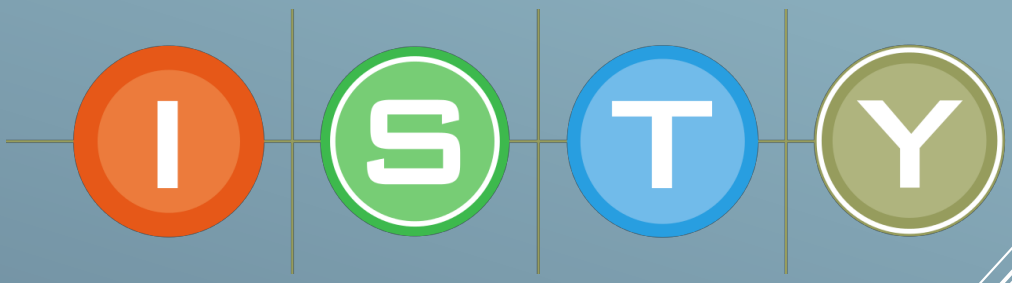


PROJET COMPILATION

Analyseur Lexical en C

BONNAFOUS Camille, LEFEVRE Clément – IATIC 4
Encadrant : M. SOHIER

Table des matières

Introduction

Ce projet nous a été confié dans le cadre du module de Compilation enseigné en IATIC 4 par M. SOHIER.

Le but de ce projet est d'implémenter en langage C les **outils de bases** nécessaires à la réalisation d'un **analyseur lexical**.

L'analyse lexicale, est, avec l'analyse syntaxique et l'analyse sémantique, **une des phases de la compilation**.

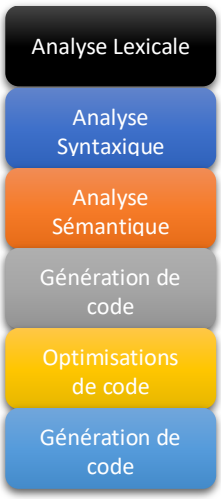


Figure 1 : Chaîne de compilation

Cette phase possède deux principaux objectifs :

1. Vérifier que le texte fourni en entrée est **découpable en mots**, de manière à ce que **chaque mot appartienne au langage utilisé.**
2. **Découper le texte** (code-source) en **tokens** (également appelés unités lexicales)

De manière générale, cette étape vise donc à **abstraire le programme source**, pour **simplifier le travail de l'analyse syntaxique** à venir.

Afin de construire un analyseur lexical, on utilise des **expression régulières** (représentant un langage régulier), que l'on associe à des **automates finis**, que nous allons implémenter au sein de ce projet.

Nous commencerons donc, dans un premier temps, par nous intéresser aux automates finis non-déterministes (AFN) ainsi que leur construction par la méthode des automates standards, avant de traiter les automates finis déterministes (AFD), et les diverses fonctions de manipulations associées (déterminisation, minimisation, simulation).

Le code source associé au projet est compilable via un Makefile, possédant trois principales cibles :

- **make** : compile l'ensemble du projet (AFN, AFD, etc.) et présente quelques exemples. L'exécutable généré se nomme '*lexa*'
- **make afn** : compile uniquement la première partie du projet (relative aux AFN et à la méthode des automates standards). L'exécutable généré se nomme '*afn*'
- **make afd** : compile uniquement la seconde partie du projet (relative aux AFD) L'exécutable généré se nomme '*afd*'

I. Automate Fini Non-déterministe (AFN)

A. Définition

Un automate fini non-déterministe (AFN) est un quintuplet : $\mathcal{A}(Q, \Sigma, \Delta, s, F)$, avec :

- Q : un ensemble fini d'états
- Σ : un alphabet
- Δ : l'ensemble de transitions de $\mathcal{A}$. On a : $\Delta \subset (Q \times \Sigma \times Q)$
- s : l'état initial de $\mathcal{A}$. On a : $s \in Q$
- $F \subset Q$: l'ensemble des états accepteurs de $\mathcal{A}$

Une transition de l'automate $\mathcal{A}$ est un triplet (p, α, q), avec :

- p : l'état de départ
- α : le caractère sur lequel s'effectue la transition
- q : l'état d'arrivée

Cela signifie donc qu'il existe une possibilité de passer de l'état p à l'état q en lisant le caractère α depuis l'état p.

B. Structures de données

A partir de la définition ci-dessus, j'ai donc choisi de mettre en place les structures de données suivantes :

État :

```c
typedef struct {
    int id; // Numéro de l'état
    bool isAcceptor; // Etat accepteur
} state_t;
```

Figure 2 : Structure d'un état

J'ai fait le choix de regrouper dans le même ensemble les états ainsi que les états finaux.
Bien que ce choix soit discutable, puisqu'il augmente certaines complexités, il permet quelques simplifications au sein de plusieurs fonctions.

Transition :

```c
typedef struct {
    state_t* starting; // Etat de départ
    char c; // Caractère sur lequel s'effectue la transition
    state_t* arrival; // Etat d'arrivée
} transition_t;
```

Figure 3 : Structure d'une transition

Comme dans la définition précédente, une transition est composée d'un état de départ, d'un caractère et d'un état d'arrivée.
Utiliser des pointeurs pour les états permet de conserver le lien entre ceux-ci et la transition, ce qui sera utile pour les manipuler.

Caractère de l'alphabet :

Comme précisé dans l'énoncé, un caractère de l'alphabet est représenté par un type *char*.

Ensembles :

Pour représenter les ensembles (d'états, de transitions, de caractères), j'ai fait le choix d'implémenter une structure relativement proche des *vector* du C++, c'est à dire un **tableau dynamique** (ou vecteur), afin de représenter les différents ensembles de la structure.

Cette structure est donc semblable pour l'ensemble des états, l'alphabet, et l'ensemble des transitions, seul le type du champ *data* diffère.
Ainsi, pour un ensemble d'états on a :

```c
typedef struct {
  int size; // Taille utilisée
  int capacity; // Taille totale disponible
  state_t* data; // Données
} states_v;
```

Figure 4 : Structure représentant un ensemble d'états

Avec cette structure, il convient de définir quelques fonctions de manipulations :

```c
void init_states(states_v* states) {
  states->size = 0;
  states->capacity = VECTOR_INITIAL_CAPACITY;
  states->data = malloc(sizeof(state_t) * states->capacity);
}
```

Figure 5 : Initialisation du tableau dynamique d'états

On initialise le tableau, en lui allouant de l'espace mémoire dynamiquement via la fonction *malloc*.

```c
void append_state(states_v* states, state_t value) {
  double_capacity_states(states);
  states->data[states->size++] = value;
}
```

Figure 6 : Ajout d'un élément au tableau

Lorsque l'on ajoute un élément au tableau, il convient dans un premier temps de s'assurer qu'il reste de l'espace disponible via la fonction *double_capacity_states.*

```c
void double_capacity_states(states_v* states) {
  if (states->size >= states->capacity) {
    states->capacity *= 2;
    states->data = realloc(states->data, sizeof(state_t) * states->capacity);
  }
}
```

Figure 7 : Redimensionnement du tableau si nécessaire

Lorsque la taille utilisée atteint la taille totale disponible, de l'espace mémoire est réalloué au tableau, grâce à la fonction *realloc*.

Il faut donc trouver le juste équilibre entre un surplus de mémoire allouée non-utilisée ou un trop grand nombre d'appels à *realloc*.

Enfin, on peut également mettre en place des mutateurs et accesseurs (getters et setters), qui ne sont ici pas nécessaire, le C n'étant pas un langage orienté-objet, mais ceux-ci permettent toutefois de s'assurer de la validité de l'index.

```c
state_t get_state(states_v* states, int index) {
  if (index >= states->size || index < 0) {
    printf("Index %d out of bounds for vector of size %d\n", index, states->size);
    exit(EXIT_FAILURE);
  }
  return states->data[index];
}
```

Figure 8 : Accesseur

```c
void set_state(states_v* states, int index, state_t value) {
  if (index >= states->size || index < 0) {
    printf("Index %d out of bounds for vector of size %d\n", index, states->size);
    exit(EXIT_FAILURE);
  }
  states->data[index] = value;
}
```

Figure 9 : Mutateur

AFN :

On peut donc définir la structure d'un automate fini non-déterministe, composée :

- D'un tableau dynamique d'états (pouvant être accepteurs ou non)
- D'un tableau dynamique de caractères (représentant l'alphabet)
- D'un index désignant l'identifiant de l'état initial. Par convention cette valeur sera fixée à 0 pour l'ensemble du projet.
- D'un tableau dynamique de transitions

```c
typedef struct {
    states_v states; // Ensemble de tous les états
    chars_v alphabet; // Alphabet
    int idInitial; // Etat initial
    transitions_v transitions; // Transitions
} afn_t;
```

Figure 10 : Structure représentant un AFN

Dans la partie suivante, nous détaillerons la construction de ces automates finis non-déterministes par la méthode des automates standards.

II. Méthode des automates standards

A. Définition

Un automate fini non-déterministe (AFN) est standard si aucune de ses transitions ne pointe vers l'état initial, ce qui peut s'exprimer de la manière suivante :

$$\forall (p, \alpha, q) \in \Delta, q \neq s$$

Pour construire des automates standards, il faut donc écrire dans un premier temps des automates standards reconnaissants les langages de base :

- Langage vide,
- Langage composé du seul mot vide,
- Langage composé d'un mot d'un caractère,

Puis, écrire des automates standards reconnaissant la concaténation, l'union, et la fermeture itérative (étoile de Kleene) des langages de deux automates.

B. Automates de base

1. Automate standard reconnaissant le langage vide (∅)

Définition :
Cet automate est composé d'un seul état non-accepteur. On le représente de la manière suivante :

$$Q = \{0\}, \quad \Delta = \emptyset, \quad s = 0, \quad F = \emptyset, \quad \Sigma = \emptyset$$

Fonction associée :

```
afn_t afn_empty_language() {
    afn_t a;

    a.idInitial = 0; // État initial = 0

    init_alphabet(&a.alphabet); // Langage vide

    init_states(&a.states);
    state_t s = {0,false}; // 1 seul état non-accepteur
    append_state(&a.states,s);

    init_transitions(&a.transitions); // Pas de transitions

    return a;
}
```

Figure 2 : Fonction renvoyant un automate reconnaissant le langage vide

Grâce aux fonctions de manipulations de tableaux définies préalablement, on peut mettre en place cet automate de manière assez simple.
Cet automate est composé d'un seul état (initial) non-accepteur, et ne possède aucune transition.

Résultat :
On observe le résultat suivant, qui correspond à la définition.

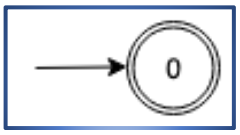
```
** Automate reconnaissant le langage vide **
Etats = {0}
Etat initial = 0
Etats accepteurs = {}
Transitions = {}
Alphabet = {}
```

Figure 3 : Automate reconnaissant le langage vide

2. Automate standard reconnaissant le langage composé du seul mot vide (ϵ)

Définition :
Cet automate est composé d'un seul état accepteur. On le représente de la manière suivante :

$$Q = \{0\}, \quad \Delta = \emptyset, \quad s = 0, \quad F = \{0\}, \quad \Sigma = \{\epsilon\}$$



Fonction associée :

```
afn_t afn_empty_word() {
    afn_t a;

    a.idInitial = 0; // État initial = 0

    init_alphabet(&a.alphabet);
    append_char(&a.alphabet,'\0'); // On peut représenter le mot vide par le caractère '\0'

    init_states(&a.states);
    state_t s = {0,true}; // 1 seul état accepteur
    append_state(&a.states,s);

    init_transitions(&a.transitions); // Pas de transitions

    return a;
}
```

Figure 4 : Fonction renvoyant un automate reconnaissant le langage composé du seul mot vide

J'ai choisi de représenter le mot vide ϵ par le caractère de fin de chaîne '\0'.
Ainsi, on ne réserve aucun caractère spécial pour ce mot.
Pour une meilleure compréhension, j'ai choisi de l'afficher différemment sur la console (*eps.*).

Résultat :
On observe le résultat suivant, qui correspond bien à la définition.

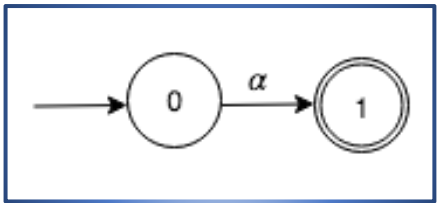
```
** Automate reconnaissant le langage composé du mot vide **
Etats = {0}
Etat initial = 0
Etats accepteurs = {0}
Transitions = {}
Alphabet = {eps.}
```

Figure 5 : Automate reconnaissant le langage composé du seul mot vide

3. Automate standard reconnaissant le langage d'un mot d'un caractère (α)

Définition :
Cet automate est composé de deux états et d'une transition de l'état initial vers l'état accepteur.
On le représente de la manière suivante :

$$Q = \{0, 1\}, \quad \Delta = \{(0, \alpha, 1)\}, \quad s = 0, \quad F = \{1\}, \quad \Sigma = \{\alpha\}$$



Fonction associée :

```
afn_t afn_single_char(char c) {
    afn_t a;

    a.idInitial = 0; // État initial = 0

    init_alphabet(&a.alphabet); // L'alphabet contient le car. passé en paramètre
    append_char(&a.alphabet,c);

    init_states(&a.states); // 2 états
    state_t s1 = {0,false}; // Le premier (initial) non-accepteur
    state_t s2 = {1,true}; // Le 2nd est accepteur
    append_state(&a.states,s1);
    append_state(&a.states,s2);

    init_transitions(&a.transitions); // On a une transition, de 0 par c vers 1
    transition_t tr = {&a.states.data[0],c,&a.states.data[1]};
    append_transition(&a.transitions,tr);

    return a;
}
```

Figure 6 : Fonction renvoyant un automate reconnaissant le langage composé d'un mot d'un caractère

Résultat :
Lorsqu'n exécute cette fonction en lui passant le caractère 'a' en paramètre, on observe le résultat suivant, conforme à la définition :

```
** Automate reconnaissant le langage d'un mot d'un caractère **
Etats = {0,1}
Etat initial = 0
Etats accepteurs = {1}
Transitions = {(0, a, 1)}
Alphabet = {a}
```

Figure 7 : Automate reconnaissant le langage composé du mot 'a'

B. Union, Concaténation et Étoile de Kleene

Pour cette partie, considérons deux automates standards $\mathcal{A}_1 = (Q_1, \Sigma_1, \Delta_1, s_1, F_1)$ et $\mathcal{A}_2 = (Q_2, \Sigma_2, \Delta_2, s_2, F_2)$, et $\mathcal{A} = (Q, \Sigma, \Delta, s, F)$, un automate standard reconnaissant la réunion, la concaténation et la fermeture itérative de leurs langages.

1. Automate standard reconnaissant la réunion de deux langages

On cherche à construire $\mathcal{A}$ tel que $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_1) \cup \mathcal{L}(\mathcal{A}_2)$

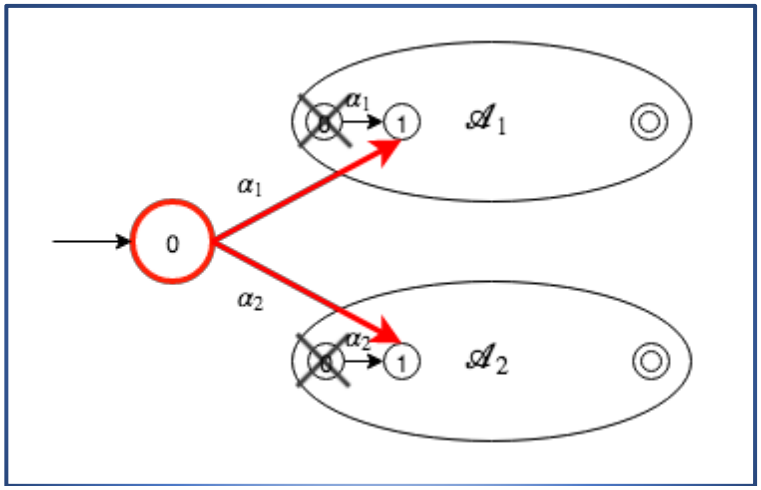


Figure 8 : Schéma représentant un automate reconnaissant l'union de deux langages

Pour commencer, j'ai choisi de partir de l'automate $\mathcal{A}_1$, passé en paramètre :

```
afn_t a = a1;
```

Alphabet :

Concernant l'alphabet de l'automate obtenu, on a :

$\Sigma = \Sigma_1 \cup \Sigma_2$.

On va donc recopier les caractères des deux automates standards en paramètre.

```
// ALPHABET
bool contains = false;
for(int i = 0; i < a2.alphabet.size; i++) {
    for(int j = 0; j < a.alphabet.size; j++) {
        if(a1.alphabet.data[j] == a2.alphabet.data[i]) {
            contains = true;
            break;
        }
    }
    if(!contains)
        append_char(&a.alphabet,get_char(&a2.alphabet,i));
}
```

A l'aide d'une double boucle for, itérant sur les alphabets des deux automates, on vérifie qu'il n'y a pas de doublons, et le cas échéant, on ajoute le caractère au nouvel alphabet.

États :

Concernant les états, on a :

$Q = Q_1 \backslash \{s_1\} \cup Q_2 \backslash \{s_2\} \cup s$

$F = \begin{cases} F_1 \cup F_2 \text{ si } s_1 \notin F_1 \text{et } s_2 \notin F_2 \\ F_1 \backslash \{s_1\} \cup F_2 \backslash \{s_2\} \cup s \text{ sinon} \end{cases}$

En effet, si l'un des précédents états initiaux était accepteur, le nouvel état initial l'est lui-aussi, ce qui peut se traduire par la condition suivante.

```
if(a2.states.data[a2.idInitial].isAcceptor)
    a.states.data[a.idInitial].isAcceptor = true;
```

Il faut noter que nous n'avons pas besoin d'émettre la condition sur $\mathcal{A}_1$, puisque nous avons au préalable dupliqué $\mathcal{A}_1$ en A, ainsi, si l'état initial de $\mathcal{A}_1$ était accepteur, il en sera de même pour $\mathcal{A}$.

Ensuite, on ajoute à $\mathcal{A}$ tous les états de $\mathcal{A}_2$, à l'exception de son état initial.

```
// On ajoute tous les états de a2
for(int i = 0; i < a2.states.size; i++) {
    if(i != a2.idInitial) { // Sauf son état initial
        a2.states.data[i].id = a.states.size;
        append_state(&a.states,a2.states.data[i]);
    }
}
```

Transitions :

Concernant les transitions, on a :

$\Delta = \{(p,\gamma,q) \in \Delta_1 / p \neq s_1\} \cup \{(p,\delta,q) \in \Delta_2 / \ p \neq s_2\}$
$\cup \{(s,\alpha_1,q)/(s_1,\alpha_1,q) \in \Delta_1\} \cup \{(s,\alpha_2,q)/(s_2,\alpha_2,q) \in \Delta_2$

En d'autres termes, toutes les transitions ayant pour états de départs les précédents états initiaux des deux automates sont modifiées, et partent du nouvel état initial s.

```
// TRANSITIONS
for(int i = 0; i < a2.transitions.size; i++) {
    if(a2.transitions.data[i].starting->id == a2.idInitial)
        a2.transitions.data[i].starting = &a.states.data[a.idInitial];
    append_transition(&a.transitions,a2.transitions.data[i]);
}
```

Exemple :

L'automate reconnaissant l'expression régulière $a|b|\epsilon$.

Pour construire cet automate, on construit d'abord l'automate reconnaissant *a*, puis celui reconnaissant *b*, puis l'automate reconnaissant $a|b$, puis l'automate reconnaissant le mot vide ϵ, et enfin l'automate reconnaissant $a|b|\epsilon$.

```
afn_union(afn_union(afn_single_char('a'),afn_single_char('b')),afn_empty_word())
```

On obtient le résultat suivant, qui est bien un automate standard reconnaissant le langage.

```
** Automate reconnaissant l'union de langages (a|b|eps.) **
Etats = {0,1,2}
Etat initial = 0
Etats accepteurs = {0,1,2}
Transitions = {(0, a, 1),(0, b, 2)}
Alphabet = {a,b,eps.}
```

Figure 9 : Automate reconnaissant le langage $a|b|\epsilon$.

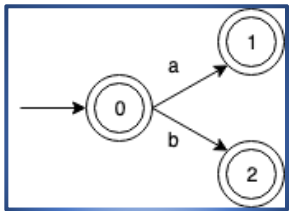


Figure 10 : Représentation graphique de l'automate reconnaissant $a|b|\epsilon$

2. Automate standard reconnaissant la concaténation de deux langages

On cherche à construire $\mathcal{A}$ tel que $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_1).\mathcal{L}(\mathcal{A}_2)$

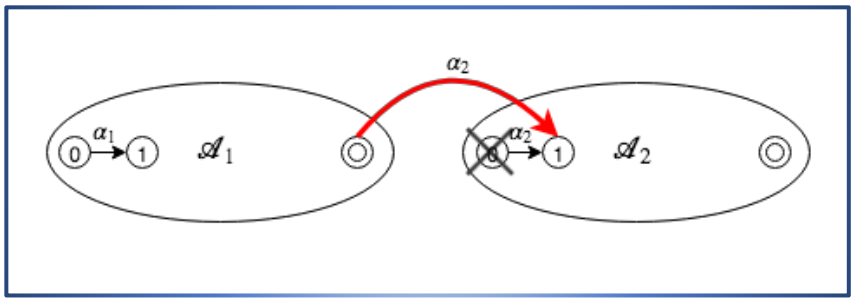


Figure 11 : Schéma représentant un automate reconnaissant la concaténation de deux langages

Ici encore, il paraît judicieux de partir de l'automate $\mathcal{A}_1$ pour obtenir l'automate $\mathcal{A}$.

Alphabet :

De la même manière que pour l'union, concernant l'alphabet de l'automate obtenu, on a :
$\Sigma = \Sigma_1 \cup \Sigma_2$.

Transitions :

Concernant les transitions, on a :
$\Delta = \Delta_1 \cup \{(p, \beta, q) \in \Delta_2 / p \neq s_2\} \cup \{(f_1, \alpha, q) / (s_2, \alpha, q) \in \Delta_2, f_1 \in F_1\}$

```
// TRANSITIONS
// On parcours les transitions de a2
for(int i = 0; i < a2.transitions.size; i++) {
    // Si elles ont pour état de départ l'ancien état initial s2
    if(a2.transitions.data[i].starting->id == a2.idInitial) {
        for(int j = 0; j < a1.states.size; j++) { // Alors on ajoute autant de transitions
            if(a1.states.data[j].isAcceptor) { // qu'il n'y a d'états accepteurs dans a1
                transition_t tr = {&a1.states.data[j],a2.transitions.data[i].c,a2.transitions.data[i].arrival};
                append_transition(&a.transitions,tr);
            }
        }
    }
    else // Sinon, on recopie simplement la transition
        append_transition(&a.transitions,a2.transitions.data[i]);
}
```

On recopie donc toutes les transitions de $\mathcal{A}_1$, ainsi que celles de $\mathcal{A}_2$ n'ayant pas pour état de départ s_2.
Pour chaque transition ayant pour état de départ s_2, on ajoute autant de transition à l'automate $\mathcal{A}$ qu'il n'y a d'états accepteurs dans $\mathcal{A}_1$(c'est à dire $|F_1|$), et l'état de départ de ces transitions devient l'un des états accepteurs de $\mathcal{A}_1$ (le caractère sur lequel s'effectue la transition et l'état d'arrivée ne sont pas modifiés).

États :

Concernant les états, on a :
$Q = Q_1 \cup Q_2 \backslash \{s_2\}$
$F = \begin{cases} F_2 \text{ si } s_2 \notin F_2 \\ F_2 \backslash \{s_2\} \cup F_1 \text{ sinon} \end{cases}$

```
// ETATS
// Si l'état initial de a2 n'est pas accepteur
if(!a2.states.data[a2.idInitial].isAcceptor) {
    for(int i = 0; i < a.states.size; i++) { // Les états finaux de a1 ne sont plus accepteurs
        if(a.states.data[i].isAcceptor)
            a.states.data[i].isAcceptor = false;
    }
}
// On ajoute tous les etats de a2
for(int i = 0; i < a2.states.size; i++) {
    if(i != a2.idInitial) { // Sauf son état initial
        a2.states.data[i].id = a.states.size;
        append_state(&a.states,a2.states.data[i]);
    }
}
```

Si s_2 n'est pas accepteur dans $\mathcal{A}_2$, alors les états accepteurs F_1de $\mathcal{A}_1$ ne sont plus accepteurs dans $\mathcal{A}$.
On recopie ensuite tous les états de $\mathcal{A}_2$ à l'exception de son état initial, qui n'est plus utilisé.

La ligne ci-dessous permet de modifier la valeur de l'identifiant de l'automate, de manière à ne pas avoir de doublons et conserver ainsi une certaine cohérence.

```
a2.states.data[i].id = a.states.size;
```

Comme les transitions pointent sur les états qui les composent, les modifications y sont également prises en compte.

Exemple :
L'automate reconnaissant l'expression régulière $a.b.c$

```
afn_concat(afn_concat(afn_single_char('a'),afn_single_char('b')),afn_single_char('c'))
```

On obtient le résultat suivant, qui est bien un automate standard reconnaissant le langage.

```
** Automate reconnaissant la concaténation de deux langages (a.b.c) **
Etats = {0,1,2,3}
Etat initial = 0
Etats accepteurs = {3}
Transitions = {(0, a, 1),(1, b, 2),(2, c, 3)}
Alphabet = {a,b,c}
```

Figure 12 : Automate reconnaissant le langage $a.b.c$

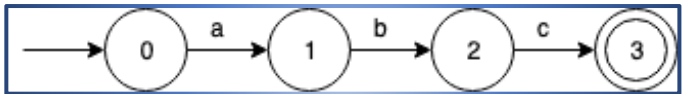


Figure 13 : Représentation graphique de l'automate reconnaissant $a.b.c$

3. Automate standard reconnaissant la fermeture itérative de son langage

On cherche à construire $\mathcal{A}$ tel que $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_1)*$

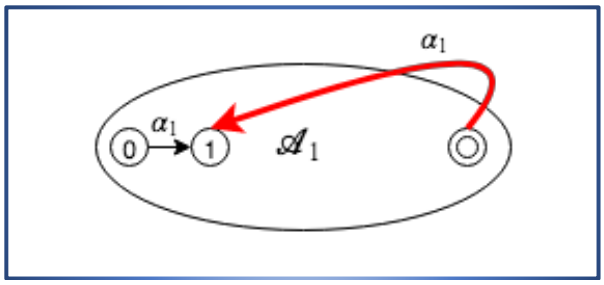


Figure 14 : Schéma d'un automate reconnaissant la fermeture itérative de son langage

États :

$Q = Q_1$
$F = F_1 \cup \{s_1\}$
$s = s_1$

Il n'y a quasiment aucune modification au niveau des états, hormis l'état initial qui devient accepteur.

```
// L'état initial devient accepteur
a.states.data[a.idInitial].isAcceptor = true;
```

Transitions :

Les principales modifications ont lieu sur les transitions, puisqu'on a :

$\Delta = \Delta_1 \cup \{(f, \alpha_1, q)/f \in F_1, (s_1, \alpha_1, q) \in \Delta_1\}$

```
// TRANSITIONS
for(int i = 0; i < a.transitions.size; i++) {
    if(a.transitions.data[i].starting->id == a.idInitial) {
        for(int j = 0; j < a.states.size; j++) {
            if(a.states.data[j].isAcceptor && j != a.idInitial) {
                transition_t tr = {&a.states.data[j],a.transitions.data[i].c,a.transitions.data[i].arrival};
                append_transition(&a.transitions,tr);
            }
        }
    }
}
```

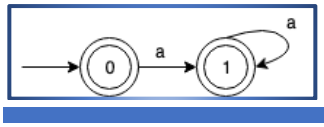
- On parcourt donc l'ensemble des transitions Δ_1 de $\mathcal{A}_1$
- Si la transition δ_i parcourue a pour état de départ l'état initial s_1 de l'automate, alors, pour chaque état accepteur, on ajoute une transition allant de l'état accepteur vers l'état d'arrivée de δ_i par le même caractère que δ_i.

Exemple :

L'automate reconnaissant l'expression régulière $a*$:

```
** Automate reconnaissant la fermeture itérative de Kleene de son langage (a*) **
Etats = {0,1}
Etat initial = 0
Etats accepteurs = {0,1}
Transitions = {(0, a, 1),(1, a, 1)}
Alphabet = {a}
```

*Figure 15 : Automate reconnaissant le langage a**

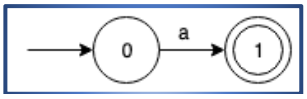


C. Exemples

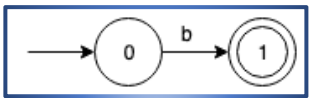
Mots ne contenant pas deux a consécutifs : $(ab|b)^*(a|\epsilon)$.
On va donc, par la méthode des automates standard, construire l'automate reconnaissant ce langage, puis nous exécuterons l'automate obtenu sur divers mots.

Construction :
En utilisant la méthode des automates standards, on procède de la manière suivante :

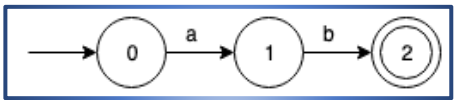
1. a :



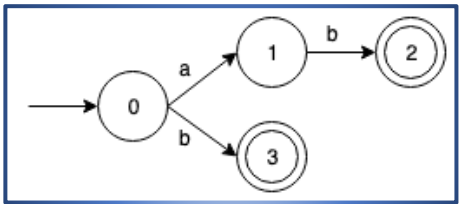
2. b :



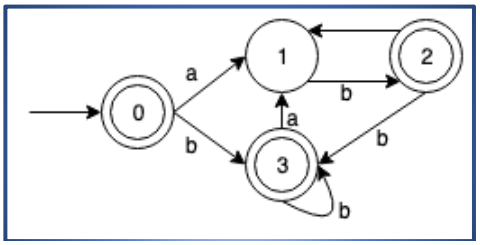
3. ab :



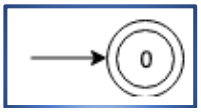
4. $ab|b$:



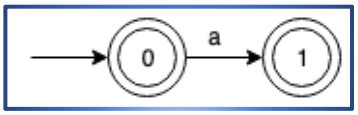
5. $(ab|b)^*$:



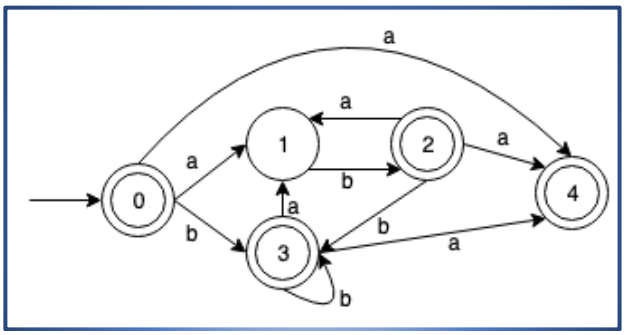
6. ϵ :



7. $a|\epsilon$:



8. $(ab|b)^*\ (a|\epsilon)$:



Lorsque l'on exécute notre programme, on obtient le résultat suivant, qui correspond à la représentation graphique ci-dessus.

```
** Automate reconnaissant (ab|b)*(a|eps.) **
doneEtats = {0,1,2,3,4}
Etat initial = 0
Etats accepteurs = {0,2,3,4}
Transitions = {(0, a, 1),(1, b, 2),(0, b, 3),(2, a, 1),(3, a, 1),(2, b, 3),(3, b, 3),(0, a, 4),(2, a, 4),(3, a, 4)}
Alphabet = {a,b}
```

Exécution sur des mots :

Mot	Statut
ϵ	Accepté
a	Accepté
aa	Refusé
$abbabba$	Accepté
$bbbabab$	Accepté
aba	Refusé

III. Automate Fini Déterministe (AFD)

A. Définition

Un automate fini déterministe (AFD) est un quintuplet : $\mathcal{A}(Q, \Sigma, \delta, s, F)$, avec :

- Q : un ensemble fini d'états
- Σ : un alphabet
- δ : la fonction de de transition de $\mathcal{A}$. On a : $\delta : Q \times \Sigma \rightarrow Q \cup \{\perp\}$
- s : l'état initial de $\mathcal{A}$. On a : $s \in Q$
- $F \subset Q$: l'ensemble des états accepteurs de $\mathcal{A}$

B. Structures de données

A partir de la définition ci-dessus, on peut définir la structure de données suivante pour un automate fini déterministe (AFD) :

```
typedef struct {
    states_v states; // Ensemble de tous les états
    chars_v alphabet; // Alphabet
    int idInitial; // Etat initial
    int** transitionTable; // Table des transitions
} afd_t;
```

Figure 16 : Structure de données représentant un AFD

Cette structure est similaire à celle de l'AFN, la seule différence concerne les transitions.
En effet, les transitions d'un AFD sont déterminées par une **fonction de transition δ**, de manière à ce que les **transitions** pour chaque état, sont **déterminées de façon unique par le symbole d'entrée**.

J'ai donc choisi de représenter cette fonction de transition sous la forme d'une **matrice d'entiers**, de **$|Q|$ lignes et de $|\Sigma|$ colonnes**.

C. Fonctions

1. Simulation d'un AFD

Fonction :
Un AFD peut être simulé par un algorithme prenant en entrée un mot ω et décidant s'il appartient ou non au langage de l'automate.

L'algorithme est le suivant :

Algorithme : $simulation(\omega$: chaîne de caractères)
  $état \leftarrow s$
  Pour i allant de 0 à $|\omega| - 1$ faire :
    $état \leftarrow \delta(état, \omega[i])$
  FinPour
  $Renvoyer\ (état \in F)$

Figure 17 : Algorithme de simulation d'un AFD

Il est de complexité $\theta(|\omega|)$ pour un AFD.

```c
bool simulate(afd_t a, char* word, int sizeOfWord) {
    int idState = a.idInitial;
    int idChar;
    // Parcours de chaque caractère du mot
    for(int i = 0; i < sizeOfWord; i++) {
        idChar = get_alphabet_id(a,word[i]);
        if(idChar == CHAR_NOT_FOUND) // Si un caractère n'est pas présent dans l'alphabet de l'automate
            return false; // Alors le mot n'est pas reconnu par l'automate
        if(a.transitionTable[idState][idChar] == NO_TRANSITION)
            return false;
        idState = a.transitionTable[idState][idChar];
    }

    if(a.states.data[idState].isAcceptor) // Si l'état d'arrivée est accepteur
        return true; // Le mot appartient au langage
    return false;
}
```

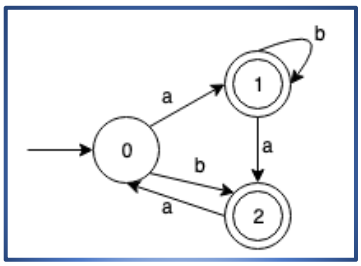
Figure 18 : Fonction de simulation d'un AFD

La fonction implémentée suit donc de manière assez proche l'algorithme précédent.
On parcours donc les caractères du mot en entrée, et pour chaque caractère :

- On vérifie que le caractère est bien présent dans l'alphabet de l'automate, grâce à la fonction *get_alphabet_id()*
  → Si c'est le cas, on renvoie son index (sa position),
  → Sinon on renvoie un code d'erreur correspondant, et on renvoie *faux*, puisque cela implique que l'automate ne reconnaît pas le mot.
- On vérifie ensuite qu'il existe bien une transition dans la table, depuis l'état actuel (représenté par *idState*), par le caractère correspondant (représenté par *idChar*).
  → Si c'est le cas, alors on effectue cette transition
  → Sinon, alors cela signifie que l'automate ne reconnaît pas le mot, on renvoie donc *faux*.

Une fois la totalité du mot parcourue, il faut vérifier que le dernier état dans lequel on se trouve est bien accepteur.
→ Si c'est le cas, alors l'automate a bien reconnu le mot
→ Sinon, il n'a pas reconnu le mot.

Exemple :
Considérons l'AFD suivant :




```
** AFD construit **

Etats = {0,1,2}
Etat initial = 0
Etats accepteurs = {1,2}
Table des transitions:
  || a | b
----------
0 || 1 | 2
1 || 2 | 1
2 || 0 | -

Alphabet = {a,b}
```

On va simuler cet AFD avec les mots suivants :

Mot	Résultat attendu
a	Reconnu
abb	Reconnu
abab	Non-reconnu
abaa	Non-reconnu
abaab	Reconnu
abaabaaba	Reconnu

En exécutant le programme, on obtient le résultat suivant, conforme à nos attentes.

```
Mot à tester : a
L'automate a bien reconnu le mot a
Continuer ? (y/n) y
Mot à tester : abb
L'automate a bien reconnu le mot abb
Continuer ? (y/n) y
Mot à tester : abab
L'automate n'a pas reconnu le mot abab
Continuer ? (y/n) y
Mot à tester : abaa
L'automate n'a pas reconnu le mot abaa
Continuer ? (y/n) y
Mot à tester : abaab
L'automate a bien reconnu le mot abaab
Continuer ? (y/n) y
Mot à tester : abaabaaba
L'automate a bien reconnu le mot abaabaaba
Continuer ? (y/n) n
```

2. Déterminisation d'un AFN

3. Minimisation d'un AFD

Une fois un AFN déterminisé, le nombre d'états de l'AFD engendré peut être important. C'est pourquoi il est judicieux de minimiser cet AFD.
Minimiser un AFD signifie construire le plus petit AFD reconnaissant le même langage.
Pour ce faire, on regroupe des états de l'automate au sein de groupes cohérents.

Pour la minimisation, nous avons mis en place trois fonctions :

- minimize : qui est la fonction principale de la minimisation
- make_bilan : qui permet d'effectuer le bilan
- construct_afd : qui permet, à partir de la table de minimisation remplie, et du précédent AFD, de construire le plus petit AFD reconnaissant le même langage.

Minimize :
Cette fonction prend en paramètre un AFD, et renvoie l'AFD minimal reconnaissant le même langage.

La table de minimisation, est stockée dans un tableau a deux dimensions, de $|\Sigma] + 2$ lignes (les deux lignes supplémentaires correspondent aux deux bilans), et de $|Q|$ états.

```
int minimTable[a.alphabet.size + 2][a.states.size];
```

Comme mentionné précédemment, nous allons grouper les états de l'automate.
Dans un premier temps, on distingue donc deux groupes : les états accepteurs, et les non-accepteurs.

```
// 1er bilan (groupe accepteur - non accepteur)
for(int i = 0; i < a.states.size; i++) {
    if(a.states.data[i].isAcceptor)
        minimTable[0][i] = 1;
    else
        minimTable[0][i] = 0;
}
```

Suite à cela, nous allons mettre en place une boucle *tant que*, qui rebouclera si les deux bilans obtenus sont différents.

```
do { // Tant Que les deux bilans sont différents
    loop = false;
    // Remplissage de la table
    for(int i = 1; i < a.alphabet.size + 1; i++) {
        for(int j = 0; j < a.states.size; j++) {
            destState = a.transitionTable[j][i-1];
            minimTable[i][j] = minimTable[0][destState];
        }
    }

    // Post Bilan
    make_bilan(a.alphabet.size+1,a.states.size,minimTable);

    // Comparaison des bilans
    for(int i = 0; i < a.states.size; i++) {
        if(minimTable[0][i] != minimTable[a.alphabet.size+1][i]) {
            loop = true;
            break;
        }
    }
    if(loop) // Nouveau bilan devient le pré-bilan
        for(int i = 0; i < a.states.size; i++)
            minimTable[0][i] = minimTable[a.alphabet.size+1][i];

} while(loop);
```

On commence donc par remplir, dans la table de minimisation, les groupes atteints par les états lorsque ceux-ci effectuent leurs transitions.

On effectue ensuite le bilan de la table de minimisation, qui sera stocké dans sa dernière ligne, en appelant la fonction *make_bilan.*

Suite à cela, si les deux bilans sont identiques, alors l'AFD est minimal, sinon, le dernier bilan devient le premier, et on reboucle.

Une fois ceci-fait, on renvoie le résultat de la fonction *construct_afd*, qui va construire l'AFD grâce au précédent AFD et à la table de minimalisation.

```
return construct_afd(a,minimTable);
```

Make_bilan :

Cette fonction prend en paramètre une table de minimisation ainsi que ses dimensions.

Elle va permettre de regrouper les états de l'AFD entre eux, en comparant les colonnes de la table de minimisation.
Le bilan obtenu sera affecté à la dernière ligne de la table de minimisation.

	0	1	2
(Pré-bilan)	0	0	1
a	0	1	1
b	0	1	1
(Post-bilan)	**0**	**1**	**2**

Tableau 1 : Exemple de table de minimisation

Ainsi, si on considère la table de minimisation ci-dessus, la fonction *make_bilan* permet d'obtenir la dernière ligne, en gras, en regroupant les colonnes identiques sous le même identifiant, et les colonnes différentes sous un identifiant différent.

Pour cela, on commence par initialiser la ligne du bilan à 0.
On parcourt ensuite les colonnes de la table, en les comparant case par case.
Si les deux colonnes sont identiques, alors on ne modifie pas l'identifiant associé, sinon, on incrémente de 1 cette valeur.

```
// Effectue le second bilan (groupe les colonnes identiques)
void make_bilan(int nbRows, int nbCols, int minimTable[nbRows][nbCols]) {
    bool identic;

    for(int i = 0; i < nbCols; i++)
        minimTable[nbRows][i] = 0;

    printf("Bilan = {");
    for(int i = 0; i < nbCols; i++) {
        for(int j = i; j < nbCols; j++) {
            identic = true;
            for(int k = 0; k < nbRows; k++) {
                if(minimTable[k][i] != minimTable[k][j]) {
                    identic = false;
                    break;
                }
            }
            if(identic)
                minimTable[nbRows][j] = minimTable[nbRows][i];
            else
                minimTable[nbRows][j] = minimTable[nbRows][i] + 1;
        }
        printf("%d,",minimTable[nbRows][i]);
    }
    printf("}\n");
}
```

Construct afd :

Pour construire l'AFD minimal reconnaissant le même langage, à partir du précédent AFD et de la table de minimalisation, nous avons choisi de procéder de la manière suivante :

1. Les alphabets des deux AFDs étant identiques, on les recopie simplement
   `amin.alphabet = a.alphabet;`

2. Concernant les états, on parcourt la ligne du bilan de la table de minimisation, et on ajoute l'état à l'AFD. (Il faut également vérifier que l'AFD minimal ne comporte pas déjà l'état).

```
for(int i = 0; i < a.states.size; i++) {
    bool in = false;
    for(int j = 0; j < amin.states.size; j++) {
        if(amin.states.data[j].id == minimTable[a.alphabet.size+1][i]) {
            in = true;
            break;
        }
    }
    if(!in) {
        state_t s = {minimTable[a.alphabet.size+1][i],false};
        if(a.states.data[i].isAcceptor)
            s.isAcceptor = true;
        append_state(&amin.states,s);
    }
}
```

   Pour vérifier si le nouvel état à ajouter à l'AFD minimal est accepteur, il faut vérifier si l'état du précédent AFD est accepteur. Si c'est le cas, alors le nouvel état l'est aussi.

3. Enfin, grâce aux informations précédentes, on connaît la taille de la nouvelle table de transitions (nombre d'états et taille de l'alphabet).

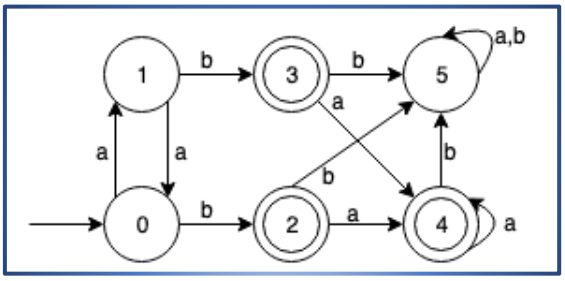
On alloue donc la mémoire nécessaire :

```
amin.transitionTable = (int**) malloc(amin.states.size * sizeof(int*));
for(int i = 0; i < amin.states.size; i++)
    amin.transitionTable[i] = (int*) malloc(amin.alphabet.size * sizeof(int));
```

Avant de la remplir :

```
for(int i = 0; i < a.states.size; i++)
    for(int j = 1; j < amin.alphabet.size + 1; j++)
        amin.transitionTable[minimTable[a.alphabet.size+1][i]][j-1] = minimTable[j][i];
```

Exemple :
Considérons l'AFD suivant (il n'est pas standard) :

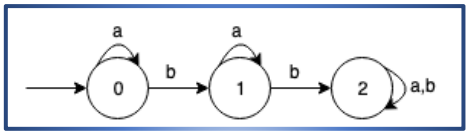


Soit :
$Q = \{0,1,2,3,4,5\}$
$F = \{2,3,4\}$
$s = 0$
$\Sigma = \{a, b\}$

δ	a	b
0	1	2
1	0	3
2	4	5
3	4	5
4	4	5
5	5	5

En le minimisant par notre fonction, on obtient le résultat suivant :

```
Etats = {0,1,2}
Etat initial = 0
Etats accepteurs = {1}
Table des transitions:
  || a | b
----------
0 || 0 | 1
1 || 1 | 2
2 || 2 | 2
```



Lorsque l'on s'intéresse au bilan, on peut voir apparaître les groupes formés :

```
Bilan = {0,0,1,1,1,2,}
```

On remarque donc que les précédents états 0 et 1 correspondent désormais à l'état 0, les précédents 2,3 et 4 ont été regroupés pour former l'état 1, et enfin l'ancien état 5 est resté seul.
➔ Les états indistinguables sont regroupés en un seul état.

IV. Axes d'améliorations

A. Déterminisation

Nous n'avons malheureusement pas réussi à implémenter cette fonction, primordiale à la réalisation d'un analyseur lexical.

En effet, un algorithme de simulation de mot sur AFN est beaucoup plus complexe que sur un AFD ($\theta(|\omega|)$ pour un AFD contre jusqu'à $\theta(|Q|^{|\omega|})$ pour un AFN), et, comme nous l'avons démontré en cours, pour tout AFN, il existe un AFD équivalent, reconnaissant le même langage.

Il peut donc être judicieux de construire un AFN par la méthode des automates standards, puis de le déterminiser, et enfin de le minimiser.
La déterminisation d'AFN occupe donc une place primordiale pour la réalisation d'un analyseur lexical.

B. Structures de données

Une des raisons pour lesquelles nous ne sommes pas parvenus à faire fonctionner ces deux dernières fonctions est très certainement le choix de nos structures de données, qui rendait peu aisée la manipulation d'ensembles d'états, de transitions associées, etc.

Nous sommes également conscients que ce choix a un impact non-négligeable sur la complexité des fonctions mises en place dans ce projet.

Par exemple, pour un AFN, il aurait sans doute été préférable de lier les états à leurs transitions associées, par exemple par le biais d'un tableau à deux dimensions, de type *states_v* (tableau dynamique d'états) indexé par les états et par les caractères de l'alphabet de l'AFN.

C. Complétion d'un AFD

Afin de minimiser un AFD, il conviendrait de s'assurer que celui-ci est complet, c'est à dire que, si, pour tout état p et tout symbole α, il existe un état q, tel que $\delta(p, \alpha) = q$.

Si ce n'est pas le cas, alors il serait nécessaire d'y ajouter un état mort (qu'il faudrait distinguer des autres états), et d'ajouter des transitions vers cet état lorsqu'aucun état q n'est défini dans la table de transitions de l'AFD.

Index des fonctions

A. AFN et méthode des automates standards

Fonctions principales :

1 `afn_t afn_empty_language();`

⇒ Renvoie un automate standard reconnaissant le langage vide.

2 `afn_t afn_empty_word();`

⇒ Renvoie un automate standard reconnaissant le langage composé du seul mot vide.

3 `afn_t afn_single_char(char car);`

⇒ Renvoie un automate standard reconnaissant le langage composé d'un mot d'un caractère, passé en paramètre.

4 `afn_t afn_union(afn_t a1, afn_t a2);`

⇒ Prend deux automates standards en paramètres, et renvoie un automate standard reconnaissant l'union de leurs deux langages.

5 `afn_t afn_concat(afn_t a1, afn_t a2);`

⇒ Prend deux automates standards en paramètres, et renvoie un automate standard reconnaissant la concaténation de leurs deux langages.

6 `afn_t afn_kleene_star(afn_t a);`

⇒ Prend un automate standard en paramètre, et renvoie un automate standard reconnaissant la fermeture itérative de Kleene de son langage.

Fonctions auxiliaires :

7 `void afn_print(afn_t a);`

⇒ Affiche sur la console les données d'un AFN passé en paramètre.

8 `void afn_free(afn_t a);`

⇒ Libère la mémoire allouée pour un AFN passé en paramètre

B. AFD

Fonctions principales :

```
1 bool simulate(afd_t a, char* word, int sizeOfWord);
```

⇒ Prend en paramètre un AFD, et un mot d'une taille donnée. Renvoie vrai si le mot est reconnu par l'AFD, ou faux sinon.

```
2 afd_t minimize(afd_t a);
```

⇒ Prend en paramètre un AFD, et renvoie l'AFD minimal reconnaissant le même langage.

Fonctions auxiliaires :

```
3 int get_alphabet_id(afd_t a, char c);
```

⇒ Prend en paramètre un AFD et une caractère. Renvoie la position de ce caractère dans l'alphabet de l'automate, ou un code d'erreur correspondant sinon (CHAR_NOT_FOUND).

- Appelée dans **simulate**

```
4 void make_bilan(int nbRows, int nbCols, int minimTable[nbRows][nbCols]);
```

⇒ Prend en paramètre une table de minimisation (i.e. une matrice) d'une taille donnée, et regroupe les états indistinguables d'un AFD entre eux. (Stocke le résultat dans la dernière ligne de la table).

- Appelée dans **minimize**

```
5 afd_t construct_afd(afd_t a, int minimTable[a.alphabet.size +
1][a.states.size]);
```

⇒ Prend en paramètre un AFD et une table de minimisation associée. Renvoie l'AFD minimal reconnaissant le même langage.

- Appelée dans **minimize**

```
6 void afd_print(afd_t a);
```

⇒ Affiche sur la console les données d'un AFD passé en paramètre.

```
7 void afd_free(afd_t a);
```

⇒ Libère les ressources allouées pour un AFD passé en paramètre.

```
8 afd_t afd_create_from_input();
```

⇒ Récupère les informations nécessaires à la création d'un AFD auprès de l'utilisateur (sur la console). Renvoie l'AFN créé.