Projet Compilation

Outils de base pour la construction d'un analyseur lexical en C

Travail réalisé par : Karim Kharraz
Travail encadré par : Monsieur Devan Sohier
Année universitaire : 2019/2020

Introduction

Dans le cadre du module de compilation, en IATIC-4, on nous a demandé d'implémenter les outils de base permettant de construire un analyseur lexical en C.
Le but de ce projet est de construire en premier lieu un automate fini non déterministe par la méthode des automates standard. Puis le déterminiser et le minimiser.
De ce fait, a il fallut en premier temps choisir la structure de données qui va représenter nos automates.
Ensuite, j'ai eu à implémenter la méthode standard de construction d'automate simple permettant la reconnaissance du langage vide, du mot vide et d'un caractère.
Les méthodes de concaténation, de réunion et de fermeture de Kleene ont été implémentées par la suite, afin de permettre la construction d'automates plus évolués.
Une fois que la construction a été faite, j'ai implémenté les méthodes de détermination, et de minimisation d'un automate standard non déterministe.
Ce rapport expliquera d'une manière détaillée tous les conceptes mentionnés précédemment et présentera un jeux de testes concluants.
Un makefile est rendu avec les fichiers sources afin d'exécuter le programme facilement (make et make run).

A- *Structure des données*

Un automate fini non déterministe est défini par le quintuplet suivant A= (Q,∑,Δ,s,F) tels que:
Q: est l'ensemble des états de l'automate.
∑ : est l'alphabet.
Δ: est l'ensemble des transitions.
s: est l'état initial.
F: est l'ensemble des états finaux.

```c
typedef struct Automate {
    int *Q;  //ensemble des etats
    char* symbole; //ensemble des symboles
    transition *delta; //Ensemble des transitions
    int nbTrans; // nombre des transitions
    int initial; // l'etat inital
    int *accepteur;  // Ensemble des etats accepteurs
    int nbEtat; // nombre des etats
    int nbaccept; // nombre des etats accepteurs
} Automate;
```

Figure 1 : Structure d'un Automate.

- Chaque état de l'automate est un entier.L'ensemble de ces états sont stockés dans Q et leurs nombre dans nbEtat.
- L'alphabet du langage est décrit dans le tableau symbole.
- Une transition delta est la représentation du passage de l'automate d'un état k, à un état j via le symbole z tels que (k,j)∈ Q , j !=s et z ∈ symbole .
- Une transition est décrite par la structure suivante.

```c
typedef struct transition //Transition d'un automate
{
    int start; //etat de start
    char symbole; // symbole
    int end; //etat d'end
} transition;
```

Figure 2: Structure d'une transition

- Initial représente l'état initial de l'automate.
- L'ensemble des états accepteur de l'automate sont stockés dans un tableau d'entiers "accepteur".
- Le nombre d'états, de transitions et des états accepteur sont stockés dans des variables de type int.

Remarque: La structure utilisé est polyvalente. Elle peut donc définir un automate déterministe ou pas. L'unique différence entre les deux types d'automates, est que les états d'un automate déterministe sont interprétés comme des caractères grâce à leur code ascii.

B- Méthode des automates standards

La méthode consiste à :

1. Construire les automates standards de base.
2. Etant donné deux automates standard construire les automates standards de la réunion, la concaténation et la fermeture itérative de leurs langages.

1. Construire les automates standards de base

1.1 Fonctions de base

Automate construire_automate_vide()

Cette fonction permet la construction de l'automate vide A définit par :
A= (Q={0},∑=∅,Δ=∅,s=0,F=∅)

```
kharraz@kharraz-Lenovo-ideapad-320S-14IKB:~/Bureau/compilation$ make run
chmod u+x compil
./compil
L'ensemble des etats de l'automate
(0|)
Symboles de l'automate
VIDE
L'ensemble des transitions de l'automate
VIDE
Etat initial : { 0 }
L'ensemble des etats accepteurs de l'automate
VIDE
```

Figure3: Exécution de la fonction construire_automate_vide()

Automate construire_automate_mot_vide()

Cette fonction permet la construction de l'automate A du mot vide définit par:

A= (Q={0},∑=∅,Δ=∅,s=0,F={0})

```
kharraz@kharraz-Lenovo-ideapad-320S-14IKB:~/Bureau/compilation$ make run
chmod u+x compil
./compil
L'ensemble des etats de l'automate
(0|)
Symboles de l'automate
VIDE
L'ensemble des transitions de l'automate
VIDE
Etat initial : { 0 }
L'ensemble des etats accepteurs de l'automate
( 0 | )
```

Figure 4: Exécution de la fonction construire_automate_mot_vide()

Automate construire_automate_symbole(char symbole)

Cette fonction permet la construction de l'automate A composé d'un seul symbole et définit par:

A= (Q={0,1},∑={***a***},Δ={(0, ***a***,1)},s=0,F={1})

```
kharraz@kharraz-Lenovo-ideapad-320S-14IKB:~/Bureau/compilation$ make run
chmod u+x compil
./compil
L'ensemble des etats de l'automate
(0|1|)
Symboles de l'automate
(a|)
L'ensemble des transitions de l'automate
({0,a,1} )
Etat initial : { 0 }
L'ensemble des etats accepteurs de l'automate
( 1 | )
```

Figure 5: Exécution de la fonction construire_auromate_symbole('a')

1.2 Fonctions utiles

Automate initialiser_automate(Automate A, int nbEtat, int delta, int nbsymboles, int F)

Cette fonction permet d'allouer dynamiquement l'espace mémoire nécessaire pour l'automate A. Elle, a été la base de la construction des automates de bases standards, garantissant ainsi une exploitation optimale de la mémoire.

Automate ajouter_transition(Automate A, int start, char symbole, int end,int alloue)

Cette fonction permet d'ajouter une transition au tableau delta de l'automate A et incrémente de 1 son nombre de transitions, si cette transition respecte les règles suivantes:

- Les états de départ et d'arrivée existent ({start, end} ∈ A.Q).
- Le caractère de transition appartient à l'alphabet (symbole ∈ A.symbole).
- Respecte la norme pour un automate standard (end != A.initial).

La variable "alloue" passée en paramètre permet d'autoriser ou non la réallocation de la mémoire dédiée au tableau de transitions delta.
Cette fonction a été lors de l'implémentation des fonctions de manipulations d'automates ce qui explique l'utilité de la variable alloue.

void afficher_automate(Automate A, int dt)

Cette fonction permet d'afficher un automate non déterministe si dt=0 et un automate déterministe si dt=1.

2. *Construire les automates évolués*

Soit A et B deux automates non déterministe non minimaliste.

2.1 Concaténation des deux automates

Automate concat_automate(Automate A, Automate B)

Soit C l'automate résultant de la concaténation de A et B. La construction de C se fait comme suit:

1. Construction de l'alphabet de C: Il s'agit de l'union des deux tableaux de symboles de A et B dans le tableau symbole de C(strcpy(A.symbole,B.symbole)).Pour éviter la redondance des symboles on procède à une vérification de l'existence du symbole avant de l'ajouter dans le tableau de C. Pour cela j'utilise la fonction strchr qui renvoie NULL si le symbole n'existe pas dans C.symbole .
2. Construction de l'ensemble des états de C (C.Q[]): Se sont les états de A et B tels que les indexes de B sont incrémentées du nombre d'états de A(A.nbEtat).
3. Construction des transitions de C (C.delta) : On va garder les mêmes transitions de l'automate A et au niveau de l'automate B on supprime l'état initiale et on relie par le dernier état de A l'ensemble des états de B qui étaient reliés à son état initial.
4. Construction de l'ensemble des états accepteurs de C (C.accepteur) : Si l'état initiale de B est un état accepteur alors on garde les états accepteurs de A et B sinon on garde que les états accepteurs de B.

```
kharraz@kharraz-Lenovo-ideapad-320S-14IKB:~/Bureau/compilation$ make run
chmod u+x compil
./compil
Construction de A
L'ensemble des etats de l'automate
(0|1|)
Symboles de l'automate
(k|)
L'ensemble des transitions de l'automate
({0,k,1} )
Etat initial : { 0 }
L'ensemble des etats accepteurs de l'automate
( 1 | )
Construction de B
L'ensemble des etats de l'automate
(0|1|)
Symboles de l'automate
(a|)
L'ensemble des transitions de l'automate
({0,a,1} )
Etat initial : { 0 }
L'ensemble des etats accepteurs de l'automate
( 1 | )
Construction de C la concatenation de A et B
création de la transition (0,k,1)
création de la transition (1,a,2)
L'ensemble des etats de l'automate
(0|1|2|)
Symboles de l'automate
(k|a|)
L'ensemble des transitions de l'automate
({0,k,1} {1,a,2} )
Etat initial : { 0 }
L'ensemble des etats accepteurs de l'automate
( 2 | )
```

Figure 6: Exécution de la fonction *concat_automate(Automate A, Automate B)*

2.2 Réunion des deux automates

Automate reunion_automate(Automate A, Automate B)

Soit C l'automate résultant de la réunion de A et B. La construction de C se fait comme suit:

1. Addition des variables de grandeur (nbTrans,nbaccept et nbEtat) des deux automates.
2. Initialisation du nouvel automate C via la fonction **initialiser_automate** en lui passant les nouvelles variables de grandeur. => Allocation dynamique de l'espace mémoire.
3. Construction du nouvel alphabet de C (C.symbole[]) avec la même méthode précisé dans la concaténation.
4. Construction du tableau des états de C (C.Q[] : il s'agit d'une simple boucle for de 0 à A.nbEtat + B.nbEtat qui copie les états Q[i] de A et B dans C.Q[i] avec Q[i]=i .
5. Construction des transitions de C : La réunion des transitions se fait en copiant les transitions de A dans C (C.delta[]) puis une deuxiéeme boucle sur les transitions de B.
   Si le start d'une transition de B est égal à la l'état initial de B alors la transition aura comme état de départ l'état initial de A sinon je garde la même transition. L'ajout de A.nbEtat aux transitions de B garantit une bonne indexation.

```
for (int i = 0; i < B.nbTrans; i++) {
    if (B.delta[i].start == B.initial)
        res.delta[k].start = res.initial;
    else
        res.delta[k].start = A.nbEtat + B.delta[i].start - 1;
    res.delta[k].end = A.nbEtat + B.delta[i].end - 1;
    res.delta[k].symbole = B.delta[i].symbole;
    k++;
}
```

Figure 7: Ajout des transitions de B dans C

6. Construction des état accepteurs de C (C.accept[]) : tous les accepteurs de A et B sont recopiés.

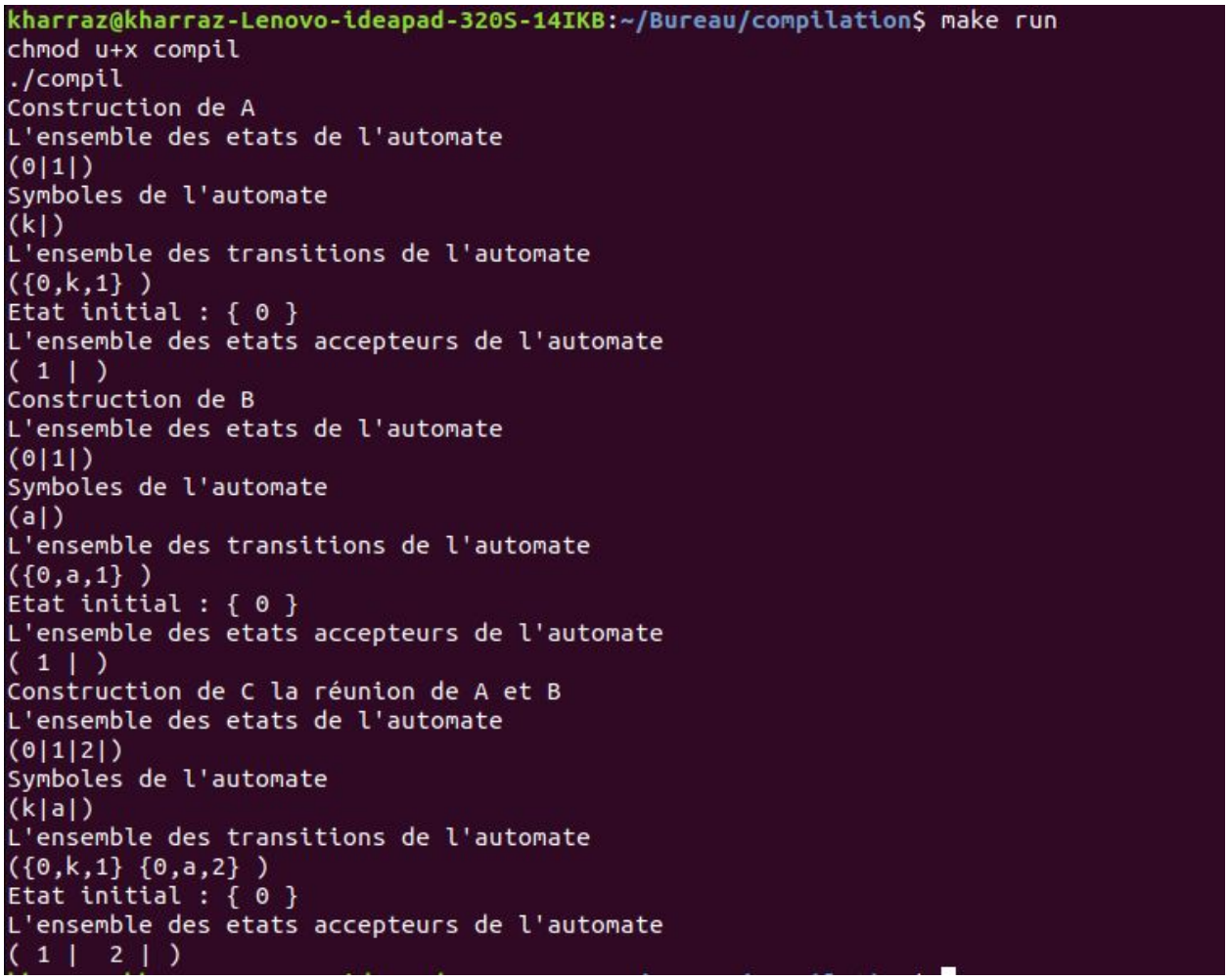

Figure 8: Exécution de la fonction reunion_automate(A,B)

2.3 Fermeture itérative de Kleene

Automate fermeture_it_kleene(Automate A)

La construction de la fermeture itérative de Kleene se fait comme suit:

1. Copie de l'automate A dans un automate temporaire C.
2. Calcul du nombre de transition maximal que peut avoir C (|delta|= A.nbTrans*(A.nbEtat-1).
3. Réallocation de la taille du tableau de transitions (C.delta[]).
4. Pour chaque transition i au départ de l'état intial faire
   pour chaque état accepteur j ≠ état initial faire
   C=***ajouter_transition(C,A.accepteur[k],A.delta[i].symbole,A.delta[i].end,0)***
5. Réallocation de la taille de delta pour ne pas gaspiller la mémoire.

```
kharraz@kharraz-Lenovo-ideapad-320S-14IKB:~/Bureau/compilation$ make run
chmod u+x compil
./compil
creation de la transition (1,k,1)
L'ensemble des etats de l'automate
(0|1|)
Symboles de l'automate
(k|)
L'ensemble des transitions de l'automate
({0,k,1} {1,k,1} )
Etat initial : { 0 }
L'ensemble des etats accepteurs de l'automate
( 1 | )
```

Figure 9: Execution de la fonction *fermeture_it_kleene(Automate A)*

C- *Déterminisation d'un Automate*

"Un automate fini déterministe (AFD), est un automate fini dont les transitions à partir de chaque état sont déterminées de façon unique par le symbole d'entrée."
Afin de déterminiser un automate, j'ai eu recours à une structure intermédiaire :

```
typedef struct deterministe {
    char nom;  //nom de l'etat
    int* etat_nd; // etats non deterministes
    int nbEtat; //nombre d'etats
} deterministe;
```

Figure 10: Structure deterministe

Chaque état déterministe est composé d'un ensemble d'états non déterministe représenté par un tableau etat_nd ,le nombre d'états du sous ensemble et son nom .

Automate determiniser(Automate autom)

Il s'agit de la fonction principale de déterminisation d'un automate. Cette dernière est composée de plusieurs fonctions internes que je vais détailler par la suite.

1. *Fonctionnement global*

1. La fonction ***determiniser(Automate autom):*** créer les variables temporaires qui seront utilisés par la suite afin de créer l'automate déterministe.
   - déterministe* etat_dt : tableau des états déterministes
   - transition* delta, : tableau des transitions de l'automate déterministe
   - dernier_ajoute_dt : Dernier état ajouté dans etat_dt
   - idxdelta : index sur le tableau des transitions .
2. Elle définit le premier état déterministe qui est toujours l'état initial.
3. Fait appel à la fonction ***determiniser_etat*** pour déterminiser tous les autres états:

   tant que (i<= dernier état ajouté)

   void determiniser_etat(Automate Automate, deterministe* etat_dt, transition* delta, int *dernier_ajoute_dt, int *idxdelta, deterministe etat_a_determiner)

   I++;

fin tantque.

La variable dernier_etat_ajouté joue le rôle d'index du tableau déterministe et nous permet aussi d'arrêter la boucle tantque si il n'y a plus d'états à dérterminiser.
4. Crée l'automate déterministe à l'aide des variables définis précédemment et avec la fonction **creer_automate_deterministe**.

2. Fonctions internes

1. void determiniser_etat(Automate A , deterministe etat_dt, transition* delta, int *dernier_ajoute_dt, int *idxdelta, deterministe etat_a_determiner)*

Cette fonction permet de déterminiser les prochains états de etat_dt[dernier_ajoute_dt] selon l'alphabet(A.symbole[]) et ajouter les nouvelles transitions à l'automate.

pour i dans A.symbole[] faire
- tmp <- créer le prochain etat déterministe avec la fonction **prochain_etat_dt**
- vérifier avec la fonction **verfier_etat_dt** si cet état n'est pas dupliqué et l'ajouter au tableau d'état déterministe etat_dt si c'est le cas.
- créer les transitions issues des états ajoutés en suivant les symboles de i.

fin pour

2. int verfier_etat_dt(déterministe etat, déterministe tmp, int *dernier_ajoute_dt)*

Cette fonction vérifie si l'état déterministe tmp est déjà existant dans le tableau d'états et l'ajouter sinon.

3. deterministe prochain_etat_dt(Automate Automate, deterministe etat_a_determiner, char nom, char symbole)

Détermine le prochain état de etat_a_determiner lorsqu'on consomme le symbole.
Elle permet d'assurer la transition entre l'ensemble d'états non déterministes et un état déterministe.

Pour chaque état non déterministe i dans etat_a_determiner faire
Pour chaque transition j dans l'automate faire
si (j.start = i et j.symbole=symbole et j.end ∉ etat_a_determiner.etat_nd)

Ajouter j.end dans etat_a_determiner.etat_nd
finsi
finpour
finpour

4. char nextalpha(déterministe* etat, int dernier_ajoute_dt)

Donne la prochaine lettre de l'alphabet a mettre dans le prochain état déterministe

5. int in_tableau(int* tab, int nbtab, int val)

Vaut 1 si la variable val n'existe pas dans le tableau tab.

6. Automate creer_automate_deterministe(Automate res, Automate a_determiner, transition* delta, int idxdelta, deterministe* dt, int nb_dt)

Cette fonction me permet de transformer les données traitées précédemment et les passer en paramètre à l'automate déterministe .

D- Minimisation d'un automate déterministe

J'ai exploité la technique vu en cours afin de minimiser un automate déterministe.
Voici les différentes étapes de la minimisation:

1. Créer une matrice contenant en première ligne toutes les transitions de l'automate déterministe et en colonne epsilon ainsi que l'ensemble des symboles de l'automate.
Initialiser la matrice au niveau de l'epsilon :
1 si l'etat est acceptant
0 sinon .
Cette initialisation est faite grâce à la fonction ***verifier_accepteur*** qui prend en paramètre un automate et la case à remplir, et retourne soit 1 soit 0 en fonction de l'état à traiter au niveau de la case .

2. Remplir le reste de la matrice par ***remplir_mat*** en fonction de l'état d'arrivé, de départ et du symbole traité.Cette fonction va parcourir la matrice et pour chaque case elle fait appel à la fonction ***valeur_mat*** qui va déterminer la valeur de cette case.

3. Exécuter l'algorithme suivant jusqu'à obtenir les mêmes epsilons dans les deux matrices:
Tantque (il n'y a pas de création de nouvel état) faire
{
Old_mat=New_mat
initialiser New_mat
Remplir New_epsilon
Remplir New_mat
}
Ce traitement fait appel à ces trois fonctions :
● ***nouvel_etat*** : Qui prend en paramètre l'ancienne matrice et la nouvelle matrice et leurs dimensions et retourne 0 s'il n'y a pas eu de nouvel état et 1 sinon.
● ***exist_eps***: qui prend en paramètre les deux matrices Matold et Matnew et leurs dimensions. A l'intérieur de cette fonction j'ai créé une structure qui contient une chaîne de caractères et un la valeur de l'epsilon, en fait la chaîne n'est autre qu'une colonne de Matn et l'epsilon n'est autre que l'epsilon correspondant à cette colonne. Il s'agit donc

d’un parcourir de toutes les chaînes générés à partir de la matrice et leurs comparaison pour créer le nouvel epsilon dans la nouvelle matrice.

● ***remplir_mat*** : La fonction qu’on décrit précédemment pour initialiser la matrice en fonction des états et l’epsilon
=> On a donc une matrice comportant les nouveaux états ainsi que leur symboles.

4. Construction du nouvel automate:

On commence par créer les transitions, pour faire cela on va parcourir la matrice finale et insérer dans un tableau de Transition (alloué) les champs depart, arrive, et alpha.
On fait appel à une fonction ***exist*** qui prend en paramètre le tableau de transition et les 3 champs pour vérifier si cette transition existe dans le but de ne pas avoir de doublons.
Les états du nouvel automate qui sont dans la ligne correspondant à epsilon, à ce niveau faut juste vérifier avant chaque insertion que l’état n’existe pas dans le tableau d’états.
Ensuite, vient le tour des états accepteurs, pour ce cela on fait appel à la fonction ***accepter*** qui prend en paramètre un état, les états accepteurs de l’ancien automate et retourne s’il est accepteur ou non.
Finalement, on récupère les valeurs des autres champs qui sont symbole[], nbTrans, nbEtats, et l'état Initial.

E- Tests et Validation

1. Création de l'automate

J'ai créer l'automate vu en cours puisque je dispose déjà des résultats de sa déterminisation et de sa minimisation.

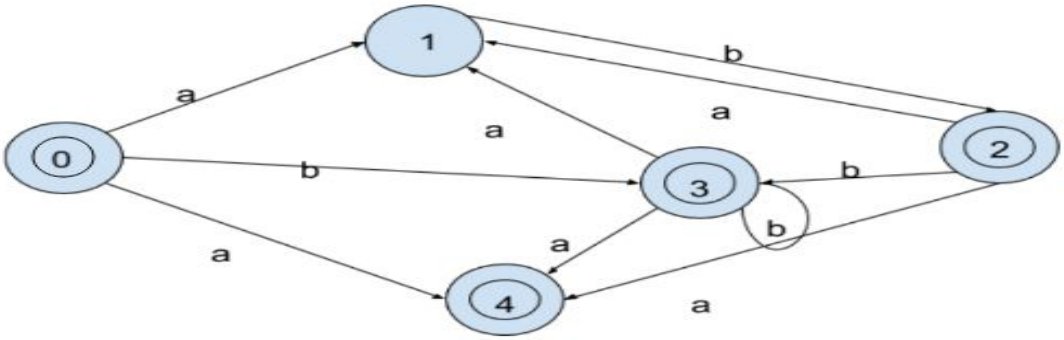

Figure 11: Modélisation graphique del'automate

```
//  Construction de l'automate du cours

    A = automate_caractere('a');
    B = automate_caractere('b');
    A = concat(A, B);
    A = reunion_automate(A, B);
    A.Q[A.nbEtat] = 4;
    A.nbEtat++;
    A.accepteur[A.nbaccept] = 4;
    A.nbaccept++;
    A.accepteur[A.nbaccept] = 0;
    A.nbaccept++;
    A = ajouter_transition(A, 0, 'a', 4, 1);
    A = ajouter_transition(A, 2, 'a', 1, 1);
    A = ajouter_transition(A, 2, 'b', 3, 1);
    A = ajouter_transition(A, 2, 'a', 4, 1);
    A = ajouter_transition(A, 3, 'a', 4, 1);
    A = ajouter_transition(A, 3, 'b', 3, 1);
    A = ajouter_transition(A, 3, 'a', 1, 1);
    afficher_automate(A, 0);
```

Figure 12: Construction de l'automate du graphe

```
kharraz@kharraz-Lenovo-ideapad-320S-14IKB:~/Bureau/compilation$ make run
chmod u+x compil
./compil
création de la transition (0,a,1)
création de la transition (1,b,2)
creation de la transition (0,a,4)
creation de la transition (2,a,1)
creation de la transition (2,b,3)
creation de la transition (2,a,4)
creation de la transition (3,a,4)
creation de la transition (3,b,3)
creation de la transition (3,a,1)
L'ensemble des etats de l'automate
(0|1|2|3|4|)
Symboles de l'automate
(a|b|)
L'ensemble des transitions de l'automate
({0,a,1} {1,b,2} {0,b,3} {0,a,4} {2,a,1} {2,b,3} {2,a,4} {3,a,4} {3,b,3} {3,a,1} )
Etat initial : { 0 }
L'ensemble des etats accepteurs de l'automate
( 2 |  3 |  4 |  0 | )
```

Figure 13: Execution de la construction de l'automate

2. *Déterminisation de l'automate*

```
##determinisation##
#### A  ## nbetat= 1
sous etat non deterministe =0
#### B  ## nbetat= 2
sous etat non deterministe =1
sous etat non deterministe =4
#### C  ## nbetat= 1
sous etat non deterministe =3
#### m  ## nbetat= 0
#### D  ## nbetat= 1
sous etat non deterministe =2
##creation de l'automate deterministe##
A est accepteur
B est accepteur
C est accepteur
D est accepteur
creation de la transition (A,a,B)
creation de la transition (A,b,C)
creation de la transition (B,a,m)
creation de la transition (B,b,D)
creation de la transition (C,a,B)
creation de la transition (C,b,C)
creation de la transition (m,a,m)
creation de la transition (m,b,m)
creation de la transition (D,a,B)
creation de la transition (D,b,C)
L'ensemble des etats de l'automate
(A|B|C|m|D|)
Symboles de l'automate
(a|b|)
L'ensemble des transitions de l'automate
({A,a,B} {A,b,C} {B,a,m} {B,b,D} {C,a,B} {C,b,C} {m,a,m} {m,b,m} {D,a,B} {D,b,C} )
Etat initial : { 0 }
L'ensemble des etats accepteurs de l'automate
( A |  B |  C |  D | )
kharraz@kharraz-Lenovo-ideapad-320S-14IKB:~/Bureau/compilation$
```

Figure 14: Exécution de la déterminisation de l'automate

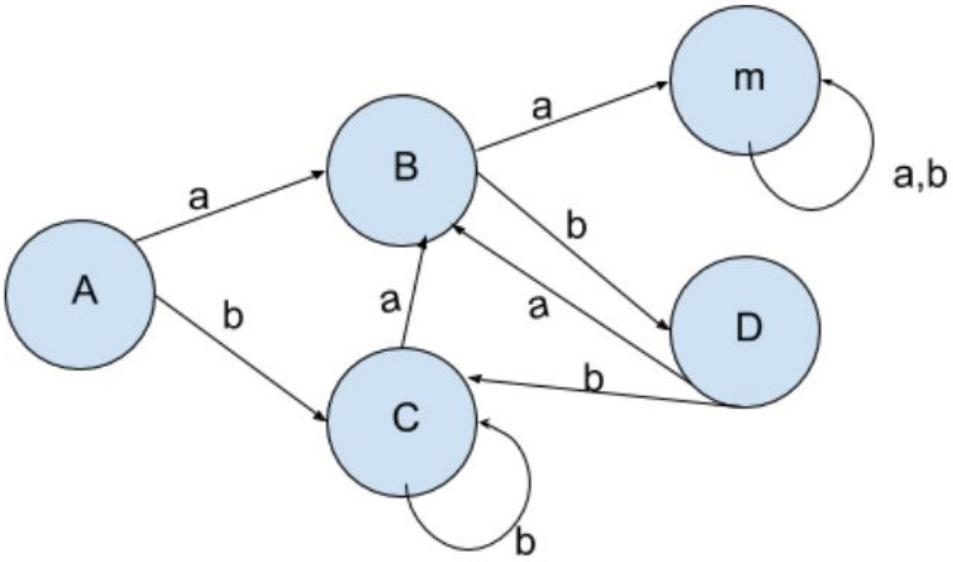

Figure 15: Modélisation graphique du graphe Déterministe.

Comme vous pouvez le remarquer l'état "m" est l'état mort.

```
| ||A||B||C||m||D|
|e||1||1||1||0||1|
|a||1||0||1||0||1|
|b||1||1||1||0||1|


| ||A||B||C||m||D|
|e||0||1||0||2||0|
|a||1||2||1||2||1|
|b||0||0||0||2||0|

| ||A||B||C||m||D|
|e||0||1||0||2||0|
|a||1||2||1||2||1|
|b||0||0||0||2||0|
 Finalisation de la minimisation
Les nouvelles transitions
{0,a,1} {1,a,2} {2,a,2} {0,b,0} {1,b,0} {2,b,2}
L'ensemble des etats de l'automate
(0|1|2|)
Symboles de l'automate
(a|b|)
L'ensemble des transitions de l'automate
({0,a,1} {1,a,2} {2,a,2} {0,b,0} {1,b,0} {2,b,2} )
Etat initial : { 0 }
L'ensemble des etats accepteurs de l'automate
( 0 |  1 | )
kharraz@kharraz-Lenovo-ideapad-320S-14IKB:~/Bureau/compilation$
```

Figure 16: Exécution de la minimisation.

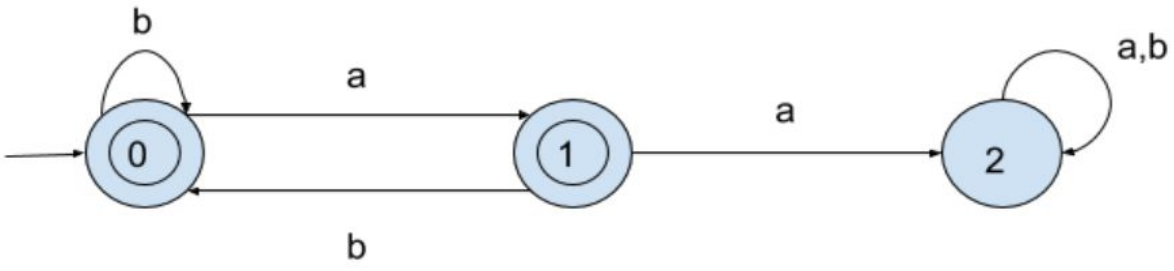

Figure 17: Modélisation graphique de l'automate minimisé.

4. *Exemple d'exécution de mots sur l'automate*

```
#### test acceptation mots ###
MOT A TESTER =abab
(0,a)=1, (1,b)=0, (0,a)=1, (1,b)=0,
mot accepté
MOT A TESTER =abaa
(0,a)=1, (1,b)=0, (0,a)=1, (1,a)=2,
caractere a n'est pas accepte par l'automate
MOT A TESTER =abababbb
(0,a)=1, (1,b)=0, (0,a)=1, (1,b)=0, (0,a)=1, (1,b)=0, (0,b)=0, (0,b)=0,
mot accepté
kharraz@kharraz-Lenovo-ideapad-320S-14IKB:~/Bureau/compilation$
```

Figure 18: Exécution mot sur le graphe minimiser.

F- Conclusion

L'ensemble des fonctionnalitées demandées ont été implémentées et marchent correctement.
Tout au long de ce projet, j'ai été confronté à plusieurs contraintes techniques relatives à la gestion de la mémoire que j'ai pu régler.
J'ai commencé une l'implémentation d'un interpréteur de commande afin de faire passer une expression régulière comme paramètres à l'exécution du programme et construire ainsi des automates d'une manière dynamique mais j'ai pas eu le temps de le finaliser.