LANGAGES ET AUTOMATES

GLIN502

TP1 : Mots et langages **(programmes en C)**

TP2 : Langages algébriques **(avec Jflap)**

TP3 : Arbres et chaînes de dérivation **(programmes en C)**

TP4 : Introduction aux automates **(avec Jflap)**

TP5 : Reconnaissance d'un mot par un automate **(programmes en C)**

TP6 : Déterminisation et minimisation d'automates **(avec Jflap)**

TP 1 : Mots et langages

Vous utiliserez le `langage C` et les fichiers `mot.h` et `mot.c` disponibles dans l'espace pédagogique (`FLIN509`, puis `Documents Et Liens`), et aller dans le dossier `TP` puis `TP1MotsEtLangages`). On y trouve aussi le fichier `testMot.c` qui contient un "main" permettant de tester les fonctionalités de `mot.c`. Le makefile est donné à titre d'exemple mais il est conseillé de l'utiliser en le copiant dans son répertoire. Vous l'adapterez et vous n'aurez plus qu'à taper "make monprojet" pour lancer les compilations et l'édition de lien.

Pour répondre aux questions, remplissez la feuille de TP.

Attention :

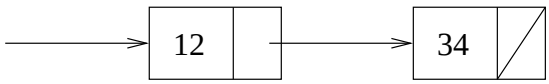
1. quand on vous demande une spécification de fonction, on la veut en terme de langage : les mots *liste* ou *élément* sont interdits, il faut utiliser les mots *mot* ou *langage*.
2. il ne suffit évidement pas de répondre *la fonction renvoie un langage*, il faut expliciter une relation entre les données et le résultat.

Mot

Un mot sera implanté par une chaîne de caractères (le mot vide sera implanté la chaîne vide "").

Langage

Un langage sera une liste de mots (le langage vide sera la liste vide).
Par exemple le langage $\{12,\ 34\}$ peut s'implanter par la liste :



Exercice 1

Donner les spécifications des fonctions :
- `motMiroir`
- `motConcatMot`

Exercice 2

Implanter les fonctions :

- `langage motConcatLangage(mot m, langage L) ;`
  qui renvoie la concaténation des deux langages $\{m\}$ et de L
- `langage langageConcatLangage(langage L1, langage L2) ;`
  qui renvoie le langage concaténation de L1 et de L2.

Exercice 3

Prévoyez puis vérifiez les résultats de l'appel `langageConcatLangage(X,Y)` pour :

- $X = \{a,\ ab,\ bb\}$, $Y = \{ba,\ bb\}$
- $X = \{\epsilon\}$, $Y = \{ba,\ bb\}$
- $X = \{\}$, $Y = \{ba,\ bb\}$

Calculer X^2 et X^3 pour $X = \{aa,\ ab,\ ba,\ bb\}$. Devinez X^* et X^+.

Exercice 4

Implanter les fonctions suivantes :

- `Langage motPrefixes(mot m)` qui renvoie le langage composé de tous les préfixes de `m`
- `Langage langagePrefixes(langage l)` qui renvoie le langage composé de tous les préfixes de tous les mots du langage `l`

Exercice 5

Ecrire et tester les fonctions identiques pour les suffixes.

Exercice 6

Soit V un alphabet. Définissons l'ensemble des *mélanges* de deux mots de V^* : u et v désignent deux mots quelconques de V^*, x et y deux lettres quelconques de V ; attention on peut avoir $u = v$ et on peut avoir $x = y$. Ecrire la fonction $\mathcal{M}$ définie par :

- $\mathcal{M}(\epsilon, v) = \{v\}$
- $\mathcal{M}(u, \epsilon) = \{u\}$
- $\mathcal{M}(xu, yv) = \{x\}\mathcal{M}(u, yv) \cup \{y\}\mathcal{M}(xu, v)$
- soit $a,\ b,\ c,\ d,\ e$ cinq lettres distinctes de V. Combien d'éléments possède $\mathcal{M}(abc, de)$?
- Généraliser le résultat précédent (mots de longueurs quelconques, lettres pas forcément différentes) : on donnera un majorant μ tel que $|\mathcal{M}(u, v)| \leq \mu(|u|, |v|)$.

Exercice 7

- Pour n donné, générer tous les mots d'au plus $2n$ lettres ayant autant de a que de b
- Pour n donné, générer tous les mots d'au plus n lettres dont tous les préfixes ont plus de a que de b.

TP 1 Annexe

```
/** mot est une chaine non mutable : interdiction de modifier un mot
 * on en cree toujours un nouveau
 */
typedef char* mot;
typedef char symbole;

/** toutes les fonctions de base sur mot commencent par motMajuscule...
 */

/** retourne un mot vide */
mot motCreer();
/** concatene un symbole et un mot pour former un nouveau mot
 * retourne le mot c.m
 * ne modifie pas le mot m
 */
mot symboleConcatMot(symbole c, mot m);
mot motConcatSymbole(mot m, symbole c);
mot motCopier(mot m);
/** retourne vrai ssi m est vide
 */
int motEstVide(mot m);
/** retourne ce mot sans sa premiere lettre
 * prerequis : ce mot n'est pas vide
 */
mot motQueue(mot m);
/** retourne la premiere lettre de ce mot
 * prerequis : ce mot n'est pas vide
 */
symbole motTete(mot m);
mot motConcatMot(mot un, mot deux);
mot motMiroir(mot m);
int motEgalMot(mot un, mot deux);
```

```
struct cellule; /* composee d'un mot et d'un ptr */
typedef struct cellule* langage; /* un langage est un ptr sur le 1er mot */
struct cellule {
  mot m;
  langage suiv;
};
/** creer un langage vide : une liste vide.
 */
langage langageCreer();
int langageEstVide(langage l);
int langageContientMot(langage l, mot m);
/** retourne un nouveau langage contenant m et l
 */
langage langageAjouterMot(langage l, mot m);
/** retourne le premier mot du langage
 * prerequis : l est non vide
 */
mot langagePremierMot(langage l);
/** retourne le langage prive de son premier mot
 * prerequis : l est non vide
 */
langage langageReste(langage l);
void langageAfficher(langage l);
langage langageUnionLangage(langage l1, langage l2);
/** retourne tous les prefixes du mot m
 */
langage motPrefixes(mot m);
```

FEUILLE DE TP 1 : Mots et langages

Nom :

Mot

Exercice 1
La fonction `motMiroir` prend en entrée

et renvoie

La fonction `motConcatMot` prend en entrée

et renvoie

Langage

Exercice 2
La fonction `motConcatLangage` prend en entrée

et renvoie

La fonction `langageConcLangage` prend en entrée

et renvoie

Nom :

Exercice 3
Ne trichez pas : remplissez les ligne suivantes avant de tester le résultat :

- pour $X = \{a,\ ab,\ bb\}, Y = \{ba,\ bb\}$ `langageConcatLangage(X,Y)` renvoie

- pour $X = \{\epsilon\}, Y = \{ba,\ bb\}$ `langageConcatLangage(X,Y)` renvoie

- pour $X = \{\}, Y = \{ba,\ bb\}$ `langageConcatLangage(X,Y)` renvoie

pour $X = \{aa,\ ab,\ ba,\ bb\}$

- X^2 est obtenu par l'appel
  et vaut

- X^3 est obtenu par l'appel
  et vaut

- X^* vaut

- X^+ vaut

Exercice 4
La fonction `motPrefixes` prend en entrée

et renvoie

La fonction `langagePrefixes` prend en entrée

et renvoie

Nom :

Exercice 5

```
langage motSuffixes(mot m) {...

   return ... }
```

a été testée sur $m =$

pour donner le résultat :

```
langage langageSuffixes(langage L) {...

return ... }
```

a été testée sur $L =$

pour donner le résultat :

TP 2 : langages algébriques

Plusieurs des TP suivants utilisent "jflap". (commande : jflap&). Pour chacun de ces TP, vous irez chercher les fichiers à charger dans l'espace pédagogique (FLIN509 → Documents Et Liens → dossier TP, sous dossier correspondant au TP.

Grammaire

Remarque : Par défaut, jflap utilise la lettre λ au lieu de notre ϵ. Pour changer la lettre par défaut, modifiez l'item du menu "preferences".

Exercice 1

On va tester la grammaire du fichier `GramDick1.jff`. Pour rentrer un mot dont on veut construire l'arbre de dérivation : `Input` → `Brute Force Parse`, puis rentrer le mot à tester, dans l'exemple *aaabbb* et enfin `Start`. On vous indique que le mot est accepté, et vous voyez alors s'afficher la racine de l'arbre de dérivation.Avec des applications successives de `Step`, vous développez l'arbre jusqu'à l'obtenir en entier. En recommançant à partir de `Input`, vous pouvez tester qu'un mot comme *aabbb* est refusé.

Si vous cliquez sur la flèche à droite de *non inversed tree*, vous voyez apparaître l'option *derivation table*. Cliquez d'abord sur *start*, sélectionnez ensuite cette option puis appuyez de façon répétée sur *step*. A quoi correspond la réponse de "jflap" ?

Exercice 2

Soit la grammaire du fichier `GramDick2.jff` et l'entrée *abaabb*. Pouvez vous deviner l'arbre de dérivation avant de le vérifier ?

Exercice 3

Même question pour la grammaire du fichier `GramDick3.jff`.

Exercice 4

Sur l'entrée *abab*, même question pour les grammaires des fichiers `GramDick4.jff` et `GramDick5.jff` ?

Exercice 5

Éditer une nouvelle grammaire avec *jflap*. Cliquez sur la première ligne et tapez *S*. Puis cliquez dans une case à droite sur cette ligne et saissisez *cTd*. Rentrez successivement dans les lignes suivantes les règles :

$T \to eTf$
$T \to eUf$
$U \to gUh$
$U \to gVh$
$V \to \epsilon$

Remarque : pour obtenir λ (ou ϵ), il suffit de ne pas remplir la partie droite de la production.

Tester que le mot *cefd* n'est pas généré par la grammaire. Trouvez alors le mot le plus court qui soit généré par cette grammaire. Sélectionnez*File* → *Dismiss Tab* puis rajoutez une seule règle telle que le mot *cefd* soit maintenant généré par la grammaire.

Exercice 6

Étude de l'importance de l'ordre dans lequel les règles sont rentrées.

Soient les quatre règles :

1. $S \to SS$
2. $S \to aSbS$
3. $S \to aSb$
4. $S \to \epsilon$

Vous laisserez toujours la quatrième règle en dernier, mais fabriquez les 6 grammaires qui correspondent aux six ordres possibles sur les 3 premières règles. Puis testez chacune de ces grammaires sur le mot *abab*. Le but est de deviner à l'avance l'arbre de dérivation que fournira la grammaire que vous êtes en train de tester.

Exercice 7

Fabriquez et testez la grammaire qui engendre l'ensemble des mots construits avec l'alphabet $\{a, b\}$, où tous les a sont avant le premier b, et qui ont autant de a que de b.

Exercice 8

Fabriquez et testez la grammaire qui engendre l'ensemble des mots palindromes sur l'alphabet $\{a, b\}$.

Exercice 9

Fabriquez et testez la grammaire qui engendre l'ensemble des mots palindromes sur l'alphabet $\{a, b, c\}$.

Exercice 10

Fabriquez et testez la grammaire qui engendre l'ensemble des mots qui sont la concaténation de deux mots palindromes sur l'alphabet $\{a, b\}$.

Exercice 11

Fabriquez et testez la grammaire qui engendre l'ensemble des mots qui sont la concaténation de trois mots palindromes sur l'alphabet $\{a, b\}$.

Exercice 12

On veut tester la grammaire qui engendre l'ensemble des mots qui sont concaténation de 1 ou plusieurs mots palindromes sur $\{a, b\}$, chacun de ces mots palindromes étant composé d'au moins 2 lettres. Cette grammaire se trouve dans le fichier *ConcatMultiple.jff*. Testez la d'abord sur *aba*, puis après *File* → *Dismiss Tab*, testez la sur *abbabbabbabb* Vous voyez que *jflap* peine à trouver la solution. Cliquez sur *pause*, puis *File* → *Dismiss Tab*. Nous allons apprendre à aider *jflap*. Cliquez sur *input* → *User Control Parse* et rentrez *abbabbabbabb* dans *input*, puis cliquez sur *start*.

1. Cliquez maintenant sur la première règle et sur le S qui est affiché dans le panneau du bas, puis sur *step*.
2. Cliquez maintenant sur la deuxième règle et sur le premier S qui est affiché dans le panneau du bas, puis sur *step*.
3. Cliquez maintenant sur la troisime règle et sur le premier S qui est affiché dans le panneau du bas, puis sur *step*.
4. Si vous cliquez sur le premier S qui est affiché dans le panneau du bas, sur quelle règle allez vous cliquer avant de cliquer sur *step*.
5. Finissez de diriger *jflap* pour obtenir la dérivation complète, en s'electionnant systématiquement le S le plus à gauche (on parle de dérivation gauche. Trouvez de même la dérivation droite de *aaabbabbaaababbbababb*.

Exercice 13

Fabriquez la grammaire de *Lukasiewicz*, dont les règles de production sont $S \rightarrow aSS \mid b$.

Trouvez (et testez) les quatre mots de longueur au plus 6 qui appartiennent au langage généré par cette grammaire. Trouvez un mot de longueur 15 dont l'arbre de dérivation soit le moins profond possible, et un autre mot de longueur 13 dont l'arbre de dérivation soit le plus profond possible.

Les grammaires fournies.

GramDick1
$S \rightarrow aSb \mid \epsilon$

GramDick2
$S \rightarrow aSb \mid SS \mid \epsilon$

GramDick3
$S \rightarrow aSbS \mid \epsilon$

GramDick4
$S \rightarrow aSbS \mid SS \mid aSb \mid \epsilon$

GramDick5
$S \rightarrow SS \mid aSb \mid aSbS \mid \epsilon$

ConcatMultiple
$S \rightarrow SS \mid aSa \mid bSb \mid \epsilon$
$S \rightarrow bbb \mid aaa \mid bab \mid aba$

FEUILLE TP 2 : Grammaires

Nom :

Exercice 1

Fichier GramDick1.jff : la réponse obtenue avec *Derivation table* correspond

on ne fournit pas d'explications au fait qu'un mot soit refusé parce que

Exercice 2

Fichier GramDick2.jff : l'arbre de dérivation deviné est dessiné ci-dessous :

La chaîne de dérivation sera

Exercice 3

Fichier GramDick3.jff : l'arbre de dérivation deviné est dessiné ci-dessous :

La chaîne de dérivation sera

Exercice 4

Fichiers GramDick4.jff et GramDick5.jff : les arbres de dérivation devinés sont dessinés ci-dessous :

Les chaînes de dérivation seront

Exercice 5
Le mot le plus court est :

La règle à rajouter est :

Exercice 6

voici pour chacun des ordres l'arbre de dérivation deviné :

1. ordre :
arbre :

2. ordre :
arbre :

3. ordre :
arbre :

4. ordre :
arbre :

5. ordre :
arbre :

6. ordre :
arbre :

Exercice 7

La grammaire suivante :

sera testée sur les mots :

Exercice 8

La grammaire suivante :

sera testée sur les mots :

Exercice 9

La grammaire suivante :

sera testée sur les mots :

Exercice 10

La grammaire suivante :

sera testée sur les mots :

Exercice 11

La grammaire suivante :

sera testée sur les mots :

Exercice 12

on n'est pas interessé par l'ensemble des mots qui sont concaténation de mots palindromes d'une lettre parce que

La dérivation gauche de abbabbabbabb est

La dérivation droite de abbabbabbabb est

Exercice 13

Les quatre mots de longueur au plus 6 qui appartiennent au langage sont :

Le mot de longueur 15 dont l'arbre de dérivation est le moins profond possible est :

et voici son arbre de dérivation

Le mot de longueur 13 dont l'arbre de dérivation est le plus profond possible est :

et voici son arbre de dérivation

TP3 : Arbres et chaînes de dérivation

Ce TP vous demande de programmer en C. Récupérer les fichiers nécessaires dans l'espace pédagogique (ENT) : FLIN509 Documents et Liens →, dossier TP, puis le TP correspondant.

La grammaire :

On dispose d'un ensemble C de lettres $\{a, b, c, ..\}$. On dispose de la grammaire $G =< T, N, S, P >$, avec
$T = C\,U\,\{(,), +, *\}$ (les lettres, les parenthèses et les signes "+" et "*"). l'ensemble des terminaux.
$N = \{S\}$ Un seul non terminal, l'axiome S ;
$P = \{P_1, P_2, P_3, P_4\}$ avec :
- P_1 : $S \rightarrow c$ où $c \in C$
- P_2 : $S \rightarrow (S + S)$
- P_3 : $S \rightarrow (SS)$
- P_4 : $S \rightarrow (S)*$

Les productions de cette grammaire, par exemple $(((a(a)*) + ((((bb) + c)) * +d)))*$, seront appelées des expressions régulières.

Les trois structures de données :

- Le type **Expression régulière** pour les mots produits par la grammaire.
- Le type **ArbreDerivation** pour la représentation d'un mot par une arborescence explicitant les productions intervenues pour obtenir le mot.
- Le type **ChaineDerivation** pour les chaînes de dérivations représentant la séquence de productions générant un mot à partir de la grammaire.

Etudier les contenus des fichiers *tp3.h* et *tp3.c* pour identifier les fonctions de base permettant la gestion de ces structures de données. Chaque structure poura être exploités en utilisant exclusivement les fonctions déclarées dans *tp3.h* et implantées dans *tp3.c*. Les fonctions sont regroupées en 3 catégories :

- Les fonctions de "test", notées *estXxxP()*, qui teste si l'argument est bien un "xxx". Par exemple :
  - La fonction *estConcatenationP()* teste si son argument est une concaténation de deux mots (de deux expressions régulières).
  - La fonction *estArbreUnionP()* teste si son argument est un arbre de dérivation dont le noeud racine est une production $S \rightarrow (S + S)$.
- Les fonctions d'accès aux composants des structures, notées *getXxx()*. Par exemple :
  - La fonction *getLettre()* rend en valeur la lettre d'une expression régulière réduite à une lettre.
  - La fonction *premDerivation()* rend en valeur la première dérivation de la chaîne de dérivation passée en argument.
- Les fonctions de construction de structures, notées *consXxx*. Par exemple :
  - La fonction *consChaineDerivation()* qui construit une chaîne de dérivation en ajoutant un nouvel arbre de dérivation à une chaine de dérivations.
  - La fonction *consArbreEtoile()* qui construit un arbre de dérivation dont la racine est une fermeture de Kleene.

Travail demandé.

Définir les fonctions suivantes :

1. ***ArbreDerivation Construire(ExpressionReguliere E) ;***
   qui construit l'arbre de dérivation correspondant à l'expression régulière E.
2. ***int remplacerArbreDerivationGauche(ArbreDerivation ad, ArbreDerivation r)*** ;
   qui remplace (écrasement) le non terminal$S$ le plus à gauche dans ad par r. Cette fonction retourne 1 en cas de succès et 0 si aucun S n'a été trouvé. Attention : l'arbre ad étant modifié par cette fonction, il faudra en faire une duplication préalablement.
3. **ChaineDerivation consChaineDerivationGauche(ArbreDerivation ad) ;**
   qui construit la chaîne des dérivations les plus à gauche permettant d'obtenir l'arbre de dérivation ad supposé composé que de lettres terminales et d'aucune occurrence de S.

4. **ChaineDerivation consChaineDerivationDroite(ArbreDerivation ad) ;**
   qui construit la chaîne des dérivations les plus à droite permettant d'obtenir l'arbre de dérivation *ad* supposé composé que de lettres terminales et d'aucune occurrence de S.

Pour cela, on s'aidera des fonctions déjà écrites dans le fichier *testTP3.c* et en particulier de la fonction *main* qu'il faudra mettre partiellement en commentaire au fur et à mesure de l'écriture des fonctions.

Annexes

```
/*************** La structure ExpressionReguliere ***************/

typedef char *ExpressionReguliere;

/* Une expression régulière est un mot (séquence de lettres) reconnu par la grammaire.
* Une expression régulière est codée en C par une chaîne de caractères.
* Elle doit être reconnaissable par la grammaire, car il n'est pas fait de vérification.
* Les caractères spéciaux sont : +, * et les parenthèses ( et ).
* Tous les autres caractères sont considérés comme des lettres du langage (étendu).
* Remarque : Les espaces sont considérés comme des caractères normaux, i.e. des lettres.
*/

/* TESTS */

int estLettreP(ExpressionReguliere E);
int estEtoileP(ExpressionReguliere E);
int estUnionP(ExpressionReguliere E);
int estConcatenationP(ExpressionReguliere E);

/* ACCESSEURS */

char getLettre(ExpressionReguliere E);
ExpressionReguliere getEtoile(ExpressionReguliere E);
ExpressionReguliere getUnion1(ExpressionReguliere E);
ExpressionReguliere getUnion2(ExpressionReguliere E);
ExpressionReguliere getConcatenation1(ExpressionReguliere E);
ExpressionReguliere getConcatenation2(ExpressionReguliere E);

/****************** La structure de l'arbre de dérivation *********************/

/* Dans la structure d'arbre de dérivation, le type du noeud sera :
 *      '+', '*', ou '.'    pour un noeud union, étoile ou concaténation.
 *      'a','b','c',...     pour les lettres de l'alphabet de la grammaire.
 *      'S'                   pour l'axiome S (présent dans le langage étendu)
 *
 * Les sous termes sont mis dans sa1 s'il n'y en a qu'un seul (pour l'étoile) ou deux pour
 * les noeuds union et concaténation. Les lettres et l'axiome S n'ont pas de sous terme.
 */

struct noeudDerivation
{    char typeNoeud;
     struct noeudDerivation *sa1;
     struct noeudDerivation *sa2;
};

typedef struct noeudDerivation *ArbreDerivation;

/* TESTS (L'arbre est supposé non vide) */

/* Une lettre a,b,c,... , y compris le non terminal S */
int estArbreLettreP(ArbreDerivation a);
/* Test si l'arbre de dérivation représente juste le non terminal S */
int estArbreSP(ArbreDerivation a);
int estArbreEtoileP(ArbreDerivation a);
int estArbreUnionP(ArbreDerivation a);
int estArbreConcatenationP(ArbreDerivation a);
```

```
/* ACCESSEURS */

char getArbreLettre(ArbreDerivation d);
ArbreDerivation getArbreEtoile(ArbreDerivation d);
ArbreDerivation getArbreUnion1(ArbreDerivation d);
ArbreDerivation getArbreUnion2(ArbreDerivation d);
ArbreDerivation getArbreConcatenation1(ArbreDerivation d);
ArbreDerivation getArbreConcatenation2(ArbreDerivation d);

/* CONSTRUCTEURS */

/* La lettre c est mise dans typeNoeud. Elle ne doit être ni '+', ni '*', ni '.' */
ArbreDerivation consArbreLettre(char c);
/* typeNoeud = '*' et getEtoile est mis dans sa1 */
ArbreDerivation consArbreEtoile(ArbreDerivation d);
/* typeNoeud = '+' */
ArbreDerivation consArbreUnion(ArbreDerivation d1, ArbreDerivation d2);
/* typeNoeud = '.' */
ArbreDerivation consArbreConcatenation(ArbreDerivation d1, ArbreDerivation d2);

/***************** La structure ChaineDerivation *****************/

/* Une chaine de dérivation est une liste d'arbres de dérivation. Le délimiteur de fin de liste sera NULL. */

struct noeudChaineDerivation
{       ArbreDerivation prem;
        struct noeudChaineDerivation *suiv;
};

typedef struct noeudChaineDerivation *ChaineDerivation;

/* TESTS */

int estChaineVideP(ChaineDerivation cd);

/* ACCESSEURS */

/* Le premier arbre de dérivation de la chaîne de dérivation */
ArbreDerivation premDerivation(ChaineDerivation cd);
/* La chaîne de dérivation privée du premier arbre de dérivation */
ChaineDerivation suivDerivation(ChaineDerivation cd);

/* CONSTRUCTEURS */

/* Création d'une chaîne de dérivation vide */
ChaineDerivation consChaineDerivationVide();
/* Ajout d'un arbre de dérivation ad en tête de la chaine de dérivation cd */
ChaineDerivation consChaineDerivation(ArbreDerivation ad, ChaineDerivation cd);
/* Ajouter un arbre de dérivation à la fin de la chaîne de dérivation cd.
 * Cette fonction reconstruit une nouvelle chaîne sans détruire
 * la chaine cd en argument.
 */
ChaineDerivation consChaineDerivationFin(ArbreDerivation ad, ChaineDerivation cd);
```

TP4 : Introduction aux automates

Ce TP se fera en utilisant le logiciel "jflap". (commande : `jflap&`).
Attention, on ne crée pas une seule transition avec plusieurs étiquettes, mais plusieurs transitions chacune avec une seule étiquette.

Automates déterministes

Exercice 1 : Dans l'automate du fichier `01TresSimple.jff`, quel est le seul mot reconnu ? Faites tourner l'automate successivement (à chaque fois grâce à `input→Step by State` puis `step` jusqu'à épuisement des lettres du mot) sur les mots `ab`, `abba`, `ac`. Les résultats vous surprennent-ils ? Quel est le langage reconnu par cet automate ?

Exercice 2 : Définir formellement l'automate de l'exercice précédent, comme est défini celui de l'exercice suivant.

Exercice 3 : Dessiner l'automate $\mathcal{A} = (\Sigma, E, i, F, \delta)$ avec

- $\Sigma = \{a, b\}$
- $E = \{e_1, e_2\}$
- $i = e_1$
- $F = \{e_2\}$
- $\delta = \{e_1\, a\, e_1\,,\, e_1\, b\, e_2\,,\, e_2\, a\, e_2\,,\, e_2\, b\, e_1\,\}$

Quel langage reconnaît cet automate ? Quel est dans cet automate le chemin de trace `aabba` ?

Exercice 4 : Dessiner un automate qui reconnaisse la langage {a, aab, aaaab, b, bbaa}. Pouvez-vous supprimer des états (tout en continuant à reconnaître le même langage) ?
Pouvez-vous alors le rendre complet en rajoutant au plus un état ?

Exercice 5 : Sur l'alphabet `{a, b, c}`, dessinez l'automate déterministe qui accepte les mots qui ont exactement une occurrence de `c`.

Exercice 6 : Sans rajouter d'état, mais seulement en en supprimant (et en rajoutant des transitions) peut-on transformer l'automate donné dans le fichier `09PlusieursEtatsDeSortie_A` en un automate fini déterministes n'ayant qu'un état de sortie (et acceptant le même langage) ?
Pourquoi la même astuce ne fonctionne-t-elle pas pour les automates des fichiers `09PlusieursEtatsDeSortie_B` et `09PlusieursEtatsDeSortie_C` ?

Exercice 7 : Quel langage reconnaît l'automate du fichier `11Bmod4.jff` ?

Exercice 8 : Soit l'automate du fichier `02EtatsInnaccessibles.jff`. Dessiner un automate qui reconnaisse le même langage, mais dont tous les états soient à la fois accessibles et co-accessibles, puis dessiner un automate complet qui reconnaisse le même langage et dont tous les états soient accessibles.

Automate indéterministe

Exercice 9 : Soit l'automate du fichier `03IndeterministeUltraSimple.jff`. Quand vous testez si le mot `a` est accepté par cet automate, comment voyez vous que c'est le cas ?
Quel est le langage reconnu par l'automate de `04IndeterministeTresSimple.jff` ? Vérifiez que tous les mots que vous avez trouvés sont reconnus. Testez aussi quelques mots non reconnus.

Exercice 10 : Testez sur l'automate du fichier `06IndeterministeABA.jff` les mots suivants : abba, aba, aabab et aabbab. Pouvez vous décrire le langage reconnu par cet automate ?

Exercice 11 : Dessiner un automate indéterministe qui reconnaisse tous les mots qui aient au moins une occurrence du facteur ab et seulement eux.

Exercice 12 : Dessiner un automate indéterministe qui reconnaisse tous les mots qui aient au moins deux occurrences du facteur ab et seulement eux.

Complémentaire

Exercice 13 : Quel est le langage reconnu par l'automate de `12Complementaire.jff` ?

Exercice 14 : Dessiner un automate qui reconnaisse sur l'alphabet `{a, b}` tous les mots sauf {a, aab, aaaab, b, bbaa}.

ϵ-transition

Attention, sous `jflap` les ϵ-transitions sont étiquetées λ par défaut. Pour les obtenir, il suffit de ne pas mettre d'étiquette à la transition.

Exercice 15 : `jflap` ne permet pas d'avoir plusieurs états d'entrée. Modifiez l'automate du fichier `05DeuxEtatsEntree.jff` en ajoutant un état (d'entrée) et deux ϵ-transitions de façon à ce qu'il accepte le même langage que l'automate initial dans lequel les états d'entrée seraient q_0 et q_1. Testez votre résultat.

Exercice 16 : Quel est le langage reconnu par l'automate donné dans le fichier `07UltraSimpleEpsilon.jff` ?

Exercice 17 : Quel est le langage reconnu par l'automate donné dans le fichier `08Epsilon.jff` ?

Exercice 18 : Dessiner un automate qui reconnaisse sur l'alphabet `{a, b }` tous les mots qui admettent au moins une factorisation uv où u aie un nombre pair de `a` et v aie un nombre impair de `b`.

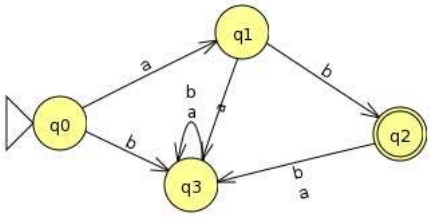
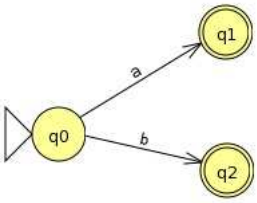
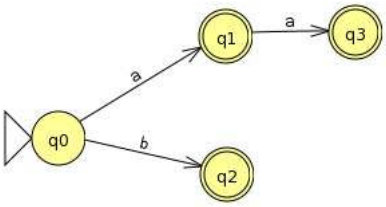


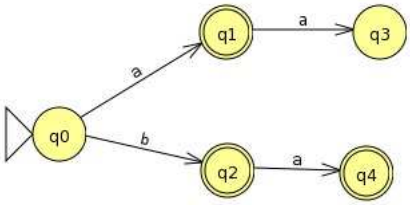
Figure 1: Exercices 1 et 2 : 01TresSimple



(a) 09PlusieursEtatsDeSortieA

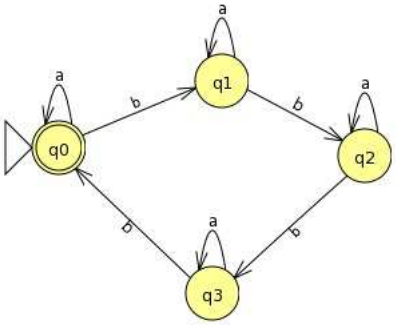


(b) 09PlusieursEtatsDeSortieB

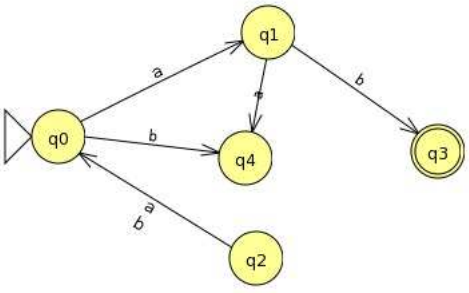


(c) 09PlusieursEtatsDeSortieC

Figure 2: Exercice 6

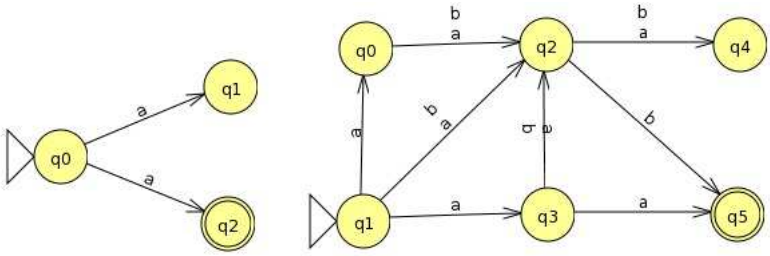


(a) 11Bmod4



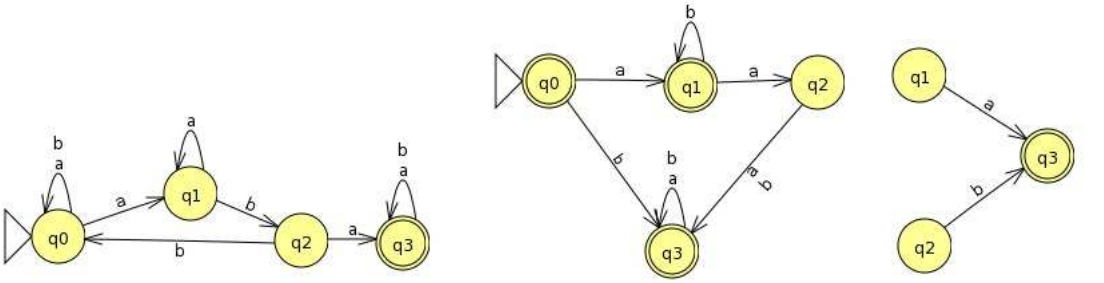
(b) 02EtatsInnaccessibles

Figure 3: Exercices 7 et 8



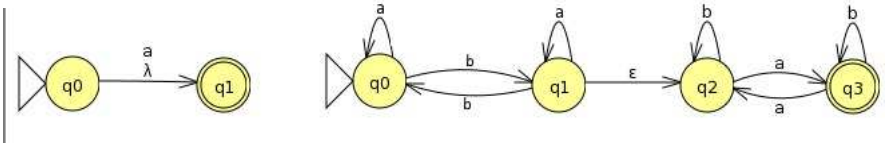
(a) 03IndeterministeUltaSimple (b) 04IndeterministeTresSimple

Figure 4: Exercice 9



(a) 06IndeterministeABA (b) 12Complementaire (c) 05DeuxEtatsEntree

Figure 5: Exercices 10, 13 et 15



(a) 07UltraSimpleEpsilon (b) 08Epsilon

Figure 6: Exercices 16 et 17

FEUILLE TP 4 : Introduction aux automates

Nom :

Automate déterministe

Exercice 1 et 2 : Fichier 01TresSimple
Répondez aux questions suivants avant de vérifier par la trace.

Le seul mot reconnu par l'automate est

- de q_0 par a on va en

  puis par b en

  donc le mot ab est

  ensuite par b on va en

  puis par a on va en

  donc le mot $abba$ est

- et ac n'est pas reconnu parce que

- le langage reconnu par l'automate est

- et la définition formelle de l'automate est

Exercice 3 : dessin d'un automate défini formellement
Dessin de l'automate de cet exercice :

Cet automate reconnait le langage composé des mots :

Le chemin de trace *aabba* est

Exercice 4 : dessin d'un automate minimum reconnaissant un langage fini donné
Dessin de l'automate demandé et ayant le moins d'états possible, tout en étant complet :

Cet automate a été testé sur les mots

- qui appartiennent au langage :

- qui n'appartiennent pas au langage :

Exercice 5 : dessin d'un automate reconnaissant les mots avec 1 occurrence de c Dessin de l'automate demandé qui reconnait le même langage, mais dont tous les états sont et accessibles et co-accessibles :

Testé sur les mots :

Exercice 6 :

- Fichier ***09PlusieursEtatsDeSortieA***
  La solution la plus simple est de construire l'automate à deux états suivant :

- Fichier ***09PlusieursEtatsDeSortieB***
  On ne peut pas mettre une transition étiquetée b entre q_0 et q_1 parce que

Exercice 7 : Le langage reconnu par l'automate est :

Exercice 8 : Dessin des deux automates :

Automate indéterministe

Exercice 9 :

- Fichier **03IndeterministeUltraSimple**
  On voit que a est reconnu parce que

- Fichier **04IndeterministeTresSimple**
  Il y a mots dans le langage reconnu parce que

Exercice 10 : fichier 06IndeterministeABA
Le langage reconnu par l'automate est celui des mots :
Seuls ces mots sont reconnus par l'automate parce que :

Tous ces mots sont reconnus par l'automate parce que :

Exercice 11 : automate reconnaissant tous les mots ayant une occurrence au moins de ab et seulement eux

Dessin de l'automate :

Tous ces mots sont reconnus par l'automate parce que :

Seuls ces mots sont reconnus par l'automate parce que :

Exercice 12 : automate reconnaissant tous les mots ayant deux occurrences au moins de ab et seulement eux

Dessin de l'automate :

Tous ces mots sont reconnus par l'automate parce que :

seuls ces mots sont reconnus par l'automate parce que :

Complémentaire

Exercice 13 : fichier 12Complementaire

Pour trouver le langage reconnu par cet automate, on étudie d'abord l'automate dont le dessin est :
et qui reconnaît le langage des mots

et on en déduit que le langage reconnu par l'automate donné est

Exercice 14.
Le dessin d'un automate qui reconnaisse sur l'alphabet `{a, b }` tous les mots sauf $\{a, aab, aaaab, b, bbaa\}$ est :

ε-transition

Remarque : Pour obtenir sous jflap les ε-transitions, il suffit de ne pas mettre d'étiquette à la transition.

Exercice 15 : fichier 05DeuxEtatsEntree
Pour obtenir un automate qui reconnaisse le langage reconnu par l'automate de la figure dont les états d'entrée seraient q_0 et q_1, il faut :

Exercice 16 : fichier 07UltraSimpleEpsilon
Le langage reconnu est composé des deux mots :

Exercice 17 : fichier 08Epsilon
Le langage reconnu par l'automate du fichier `08Epsilon.jff` est celui des mots sur { } qui

Exercice 18 : Le dessin d'un automate qui reconnaisse sur l'alphabet `{a,b}` tous les mots qui admettent au moins une factorisation uv où u aie un nombre pair de `a` et v aie un nombre impair de `b` est :

TP 5 Reconnaissance d'un mot par un automate

Présentation

Il s'agit d'écrire (dans le langage C) les fonctions permettant de décider si un mot appartient à un langage défini par un automate.
Les structures de données nécessaires pour implanter les automates sont spécifiées et définies dans les fichiers *tp5**. Les structures sont identiques quel que soit le type d'automate : déterministe, indéterministe, avec ε-transitions, avec quand même une structure supplémentaire pour les automates indéterministes : la structure *ListeEtat* qui implante les ensembles d'états.

Les structures de données :

- La structure des mots (type ***Mot***) et lettres (type ***Lettre***) : un mot est une séquence de lettres, naturellement codé par un chaîne de caractères en C.
- Le type ***Etat*** qui implante un état de l'automate, en fait un simple entier positif ou nul qui l'identifiera.
- Le type **Automate** et les sous structures :
  - **ListeEtatStatut**
  - **FonctionDelta**

  La connaissance de l'implantation de ces sous structures n'est pas nécessaire a priori. Elles sont déclarées dans le fichier *tp5_implantation.h* et elles sont définies dans le fichier *tp5_implantation.c*. Les listes "ListeEtatStatut" sont des listes d'états pécisant pour chacun d'eux leur statut (état initial, final, les deux , aucun des deux). Les listes "FonctionDelta" implantent la fonction de transition de l'automate.
- La structure **ListeEtat** : Cette structure implante les ensembles d'états qui seront introduits dans les automates indéterministes.

L'ensemble des fonctionnalités utiles pour manipuler les automates sont données dans le fichier *tp5_interface.h*. Les signatures des fonctions sont explicites. En cas d'incertitude, il faut aller voir leur implantation. Le fichier *tp5_interface.h* est donné en annexe.

Travail à faire

Définir les fonctions suivantes afin que le programme défini dans le fichier *testTP5.c* fonctionne :

1. Pour les automates déterministes
   (a) Etat deltaEtoile(Automate A, Etat e, Mot m) // La fonction δ* pour un automate déterministe
   (b) int estAccepte(Automate A, Mot m) // Test si le mot m est reconnu par l'automate A
2. Pour les automates indéterministes
   (a) ListeEtat deltaEtenduIND(Automate A, ListeEtat E, Lettre L) // La fonction δ étendue
   (b) ListeEtat deltaEtoileIND(Automate A, ListeEtat E, Mot m) // La fonction δ* étendue
   (c) int estAccepteIND(Automate A, Mot m) // Test si le mot m est reconnu par l'automate A
3. Pour les automates avec ε-transitions
   (a) ListeEtat epsilonChapeau(Automate A, ListeEtat E) // La fonction $\hat{\epsilon}$
   (b) ListeEtat deltaEtoileEpsilon(Automate A, ListeEtat E, Mot m) // La fonction δ* étendue
   (c) int estAccepteEpsilon(Automate A, Mot m) // Test si le mot m est reconnu par l'automate A

Annexe : le fichier *tp5_interface.h*

```
/********************** La structure des mots **********************/
/* TESTS */
int estMotVideP(Mot);

/* ACCESSEURS */
Lettre prem(Mot);
Mot suiv(Mot);

/* CONSTRUCTEURS */
Mot ajouterLettre(Lettre L, Mot m);

/*************************** * Les états et listes d'états *************************** */
/* Les différentes statuts possibles d'un état */
#define PAS_ETAT                -1          /* L'état n'a pas été trouvé */
#define ETAT_NORMAL              0
#define ETAT_INITIAL             1
#define ETAT_FINAL              10
#define ETAT_INITIAL_FINAL      11
#define ETAT_AU_MOINS_INITIAL(e)    (e & 1)
#define ETAT_AU_MOINS_FINAL(e)      (e & 10)

/* TESTS */
int estListeEtatVideP(ListeEtat L);

/* ACCESSEURS */
Etat premEtat(ListeEtat L);
ListeEtat suivEtats(ListeEtat L);

/* CONSTRUCTEUR */
ListeEtat ajouterEtat(Etat, ListeEtat);
ListeEtat unionEtat(ListeEtat, ListeEtat);

/* OUTILS */
int appart(Etat e, ListeEtat L);
int intersectionNonVide(ListeEtat L1, ListeEtat L2);
ListeEtat differenceEtat(ListeEtat L1, ListeEtat L2);
void afficherListeEtat(ListeEtat L);

/********************** * La structure Automate ********************** */
/* TESTS */
int estEtatFinalP(Automate, Etat);
int existeTransitionP(Automate, Etat, Lettre);

/* CONSTRUCTEURS */
Automate creerAutomate(Mot alphabet);
void ajouterEtatAutomate(Automate A, Etat e, char statut);
void ajouterTransition(Automate, Etat, Lettre, Etat);

/* ACCESSEURS */
Etat getEtatInitial(Automate);
ListeEtat getEtatsTerminaux(Automate A);
Etat delta(Automate A,Etat e, Lettre l);          // La fonction de transition delta

/* Pour les automates indéterministes */
ListeEtat getEtatsInitiaux(Automate A);
ListeEtat deltaIND(Automate A, Etat e, Lettre L);
ListeEtat deltaEtenduIND(Automate A, ListeEtat E, Lettre L);
```

TP 6 : Déterminisation et minimisation d'automates

Déterminisation d'automates

Exercice 1

Chargez l'automate `01AB.jff`. Quel langage reconnaît-il ? Quel est l'automate déterministe correspondant ?

Vérifiez votre résultat avec l'outil de déterminisation de `jflap` : `Convert` → `Convert to DFA`, puis expansion d'un état à la fois grâce à l'icône du milieu (`(S)tate expander`) ; utilisez l'icône de gauche (la flèche) pour déplacer les états et tenir le dessin dans les limites de l'écran.

Exercice 2

Fabriquez un automate indéterministe qui reconnaisse les mots (sur $\{a, b\}$) qui ont au moins une occurrence du facteur `aba`, et testez votre automate sur des exemples et des contre exemples ;

utilisez *Input* → *Multiple Run* pour rentrer vos exemples et contre exemples, puis *Run Inputs* pour tester..

Exercice 3

Fabriquez un automate indéterministe qui reconnaisse les mots (sur $\{a, b\}$) qui n'ont aucune occurrence du facteur `aba`, et testez votre automate sur des exemples et des contre exemples.

Exercice 4

Fabriquez un automate indéterministe qui reconnaisse les mots (sur $\{a, b\}$) qui ont au moins deux occurrences du facteur `aba`, et testez votre automate sur des exemples et des contre exemples.

Exercice 5

Fabriquez un automate déterministe qui reconnaisse les mots (sur $\{a, b\}$) qui ont exactement une occurrence du facteur `aba`, et testez votre automate sur des exemples et des contre exemples.

Exercice 6

Fabriquez un automate indéterministe qui reconnaisse les mots (sur $\{a, b\}$) qui ont au moins une occurrence du facteur `ababa` ou une occurrence du facteur `aaabbb`, et testez votre automate sur des exemples et des contre exemples.

Exercice 7

Fabriquez un automate déterministe qui reconnaisse les mots (sur $\{a, b\}$) qui n' ont ni occurrence du facteur `ababa` ni occurrence du facteur `aaabbb`, et testez votre automate sur des exemples et des contre exemples.

Exercice 8

Fabriquez un automate déterministe qui reconnaisse les mots (sur $\{a, b\}$) qui ont au moins une occurrence du facteur `ababa` suivie d'au moins une occurrence du facteur `aaabbb`, et testez votre automate sur des exemples et des contre exemples.

Exercice 9

Fabriquez un automate déterministe qui reconnaisse les mots (sur $\{a, b\}$) dont chaque occurrence du facteur `ababa` est suivie d'exactement une occurrence du facteur `aaabbb`, et testez votre automate sur des exemples et des contre exemples.

Minimisation d'automates

Pour minimiser un automate
- une fois l'automate déterministe complet chargé
- utilisez `Convert` → `Minimize DFA`
- apparaissent alors deux figures :
  - à gauche l'automate à minimiser
  - à droite l'arbre d'affinage des partitions, arbre initialisé à
    - la racine, qui correspond à l'ensemble des états
    - les deux fils de la racine, qui correspondent respectivement à l'ensemble des sommets finaux et l'ensemble des sommets non finaux

    c'est-à-dire que les deux feuilles de l'arbre, à ce stade, correspondent aux classes d'équivalence de la relation $\equiv_0$.
- la règle du jeu consiste alors à affiner les classes d'équivalence : pour cela, on peut
  - soit tricher
    - éhontément en sélectionnant la racine de l'arbre (en cliquant dessus) pour obtenir tout l'arbre d'affinage en cliquant sur `Complete Subtree`
    - à peine moins éhontément en sélectionnant successivement chacune des deux feuilles de l'arbre (toujours en cliquant sur la feuille concernée) pour obtenir tout le sous-arbre correspondant d'affinage en cliquant sur `Complete Subtree`
  - soit se comporter plus honnêtement en construisant à la main cet arbre d'affinage : pour cela
    - on sélectionne une feuille de l'arbre qu'on pense pouvoir affiner (toujours en cliquant dessus)
    - si on tient à tricher on peut cliquer sur `Auto Partition` qui fait automatiquement le travail qu'on vous indique comment faire honnêtement ci dessous
    - on clique sur `Set Terminal`, ce qui fait apparaître une fenêtre (intitulée `what terminal?` dans laquelle vous tapez la lettre qui permet d'affiner la classe d'équivalence en deux sous classes : cliquer sur `OK` fait alors apparaître deux fils à la feuille sélectionnée de l'arbre, mais ces deux fils n'ont aucune étiquette
    - il faut maintenant indiquer à quels états correspondent chacun de ces fils. Pour cela vous sélectionnez un des fils, puis dans la figure de gauche vous cliquez sur les états dont vous pensez qu'ils ne sont pas distingués par la lettre choisie.*Il faut ensuite recommencer l'opération avec l'autre fils, qui à ce stade n'est encore étiquetté par rien du tout.*
    - Il reste à vérifier que votre affinage est correct en cliquant sur `Check Node`. Si vous avez juste, vous pouvez continuer votre affinage, sinon vous devez utiliser `Remove`, ou tout recommencer !.
- quand votre affinage est fini, cliquez sur `Finish`. Les états de l'automate minimal apparaissent alors, et pour chacun il est indiqué la classe des états non distinguables auquel il correspond.
- Il vous reste à placer les transitions,
  - soit en trichant éhontément en cliquant sur `Complete`
  - soit en trichant petit à petit en cliquant sur `Hint` (qui vous place une transition
  - soit en plaçant à la main les transitions.
- puis, quand vous pensez avoir fini, cliquez sur `Done ?`

Exercice 10

Fabriquez un automate indéterministe qui reconnaisse tous les mots ayant au moins un facteur `abab` ou au moins un facteur `aabb`. Déterminisez le puis minimisez le :
- fabriquez la relation de Nérode en trichant éhontément. A quoi correspond chaque ligne de l'arbre obtenu ? Expliquez chacune des étiquettes qui apparaît sur l'arbre d'affinage.
- Construction de l'automate : fabriquez l'ensemble des états en cliquant sur la commande `Finish` puis placez l'ensemble des transitions une par une en trichant à peine moins éhontément (en utilisant la commande `Hint`). Justifiez une des transitions qui apparaît à partir des transitions de l'automate déterministe initial.

Exercice 11

Minimisez sans tricher l'automate du fichier `1UltraSimple.jff`

Exercice 12

Minimisez sans tricher l'automate du fichier `2TresSimple.jff`

Exercice 13

Minimisez sans tricher l'automate du fichier `3Simple.jff`

Exercice 14
Minimisez sans tricher l'automate du fichier `4Moyen.jff`

Exercice 15
Minimisez sans tricher l'automate du fichier `5PlusDifficile.jff`
Le problème est de trouver le bon moment pour distinguer les deux états terminaux.

Exercice 16
Fabriquez l'automate déterministe minimal qui contienne exactement une occurrence du facteur `ababab` *et* exactement une occurrence du facteur `aaabbb`. Il est conseillé de procéder par étapes, et vous pouvez utiliser l'outil `Convert` → `Combine Automata` qui dessine sur une même fichier deux automates définis dans deux fichiers différents.

Annexes

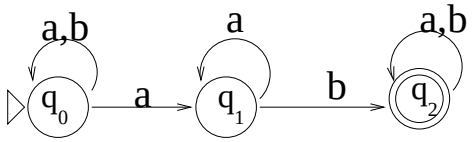


FIG. 1 – 01AB.jff

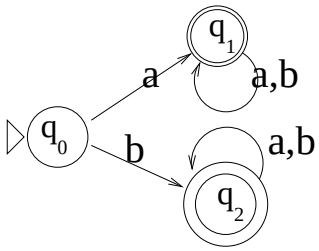


FIG. 2 – 1UltraSimple.jff

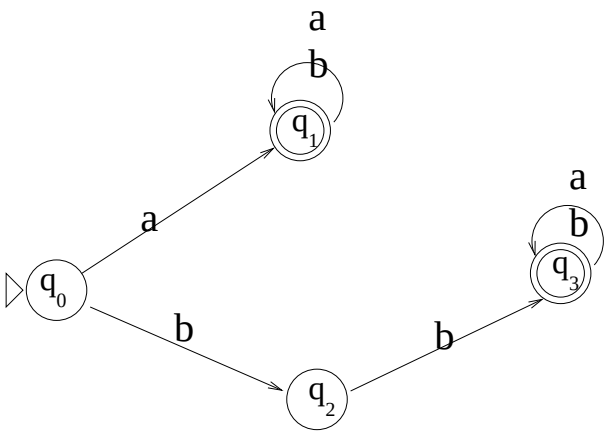


FIG. 3 – 2TresSimple.jff

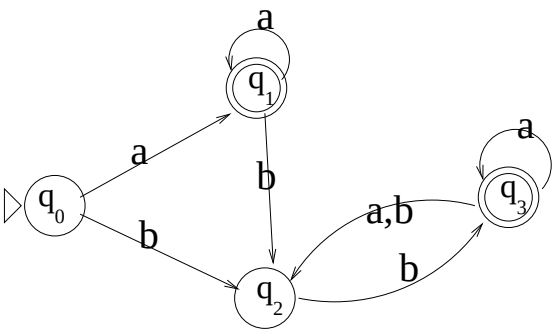


FIG. 4 – 3Simple.jff

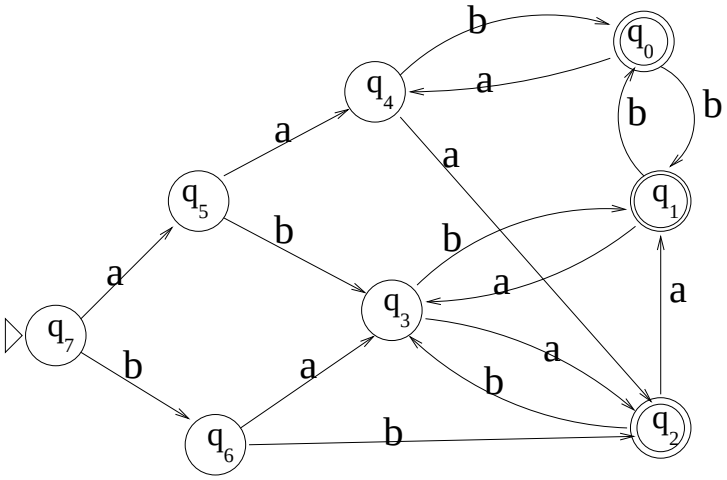


FIG. 5 – 4Moyen.jff

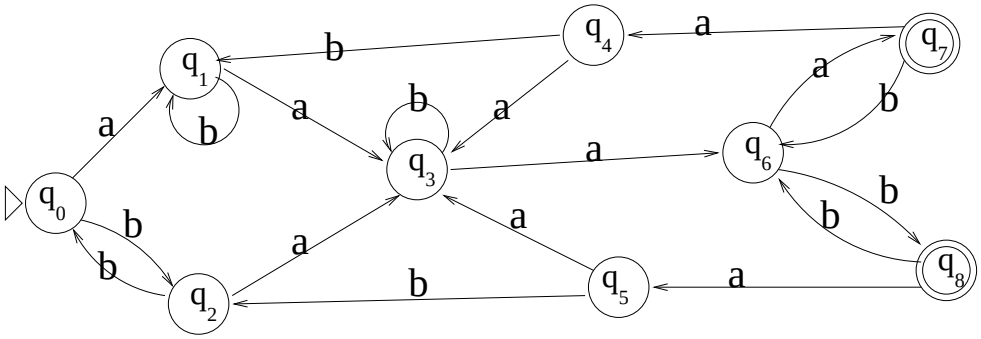


FIG. 6 – 5PlusDifficile.jff

FEUILLE TP 6 : Déterminisation et minimisation d'automates

Nom

Déterminisation d'automates

Fichier 01ABB
Le langage reconnu est le langage des mots sur $\{a, b\}$ qui

L'automate déterministe obtenu à la main est dessiné ci-dessous :

automate indéterministe qui reconnaît les mots qui ont au moins une occurrence du facteur `aba`
L'automate déterministe obtenu à la main est dessiné ci-dessous :

Exemples testés

Contre exemples testés :

automate indéterministe qui reconnaît les mots qui n'ont aucune occurrence du facteur `aba`

L'automate déterministe obtenu à la main est dessiné ci-dessous :

Exemples testés

Contre exemples testés :

automate indéterministe qui reconnaît les mots qui ont au moins deux occurrence du facteur `aba`

L'automate déterministe obtenu à la main est dessiné ci-dessous :

Exemples testés

Contre exemples testés :

automate déterministe qui reconnaît les mots qui ont exactement une occurrence du facteur `aba`

L'automate déterministe obtenu à la main est dessiné ci-dessous :

Exemples testés

Contre exemples testés :

automate indéterministe qui reconnaît les mots qui ont au moins une occurrence du facteur `ababa` **ou une occurrence du facteur** `aaabbb`

L'automate déterministe obtenu à la main est dessiné ci-dessous :

Exemples testés

Contre exemples testés :

automate déterministe qui reconnaît les mots qui n'ont ni occurrence du facteur `ababa` **ni occurrence du facteur** `aaabbb`

L'automate déterministe obtenu à la main est dessiné ci-dessous :

Exemples testés

Contre exemples testés :

automate déterministe qui reconnaît les mots qui ont au moins une occurrence du facteur `ababa` suivie d'au moins une occurrence du facteur `aaabbb`

L'automate déterministe obtenu à la main est dessiné ci-dessous :

Exemples testés

Contre exemples testés :

automate déterministe qui reconnaît les mots dont chaque occurrence du facteur `ababa` est suivie d'exactement une occurrence du facteur `aaabbb`

L'automate déterministe obtenu à la main est dessiné ci-dessous :

Exemples testés

Contre exemples testés :

Minimisation d'automates

automate indéterministe qui reconnaisse tous les mots ayant au moins un facteur `abab` ou au moins un facteur `aabb`

L'automate indéterministe obtenu à la main est dessiné ci-dessous :

Jeu de tests :

automate déterministe fabriqué à la main qui reconnaisse tous les mots ayant au moins un facteur `abab` ou au moins un facteur `aabb`

L'automate indéterministe obtenu à la main est dessiné ci-dessous :

Explication de l'arbre obtenu.

La ligne des composée des deux sommets de la racine correspond aux deux sous-ensembles d'états séparés par
L'étiquette sous le sommet qualifié de *non final* correspond à l'ensemble des sommets de l'automate initial qui ne sont pas séparés par
Les classes d'équivalence de la relation $\equiv_3$ se retrouvent

une fois que vous avez cliqué sur *Finish*

à quoi correspond chaque état ?
Il y a l'etiquette sous le sommet q_3 parce que
Plus généralement, les étiquettes qu'il y a sous chaque état correspondent à
Il y a, entre les états q_3 et $q_{??}$ une transition étiquetée ? ? parce que dans l'automate initial

Relation de Nérode obtenue

Dessiner ci-dessous l'arbre obtenu, indiquer à quoi correspond chaque niveau de cet arbre et chaque étiquette de chaque noeud.

Faire aussi un tableau qui indique pour chaque transition de l'automate obtenu à quelle(s) transition(s) elle correspond dans l'automate déterministe non minimal d'une part, dans l'automate indéterministe de l'autre.

Fichier 1UltraSimple.jff : résultat prévu avant l'utilisation de `jflap`

L'automate déterministe obtenu à la main est dessiné ci-dessous :

Fichier 2TresSimple.jff : résultat prévu avant l'utilisation de `jflap`
L'automate déteomate déterministe obtenu à la main est dessiné ci-dessous :

Fichier 3Simple.jff : résultat prévu avant l'utilisation de `jflap`
omate déterministe obtenu à la main est dessiné ci-dessous :

Fichier 4Moyen.jff
omate déterministe obtenu à la main est dessiné ci-dessous :

Fichier 5PlusDifficile.jff
A quel moment arrivez vous à distinguer les deux états terminaux ?
pourquoi pas avant ?

Automate déterministe minimal qui contienne exactement une occurrence du facteur `ababab` *et* exactement une occurrence du facteur `aaabbb`
Avant de faire l'exercice, indiquez les étapes successives par lesquelles vous allez passer.