.7509, -14.7509, 14.7, -14.75, +14.750

Q2- Modifiez l'automate précédent afin d'accepter au maximum deux chiffres après la virgule pour un nombre réel.

- Par exemple, l'automate doit reconnaitre :
  - Les nombres entiers : 68, +68, -68
  - Les nombres réels : 17.30, +17.30, -17.30, 17.3, +17.3, -17.3

Q3- Implémentez les AEFD correspondant à **Q1** et **Q2**.

Exercice 2 :

Q1- Définissez l'alphabet, le langage et l'expression régulière qui permet de reconnaitre les équations arithmétiques pour les opérandes signés.

Exemple :

- L'équation : 9*2+3= 21 acceptée
- L'équation : +9.00*2+3.00 = +21.000 acceptée
- L'équation : -9*2-3= -21 acceptée
- L'équation : -9*2+-3.87= -21 non acceptée
- L'équation : -9*2+3.87*= -21 non acceptée

Q2- Implémentez l'AEFD correspondant.

Exercice 3 :

Q1- Définissez l'alphabet, le langage et l'expression régulière qui permet de reconnaitre une adresse IP version IPv4. Les adresses IP sont codées sur 4 octets dont la forme est la suivante :

octet1.octet2.octet3.octet4

- Chaque octet prend les valeurs entre **0** et **255**.

Q2- Implémentez l'AEFD correspondant.

TP N°4 : Analyse Lexicale

1- Objectif :

L'objectif de ce TP est de développer un analyseur lexical d'un langage d'affectation simple. Le résultat produit par cet analyseur (**Suite des unités lexicales**) sera utilisé par l'analyseur syntaxique **LL1** dans le TP suivant.

2- Définition :

L'analyse lexicale est la 1ère étape de la compilation. Elle permet de transformer le code source en **une suite d'U.L** qui peuvent être :

- Mots clés ou réservés
- Identificateurs : (en langage d'affectation simple : x, y, x1, somme, …)
- Nombres : les entiers non signés

L'analyseur lexical doit assurer les tâches suivantes :

- Lecture caractère/caractère du code source.
- Élimination des informations inutiles (dans notre cas : les espaces, les tabulations, …).
- Identification des U.L. (type de chaque UL, ligne) :
  - La spécification des U.L se fait par des expressions régulières.
  - La reconnaissance des U.L. se fait via l'A.E.F relatif à l'expression régulière.
- Détection des erreurs lexicales (exemple : caractère n'appartenant pas à l'alphabet du langage, nombre illégal…).

3- Etapes à suivre :

3.1- ***Définition du langage d'affectation simple :***

L'analyseur lexical du sous langage souhaité est un AEFD (Automate à Etats Fini Déterministe) permettant de reconnaitre les unités lexicales (**terminaux**) suivantes :

a) Identifiant : L'identificateur sont analogues aux noms de variables dans les langages de programmation.

La forme générale d'un identifiant valide est exprimée par l'expression régulière suivante :

r ::=\w[\d\w_]*

b) Nombres : Les nombres entiers, non signés, valides sont représentés par l'expression régulière suivante :

r ::=(([\d]+) | (- [\d]+))

c) **Parenthèse ouvrante / fermante :** les Parenthèse ouvrantes et fermante () doivent être reconnues

d) **SYMBOLE_AFFECTATION :** =

e) **Opérateurs Arithmétiques :** ***r := [+-*/]***

f) **Séparateur Point-virgule**

3.2- Reconnaissance des U.L. :

A partir de l'expression régulière qui définit chaque U.L, on donne son AEFD correspondant, puis réunir tous les automates dans un seul automate globale, qui commence par l'état initial 0. A partir de cet état-là, on commence à lire une expression d'affectation caractère par caractère.

Par exemple : La lecture d'une lettre nous ramène à la reconnaissance d'un identificateur, alors que la lecture d'un chiffre nous ramène à la reconnaissance d'un nombre entier ou réel, etc.….

Étendre cet automate pour qu'il ignore les espaces et signale les erreurs lexicales telles que la lecture d'un nombre erroné ou la lecture d'un caractère non autorisé.

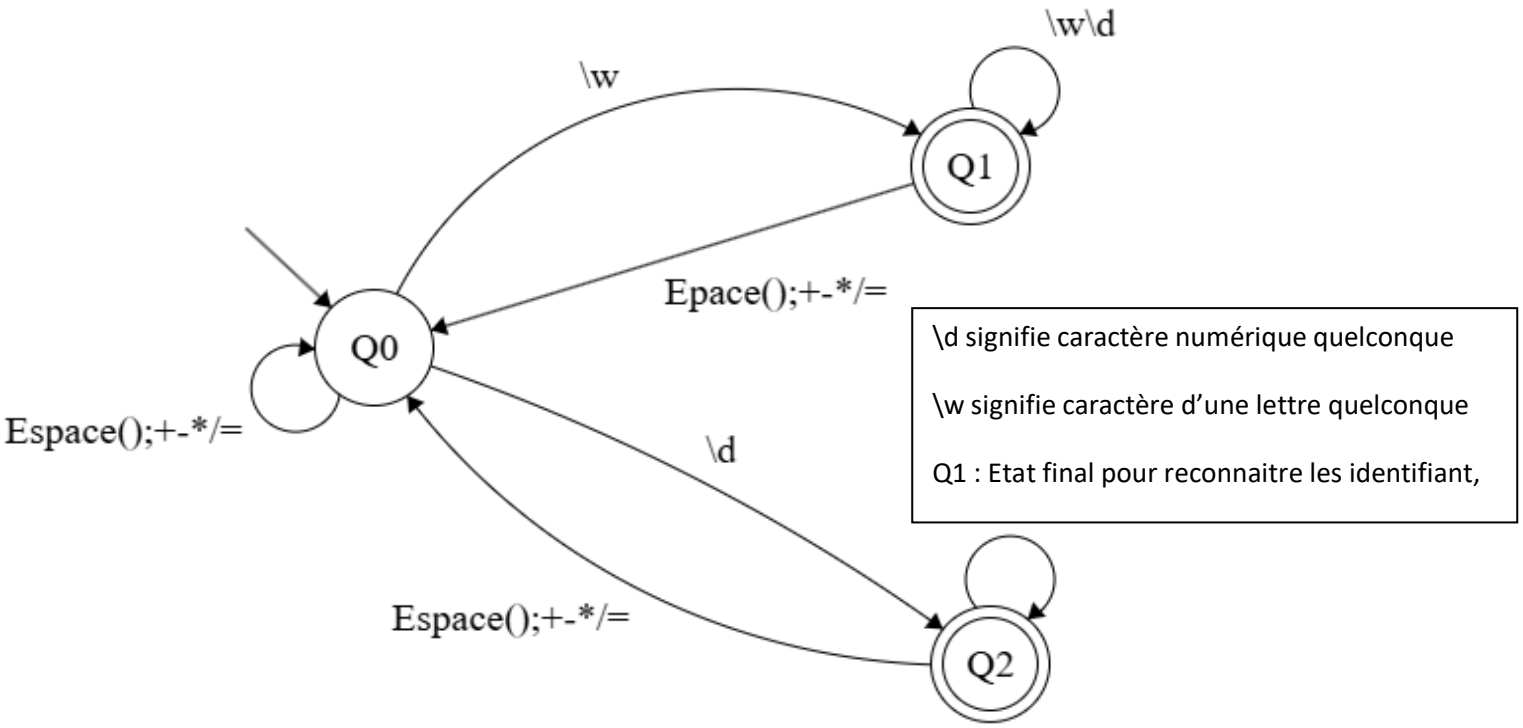


Figure1 Automate global du langage d'affectation simple

3.3- **Attribution des codes aux U.L. :**

Chaque **unité lexicale** est représentée par **un code**. Il est préférable d'utiliser un intervalle contigu pour représenter les codes des unités lexicales. Par exemple :

Unité Lexicale	Code (Type_UL)
IDENTIFIANT_UL	260
NOMBRE_UL	261
PLUS	262
MINUS	263
MULTIPLY	264
DIVIDE	265
PARENTHESE_OUVRANTE	266
PARENTHESE_FERMANTE	267
SYMBOLE_AFFECTATION	268
SEPARATEUR_POINT_VIRGULE	269
EPSILON	111
FIN_SUITE_UL (#)	999
ERREUR	-9

3.4- **Structure d'une U.L. :**

Une unité lexicale est représentée par la structure suivante :

```
typedef struct Unite_Lexicale {
                    char   Lexeme[20];
                    int    Code;
                    int    Ligne;
                    struct Unite_Lexicale* Suivant;
} UL;
```

Remarque :

A la fin de l'analyse lexicale, il faut retourner la liste chainée des U.L, appelée suite des U.L

3.5- Algorithme :

a- Lire l'expression d'affectation (code source) à partir d'un fichier texte.

b- Analyser le code source caractère par caractère en utilisant l'automate global.

c- Une unité lexicale est reconnue si l'automate est dans l'état final de celle-ci et un caractère séparateur est lu.

d- Chaque U.L reconnue par l'automate sera insérée à la fin de la liste chainée des suite U.L et l'automate transite à l'état initial.

e- En cas où une erreur lexicale est détectée, l'analyseur doit avorter le traitement et retourne la position et la ligne du caractère où l'erreur a été détectée.

f- Si l'analyse lexicale se termine avec succès, insérer le « # » à la fin de la suite U.L.

Exemple d'exécution :

1- ***Analyse lexicale avec succès :***

a=(((2+3)*5)-4);

```
        CODE SOURCE A ANALYSER :       a=-(((2+3)*5)-4);

        ANALYSE LEXICALE REUSSITE - LISTE DES ULs :

Lexeme [a] , Code :[260] , Ligne[ligne] 1
Lexeme [=] , Code :[268] , Ligne[ligne] 1
Lexeme [-] , Code :[263] , Ligne[ligne] 1
Lexeme [(] , Code :[266] , Ligne[ligne] 1
Lexeme [(] , Code :[266] , Ligne[ligne] 1
Lexeme [(] , Code :[266] , Ligne[ligne] 1
Lexeme [2] , Code :[261] , Ligne[ligne] 1
Lexeme [+] , Code :[262] , Ligne[ligne] 1
Lexeme [3] , Code :[261] , Ligne[ligne] 1
Lexeme [)] , Code :[267] , Ligne[ligne] 1
Lexeme [*] , Code :[264] , Ligne[ligne] 1
Lexeme [5] , Code :[261] , Ligne[ligne] 1
Lexeme [)] , Code :[267] , Ligne[ligne] 1
Lexeme [-] , Code :[263] , Ligne[ligne] 1
Lexeme [4] , Code :[261] , Ligne[ligne] 1
Lexeme [)] , Code :[267] , Ligne[ligne] 1
Lexeme [;] , Code :[269] , Ligne[ligne] 1
Lexeme [#] , Code :[999] , Ligne[ligne] 2
```

2- ***Analyse lexicale échouée (erreur) :***

a=(((! 2+3)*5)-4);

```
        CODE SOURCE A ANALYSER :       a=(((  ! 2+3)*5)-4);

 Lexeme [a] , Code :[260] , Ligne[ligne] 1
 Lexeme [=] , Code :[268] , Ligne[ligne] 1
 Lexeme [(] , Code :[266] , Ligne[ligne] 1
 Lexeme [(] , Code :[266] , Ligne[ligne] 1
 Lexeme [(] , Code :[266] , Ligne[ligne] 1
Erreur lexicale '!' ligne 1
```

Solution TP 4

```c
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>

/**=========================================
=       codes        **/

#define ID       260
#define NUM      261
#define PLUS     262
#define MINUS    263
#define MULT     264
#define DIV      265
#define LPAREN   266
#define RPAREN   267
#define EQ       268
#define SEMI     269
#define EPSILON  111
#define END      999
#define ERROR    -9

/**=======  ETATS DE L'AUTOMATE   ========**/
#define q0 0    /** état initial **/
#define q1 1    /** lecture d'un identificateur **/
#define q2 2    /** lecture d'un nombre **/
```

```c

/**========  STRUCTURE TOKEN ==========**/
typedef struct UniteLexicale {
    char lexeme[100];
    int  code;
    int  ligne;
    struct UniteLexicale* suivant;
} UniteLexicale;

/**========  AJOUTER UN TOKEN À LA LISTE CHAÎNÉE =======**/
UniteLexicale* ajout_token(UniteLexicale* liste, char* lexeme, int code, int ligne) {

    UniteLexicale* nouveau = (UniteLexicale*)malloc(sizeof(UniteLexicale));
    strcpy(nouveau->lexeme, lexeme);
    nouveau->code = code;
    nouveau->ligne = ligne;
    nouveau->suivant = NULL;

    if (liste == NULL)
        return nouveau; /** if list is empty , the element becomes the first one **/

    UniteLexicale* temp = liste;
    while (temp->suivant != NULL)
        temp = temp->suivant;
    temp->suivant = nouveau;

    return liste;
}
```

```c
/**========== AFFICHER LA LISTE DE TOKENS =========**/
void afficher_tokens(UniteLexicale* liste) {

    printf("\n===== TABLE DES TOKENS =====\n");
    UniteLexicale* courant = liste;

    while (courant != NULL) {
        printf("Lexeme: %-10s | Code: %-2d | Ligne: %d\n",
               courant->lexeme, courant->code, courant->ligne);
        courant = courant->suivant;
    }
}

/**==========  AFFICHER LE CONTENU FICHIER =========**/
void afficher_contenu_fichier(const char* contenu) {
    printf("========= CONTENU DU FICHIER ==========\n");
    printf("%s\n", contenu);
    printf("=======================================\n");
}


/**==========  ANALYSEUR LEXICAL (AUTOMATE)  =========**/
int Analyseur_Lexical(UniteLexicale* liste, char* source) {

    int i = 0;
    int state = q0;      /** état initial **/
    char buffer[100];
    int buf_index = 0;
    int ligne = 1;


    while (1) {
        char c = source[i];
        switch (state) {
            /**-------------------------------------
            -   ÉTAT q0 : état initial
            -   Decider quel type de Token
            -------------------------------------**/
            case q0: /** start **/
                if (c == '\0') {
                    return liste;
                }
                else if (isspace(c)) {
                    if (c == '\n') ligne++; /** increment ligne **/
                    i++;
                }
                else if (isalpha(c)) {
                    buf_index = 0;
                    buffer[buf_index++] = c;
                    i++;
                    state = q1;  /** identificateur **/
                }
                else if (isdigit(c)) {
                    buf_index = 0;
                    buffer[buf_index++] = c;
                    i++;
                    state = q2; /** nombre **/
                }
```

```
114                 else if (c == '+') {
115                     liste = ajout_token(liste, "+", PLUS, ligne);
116                     i++;
117                 }
118                 else if (c == '-') {
119                     liste = ajout_token(liste, "-", MINUS, ligne);
120                     i++;
121                 }
122                 else if (c == '=') {
123                     liste = ajout_token(liste, "=", EQ, ligne);
124                     i++;
125                 }
126                 else if (c == '(') {
127                     liste = ajout_token(liste, "(", LPAREN, ligne);
128                     i++;
129                 }
130                 else if (c == ')') {
131                     liste = ajout_token(liste, ")", RPAREN, ligne);
132                     i++;
133                 }
134                 else if (c == ';') {
135                     liste = ajout_token(liste, ";", SEMI, ligne);
136                     i++;
137                 }
138               else if (c == '*') {
139                   liste = ajout_token(liste, "*", MULT, ligne);
140                   i++;
141               }
142               else if (c == '/') {
143                   liste = ajout_token(liste, "/", DIV, ligne);
144                   i++;
145               }
146               else {
147                   printf("Erreur Lexicale : caractere '%c' ligne %d\n", c, ligne);
148                   i++;
149                   return ERROR;
150               }
151               break;
153             /**----------------------------------------
154             -   ÉTAT q1 : on lit un identificateur **/
155
156             case q1:
157                 if (isalnum(c)) {
158                     buffer[buf_index++] = c;
159                     i++;
160                 } else {
161                     buffer[buf_index] = '\0';
162                     liste = ajout_token(liste, buffer, ID, ligne);
163                     state = q0;
164                 }
165                 break;
```

```
167             /**----------------------------------------
168             -   ÉTAT q2 : lecture d'un nombre       */
169             case q2:
170                 if (isdigit(c)) {
171                     buffer[buf_index++] = c;
172                     i++;
173                 } else {
174                     buffer[buf_index] = '\0';
175                     liste = ajout_token(liste, buffer, NUM, ligne);
176                     state = q0;
177                 }
178                 break;
179         }
180     }
181     return 1;
182 }
```

Exercice supplémentaire :

Développer un analyseur lexical d'un langage des ensembles.

L'analyseur lexical du sous langage souhaité est un AEFD (Automate à Etats Fini Déterministe) permettant de reconnaitre les unités lexicales (**terminaux**) suivantes :

- Identifiant : L'identificateur d'un champ et les noms d'ensemble (A, B.… lettre majuscule)
- Ensemble explicite : {1, 2 ,3} ou {} (vide)
- Nombres : Les nombre entiers / réels valides.
- Accolade ouvrante / fermante : les accolades ouvrantes, **{,** et fermante, **},** doivent être reconnues.
- Les Opération ensembliste :
    a. Union : +
    b. Intersection : /
    c. Différence : -
- Séparateur Virgule : **,**
- Parenthèses : **(** et **)**
- Egale : =

1- Donner l'AEFD de ce langage.

2- Implémenter l'analyseur lexical en utilisant l'AEFD trouvé.

TP N°5 : Analyse Syntaxique

1- Objectif :

L'objectif de ce TP est de concevoir et d'implémenter un analyseur syntaxique LL (1) permettant de reconnaître les phrases d'un langage d'affectation simple à partir de la suite d'unités lexicales générée par l'analyseur lexical. Ce TP vise à :

- **Définir la grammaire du langage** et vérifier qu'elle est compatible avec la méthode LL (1) (élimination des ambiguïtés, des récursions à gauche, facteur commun…).
- **Construire les ensembles FIRST et FOLLOW**, pour la génération de la table d'analyse.
- **Élaborer la table LL (1)** associé à la grammaire.
- **Implémenter l'analyseur syntaxique** capable de :
  - Lire la suite de lexèmes en entrée,
  - Appliquer la stratégie d'analyse LL (1),
  - Détecter et signaler les erreurs syntaxiques,
  - Valider les phrases conformes à la grammaire.

Partie 1 : PREPARATION DE LA TABLE D'ANALYSE

1.1- Définition de la grammaire : elle est constituée de **:**

- **Les terminaux**

IDENT	NOMBRE	PLUS +	MINUS -	MULTIPLY *	DIVIDE /	PAR_(	PAR_)	SYMBOLE_AFFECTATION =	SEP_;

- **Les non-terminaux**

S (axiome)
A
E
E1
T
T1
F

Les règles de production

- S -> ID = A;
- A -> E
- E -> TE1
- E1 -> +TE1 | -TE1| EPSILON
- T -> FT1
- T1 -> *FT1 | /FT1| EPSILONE
- F-> (A)| NUMBER | ID

1.2- Codage et sauvegarde des règles de production :

a- Affecter **des codes** pour chaque **terminal.**

Utiliser les codes affectés dans l'étape de l'Analyse Lexicale.

```
10  #define IDENTIFIANT_UL           260
11  #define NOMBRE_UL                261
12  #define PLUS                     262
13  #define MINUS                    263
14  #define MULTIPLY                 264
15  #define DIVIDE                   265
16  #define PARENTHESE_OUVRANTE      266
17  #define PARENTHESE_FERMANTE      267
18  #define SYMBOLE_AFFECTATION      268
19  #define SEPARATEUR_POINT_VIRGULE 269
20  #define EPSILONE                 111
21  #define FIN_SUITE_UL             999
22  #define ERREUR                   -1
```

b- Affecter **des codes** pour chaque **non-terminal**.

```
#define VN_S   0
#define VN_A   1
#define VN_E   2
#define VN_E1  3
#define VN_T   4
#define VN_T1  5
#define VN_F   6
```

c- Affecter **un code** pour chaque **règle de production**.

Règle de production	**Code**
S -> ID= A;	**0**
A -> E	**1**
E -> TE1	**2**
E1 -> +TE1	**3**
E1 -> -TE1	**4**
E1 -> EPSILON	**5**
T -> FT1	**6**
T1 -> *FT1	**7**
T1 -> /FT1	**8**
T1 -> EPSILONE	**9**
F -> (A)	**10**
F-> NUMBER	**11**
F-> ID	**12**

d- Sauvegarder toutes **les règles de production** dans une matrice sous la forme suivante :

0	**260**	**268**	**1**	**269**	**-1**
1	**2**	**-1**	**-1**	**-1**	**-1**
2	**4**	**3**	**-1**	**-1**	**-1**
3	**262**	**4**	**3**	**-1**	**-1**
4	**263**	**4**	**3**	**-1**	**-1**
5	**111**	**-1**	**-1**	**-1**	**-1**
6	**6**	**5**	**-1**	**-1**	**-1**
7	**264**	**6**	**5**	**-1**	**-1**
8	**265**	**6**	**5**	**-1**	**-1**
9	**111**	**-1**	**-1**	**-1**	**-1**
10	**266**	**1**	**267**	**-1**	**-1**
11	**261**	**-1**	**-1**	**-1**	**-1**
12	**260**	**-1**	**-1**	**-1**	**-1**

Où :

- Chaque **ligne de la matrice** représente **une règle de production**.
- Chaque **cellule de la matrice** contient **le code d'un terminal** ou **d'un non-terminal**.
- **Le code -1** indique la fin **d'une règle de production**.

- **Exemple de code source :**

```
int RPs[13][5] = {
    {IDENTIFIANT_UL, SYMBOLE_AFFECTATION, VN_A, SEPARATEUR_POINT_VIRGULE, -1},    // 0: S -> ID = A;
    {VN_E, -1, -1, -1, -1},                                                       // 1: A -> E
    {VN_T, VN_E1, -1, -1, -1},                                                    // 2: E -> TE1
    {PLUS, VN_T, VN_E1, -1, -1},                                                  // 3: E1 -> +TE1
    {MINUS, VN_T, VN_E1, -1, -1},                                                 // 4: E1 -> -TE1
    {EPSILONE, -1, -1, -1, -1},                                                   // 5: E1 -> EPSILON
    {VN_F, VN_T1, -1, -1, -1},                                                    // 6: T -> FT1
    {MULTIPLY, VN_F, VN_T1, -1, -1},                                              // 7: T1 -> *FT1
    {DIVIDE, VN_F, VN_T1, -1, -1},                                                // 8: T1 -> /FT1
    {EPSILONE, -1, -1, -1, -1},                                                   // 9: T1 -> EPSILONE
    {PARENTHESE_OUVRANTE, VN_A, PARENTHESE_FERMANTE, -1, -1},                     // 10: F -> (A)
    {NOMBRE_UL, -1, -1, -1, -1},                                                  // 11: F -> NUMBER
    {IDENTIFIANT_UL, -1, -1, -1, -1},                                             // 12: F -> ID
};
```

1.3- Calcul du First et Follow :

a- Pour chaque **non-terminal R**, calculer ***First(R)*** afin de remplir l'entrée correspondante de la Table d'Analyse ***(TA).***

b- En cas où le **non-terminal** produit **Epsilone**, calculer ***Follow(A)*** est le rajouté au résultat précédent.

Calcul:

TA (S)= First(S) ={IDENTIFIANT_UL}

TA (A)= First(A) ={ PAR_(, NOMBRE_UL, IDENTIFIANT_UL}

TA (E)= First(E) ={PAR_(, NOMBRE_UL, IDENTIFIANT_UL}

TA (E1)= First(E1)U Follow(E1) ={PAR_), PLUS, MINUS, SEPARATEUR_POINT_VIRGULE}

TA (T)= First(T) ={PAR_(, NOMBRE_UL, IDENTIFIANT_UL}

TA (T1)= First(T1)U Follow(T1) ={PLUS, MINUS, MULTIPLY, DIVIDE , PAR_) ,
SEPARATEUR_POINT_VIRGULE}

TA (F)= First(F) ={PAR_(, NOMBRE_UL, IDENTIFIANT_UL}

1.4- Construction et remplissage de la Table d'Analyse (TA)

La Table d'Analyse (TA) est représentée sous forme d'une **matrice** où :

- **Chaque ligne** représente **un non-terminal**.
- **Chaque colonne** représente **un terminal**.

Si ***First*** ou ***Follow*** **d'un non-terminal** correspond à **un terminal** (ou plusieurs), **la cellule** correspondante dans **la Table d'Analyse (TA)** est remplie par **le code** de **la règle de production** qui a généré ce **terminal**.

Les cellules vides sont remplies par **un code d'erreur** : **#define ERREUR -1**

Table d'Analyse :

	IDENT	NOMBRE	PLUS	MINUS	MULTIPLY	DIVIDE	PAR_(	PAR_)	SYM_'='	SEP_;	#
S	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
A	1	1	-1	-1	-1	-1	1	-1	-1	-1	-1
E	2	2	-1	-1	-1	-1	2	-1	-1	-1	-1
E1	-1	-1	3	4	-1	-1	-1	5	-1	5	-1
T	6	6	-1	-1	-1	-1	6	-1	-1	-1	-1
T1	-1	-1	9	9	7	8	-1	9	-1	9	-1
F	12	11	-1	-1	-1	-1	10	-1	-1	-1	-1

- **Exemple de code source :**

```
/**                                Table d'analyse                                    **/
/** Colonnes: IDENTIFIANT_UL, NOMBRE_UL, PLUS, MINUS, MULTIPLY, DIVIDE, PAR_(, PAR_), =, ;, FIN_# **/
int TA[7][11] = {
    // ID      NUM     +       -       *       /       (       )       =       ;       FIN
    {0,     ERREUR, ERREUR, ERREUR, ERREUR, ERREUR, ERREUR, ERREUR, ERREUR, ERREUR, ERREUR}, // S
    {1,     1,      ERREUR, ERREUR, ERREUR, ERREUR, 1,      ERREUR, ERREUR, ERREUR, ERREUR}, // A (simplifié)
    {2,     2,      ERREUR, ERREUR, ERREUR, ERREUR, 2,      ERREUR, ERREUR, ERREUR, ERREUR}, // E
    {ERREUR,ERREUR, 3,      4,      ERREUR, ERREUR, ERREUR, 5,      ERREUR, 5,      ERREUR}, // E1
    {6,     6,      ERREUR, ERREUR, ERREUR, ERREUR, 6,      ERREUR, ERREUR, ERREUR, ERREUR}, // T
    {ERREUR,ERREUR, 9,      9,      7,      8,      ERREUR, 9,      ERREUR, 9,      ERREUR}, // T1
    {12,    11,     ERREUR, ERREUR, ERREUR, ERREUR, 10,     ERREUR, ERREUR, ERREUR, ERREUR}, // F
};
```

Partie 2 : PROGRAMMATION DE L'ANALYSEUR SYNTAXIQUE (LL1)

2.1- Structure de données :

On va utiliser deux structures de données :

- **Première structure** : c'est une **Liste Chainée** des **Unités Lexicales** générée par l'Analyseur **Lexical** qui se termine par « # ».
- **Deuxième structure** : c'est une **Pile** pour **l'Analyseur Syntaxique** initialisée à « **S #** ».
- **Remarque** : la structure des éléments de la **Pile** est identique à celle de **Liste Chainée** des **Unités Lexicales**.

```
typedef struct pile {
    UL* sommet;
} Pile;

Pile* InitialiserPile(Pile* p) {
    p = malloc(sizeof(Pile));
    p->sommet = NULL;
    return p;
}
```

2.2- L'Analyse Syntaxique :

Faire dérouler **l'Algorithme d'Analyse Syntaxique LL1** suivant :

PileAnalyseurSyntaxique=**S #** //Initialisation de la pile de l'analyseur syntaxique

CopieSuiteUL= une copie de la SuiteUL produit par l'analyseur lexicale

Répété tant que (Tete(**CopieSuiteUL**) ->Code !=# || Tete(**PileAnalyseurSyntaxique**) ->Code!=#) :{

Si Tete(**CopieSuiteUL**) ->Code== Tete(**PileAnalyseurSyntaxique**) ->Code { **//Action1**

Supprimer l'entête de **CopieSuiteUL**

Dépiler l'élément tète de **PileAnalyseurSyntaxique }**

Sinon

Si Tete(**PileAnalyseurSyntaxique**) est un non-Terminal {

Si TA[Tete(**PileAnalyseurSyntaxique**)->code][Tete(**CopieSuiteUL**) ->Code] !=**Erreur** { **//Action2**

Dépiler l'élément tète de **PileAnalyseurSyntaxique**

Empiler la Règle de production TA[Tete(**PileAnalyseurSyntaxique**)->code][Tete(**CopieSuiteUL**) ->Code] dans

PileAnalyseurSyntaxique }

Sinon **Erreur Syntaxique** }

Sinon **Erreur Syntaxique**

Afficher (**PileAnalyseurSyntaxique et CopieSuiteUL)**

}

Si (Tete(**SuiteUL**) ->Code ==# && Tete(**PileAnalyseurSyntaxique**) ->Code==#) **Analyse Syntaxique réussite**

Sinon **Erreur Syntaxique**

Code source à analyser : a=(2+3)*5-4;

Pile :S ,#.
SuiteUl: IDENTIFIANT_UL(a), SYMBOLE_AFFECTATION(=) ,PARENTHESE_OUVRANTE('(') ,NOMBRE(2), PLUS(+) ,NOMBRE(3), PARENTHESE_FERMANTE(')') ,MULTIPLY(*) ,NOMBRE(5), MINUS(-) ,NOMBRE(4), SEPARATEUR_POINT_VIRGULE(;) ,#.

Iteration 1 :
SuiteUL: IDENTIFIANT_UL Pile: S
Table_DAnalyse[0, 0] = 0
Depiler S & Empiler RP=0 (Action2)
Pile :IDENTIFIANT_UL, SYMBOLE_AFFECTATION(=) ,A ,SEPARATEUR_POINT_VIRGULE(;) ,#.
SuiteUl: IDENTIFIANT_UL(a), SYMBOLE_AFFECTATION(=) ,PARENTHESE_OUVRANTE('(') ,NOMBRE(2), PLUS(+) ,NOMBRE(3), PARENTHESE_FERMANTE(')') ,MULTIPLY(*) ,NOMBRE(5), MINUS(-) ,NOMBRE(4), SEPARATEUR_POINT_VIRGULE(;) ,#.

Iteration 2 :
SuiteUL: IDENTIFIANT_UL Pile: IDENTIFIANT_UL
Depiler et suprimer de la SuiteUL: IDENTIFIANT_UL (Action1)
Pile :SYMBOLE_AFFECTATION(=) ,A ,SEPARATEUR_POINT_VIRGULE(;) ,#.

SuiteUl: SYMBOLE_AFFECTATION(=) ,PARENTHESE_OUVRANTE('(') ,NOMBRE(2), PLUS(+) ,NOMBRE(3), PARENTHESE_FERMANTE(')') ,MULTIPLY(*) ,NOMBRE(5), MINUS(-) ,NOMBRE(4), SEPARATEUR_POINT_VIRGULE(;) ,#.

Iteration 3 :
SuiteUL: SYMBOLE_AFFECTATION(=) Pile: SYMBOLE_AFFECTATION(=)
Depiler et suprimer de la SuiteUL: SYMBOLE_AFFECTATION(=) (Action1)
Pile :A ,SEPARATEUR_POINT_VIRGULE(;) ,#.
SuiteUl: PARENTHESE_OUVRANTE('(') ,NOMBRE(2), PLUS(+) ,NOMBRE(3), PARENTHESE_FERMANTE(')') ,MULTIPLY(*) ,NOMBRE(5), MINUS(-) ,NOMBRE(4), SEPARATEUR_POINT_VIRGULE(;) ,#.

Iteration 4 :
SuiteUL: PARENTHESE_OUVRANTE('(') Pile: A
Table_DAnalyse[1, 6] = 1
Depiler A & Empiler RP=1 (Action2)
Pile :E ,SEPARATEUR_POINT_VIRGULE(;) ,#.
SuiteUl: PARENTHESE_OUVRANTE('(') ,NOMBRE(2), PLUS(+) ,NOMBRE(3), PARENTHESE_FERMANTE(')') ,MULTIPLY(*) ,NOMBRE(5), MINUS(-) ,NOMBRE(4), SEPARATEUR_POINT_VIRGULE(;) ,#.

Iteration 5 :
SuiteUL: PARENTHESE_OUVRANTE('(') Pile: E
Table_DAnalyse[2, 6] = 3
Depiler E & Empiler RP=3 (Action2)
Pile :T ,E1 ,SEPARATEUR_POINT_VIRGULE(;) ,#.
SuiteUl: PARENTHESE_OUVRANTE('(') ,NOMBRE(2), PLUS(+) ,NOMBRE(3), PARENTHESE_FERMANTE(')') ,MULTIPLY(*) ,NOMBRE(5), MINUS(-) ,NOMBRE(4), SEPARATEUR_POINT_VIRGULE(;) ,#.

Iteration 6 :
SuiteUL: PARENTHESE_OUVRANTE('(') Pile: T
Table_DAnalyse[4, 6] = 7
Depiler T & Empiler RP=7 (Action2)
Pile :F ,T1 ,E1 ,SEPARATEUR_POINT_VIRGULE(;) ,#.
SuiteUl: PARENTHESE_OUVRANTE('(') ,NOMBRE(2), PLUS(+) ,NOMBRE(3), PARENTHESE_FERMANTE(')') ,MULTIPLY(*) ,NOMBRE(5), MINUS(-) ,NOMBRE(4), SEPARATEUR_POINT_VIRGULE(;) ,#.

Iteration 7 :
SuiteUL: PARENTHESE_OUVRANTE('(') Pile: F
Table_DAnalyse[6, 6] = 11
Depiler F & Empiler RP=11 (Action2)
Pile :PARENTHESE_OUVRANTE('(') ,A ,PARENTHESE_FERMANTE(')') ,T1 ,E1 ,SEPARATEUR_POINT_VIRGULE(;) ,#.
SuiteUl: PARENTHESE_OUVRANTE('(') ,NOMBRE(2), PLUS(+) ,NOMBRE(3), PARENTHESE_FERMANTE(')') ,MULTIPLY(*) ,NOMBRE(5), MINUS(-) ,NOMBRE(4), SEPARATEUR_POINT_VIRGULE(;) ,#.

Iteration 8 :
SuiteUL: PARENTHESE_OUVRANTE('(') Pile: PARENTHESE_OUVRANTE('(')
Depiler et suprimer de la SuiteUL: PARENTHESE_OUVRANTE('(') (Action1)
Pile :A ,PARENTHESE_FERMANTE(')') ,T1 ,E1 ,SEPARATEUR_POINT_VIRGULE(;) ,#.
SuiteUl: NOMBRE(2), PLUS(+) ,NOMBRE(3), PARENTHESE_FERMANTE(')') ,MULTIPLY(*) ,NOMBRE(5), MINUS(-) ,NOMBRE(4), SEPARATEUR_POINT_VIRGULE(;) ,#.

Iteration 9 :
SuiteUL: NOMBRE Pile: A
Table_DAnalyse[1, 1] = 1
Depiler A & Empiler RP=1 (Action2)
Pile :E ,PARENTHESE_FERMANTE(')') ,T1 ,E1 ,SEPARATEUR_POINT_VIRGULE(;) ,#.
SuiteUl: NOMBRE(2), PLUS(+) ,NOMBRE(3), PARENTHESE_FERMANTE(')') ,MULTIPLY(*) ,NOMBRE(5), MINUS(-) ,NOMBRE(4), SEPARATEUR_POINT_VIRGULE(;) ,#.

Iteration 10 :
SuiteUL: NOMBRE Pile: E
Table_DAnalyse[2, 1] = 3
Depiler E & Empiler RP=3 (Action2)
Pile :T ,E1 ,PARENTHESE_FERMANTE(')') ,T1 ,E1 ,SEPARATEUR_POINT_VIRGULE(;) ,#.
SuiteUl: NOMBRE(2), PLUS(+) ,NOMBRE(3), PARENTHESE_FERMANTE(')') ,MULTIPLY(*) ,NOMBRE(5), MINUS(-) ,NOMBRE(4), SEPARATEUR_POINT_VIRGULE(;) ,#.

Iteration 11 :
SuiteUL: NOMBRE Pile: T
Table_DAnalyse[4, 1] = 7
Depiler T & Empiler RP=7 (Action2)
Pile :F ,T1 ,E1 ,PARENTHESE_FERMANTE(')') ,T1 ,E1 ,SEPARATEUR_POINT_VIRGULE(;) ,#.
SuiteUl: NOMBRE(2), PLUS(+) ,NOMBRE(3), PARENTHESE_FERMANTE(')') ,MULTIPLY(*) ,NOMBRE(5), MINUS(-) ,NOMBRE(4), SEPARATEUR_POINT_VIRGULE(;) ,#.

Iteration 12 :
SuiteUL: NOMBRE Pile: F
Table_DAnalyse[6, 1] = 12
Depiler F & Empiler RP=12 (Action2)
Pile :NOMBRE, T1 ,E1 ,PARENTHESE_FERMANTE(')') ,T1 ,E1 ,SEPARATEUR_POINT_VIRGULE(;) ,#.
SuiteUl: NOMBRE(2), PLUS(+) ,NOMBRE(3), PARENTHESE_FERMANTE(')') ,MULTIPLY(*) ,NOMBRE(5), MINUS(-) ,NOMBRE(4), SEPARATEUR_POINT_VIRGULE(;) ,#.

Iteration 13 :
SuiteUL: NOMBRE Pile: NOMBRE
Depiler et suprimer de la SuiteUL: NOMBRE (Action1)
Pile :T1 ,E1 ,PARENTHESE_FERMANTE(')') ,T1 ,E1 ,SEPARATEUR_POINT_VIRGULE(;) ,#.
SuiteUl: PLUS(+) ,NOMBRE(3), PARENTHESE_FERMANTE(')') ,MULTIPLY(*) ,NOMBRE(5), MINUS(-) ,NOMBRE(4), SEPARATEUR_POINT_VIRGULE(;) ,#.

Iteration 14 :
SuiteUL: PLUS(+) Pile: T1
Table_DAnalyse[5, 2] = 10
Depiler T1 & Empiler RP=10 (Action2)
Pile :E1 ,PARENTHESE_FERMANTE(')') ,T1 ,E1 ,SEPARATEUR_POINT_VIRGULE(;) ,#.
SuiteUl: PLUS(+) ,NOMBRE(3), PARENTHESE_FERMANTE(')') ,MULTIPLY(*) ,NOMBRE(5), MINUS(-) ,NOMBRE(4), SEPARATEUR_POINT_VIRGULE(;) ,#.

Iteration 15 :
SuiteUL: PLUS(+) Pile: E1
Table_DAnalyse[3, 2] = 4
Depiler E1 & Empiler RP=4 (Action2)

Pile :PLUS(+) ,T ,E1 ,PARENTHESE_FERMANTE(')') ,T1 ,E1 ,SEPARATEUR_POINT_VIRGULE(;) ,#.
SuiteUl: PLUS(+) ,NOMBRE(3), PARENTHESE_FERMANTE(')') ,MULTIPLY(*) ,NOMBRE(5), MINUS(-) ,NOMBRE(4), SEPARATEUR_POINT_VIRGULE(;) ,#.

Iteration 16 :
SuiteUL: PLUS(+) Pile: PLUS(+)
Depiler et suprimer de la SuiteUL: PLUS(+) (Action1)
Pile :T ,E1 ,PARENTHESE_FERMANTE(')') ,T1 ,E1 ,SEPARATEUR_POINT_VIRGULE(;) ,#.
SuiteUl: NOMBRE(3), PARENTHESE_FERMANTE(')') ,MULTIPLY(*) ,NOMBRE(5), MINUS(-) ,NOMBRE(4), SEPARATEUR_POINT_VIRGULE(;) ,#.

Iteration 17 :
SuiteUL: NOMBRE Pile: T
Table_DAnalyse[4, 1] = 7
Depiler T & Empiler RP=7 (Action2)
Pile :F ,T1 ,E1 ,PARENTHESE_FERMANTE(')') ,T1 ,E1 ,SEPARATEUR_POINT_VIRGULE(;) ,#.
SuiteUl: NOMBRE(3), PARENTHESE_FERMANTE(')') ,MULTIPLY(*) ,NOMBRE(5), MINUS(-) ,NOMBRE(4), SEPARATEUR_POINT_VIRGULE(;) ,#.

Iteration 18 :
SuiteUL: NOMBRE Pile: F
Table_DAnalyse[6, 1] = 12
Depiler F & Empiler RP=12 (Action2)
Pile :NOMBRE, T1 ,E1 ,PARENTHESE_FERMANTE(')') ,T1 ,E1 ,SEPARATEUR_POINT_VIRGULE(;) ,#.
SuiteUl: NOMBRE(3), PARENTHESE_FERMANTE(')') ,MULTIPLY(*) ,NOMBRE(5), MINUS(-) ,NOMBRE(4), SEPARATEUR_POINT_VIRGULE(;) ,#.

Iteration 19 :
SuiteUL: NOMBRE Pile: NOMBRE
Depiler et suprimer de la SuiteUL: NOMBRE (Action1)
Pile :T1 ,E1 ,PARENTHESE_FERMANTE(')') ,T1 ,E1 ,SEPARATEUR_POINT_VIRGULE(;) ,#.
SuiteUl: PARENTHESE_FERMANTE(')') ,MULTIPLY(*) ,NOMBRE(5), MINUS(-) ,NOMBRE(4), SEPARATEUR_POINT_VIRGULE(;) ,#.

Iteration 20 :
SuiteUL: PARENTHESE_FERMANTE(')') Pile: T1
Table_DAnalyse[5, 7] = 10
Depiler T1 & Empiler RP=10 (Action2)
Pile :E1 ,PARENTHESE_FERMANTE(')') ,T1 ,E1 ,SEPARATEUR_POINT_VIRGULE(;) ,#.
SuiteUl: PARENTHESE_FERMANTE(')') ,MULTIPLY(*) ,NOMBRE(5), MINUS(-) ,NOMBRE(4), SEPARATEUR_POINT_VIRGULE(;) ,#.

Iteration 21 :
SuiteUL: PARENTHESE_FERMANTE(')') Pile: E1
Table_DAnalyse[3, 7] = 6
Depiler E1 & Empiler RP=6 (Action2)
Pile :PARENTHESE_FERMANTE(')') ,T1 ,E1 ,SEPARATEUR_POINT_VIRGULE(;) ,#.
SuiteUl: PARENTHESE_FERMANTE(')') ,MULTIPLY(*) ,NOMBRE(5), MINUS(-) ,NOMBRE(4), SEPARATEUR_POINT_VIRGULE(;) ,#.

Iteration 22 :
SuiteUL: PARENTHESE_FERMANTE(')') Pile: PARENTHESE_FERMANTE(')')
Depiler et suprimer de la SuiteUL: PARENTHESE_FERMANTE(')') (Action1)
Pile :T1 ,E1 ,SEPARATEUR_POINT_VIRGULE(;) ,#.
SuiteUl: MULTIPLY(*) ,NOMBRE(5), MINUS(-) ,NOMBRE(4), SEPARATEUR_POINT_VIRGULE(;) ,#.

Iteration 23 :
SuiteUL: MULTIPLY(*)) Pile: T1
Table_DAnalyse[5, 4] = 8
Depiler T1 & Empiler RP=8 (Action2)
Pile :MULTIPLY(*) ,F ,T1 ,E1 ,SEPARATEUR_POINT_VIRGULE(;) ,#.
SuiteUl: MULTIPLY(*) ,NOMBRE(5), MINUS(-) ,NOMBRE(4), SEPARATEUR_POINT_VIRGULE(;) ,#.

Iteration 24 :
SuiteUL: MULTIPLY(*)) Pile: MULTIPLY(*))
Depiler et suprimer de la SuiteUL: MULTIPLY(*)) (Action1)
Pile :F ,T1 ,E1 ,SEPARATEUR_POINT_VIRGULE(;) ,#.
SuiteUl: NOMBRE(5), MINUS(-) ,NOMBRE(4), SEPARATEUR_POINT_VIRGULE(;) ,#.

Iteration 25 :
SuiteUL: NOMBRE Pile: F
Table_DAnalyse[6, 1] = 12
Depiler F & Empiler RP=12 (Action2)
Pile :NOMBRE, T1 ,E1 ,SEPARATEUR_POINT_VIRGULE(;) ,#.
SuiteUl: NOMBRE(5), MINUS(-) ,NOMBRE(4), SEPARATEUR_POINT_VIRGULE(;) ,#.

Iteration 26 :
SuiteUL: NOMBRE Pile: NOMBRE
Depiler et suprimer de la SuiteUL: NOMBRE (Action1)
Pile :T1 ,E1 ,SEPARATEUR_POINT_VIRGULE(;) ,#.
SuiteUl: MINUS(-) ,NOMBRE(4), SEPARATEUR_POINT_VIRGULE(;) ,#.

Iteration 27 :
SuiteUL: MINUS(-) Pile: T1
Table_DAnalyse[5, 3] = 10
Depiler T1 & Empiler RP=10 (Action2)
Pile :E1 ,SEPARATEUR_POINT_VIRGULE(;) ,#.
SuiteUl: MINUS(-) ,NOMBRE(4), SEPARATEUR_POINT_VIRGULE(;) ,#.

Iteration 28 :
SuiteUL: MINUS(-) Pile: E1
Table_DAnalyse[3, 3] = 5
Depiler E1 & Empiler RP=5 (Action2)
Pile :MINUS(-) ,T ,E1 ,SEPARATEUR_POINT_VIRGULE(;) ,#.
SuiteUl: MINUS(-) ,NOMBRE(4), SEPARATEUR_POINT_VIRGULE(;) ,#.

Iteration 29 :
SuiteUL: MINUS(-) Pile: MINUS(-)
Depiler et suprimer de la SuiteUL: MINUS(-) (Action1)
Pile :T ,E1 ,SEPARATEUR_POINT_VIRGULE(;) ,#.
SuiteUl: NOMBRE(4), SEPARATEUR_POINT_VIRGULE(;) ,#.

Iteration 30 :
SuiteUL: NOMBRE Pile: T
Table_DAnalyse[4, 1] = 7
Depiler T & Empiler RP=7 (Action2)
Pile :F ,T1 ,E1 ,SEPARATEUR_POINT_VIRGULE(;) ,#.
SuiteUl: NOMBRE(4), SEPARATEUR_POINT_VIRGULE(;) ,#.

Iteration 31 :
SuiteUL: NOMBRE Pile: F
Table_DAnalyse[6, 1] = 12
Depiler F & Empiler RP=12 (Action2)
Pile :NOMBRE, T1 ,E1 ,SEPARATEUR_POINT_VIRGULE(;) ,#.
SuiteUl: NOMBRE(4), SEPARATEUR_POINT_VIRGULE(;) ,#.

Iteration 32 :
SuiteUL: NOMBRE Pile: NOMBRE
Depiler et suprimer de la SuiteUL: NOMBRE (Action1)
Pile :T1 ,E1 ,SEPARATEUR_POINT_VIRGULE(;) ,#.
SuiteUl: SEPARATEUR_POINT_VIRGULE(;) ,#.

Iteration 33 :
SuiteUL: SEPARATEUR_POINT_VIRGULE(;) Pile: T1
Table_DAnalyse[5, 10] = 10
Depiler T1 & Empiler RP=10 (Action2)
Pile :E1 ,SEPARATEUR_POINT_VIRGULE(;) ,#.
SuiteUl: SEPARATEUR_POINT_VIRGULE(;) ,#.

Iteration 34 :
SuiteUL: SEPARATEUR_POINT_VIRGULE(;) Pile: E1
Table_DAnalyse[3, 10] = 6
Depiler E1 & Empiler RP=6 (Action2)
Pile :SEPARATEUR_POINT_VIRGULE(;) ,#.
SuiteUl: SEPARATEUR_POINT_VIRGULE(;) ,#.

Iteration 35 :
SuiteUL: SEPARATEUR_POINT_VIRGULE(;) Pile: SEPARATEUR_POINT_VIRGULE(;)
Depiler et suprimer de la SuiteUL: SEPARATEUR_POINT_VIRGULE(;) (Action1)
Pile :#.
SuiteUl: #. **Résultat : Analyse Syntaxique Réussie**

Solution TP5

```c
/** Fonction pour réupérer l'indice du Terminal
      dans la table d'analyse          **/
int indiceTA(int code) {
    switch(code) {
        case(IDENTIFIANT_UL): return 0;
        case(NOMBRE_UL): return 1;
        case(PLUS): return 2;
        case(MINUS): return 3;
        case(MULTIPLY): return 4;
        case(DIVIDE): return 5;
        case(PARENTHESE_OUVRANTE): return 6;
        case(PARENTHESE_FERMANTE): return 7;
        case(SYMBOLE_AFFECTATION): return 8;
        case(SEPARATEUR_POINT_VIRGULE): return 9;
        case(FIN_SUITE_UL): return 10;
    }
    return ERREUR;
}
```

```c
/** fonction pour vérifier un non_Terminal  **/
int VerifVNT(int code) {
    if (code == VN_S || code == VN_A || code == VN_E || code == VN_E1 ||
        code == VN_T || code == VN_T1 || code == VN_F)
        return 1;
    else return 0;
}
```

```c
int Analyseur_Syntaxique(Pile* P, UL* SuiteUL) {
    UL* SUL = SuiteUL;
    int i, j, RP, k = 1;

    while((SUL->Code != FIN_SUITE_UL) || (P->sommet->Code != FIN_SUITE_UL)) {
        printf("-------------------------------------------------------------------------------------------\n");
        j = 0;
        printf("Iteration %d :\n", k++);
        printf("SuiteUL: %s      Pile: %s\n", AfficheCode(SUL->Code), AfficheCode(P->sommet->Code));

        if(SUL->Code == P->sommet->Code) {
            printf("Depiler et suprimer de la SuiteUL: %s (Action1)\n", AfficheCode(SUL->Code));
            SUL = Suprimer_UL(SUL);
            P   = Depiler(P);
        }
        else {
            if(VerifVNT(P->sommet->Code)) {
                i = indiceTA(SUL->Code);
                RP = TA[P->sommet->Code][i];
                printf("Table_DAnalyse[%d, %d] = %d\nDepiler %s & Empiler RP=%d (Action2)\n",
                       P->sommet->Code, i, RP, AfficheCode(P->sommet->Code), RP);
```

```
                if(RP != ERREUR) {
                    P = Depiler(P);
                    while(RPs[RP][j] != -1) j++;
                    j--;
                    if(RPs[RP][0] != EPSILONE) {
                        while(j >= 0) {
                            P = Empiler(P, "", RPs[RP][j--], -1);
                        }
                    }
                } else return 0;
            } else return 0;
        }

        AfficherPile(P);
        Afficher2_UL(SUL);
    }

    if((SUL->Code == FIN_SUITE_UL) && (P->sommet->Code == FIN_SUITE_UL))
        return 1;
    else
        return 0;
}
```

Exercice supplémentaire :

A partir de l'exercice supplémentaire du TP n° 04, développer l'analyseur syntaxique correspondant.

La grammaire est constituée de : elle est constituée de :

- **Les terminaux :**

IDENT	NOMBRE	SEPA_,	ACCO_{	ACCO_}	PAR_(	PAR_)	UNION	INTERSECTION	DIFFERENCE	EGALE

- **Les non-terminaux**

S (axiome)
D
E
T
T1
T2
T3

- **Les règles de production**
  - S-> IDEN=D
  - D-> TE
  - E-> +TE | /TE | -TE | EPSILONE
  - T-> par_(Dpar_) | IDEN | accol_{T1accol_}
  - T1-> T2 | EPSILONE
  - T2-> NOMBRET3
  - T3->SEPA_, T2 | EPSILONE

1- Etablir la table d'analyse adéquate.

3- Implémenter l'algorithme de l'analyseur syntaxique.