Projet

Compilation

Fait par :
Roumec Bryan
Teruel Raphaël
le 10/02/2020

SOMMAIRE

Introduction

Ce projet à pour but de simuler le comportement d'un compilateur, notamment en générant des automates. Dans un premier temps un automate fini non déterministe, que nous allons finir par déterminiser et minimiser.
Pour ce faire, nous implémenterons certaine structure et fonctions en c que nous développerons tout au long de ce rapport.

I. Les structures de données

A. Automate fini non déterministe

Pour réaliser cette partie, nous avons implémenter 3 structures :

```
struct edge{
    vertex *nextVertex;
    char word;
};

struct vertex{
    int numSommet;
    edge *tab_edge;
    int nb_edge;
    bool accepteur;
};

struct automate{
    vertex *Vertex;
    int nbr_etats;
};
```

La structure edge permet de représenter les arêtes, elle a deux champs, un de type vertex* qui permet d'avoir un pointeur sur le prochain sommet, et un de type char qui permet de représenter le caractère porter par l'arête.
Nous avons également une structure vertex qui représente les sommets. Dans celle-ci, il y a quatre champs représenté de la manière suivante : un type int qui permet de connaître le numéro du sommet, un type edge* qui va stocker toute les arêtes liés à ce sommet, un type int qui va permettre de connaître le nombre d'arête lié au sommet et enfin un type booléen qui va permettre de savoir si le sommet est accepteur ou non.
Pour finir, nous avons également réalisé une structure automate, qui va nous permettre de représenter l'automate fini non déterministe (ici va représenter la "tête" de l'automate). Dans cette structure, il y a deux champs, un tableau de vertex, et un de type int, nous permettant de connaître le nombre d'état de l'automate. Nous avons choisi d'utiliser un tableau de vertex pour le côté pratique, car en effet, à chaque création d'un automate nous connaissons le nombre de sommet contenu dans celui-ci, il nous semblait donc plus pertinent d'utiliser ce procédé. Notre procédé mêle donc à la fois l'utilisation de tableau et de liste chaînée.

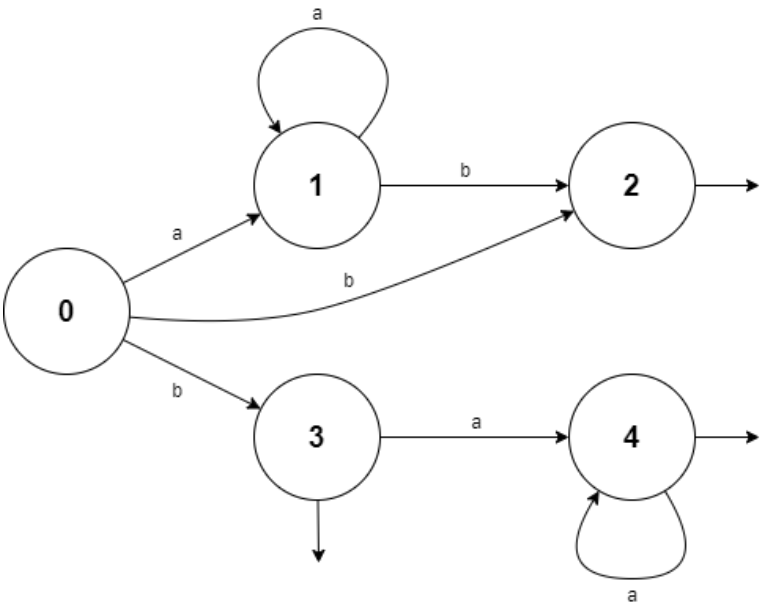
B. Automate fini déterministe

Pour cette partie, nous avons également réalisé trois structures : ces trois structures nous permettent de réaliser le tableau permettant la déterminisation comme vue en td.

```
struct Trans
{
    int *sommet;
    int size;
    char carra;
    Trans *next;
};

struct Enslist
{
    int *sommet;
    int size;
    Enslist *next;
    Trans * First;
    int nb_T;
};

struct List
{
    Enslist *first;
    int nb;
};
```

Comme le nombre final d'état de l'automate n'est pas connu au début du processus de déterminisation nous avons choisis d'utiliser une liste chaînée où chaque maillon représente un état de l'automate fini-déterministe.
Chaque maillon est constitué d'un tableau contenant les indices des etats lui correspondant dans l'automate fini non-déterministe, la taille de ce tableau (size), l'adresse du maillon suivant, et l'adresse du premier maillon d'une liste chaînée représentant la liste des transitions possibles en partant de cette état.
Donc chaque transition est constituée du caractère permettant de passer cette transition et d'un tableau des indices des états cibles lui correspondant dans l'automate fini non-déterministe.

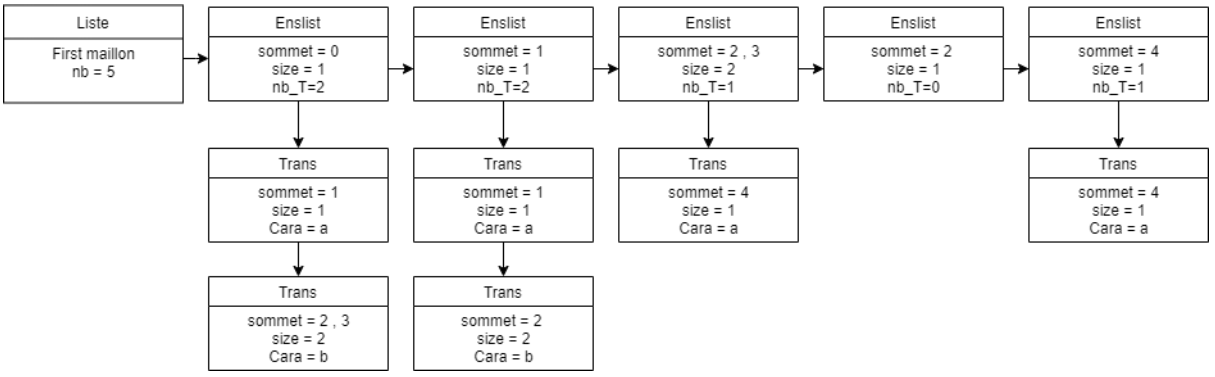
Example:
prenons l'exemple du diagramme du langage a*b|b*a



on a donc le tableau suivant :

	0	1	2,3	2	4
a	1	1	4	/	4
b	2,3	2	/	/	/

ce qui nous donne avec notre structure:



II. Les grandes fonctions

Tout au long du projet, nous avons développé plusieurs fonctions que nous allons détaillés dans cette partie :

```
void addEdge(vertex *a,vertex * b, char carac) {

    edge  E;

    a->nb_edge++;

    E.nextVertex=b;
    E.word=carac;
    a->tab_edge=(edge*)realloc(a->tab_edge,sizeof(edge)*a->nb_edge);
    a->tab_edge[a->nb_edge-1]=E;

}
```

Comme nous le voyons ci-dessus, la fonction prend en entrée trois paramètres : le premier est l'adresse du premier sommet, le deuxième paramètre est l'adresse du deuxième sommet que va relier l'arête, et le dernier paramètre est le caractère que va porter l'arête. On va alors créer la nouvelle arête et ajouter au sommet a l'arête nouvellement créée, puis on va faire pointer le champ "nextVertex" de la nouvelle arête sur l'adresse de b.

```
automate* motVide()
{
    automate * autoFiniNonDeter=malloc(sizeof(automate));
    autoFiniNonDeter->nbr_etats=1;
    autoFiniNonDeter->Vertex=malloc(sizeof(vertex));
    autoFiniNonDeter->Vertex[0].accepteur = TRUE;
    autoFiniNonDeter->Vertex[0].nb_edge = 0;
    autoFiniNonDeter->Vertex[0].tab_edge = NULL;
    return autoFiniNonDeter;
}

automate * langagevide(void){
    automate * new=malloc(sizeof(automate));
    new->nbr_etats=1;
    new->Vertex=malloc(sizeof(vertex));
    new->Vertex[0].accepteur = FALSE;
    new->Vertex[0].nb_edge = 0;
    new->Vertex[0].tab_edge = NULL;
    return new;
}
```

Les deux premières fonctions à implémenter sont les fonctions permettant de reconnaître un mot vide et le langage vide.

Les deux fonctions sont très similaires, en effet elles ne prennent aucun arguments en entrée en produisent un automate en sortie. On commence par allouer la mémoire pour un nouvel automate, puis on le paramètre avec un seul état et on alloue la mémoire pour un seul sommet. La seule différence réside dans le fait que la fonction du langage vide ne comprend aucun accepteur d'où le fait que le sommet de l'automate prend la valeur FALSE tandis que la fonction lié au mot vide tolère un état accepteur, c'est pour cela que le sommet de l'automate prend la valeur TRUE.

```
//renvoie un automate standard reconnaissant le langage composé
//d'un mot d'un caractère passé en paramètre
automate* caracter(char c)
{

    automate * new=malloc(sizeof(automate));
    new->nbr_etats=2;

    new->Vertex=(vertex*)malloc(2*sizeof(vertex));

    new->Vertex[0].accepteur = FALSE;
    new->Vertex[0].nb_edge = 0;
    new->Vertex[0].numSommet = 0;
    new->Vertex[0].tab_edge = NULL;


    new->Vertex[1].accepteur = TRUE;
    new->Vertex[1].nb_edge = 0;
    new->Vertex[1].numSommet = 1;
    new->Vertex[1].tab_edge = NULL;

    addEdge(&new->Vertex[0],&new->Vertex[1],c);



    return new;
}
```

Cette fonction prend en paramètre un caractère et sort un nouvel automate composé du caractère en entrée. Pour commencer, on alloue la place pour notre nouvel automate, puis on augmente son nombre d'état à 2, et ainsi on alloue la place pour deux sommets. On met l'état accepteur à FALSE et aucune arête sortante, tandis que pour le deuxième sommet, son état accepteur est mis à TRUE et son numéro de sommet à 1. On utilise la fonction addEdge entre les deux sommets pour créer l'arête entre les deux.

```
automate * reuninon(automate * a, automate * b){
    int decalage=0;

    automate * new=malloc(sizeof(automate));
    new->nbr_etats= a->nbr_etats+b->nbr_etats -1;
    new->Vertex=(vertex*)malloc(new->nbr_etats * sizeof(vertex));

    for (int i = 0; i <a->nbr_etats ; i++) {
        new->Vertex[i].numSommet = i;
        new->Vertex[i].accepteur = a->Vertex[i].accepteur;
    }
    for (int i = 0; i <a->nbr_etats ; i++) {

        for (int j = 0; j <a->Vertex[i].nb_edge ; j++) {
            addEdge(&new->Vertex[i],&new->Vertex[a->Vertex[i].tab_edge[j].nextVertex->numSommet],a->Vertex[i].tab_edge[j].word);
        }
    }
    decalage=a->nbr_etats;
    new->Vertex[0].accepteur=a->Vertex[0].accepteur | b->Vertex[0].accepteur;

    for (int j = 0; j <b->Vertex[0].nb_edge ; j++) {
        addEdge(&new->Vertex[0],&new->Vertex[b->Vertex[0].tab_edge[j].nextVertex->numSommet+decalage-1],b->Vertex[0].tab_edge[j].word);
    }


    for (int i = decalage; i <b->nbr_etats+decalage-1 ; i++) {

        new->Vertex[i].numSommet = i;
        new->Vertex[i].accepteur = b->Vertex[i-decalage+1].accepteur;

    }
    for (int i = decalage; i <b->nbr_etats+decalage-1 ; i++) {
        for (int j = 0; j <b->Vertex[i-decalage+1].nb_edge ; j++) {
            addEdge(&new->Vertex[i],&new->Vertex[b->Vertex[i-decalage+1].tab_edge[j].nextVertex->numSommet+decalage-1],b->Vertex[i-decalage+1].

        }

    }
    return new;

}
```

Pour cette fonction, on prend en entrée les deux automates qu'on veut réunir et on sort l'automate réuni. On commence comme les autres fonctions par allouer les places en mémoire puis on ajoute l'automate "a" sur notre nouvel automate, ensuite, on ajoute les arêtes du premier état de "b" au premier état de notre nouvel automate, les indices des sommets de notre nouvel automate représentant les sommets de "b" correspondent à tous les sommets d'indices compris entre taille de l'automate "a" inclus et la taille de notre nouvel automate (qui correspond à la taille de l'automate "a" plus la taille de l'automate "b" moins 1 exclus). Après avoir fait tout ça, on peut recopier les valeurs des sommets de l'automate "b" à l'endroit prévu à cette effet dans le nouvel automate, et pour finir, on ajoute toutes les arêtes.

```
automate * concatenation(automate * a, automate * b){
    int decalage=0;
    automate * new=malloc(sizeof(automate));
    new->nbr_etats= a->nbr_etats + (nbaccepteur(a) * (b->nbr_etats -1));
    new->Vertex=(vertex*)malloc(new->nbr_etats * sizeof(vertex));

    for (int i = 0; i <a->nbr_etats ; i++) {
        new->Vertex[i].numSommet = i;
        new->Vertex[i].accepteur = a->Vertex[i].accepteur;
    }
    for (int i = 0; i <a->nbr_etats ; i++) {
        for (int j = 0; j <a->Vertex[i].nb_edge ; j++) {
            addEdge(&new->Vertex[i],&new->Vertex[a->Vertex[i].tab_edge[j].nextVertex->numSommet],a->Vertex[i].tab_edge[j].word);
        }
    }
    decalage=a->nbr_etats;
    for (int j = 0; j <a->nbr_etats ; j++) {
        if(new->Vertex[j].accepteur==TRUE){
            new->Vertex[j].accepteur=b->Vertex[0].accepteur;
            for (int k = 0; k <b->Vertex[0].nb_edge ; k++) {
                addEdge(&new->Vertex[j],&new->Vertex[b->Vertex[0].tab_edge[k].nextVertex->numSommet+decalage-1],b->Vertex[0].tab_edge[k].word);
            }
            for (int i = decalage; i <b->nbr_etats+decalage-1 ; i++) {
                new->Vertex[i].numSommet = i;
                new->Vertex[i].accepteur = b->Vertex[i-decalage+1].accepteur;
            }
            for (int i = decalage; i <b->nbr_etats+decalage-1 ; i++) {
                for (int k = 0; k <b->Vertex[i-decalage+1].nb_edge ; k++) {
                    addEdge(&new->Vertex[i],&new->Vertex[b->Vertex[i-decalage+1].tab_edge[k].nextVertex->numSommet+decalage-1],b->Vertex[i-deca
                }
            }
            decalage+=b->nbr_etats-1;
        }
    }
    return new;
}
```

La fonction concaténation prend deux automates en entrée et crée un nouvel automate résultat de la concaténation des deux automates en entrée. Pour ce faire on ajoute simplement l'automate "b" a tout les etats accepteur de l'automate "a". Donc on crée un nouvel automate de taille |a|+(|a|*(|b|-1)), on y copie l'automate "a", et enfin pour chaque état accepteur on ajoute l'automate "b" à la suite.

```
automate * kleen(automate * a){
    a->Vertex[0].accepteur=TRUE;
    for (int  i= 1; i <a->nbr_etats ; i++) {
        if (a->Vertex[i].accepteur == TRUE) {

            for (int j = 0; j < a->Vertex[0].nb_edge; ++j) {
                addEdge(&a->Vertex[i], &a->Vertex[a->Vertex[0].tab_edge[j].nextVertex->numSommet],
                        a->Vertex[0].tab_edge[j].word);
            }
        }
    }
    return a;
}
```

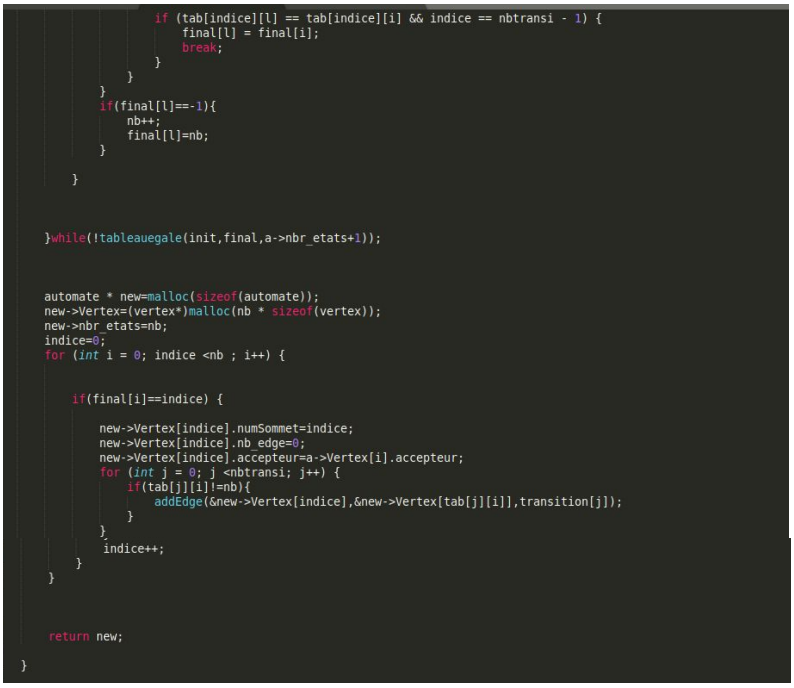
La fonction prend en entrée un automate et va parcourir tous ses sommets, si ils sont accepteurs, alors on va ajouter les arêtes nécessaires pour constituer l'automate demandé et ainsi le retourne alors en sortie.

```
automate * Determinisation(automate * a){
    List * list=malloc(sizeof(List));
    list->nb=1;

    Enslist * e=malloc(sizeof(Enslist));
    e->First=NULL;
    e->next=NULL;
    e->size=1;
    e->sommet=(int*)malloc(e->size*sizeof(int));
    e->sommet[0]=0;

    list->first=e;
    int etat=0;

    while(etat<list->nb) {

        for (int k = 0; k < e->size; ++k) {

            for (int i = 0; i < a->Vertex[e->sommet[k]].nb_edge; i++) {
                e = ajouter_Trans(e, a->Vertex[e->sommet[k]].tab_edge[i].word,
                                  a->Vertex[e->sommet[k]].tab_edge[i].nextVertex->numSommet);

            }
        }
        Trans * t=e->First;
        for (int j = 0; j <e->nb_T ; ++j) {
            if(vExiste(list,t->sommet,t->size)==-1){
               Enslist * e1=malloc(sizeof(Enslist));
               e1->size=t->size;
               e1->sommet=t->sommet;
               e1->next=NULL;
               e1->First=NULL;
               e1->nb_T=0;
               Derniermayon(list)->next=e1;
               list->nb++;
            }
            t=t->next;
        }
        e=e->next;
        etat++;
    }

    automate * new=malloc(sizeof(automate));
    new->Vertex=(vertex*)malloc(list->nb * sizeof(vertex));
    new->nbr_etats=list->nb;
    e=list->first;

    for (int i = 0; i <list->nb ; ++i) {
        new->Vertex[i].numSommet=i;
        new->Vertex[i].accepteur=FALSE;

        for (int j = 0; j <e->size ; ++j) {
            if(a->Vertex[e->sommet[j]].accepteur==TRUE){
                new->Vertex[i].accepteur=TRUE;
                break;

            }
        }
        Trans * t=e->First;
        for (int j = 0; j <e->nb_T ; ++j) {
            addEdge(&new->Vertex[i],&new->Vertex[vExiste(list,t->sommet,t->size)],t->carra);
            t=t->next;
        }
        e=e->next;
    }

    return new;
}
```

Pour cette fonction, on prend en entrée l'automate à déterminiser. On alloue d'abord en mémoire pour notre nouvel automate, en utilisant les structures liés aux automates déterministes.
On initialise une liste de type EnsListe ou chaque maillon représente un état de l'automate déterministe qu'on est entrain de créer. On lui ajoute un premier maillon(ici représenté par la variable "e") prenant les valeurs du premier état de l'automate, le tableau des sommets ne stock donc que l'indice du premier sommet de l'automate a déterminiser. On crée une boucle "tant que" qui s'arrête lorsque la taille de la liste est égale à l'indice du maillon courant. On crée une liste chaînée de type Trans contenant toutes les transitions différentes (caractère définissant la transition différente) des etats sur lesquels le maillon "e" pointe. Si il y a redondance d'une transition on ajoute le numéro de l'etat dans le tableau des états où pointe la transition (e->sommet[i]). La prochaine étape consiste à rajouter la liste obtenu au maillon courant.
Suite à ça, pour chaque transition obtenu, on regarde si le tableau des sommets de cette transition ne correspond pas à un tableau des sommets de la liste des états de notre automate déterministe. Si ce tableau des sommets ne correspond à aucun

autre tableau des sommets, on crée un nouvel état à notre automate et on incrémente sa taille.
Ensuite, on passe au maillon suivant et on recommence jusqu'à ce que la condition de la boucle soit fausse.
Pour finir on transtype cet automate de type Liste en type automate.

```
automate * minimisation(automate *a){

    int * init=(int*)malloc((a->nbr_etats+1)* sizeof(int));
    int * final=(int*)malloc((a->nbr_etats+1)* sizeof(int));
    int nbtransi=0;
    char * transition=(char *)malloc(sizeof(char));
    int indicetrensition=0;
    int nb=0;
    int indice=0;
    for (int i = 0; i <a->nbr_etats ; ++i) {
        if( a->Vertex[i].accepteur==TRUE)
            final[i]=1;
        else
            final[i]=0;

    }
    final[a->nbr_etats]=0;

    transition=transitiondiferent(a,transition);
    nbtransi= nbtransition(a, transition);
    int ** tab = (int**)malloc(nbtransi* sizeof(int*));
    for (int j = 0; j <nbtransi ; ++j) {
        tab[j]=(int*)malloc((a->nbr_etats+1)* sizeof(int ));
    }


    do{
        for (int k = 0; k <a->nbr_etats+1 ; ++k) {
            init[k]=final[k];
        }
         for (int i = 0; i < a->nbr_etats; ++i) {
             for (int j = 0; j < nbtransi; ++j) {
                 indicetrensition= transitionexiste(&a->Vertex[i], transition[j]);
                 if(indicetrensition==-1){
                     tab[j][i]=init[a->nbr_etats];

                 }
                 else{
                     tab[j][i]=init[a->Vertex[i].tab_edge[indicetrensition].nextVertex->numSommet];

                 }

             }
         }
         for (int j = 0; j < nbtransi; ++j) {

                 tab[j][a->nbr_etats]=init[a->nbr_etats];


         }

         for (int m = 1; m <a->nbr_etats+1; ++m) {
             final[m]=-1;
         }
         final[0]=0;
         nb=0;

         for (int l = 1; l < a->nbr_etats+1 ; l++) {

             for (int i = 0; i <l ; i++) {

                 if(init[i]==init[l]) {
                     indice=0;
                     while (tab[indice][l] == tab[indice][i] && indice < nbtransi - 1) {
                         indice++;
                     }
```



```
                    if (tab[indice][l] == tab[indice][i] && indice == nbtransi - 1) {
                        final[l] = final[i];
                        break;
                    }
                }
            }
            if(final[l]==-1){
                nb++;
                final[l]=nb;
            }

        }


    }while(!tableauegale(init,final,a->nbr_etats+1));


    automate * new=malloc(sizeof(automate));
    new->Vertex=(vertex*)malloc(nb * sizeof(vertex));
    new->nbr_etats=nb;
    indice=0;
    for (int i = 0; indice <nb ; i++) {


        if(final[i]==indice) {

            new->Vertex[indice].numSommet=indice;
            new->Vertex[indice].nb_edge=0;
            new->Vertex[indice].accepteur=a->Vertex[i].accepteur;
            for (int j = 0; j <nbtransi; j++) {
                if(tab[j][i]!=nb){
                    addEdge(&new->Vertex[indice],&new->Vertex[tab[j][i]],transition[j]);
                }
            }
             indice++;
        }
    }


    return new;
}
```

On alloue deux tableaux de type int appelé “init” et “final”, leur taille est défini par la taille de l’automate à minimiser + 1, pour prendre en compte l’état mort. On alloue également un tableau de char qui va stocker les poids des transitions de notre automates (cf fonction transitiondifferente) appelé “transition”. Pour finir sur les allocations, on construit un tableau de int dans lequel on va stocker les valeurs qui nous indiquent si la transition nous amène à un état prédéfini dans le tableau “init” (appelé “tab”).
On lance une boucle “tant que” qui s’arrêtera lorsque le tableau initial sera égal au tableau final. Dans cette boucle, on parcourt les transitions de chaque sommet au niveau de la double boucle “for” imbriquée. On va alors vérifier si la transition de l’automate appartient au tableau des transitions et vers quel état il nous emmène. Si la transition n’appartient pas au tableau, on rempli le tableau “tab” avec la valeur de l’état contenu dans la dernière case du tableau init, sinon on remplit avec la case correspondante du tableau “init”. Ensuite, on réinitialise notre tableau “final” avec des valeurs par défaut. Pour se faire on va initialiser la première case du tableau “final” à 0 puis on va comparer chaque case case du tableau avec les cases du tableau “init”, si le contenu est identique, on ne change pas la valeur, sinon on incrémente la valeur “nb” que l’on met dans la case du tableau “final”.
Enfin pour finir, on crée notre nouvel automate en parcourant le tableau “final” et le tableau “tab”. En effet, on parcourt le tableau “final” pour savoir quel numéro de sommet doit être ajouté à l’automate, tandis que la tableau “tab” nous permet d’ajouter les bonnes arêtes.

```c
void affichage(automate a){

    int i;
    int cpt = 1;


    for (int j = 0; j < a.nbr_etats ; j++) {

        for (i = 0; i< a.Vertex[j].nb_edge; i++){
            cpt++;
            if (a.Vertex[j].accepteur)
                couleur("31");
            printf("%d ", a.Vertex[j].numSommet);
            couleur("0");
            printf("--> %c -->",a.Vertex[j].tab_edge[i].word);
            if (a.Vertex[j].tab_edge[i].nextVertex->accepteur)
                couleur("31");
            printf(" %d\n",a.Vertex[j].tab_edge[i].nextVertex->numSommet);
            couleur("0");
        }


    }
    couleur("0");
    printf("\n\n");

}
```

Cette fonction nous sert à afficher nos automates, on prend en entrée l'automate en question, et on affiche ses sommets et ses arêtes. On met la couleur des sommets rouges lorsqu'ils sont accepteur (couleur(31);).

```c
int nbaccepteur(automate * a){
    int count=0;
    for (int i = 0; i <a->nbr_etats ; ++i) {
        if(a->Vertex[i].accepteur==TRUE)
            count++;
    }

    return count;
}
```

Cette fonction nous permet de connaître le nombre d'état accepteur d'un automate donné en entrée.

```c
Enslist * Derniermaillon(List * list){
    Enslist * e=list->first;
    while(e->next!=NULL){
        e=e->next;
    }
    return e;
}
```

Cette fonction nous permet de retourner le dernier maillon d'une liste passé en argument.

```c
int vExiste(List * l, int* e, int nb){
    Enslist  * p;

    p=l->first;

    for (int i = 0; i <l->nb ; i++) {
        if(nb==p->size){
            for (int j = 0; j <nb ; j++) {
                if(e[j]!=p->sommet[j])
                    break;
                else if(j==nb-1 && e[j]==p->sommet[j])
                    return i;
            }
        }

        p=p->next;

    }
    return -1;
}
```

Cette fonction nous retourne l'indice d'un sommet dans une liste.

```c
Enslist * ajouter_Trans(Enslist * e,char car,int voisin){
    Trans * t=malloc(sizeof(Trans));
    Trans * t1=malloc(sizeof(Trans));
    if(e->nb_T==0){
        t->carra=car;
        t->size=1;
        t->sommet=(int*)malloc(sizeof(int));
        t->sommet[0]=voisin;
        t->next=NULL;
        e->First=t;
    }
    else{
        t=e->First;
        while(t->carra!=car && t->next!=NULL){

            t=t->next;
        }
        if(t->carra==car){
           t->size++;
           t->sommet=(int*)realloc(t->sommet, sizeof(int)*t->size);
            t->sommet[t->size-1]=voisin;
            e->nb_T--;
        }
        else{
            t1->carra=car;
            t1->size=1;
            t1->sommet=(int*)malloc(sizeof(int));
            t1->sommet[0]=voisin;
            t1->next=NULL;
            t->next=t1;
        }

    }
    e->nb_T++;
    return e;
}
```

Cette fonction nous permet d'ajouter une transition à notre liste passé en argument si celle-ci n'existe pas encore, sinon la fonction ajoute l'état cible dans le tableau des sommets.

```
bool tableauegale(int * a, int * b,int nb){
    for (int i = 0; i <nb ; ++i) {
        if( a[i]!=b[i]){
            return FALSE;
        }
    }
    return TRUE;
}


int transitionexiste(vertex *v, char c){
    for (int i = 0; i <v->nb_edge ; ++i) {
        if(v->tab_edge[i].word==c)
            return i;
    }
    return -1;
}
```

La fonction “tableauegale” nous permet de savoir si deux tableaux passés en entrée sont égaux, en les comparant élément par élément.
Tandis que la fonction “transitionexiste” nous permet de savoir si une transition existe en partant d’un sommet passé en paramètre.

```
char *transitiondifferente(automate* a, char *car){
    int nb=a->Vertex[0].nb_edge;
    car=malloc(a->Vertex[0].nb_edge*sizeof(char));
    for (int j = 0; j <a->Vertex[0].nb_edge ; ++j) {
        car[j]=a->Vertex[0].tab_edge[j].word;
    }
    for (int i = 1; i <a->nbr_etats ; ++i) {
        for (int j = 0; j <a->Vertex[i].nb_edge ; ++j) {
            if(charintab(car,a->Vertex[i].tab_edge[j].word,nb)==-1){
                nb++;
                car=realloc(car,nb* sizeof(char));
                car[nb-1]=a->Vertex[i].tab_edge[j].word;
            }
        }

    }


    return car;
}
```

Cette fonction prend en entrée un automate et un tableau de caractère et elle nous permet de retourner tous les différents caractères présent dans l’automate.

```c
int nbtransition(automate *a, char *car){
    int nb=0;
    for (int i = 0; i <a->nbr_etats ; ++i) {
        for (int j = 0; j <a->Vertex[i].nb_edge ; ++j) {
            if(charintab(car,a->Vertex[i].tab_edge[j].word,nb)==-1){
                nb++;

            }
        }

    }


    return nb;
}

int charintab(char*car,char c,int nb){
    for (int i = 0; i <nb ; ++i) {
        if(car[i]==c)
            return i;

    }
    return -1;
}
```

La fonction “nbtransition” permet de récupérer le nombre de transitions différentes présent dans l’automate pris en paramètre.
La fonction “charintab”, elle, permet de savoir si un caractère pris en entrée est présent dans un tableau pris en paramètre.

```c
void executionautomate(automate * a, char * mots)
{
    int i=0,existe=-1;
    vertex * courant=&a->Vertex[0];
    printf("%d ",courant->numSommet);
    while(mots[i]!='\0'){

        existe=transitionexiste(courant,mots[i]);
        if(existe!=-1){
            printf("--> %c --> %d ",courant->tab_edge[existe].word,courant->tab_edge[existe].nextVertex->numSommet);
        } else{
            break;
        }
        courant=courant->tab_edge[existe].nextVertex;
        i++;
    }
    if(existe==-1){
        printf("\ncette automate ne verifie pas le mots\n");
    } else if(courant->accepteur==TRUE){
        printf("\nExecution terminer avec succès\n");
    } else if(courant->accepteur==FALSE){
        printf("\nLa dernier letre du mots ne se situe pas sur un etat accepteur\n");
    }

}
```

Cette fonction nous permet de savoir si une chaîne de caractère est présent dans un automate. Pour se faire, on prend en entrée un automate et une chaîne de caractère et on regarde si les lettres de la chaîne de caractère sont présents sur les arêtes de l’automate.

III. Les limites du projet

Un certain nombre de fonctionnalités ont été développé ici, mais pas toute ou pas complètement, pour reproduire un analyseur lexical.

Pour commencer, le sujet limite l'alphabet à la table ASCII, donc on pourrait améliorer notre code en augmentant l'éventail de l'alphabet.
De plus, aucune fonction n'a été demandé pour reconnaître des éléments en particulier comme par exemple la reconnaissance des variables, des fonctions etc…

Pour parler des améliorations des fonctions, le sujet nous limite à la génération d'un automate d'un mot composé d'un seul caractère, on pourrait générer un automate qui reconnaît automatiquement toute une chaîne.
Les fonctions plus avancées sont limités à deux automates en entrée, alors qu'on pourrait améliorer nos fonctions pour par exemple prendre un tableau d'automate en entrée et faire directement la concaténation ou la réunion en une seule fois.

Conclusion

Dans ce projet, nous devions reproduire un analyseur lexical, chose que nous avons réussi à faire pour ce qui était demandé, mais nous avons aussi remarqué que cet analyseur lexical était limité et que plusieurs autres améliorations pouvaient être implémentées.