$\mathbb{L}$angages $\mathbb{A}$utomates $\mathbb{N}$on-déterminisme $\mathbb{G}$rammaires $\mathbb{A}$ttibuées et $\mathbb{G}$énératives $\mathbb{E}$xpressions régulières

$\mathbb{C}$orrection $\mathbb{P}$artielle de $\mathbb{P}$rogrammes

*«Ceci n'est pas un cours de Langage C++»*

## 2.1 Comment implanter en C un reconnaisseur de mots ?

Par groupe de 4, envoyez par email votre programme à `michael.perin@imag.fr` et `eric.gascard@imag.fr`. Votre programme doit être conforme à la spécification. Ensuite, pour avoir une chance de gagner : il doit être lisible, le plus clair et le plus simple possible, être facile à maintenir et à faire évoluer, être efficace et enfin être élégant.

**a)** Donnez un programme C qui accepte un mot écrit sur l'alphabet $\{a, b\}$, entré au clavier, s'il appartient au langage $L_1$ des mots formés d'une succession d'un nombre pairs de $a$ entrecoupées d'un nombre quelconque de $b$. Par exemple, les mots $\{\epsilon, b, aa, bb, aab, baa, bbb, aabb, baab, bbaa, bbbb\}$ doivent être reconnu mais pas les mots $\{a, ab, ba, aba, bab, abab, baba\}$.

**b)** Donnez une programme C qui accepte un mot s'il appartient au langage $L_1.\{(ab)^n \mid n \in \mathbb{N}\}$

### 2.1.1 Une première implantation en C d'un reconnaisseur de mots

**Solution du a)** L'automate qui reconnaît $L_1$ est codé sous la forme d'un tableau

| `Aut` | $q_0$ | $q_1$ |
|---|---|---|
| ACCEPT | $true$ | $false$ |
| `'a'` | $q_1$ | $q_0$ |
| `'b'` | $q_0$ | |

où l'état initial est l'état et les états accepteurs sont indiqués par $true$ sur la ligne du symbole réservé ACCEPT. Ainsi $q_i$ est accepteur *si et seulement si* `Aut`$[q_i][\text{ACCEPT}] = true$. L'implantation en C est présenté en Figure 1.

**Solution du b)**
- Soit on construit sur papier l'automate déterministe qui reconnaît le langage $L$ et on le code sous forme d'un tableau `Aut`.
- Soit on code l'automate qui reconnaît $L_1$ sous la forme d'un tableau `Aut1`. On code l'automate qui reconnaît $\{(ab)^n \mid n \in \mathbb{N}\}$ sous la forme d'un tableau `Aut2`.

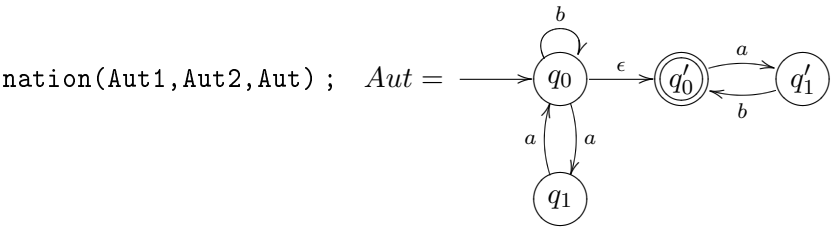
| `Aut2` | $q'_0$ | $q'_1$ |
|---|---|---|
| ACCEPT | $true$ | $false$ |
| `'a'` | $q'_1$ | |
| `'b'` | | $q'_0$ |

Ensuite on construit `Aut` à l'aide des opérations d'une bibliothèque sur les automates.

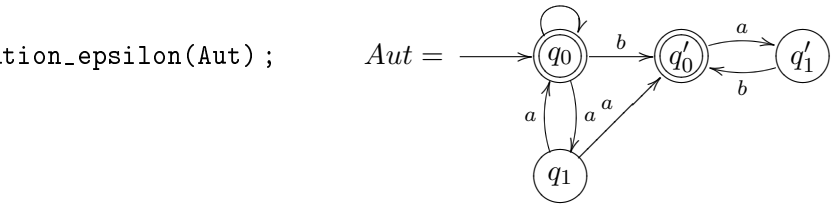
```
#include <stdio.h>

#define ENTER 10 // La touche ENTER porte le numéro 10
#define ACCEPT 'a'-1 // pour que ACCEPT corresponde à la ligne 0 du tableau Aut
#define NE 6 // nombre d'états
#define NS 3 // nombre de symboles de l'alphabet + le symbole réservé ACCEPT

// codage des automates

int Aut1[NS][NE] = { // automate de la question a)
  {1, 0}, // codage des états accepteurs
  {1, 0}, // transitions sur 'a'
  {0,-1} // transitions sur 'b': l'absence de transition est indiqué par -1
};

int Aut[NS][NE] = { // automate de la question b)
  { 1, 0, 1, 0, 1,  0 }, // codage des états accepteurs
  { 1, 2, 3, 2, 5, -1 }, // transitions sur 'a'
  { 2,-1, 4, 4,-1,  4 }  // transitions sur 'b'
};

// Fonctionnement d'un automate (1 ligne !)

int transition(int ec, char c, int Aut[NS][NE]){
 return Aut[ 1+ c-'a' ][ec];
}

// Exemple d'utilisation : reconnaissance d'un mot entré au clavier

int main(){
 char c;
 int q;

 q = 0;
 c = getchar();
 while(c!=ENTER && q>=0){
   q = transition(q,c,Aut);
   c = getchar();
  }

 if ( c==ENTER && transition(q,ACCEPT,Aut)==1 )
  printf(":accept\n");
 else
  printf(":reject\n");

return 0 ;
}
```

FIG. 1 – Implantation en C d'un AEF par un table de transitions

`concatenation(Aut1,Aut2,Aut) ;` $Aut =$

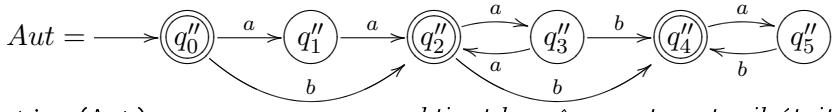



`elimination_epsilon(Aut) ;` $Aut =$




`determinisation(Aut) ;`

$Aut =$




`minimisation(Aut) ;` *on obtient le même automate, il était minimal*

**Détails de la déterminisation**

| `Aut` | $\{q_0\}$ $q''_0$ | $\{q_1\}$ $q''_1$ | $\{q_0, q'_0\}$ $q''_2$ | $\{q_1, q'_1\}$ $q''_3$ | $\{q'_0\}$ $q''_4$ | $\{q'_1\}$ $q''_5$ |
|---|---|---|---|---|---|---|
| ACCEPT | $true$ | $false$ | $true$ | $false$ | $true$ | $false$ |
| `'a'` | $\{q_1\} = q''_1$ | $\{q_0, q'_0\} = q''_2$ | $\{q_1, q'_1\} = q''_3$ | $\{q_0, q'_0\} = q''_2$ | $\{q'_1\} = q''_5$ | |
| `'b'` | $\{q_0, q'_0\} = q''_2$ | | $\{q'_0\} = q''_4$ | $\{q'_0\} = q''_4$ | | $\{q'_0\} = q''_4$ |

On obtient l'automate déterministe minimal `Aut` réprésenté par le tableau :

| `Aut` | $q''_0$ | $q''_1$ | $q''_2$ | $q''_3$ | $q''_4$ | $q''_5$ |
|---|---|---|---|---|---|---|
| ACCEPT | $true$ | $false$ | $true$ | $false$ | $true$ | $false$ |
| `'a'` | $q''_1$ | $q''_2$ | $q''_3$ | $q''_2$ | $q''_5$ | |
| `'b'` | $q''_2$ | | $q''_4$ | $q''_4$ | | $q''_4$ |

On réutilise l'implantation de la Figure 1 avec le tableau `Aut` ainsi obtenu.

## 2.2 Autre implantation en C d'un reconnaisseur de mots

On présente une autre solution qui représente l'ensembles des états courants d'un automate non-déterministe par un vecteur de booléens. La Figure 2 présente l'implantation en `C` de l'exercices (b).

Principe : Considérons un automate non-déterministe à quatre états $\{q_1, \ldots, q_4\}$ et supposons qu'il ait atteint l'ensemble d'états $\{q_1, q_3\}$, on représente cette situation par le vecteur de booléens : $\langle o_1, o_2, o_3, o_4 \rangle = \langle \underbrace{1}_{en\ q_1?}, \underbrace{0}_{en\ q_2?}, \underbrace{1}_{en\ q_3?}, \underbrace{0}_{en\ q_4?} \rangle$ qui indique les états occupés : $o_i = vrai$ si et seulement si l'automate est dans l'état $q_i$. Plusieurs booléens peuvent être à $vrai$ simultanément puisqu'un automate non déterministe peut être dans plusieurs états à la fois.

– À chaque transition de l'automate on met à jour le vecteur de booléens. Soit `carlu` la variable qui correspond à la lettre courante. On passe dans l'état $q_j$ (autrement dit $o_j$ passe à $vrai$) si `carlu` correspond à une transition qui mène en $q_j$ et si on était

L'implantation utilise deux vecteurs de booléens :
– $\langle o'_0, \ldots, o'_N \rangle$ qui représente les états occupés avant la transition
– $\langle o_0, \ldots, o_N \rangle$ qui représente les états courants, c'est-à-dire ceux occupés après la transition
Pour représenter ces vecteurs on utilise un tableau `O` de taille $2 \times N$ et un booléen `c` qui indique sur quelle ligne est le vecteur courant (l'autre ligne correspond au vecteur avant la transition). L'utilisation du booléen `c` évite la recopie des $N$ booléens $o_i$ dans les $o'_i$ avant la mise à jour ; en effet, il suffit de faire `c := !c` pour échanger les valeurs anciennes avec les valeurs courantes.

| | 0 | 1 | ... | $n$ |
|---|---|---|---|---|
| `O[!c]` | $o'_0$ | $o'_1$ | ... | $o'_n$ |
| `O[c]` | $o_0$ | $o_1$ | ... | $o_n$ |

```
#include <stdio.h>

#define ENTER 10
#define N 4

int non_bloque(int O[2][N], int c){
 // l'automate est non bloque si l'un des etats de l'automate est occupé.
 int i, s=0;
 for(i=0 ; i<N ; i++){ s = s || O[c][i] ; }
 return s;
}

int main(){
 int O[2][N] = {0} ;
 int c=0;
 char carlu;

 O[c][0]=1;  // Au depart l'automate occupe l'etat initial q0
 O[c][2] = O[c][2] || O[c][0]; // traitement des epsilon-transitions

 carlu=getchar();

 while( carlu!=ENTER && non_bloque(B,c) ){
   c=!c; // les valeurs courantes deviennent les anciennes valeurs

   // specification de l'automate sous forme d'equations booleennes
   O[c][0] = (O[!c][0] && carlu=='b') || (O[!c][1] && carlu=='a') ;
   O[c][1] = (O[!c][0] && carlu=='a') ;
   O[c][2] = (O[!c][3] && carlu=='b') ;
   O[c][3] = (O[!c][2] && carlu=='a') ;

   O[c][2] = O[c][2] || O[c][0];  // traitement des epsilon-transitions
   carlu = getchar();  // lecture d'une lettre
 }

 if (carlu==ENTER && O[c][2] ) // O[c][2] est l'état accepteur
   { printf(":accept\n") ; }
 else
   { printf(":reject\n") ; }

 return 0;
}
```

FIG. 2 – Implantation en C d'un 4AEF par des équations booléennes

précédemment dans l'état source de la transition.
Par exemple, considérons l'automate de la question (b) et toutes les transitions qui amènent dans l'état $q_0$ : $q_0 \xrightarrow{b} q_0$ et $q_1 \xrightarrow{a} q_0$. La lettre courante `carlu` fait passer dans l'état $q_0$ (autrement dit $o_0$ passe à $vrai$) si on emprunte l'une de ces 2 transitions. Les booléens $o'_0$ et $o'_1$ désignent les valeurs des booléens avant la transition.

$$o_0 := \big( \underbrace{o'_0}_{\text{on était en } q_0} \wedge \texttt{carlu} = b \big) \bigvee \big( \underbrace{o'_1}_{\text{on était en } q_1} \wedge \texttt{carlu} = a \big)$$

- Une $\epsilon$-transition telle que $q_0 \xrightarrow{\epsilon} q_2$ permet d'aller en $q_2$ dès qu'on est en $q_0$ sans lire de lettre. Pour en rendre compte, il faut ajouter l'équation $o_2 := o_2 \vee o_0$. Autrement dit, on conserve le résultat du calcul précédent auquel on ajoute la nouvelle possibilité due à l'$\epsilon$-transition.
- On indique que l'exécution de l'automate commence dans son état initial $q_i$ en assignant au booléen $o_i$ la valuer $vrai$.
- L'automate est bloqué si tous les booléens $o_i$ valent $faux$.
- Le mot est accepté si lorsqu'on a consommé toutes ses lettres, on se trouve sur un état accepteur (c'est-à-dire que l'un des booléens du vecteur qui correspond à un état accepteur vaut $vrai$).

**Avantage / inconvénient de cette implantation**

⊕ Cette implantation permet d'exécuter un automate non-déterministe – sans qu'on ait besoin de le déterminiser, ni d'éliminer les $\epsilon$-transitions. Les ensembles d'états sont codés par le vecteur de booléens $\langle o_1, \ldots, o_n \rangle$ où le booléen $o_i$ indique si l'automate est dans l'état $q_i$.

$n$ (ou $2n$) booléens suffisent pour coder l'état d'un automate non-déterministe tandis qu'avec l'implantation précédente il faut déterminiser l'automate et l'on sait que cette opération peut conduire à un automate à $2^n$ états.

| $n$ | 5 | 10 | 15 | 20 | 25 | 30 | 35 | ... |
|---|---|---|---|---|---|---|---|---|
| $2^n$ | 32 | 1024 | 32768 | 1048576 | 33554432 | 1073741824 | 34359738368 | ... |

⊖ À chaque transition de l'automate on lit une lettre et on met à jour le vecteur de booléens, ce qui nécessite $n$ (ou $2n$) affectation, tandis que la précédente implantation nécessitait seulement une affectation.

### 2.2.1 Conclusion : quelle implantation choisir ?

Considérons un automate non-déterministe à $N$ états.
- Si sa version déterministe comporte un petit nombre d'états (de l'ordre de $N$), on préfera la première implantation pour son efficacité (1 affectation par transition) malgré sa représentation (consomatrice de mémoire).
- Si la version déterministe comporte un nombre d'états très supérieure à $N$ et que la mémoire disponible est limitée, on préfera la seconde implantation pour sa représentation compacte ($N$ booléens) malgré son manque d'efficacité ($N$ affectations par transition).