# Compléments d'algorithmique : Cours

**Licence d'informatique — parcours appliqué**

## Institut Galilée & IUT de Villetaneuse

François Lévy

5 mars 2006

# Cours 1

# Information et flux d'information

On peut analyser les flux d'informations à trois niveaux :

1. au niveau des symboles : il s'agit alors de savoir ce que chaque symbole apporte en plus des précédents. La notion pertinente est alors celle d'information et de quantité d'information.
2. au niveau des mots : il s'agit alors de recherche de motifs, d'expressions régulières.
3. au niveau de l'agencement entre mots : on parle alors de grammaire

Le premier point est l'objet de ce chapitre. Le reste du cours traite du second point. Le troisième sera seulement mentionné à l'occasion, faute de temps.

## 1.1 Qu'est-ce qu'un flux d'information ?

Quand on a affaire à de grandes quantités de données, on est intéressé par des traitements qui lisent et calculent dans l'ordre de lecture, sans mémoriser les données. Cette contrainte est liée à la fois à la structure des documents traités et à celle du traitement que l'on veut faire. Par ex. une photo a une structure à deux dimensions (avec des voisins gauche/droite et haut/bas), mais peut être traitée en lisant ligne à ligne (on perd alors la notion de voisins verticaux).

**Définition formelle** : on a un alphabet $\mathcal{A} = \{a_i\bar{1} \leq i \leq T_a\}$. Un flux (ou message, ou texte) est une suite de symboles appartenant à $\mathcal{A}$. On écrit $m = \alpha_0 \cdots \alpha_{t_a-1}$ ou bien $m = m[0] \cdots m[t_a - 1]$, où chaque $\alpha_i$ ($m[i]$) est un élément de $\mathcal{A}$. Par ex :

- $\mathcal{A}_0 = \{0, 1\}$. $m_0 = 01100011$ est un message sur $\mathcal{A}_0$
- $\mathcal{A}_1 = \{a, b, c\}$. $m_1 = accabacb$ est un message sur $\mathcal{A}_1$

$\mathcal{A}_0$ est un alphabet de 2 symboles, $m_0$ un message de taille 8 sur $\mathcal{A}_0$. $m_1$ est un message de taille 8 sur l'alphabet $\mathcal{A}_1$ de taille 3. On a
$m_0[3] = 0$, $m_1[3] = a$, ou $\alpha_3^0 = 0$, $\alpha_3^1 = a$, etc.
Au besoin, on adaptera les notations.

## 1.2 Quantité d'information

**Idée 1** : on lit un flux d'information symbole après symbole. Quand on lit $\alpha$, que sait-on de plus qu'avant de l'avoir lu ?

Comme cela, la question est trop vague. Il faut savoir à quels messages on peut s'attendre. Si par ex. $\mathcal{A}$ est $\{0, 1\}$ et qu'il y a seulement deux messages $m_0 = 0000000\ldots$ et $m_1 = 111111\ldots$, $m[0]$ indique quel est le message, $m[i](i > 0)$ n'apprend rien de plus. Il en est de même si les messages sont $m_0 = 01010101\ldots$ et $m_1 = 1010101\ldots$ : après un 0 on est sûr d'avoir un 1 et réciproquement.

**Conclusion 1 :** un symbole apporte d'autant plus d'information qu'il est moins probable - donc que $P(x)$ est plus petit, ou que $\frac{1}{P(x)}$ est plus grand. C'est pourquoi on va exprimer l'information

apportée par un symbole comme une fonction croissante $I(x) = f(\frac{1}{P(x)})$. Le problème est alors : quelle fonction ?

**Idée 2 :** on considère une suite de deux symboles indépendants $x_1$, $x_2$ (donc $P(x_1x_2) = P(x_1) \times P(x_2)$). Quelle est l'information apportée par la suite $x_1x_2$ ? Comme aucun n'a d'influence sur l'autre, il est naturel de considérer que chacun apporte son information, et que celle du couple est la somme de celle de chacun pris isolément, soit encore $I(x_1x_2) = I(x_1) + I(x_2)$. La fonction f devrait alors satisfaire $f(\frac{1}{P(x_1x_2)}) = f(\frac{1}{P(x_1)}) + f(\frac{1}{P(x_2)})$, donc

$$f(\frac{1}{P(x_1)} \times \frac{1}{x_2)})) = f(\frac{1}{P(x_1)}) + f(\frac{1}{P(x_2)})$$

Dans cette dernière équation, $\frac{1}{P(x_1)}$ et $\frac{1}{P(x_2)}$ peuvent prendre n'importe quelle valeur supérieure à 1. Dans ce cas, cette équation caractérise les fonctions *log*. Il reste à choisir la base.

**Idée 3** La question est : que choisir comme unité d'information ? La réponse est un peu arbitraire. La réponse de Shannon : c'est l'information apportée par le choix entre deux possibilités (2 symboles) équiprobables. Donc si x vaut a ou b, chacun avec une probabilité 0,5, on a I(a) = I(b) = 1 — c'est à dire f(0,5) = 1, soit encore f(2) = 1. Ce qui signifie que le choix de f est $log_2$. Donc :

> L'information apportée par le symbole $x_i$ de probabilité $P(x_i)$ est $log_2(\frac{1}{P(x_i)}) = -log_2(P(x_i))$ unités d'informations (u.i.). Chez Shannon, *bit* veut dire **b**inary **i**nformation uni**t** et non binary digit.

## 1.3 Notion de source

Dans la définition de l'information, on supposait connue la probabilité de chaque symbole. Nous allons analyser un peu plus ce que signifie ce point. Pour cela, on considère que les symboles sont produits par une **source**. Ce sont les propriétés de la source qui déterminent quels messages sont possibles, et quelle est leur probabilité.

### 1.3.1 Source sans mémoire

Comme le nom l'indique, on suppose que chaque symbole du message est émis sans tenir compte de ceux qui le précèdent. La source choisit un symbole au hasard, en suivant une loi de probabilité $P(a_i)$ pour chaque symbole $a_i$ de l'alphabet $\mathcal{A}$. Les messages possibles sont m=$\alpha_0 \ldots \alpha_n$ où chaque $\alpha_j$ peut être n'importe quel $a_i$ de $\mathcal{A}$ avec la probabilité $P(a_i)$, apportant dans ce cas la quantité d'information $I_{j,i} = -log_2(P(a_i))$.

Donc l'information moyenne apportée par $\alpha_j$ est la moyenne des $I_{j,i}$ pondérée par leur probabilité. On appelle cette moyenne l'entropie de m au rang j :

$$H_j = \sum_{a_i \in \mathcal{A}} (-P(a_i) log_2(P(a_i)))$$

On constate que cette valeur ne dépend pas de j (ce qui est logique, puisque la source est sans mémoire). Elle caractérise la source. C'est ce que l'on appelle son entropie :

$$H = \sum_{a_i \in \mathcal{A}} (-P(a_i) log_2(P(a_i)))$$

(ne pas oublier que $\sum_{a_i \in \mathcal{A}} P(a_i) = 1$).

# Cours 2

# Reconnaissance des mots et automates

## 2.1 Introduction

On a un flux, que l'on appelle un texte t, et qui est constitué d'une suite de symboles appartenant à l'alphabet $\mathcal{A}$). On a aussi un ensemble de mots. Un ensemble de mots s'appelle un *langage*. Le problème est : reconnaître si les mots du langage $\mathcal{L}$ sont dans le texte. Précisons :

Pb0 : t est-il un mot de $\mathcal{L}$ ?
Pb1 : t commence-t-il par un mot de $\mathcal{L}$ ?
Pb2 : t contient-il un mot de $\mathcal{L}$ ? (i.e. au moins un mot)
Pb3 : Quelle est la position du premier mot de $\mathcal{L}$ qui est dans t ? (-1 s'il n'y en a pas)

Voici un exemple de chacun de ces problèmes.

Ex de Pb0 : vérificateur orthographique
Ex de Pb1 : décodage d'un code préfixe (type code de Huffmann)
Ex de Pb2 : sélection de lignes dans un fichier
Ex de Pb3 : recherche d'un mot dans un texte ($\mathcal{L}$ est alors un singleton) ; recherche des mots-clés dans un programme ($\mathcal{L}$ est un dictionnaire).

Tous ces problèmes se traitent grâce à deux notions parallèles :

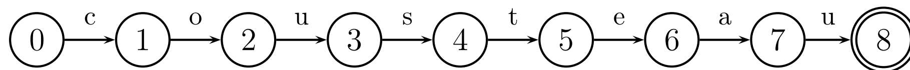
1. Un **langage** est un ensemble de mots que l'on souhaite reconnaître. La première question est : comment les décrire.
   - pour les langages finis, on peut énumérer les mots du langage (mais ce n'est pas toujours optimal).
   - pour les langages infinis, il faut nécessairement en donner une description symbolique.
2. Un **automate** est un type d'algorithme (et aussi un type de programme) capable de reconnaître un langage. Il y a des langages (et des automates) plus ou poins compliqués. Nous ne considérons dans ce cours que les plus simples.

## 2.2 Exemples

On utilise l'alphabet $\mathcal{A}$=[a-z] (uniquement les minuscules).

**Ex. 1 :** problème 1 avec $\mathcal{L}$ ={cousteau} (un seul mot). La question est "t commence-t-il par un mot de $\mathcal{L}$ ?". Pour vérifier, on lit t et on compare au fur et à mesure. On peut décrire l'algorithme en écrivant où on en est dans la lecture (la position actuelle) et le symbole qu'on attend ensuite :

| Position | Symbole |
|---|---|
| 0 | c |
| 1 | o |
| 2 | u |
| 3 | s |
| 4 | t |
| 5 | e |
| 6 | a |
| 7 | u |
| 8 | fini ! |




On verra à l'exemple 3 ce qui est derrière "où on en est dans la lecture". On utilise aussi un double cercle pour "c'est fini"

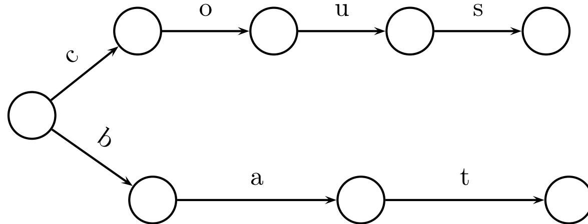
**Ex. 2 :** on veut toujours reconnaître cousteau dans un texte t en français, et traiter le problème 1 (t commence-t-il par un mot de $\mathcal{L}$), mais on pense à *mot* au sens des professeurs de français : si t commence par le mot manger, il ne commence pas par le mot mange.

On utilisera $\mathcal{A}$=[a-z␣] et $\mathcal{L}$ ={cousteau␣}.

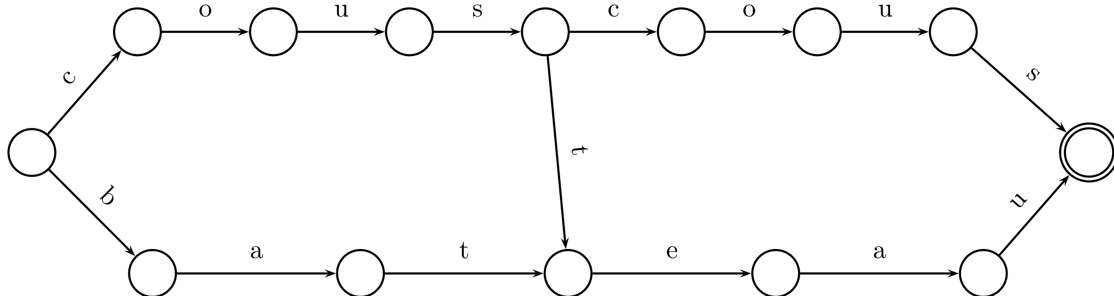
On constate qu'on est ramené à la question précédente.

**Ex. 3 :** $\mathcal{A}$=[a-z], $\mathcal{L}$ ={cousteau, couscous, bateau}, on traite toujours le problème 1.

On peut recommencer 3 fois la reconnaissance, bien sûr, mais deux mots ont le même début. On peut donc factoriser. Au début, on attend soit b, soit c :




Et maintenant ? Après `cous`, on attend c ou t – et si c'est t, la suite est la même que dans `bateau`.




Les ronds ne représentent plus une position dans t (on n'est pas à la même position quand on a reconnu `bat` ou `coust`), mais *ce qui reste à faire* (dans les 2 cas, on renvoie 'succès' si l'on trouve `eau`). On va formaliser cette idée par la notion d'état. En général, on donnera à chaque état un nom ou un numéro arbitraire.

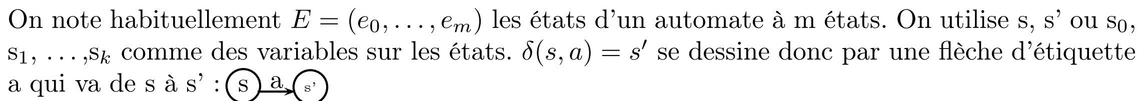
Le programme va d'un état à un autre en suivant les flèches qui correspondent à ce qu'il lit. Une flèche qui va d'un état à l'autre s'appelle une **transition**. Alternativement, c'est l'ensemble (état, flèche,état) que l'on appelle ainsi, pas la flèche seule : $\xrightarrow{a}$, $\xrightarrow{c}$, $\xrightarrow{s}$ sont en double dans le dessin, la flèche seule ne permettrait pas de savoir de quoi on parle.

## 2.3 Formalisation

**Définition 1** *Soit $\mathcal{A}$ un alphabet (les symboles). Un* mot *est une suite finie de symboles de $\mathcal{A}$. Un* langage *est un ensemble de mots.*

**Définition 2** *Un automate déterministe est un 5-uplet $\mathcal{T}$=(E, $\mathcal{A}$, $\delta$, $e_0$,F) où :*
- *E est l'ensemble des états*
- *$\mathcal{A}$ est l'alphabet*
- *$\delta$ est une fonction de $(E \times \mathcal{A}) \to E$ (la fonction de transition)*
- *$e_0$ est un état de E (l'état initial)*
- *F est une partie de E (l'ensemble des états finaux).*

On note habituellement $E = (e_0, \ldots, e_m)$ les états d'un automate à m états. On utilise s, s' ou $s_0$, $s_1$, ...,$s_k$ comme des variables sur les états. $\delta(s, a) = s'$ se dessine donc par une flèche d'étiquette a qui va de s à s' :

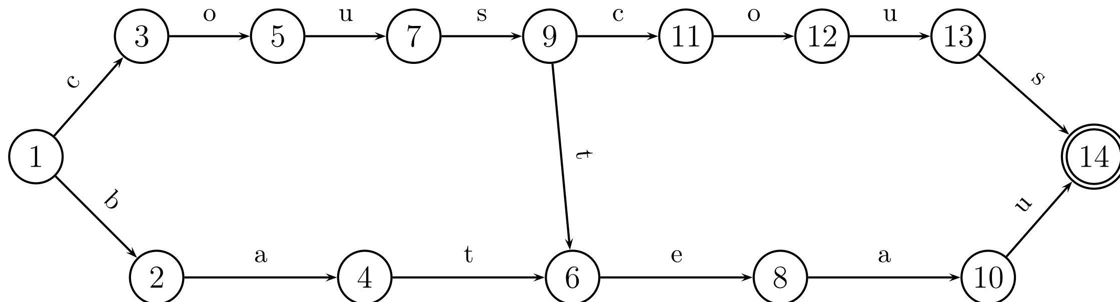



L'automate a un seul état initial (quand il n'a encore rien lu) et peut avoir plusieurs états finaux (quand on a reconnu un mot). Il est appelé déterministe parce que chaque symbole marque au plus une transition par état (donc quand on est dans un état et qu'on lit un symbole, la transition à faire est déterminée).

Avant d'aller plus loin, il faut préciser ce que veut dire 'l'automate a reconnu un mot', i.e. spécifier comment le programme doit fonctionner.

**Définition 3** *L'automate $\mathcal{T}$ reconnaît le mot m de longueur l si et seulement si il existe une suite d'états $s_0 \ldots s_l$ telle que*
- $s_0 = e_0$
- $\forall k \in [0, l-1], \delta(s_k, m[k]) = s_{k+1}$
- $s_l \in F$

Exemple :




Le chemin (1) (3) (5) (7) (9) (6) (8) (10) (14) de cet automate ou, avec une autre notation, $s_1, s_3, s_5, s_7, s_9, s_6, s_8, s_{10}, s_{14}$, reconnaît le mot `cousteau`. Remarquez que les états sont numérotés arbitrairement.

## 2.4 Représentation informatique

Il y a plusieurs techniques qui représentent chacune un compromis entre l'espace occupé et la rapidité. On utilise ici une technique rapide, et assez coûteuse en place quand il y a peu de transitions : on donne $\delta$ sous forme d'un tableau dans lequel $\delta(s, \alpha)$ est dans la ligne de s, colonne de $\alpha$.

Exemple (suite) :

| État \ Symb. | a | b | c | e | o | s | t | u |
|---|---|---|---|---|---|---|---|---|
| 1 | - | 2 | 3 | - | - | - | - | - |
| 2 | 4 | - | - | - | - | - | - | - |
| 3 | - | - | - | - | 5 | - | - | - |
| 4 | - | - | - | - | - | - | 6 | - |
| 5 | - | - | - | - | - | - | - | 7 |
| 6 | - | - | - | 8 | - | - | - | - |
| 7 | - | - | - | - | - | 9 | - | - |
| 8 | 10 | - | - | - | - | - | - | - |
| 9 | - | - | 11 | - | - | - | 6 | - |
| 10 | - | - | - | - | - | - | - | 14 |
| 11 | - | - | - | - | 12 | - | - | - |
| 12 | - | - | - | - | - | - | - | 13 |
| 13 | - | - | - | - | - | 14 | - | - |
| 14 | - | - | - | - | - | - | - | - |

Avec ce tableau, on applique en temps constant une transition donnée à un état donné.

Dans l'exemple, de nombreuse cases du tableau ne contiennent aucun état, ce qui est symbolisé par un trait. L'automate qui reçoit dans un certain état un symbole non attaché à une transition partant de cet état est implicitement en échec - mais on peut préférer éviter ces codages implicites. On va voir que c'est assez facile. La solution passe par la notion d'automate *complet*, i.e. qui n'a pas ce codage implicite de l'échec :

**Définition 4** *Un automate déterministe est complet si $\delta$ est une application de $E \times q\mathcal{A}$ dans E.*

(rappelez-vous qu'une application est partout définie). On écrit ADC pour automate déterministe complet. Le résultat important - et facile à prouver - est :

**Propriété 1** *Tout AD est équivalent à un ADC.*

Preuve : il suffit d'ajouter un état-puit ep. Toutes les transitions non définies sont remplacées par une transition vers ep, et on crée pour chaque symbole de $\mathcal{A}$ une transition de ep vers lui-même. ep est non-terminal.

Voici comme illustration la table de l'ADC complet équivalent à celui que nous avons pris pour exemple. On a ajouté l'état $s_0$ comme état-puit.

| État \ Symb. | a | b | c | e | o | s | t | u |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 2 | 3 | 0 | 0 | 0 | 0 | 0 |
| 2 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 | 5 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 0 | 0 | 6 | 0 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 7 |
| 6 | 0 | 0 | 0 | 8 | 0 | 0 | 0 | 0 |
| 7 | 0 | 0 | 0 | 0 | 0 | 9 | 0 | 0 |
| 8 | 10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 9 | 0 | 0 | 11 | 0 | 0 | 0 | 6 | 0 |
| 10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 14 |
| 11 | 0 | 0 | 0 | 0 | 12 | 0 | 0 | 0 |
| 12 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 13 |
| 13 | 0 | 0 | 0 | 0 | 0 | 14 | 0 | 0 |
| 14 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

# Cours 3

# Langages et expressions régulières

## 3.1 Langages réguliers

### 3.1.1 Introduction

Le problème est maintenant : comment définir un langage. Quand il est fini, on peut procéder par énumération, bien que ce ne soit pas nécessairement la solution optimale. Par exemple, si l'on considère le langage sur $\mathcal{A}$=$\{a,b,c\}$ défini par $\mathcal{L}$ =$\{aaa,aab,aac,aba,abb,abc,aca,acb,acc\}$, il serait mieux défini par "les mots de 3 lettres commençant par a". Dans le cas d'un langage infini, une énumération est par définition impossible. On va donc s'intéresser à la description d'un langage, éventuellement infini, par une formule finie. Pour cela, on va étudier les opérations qui permettent de définir de nouveaux langages à partir de langages déjà définis.

La première opération dont nous avons besoin est une opération sur les mots. La *concaténation* de $w_0$ et $w_1$ est définie par :

**Définition 5** *le concaténé $w = w_0.w_1$ de $w_0$ de longueur $l$ et $w_1$ de longueur $m$ est le mot de longueur $l+m$ tel que* $w[k] = \begin{matrix} w_0[k] \text{ si } k < l \\ w_1[k-l] \text{ si } l \leq k < l+m \end{matrix}$

On désigne par $\epsilon$ le mot vide. La concaténation est associative, et $\epsilon$ est son élément neutre (on appelle cette structure un demi-groupe, ou monoïde).

### 3.1.2 Opérations sur les langages

**Définition 6** *La somme de deux langages $\mathcal{L}_1$ et $\mathcal{L}_2$ est le langage $\mathcal{L} = \mathcal{L}_1 + \mathcal{L}_2$ tel que $w \in \mathcal{L}$ si et seulement si $w \in \mathcal{L}_1$ ou $w \in \mathcal{L}_2$*

On aura reconnu la réunion de deux ensembles, mais la notation additive est commode. On définit aussi la concaténation de deux langages :

**Définition 7** *On note $\mathcal{L} = \mathcal{L}_1.\mathcal{L}_2$ le langage défini par $w \in \mathcal{L}$ si et seulement si $\exists w_1 \in \mathcal{L}_1, w_2 \in \mathcal{L}_2$ t.q. $w = w_1.w_2$*

Par exemple, si $\mathcal{L}_1 = \{ab, ac\}$ et $\mathcal{L}_2 = \{ba, bc\}$, le langage $\mathcal{L}_1.\mathcal{L}_2$ est $\{abba, abbc, acba, acbc\}$.

On peut itérer la concaténation d'un langage avec lui-même : on définit $\mathcal{L}^2 = \mathcal{L}.\mathcal{L}$ et par récurrence $\mathcal{L}^n = \mathcal{L}^{n-1}.\mathcal{L}$. Il est naturel de définir $\mathcal{L}^1 = \mathcal{L}$ et $\mathcal{L}^O = \{\epsilon\}$, de sorte que la formule de récurrence soit valide pour $n > 0$. Notez que $\mathcal{L}^0$ ne dépend pas de $\mathcal{L}$, et que c'est l'élément neutre pour la concaténation des langages.

Si $\mathcal{L} = \{a\}$ est un singleton, $\mathcal{L}^n$ est formé du seul mot $a^n = a \ldots a$. Autrement, $\mathcal{L}^n$ est formé de n mots de $\mathcal{L}$, mais rien ne dit que c'est n fois le même mot. En reprenant les langages de l'exemple précédent, $\mathcal{L}_1^3 = \{ababab, ababac, abacab, abacac, acabab, acabac, acacab, acacac\}$.

On peut maintenant rassembler tous les langages formés en itérant la concaténation de $\mathcal{L}$ sur lui-même. On obtient ainsi un langage qui est infini (sauf quand $\mathcal{L}$ est $\mathcal{L}^0$).

**Définition 8** *Le langage* $\mathcal{L}^*$ *est défini par* $\mathcal{L}^* = \sum_{n=0}^{\infty} \mathcal{L}^n = \mathcal{L}^0 + \mathcal{L}^1 + \ldots + \mathcal{L}^n + \ldots$.

Autrement dit, un mot est dans $\mathcal{L}^*$ si et seulement si il est dans au moins un des $\mathcal{L}^n$ pour $n \geq 0$.
Exemple : $\mathcal{L} = \{ab\}$ (un seul mot)
On a $\mathcal{L}^0 = \{\epsilon\}$, $\mathcal{L}^1 = \mathcal{L}$, $\mathcal{L}^2 = \{abab\}$, $\mathcal{L}^3 = \{ababab\}$, $\ldots \mathcal{L}^* = \{(ab)^n \bar{n} \in n\}$.
Exemple : $\mathcal{L} = \{ab, c\}$
On a

- $\mathcal{L}^0 = \{\epsilon\}$,
- $\mathcal{L}^1 = \mathcal{L}$,
- $\mathcal{L}^2 = \{abab, abc, cab, cc\}$,
- $\mathcal{L}^3 = \{ababab, abcab, cabab, ccab, ababc, abcc, cabc, ccc\}$,
- ....
- $\mathcal{L}^*$ contient toutes les suites finies de ab et de c.

**Définition 9** *L'ensemble des langages réguliers est le plus petit ensemble de langages qui contient* $\{\epsilon\}$ *et tous les langages singletons* $\{a\}$ *pour a élément de* $\mathcal{A}$*, et qui est clos par concaténation (.), somme (+) et itération (*).*

## 3.2 Expression régulières

Un langage régulier est potentiellement infini, mais tout langage régulier peut être construit à partir d'un ensemble de singletons par une suite finie d'opérations '.', '+' et '*'. Une expression régulière est la description finie d'un langage régulier qui, en fait, décrit cette suite d'opérations. Si $\mathcal{A}$ est l'alphabet sur lequel sont construits les langages réguliers, les expressions régulières sur $\mathcal{A}$ utilisent $\mathcal{A}$ et les symboles supplémentaires $\varphi$, $\epsilon$, '.', '+', '*', '(' et ')'.

**Définition 10** *Une expression régulière est une chaîne de caractères ER vérifiant l'une des conditions suivantes :*
- $ER = \varphi$,
- $ER = \epsilon$,
- $ER = a$ *pour* $a \in \mathcal{A}$
- $ER = ER_1.ER_2$ *où* $ER_1$, $ER_2$ *sont des expressions régulières,*
- $ER = ER_1 + ER_2$ *où* $ER_1$, $ER_2$ *sont des expressions régulières,*
- $ER = ER_1^*$ *où* $ER_1$ *est une expression régulière,*
- $ER = (ER_1)$ *où* $ER_1$ *est une expression régulière*

*Rien d'autre n'est une expression régulière.*

Il est facile de définir inductivement le langage régulier $\mathcal{L}(ER)$ désigné par l'expression régulière $ER$ : par exemple $\mathcal{L}(a) = \{a\}$, ou $\mathcal{L}(ER_1.ER_2) = \mathcal{L}(ER_1).\mathcal{L}(ER_2)$, etc. Les détails sont laissés au lecteur. A l'inverse, par définition, pour tout langage régulier il existe au moins une construction finie à partir des langages atomiques et des opérations $+, ., *$, donc il existe au moins une expression régulière qui le désigne.

Attention : cette définition **n'implique pas** que deux expressions régulières différentes désignent des langages différents. Il est même très facile de trouver des contre-exemples : $(a + aa + aaa + aaaa*)$ désigne le même langage que $aa*$.

# Cours 4

# Langage reconnaissables par automates.

Tout automate reconnaît un ensemble de mots, donc un langage. Quand il existe un automate qui reconnaît tous les mots d'un langage $\mathcal{L}$ **et seulement ces mots**, on dit que $\mathcal{L}$ est reconnaissable par automate. Il s'agit maintenant de montrer que les langages reconnaissables par automate sont les langages réguliers et de donner une idée des méthodes effectives pour construire un automate qui reconnaît un langage $\mathcal{L}$ . On aura ainsi accès aux multiples logiciels basés sur ces techniques, aussi bien pour la manipulation de textes que en programmation.

## 4.1 Automate non déterministe fini (AFND)

Deux versions des automates non déterministes peuvent être employées. Elles sont fondamentalement équivalentes (dans le sens où elles reconnaissent les mêmes langages), mais la version avec $\epsilon$-transitions nous permettra une présentation plus simple de la démonstration du théorème de Kleene.

### 4.1.1 La version de base

Un automate est non déterministe quand plusieurs chemins peuvent correspondre au même mot. Cela se produit si il a plusieurs états initiaux possibles ou que plusieurs transitions portant la même étiquette peuvent partir du même état. Formellement, un automate déterministe est un 5-uplet $\mathcal{T}$=(E, $\mathcal{A}$, $\Delta$,I,F) où :

**Définition 11**
- *E est l'ensemble des états*
- *$\mathcal{A}$ est l'alphabet*
- *$\Delta$ est une relation incluse dans $(E \times \mathcal{A} \times E)$ (la relation de transition)*
- *I est une partie de E (l'ensemble des états initiaux)*
- *F est une partie de E (l'ensemble des états finaux).*

Un automate non déterministe reconnaît un mot si **il existe** un chemin étiqueté par ce mot qui va d'un état initial à un état final. D'autres chemins partant d'un état initial peuvent être étiquetés par ce mot et ne pas conduire à un état final, ou être étiquetés par un segment initial du mot et s'interrompre. On verra que dans certains cas, un automate déterministe est plus simple à construire. Mais en contrepartie, la reconnaissance d'un mot est plus difficile à programmer et suppose une capacité de retour arrière.

### 4.1.2 $\epsilon$-transitions

Une variante des automates non déterministes autorise, en plus d'avoir plusieurs états initiaux et plusieurs transitions de même étiquette partant du même état, à avoir des transitions qui sont franchies sans consommer le caractère suivant de l'entrée. Ces transitions sont étiquetées par $\epsilon$, l'élément neutre de la concaténation. Ainsi, le mot reconnu par un chemin reste la concaténation des étiquettes rencontrées.

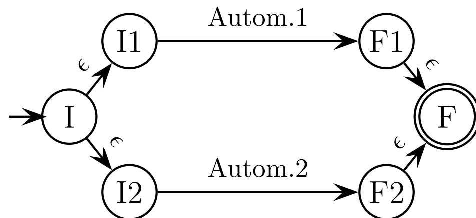
Formellement, la définition est presque la même que celle de la version de base d'un automate non déterministe. Seul l'item 3 est changé : la relation de transition $\Delta$ est incluse dans $(E \times \mathcal{A} \cup \{\epsilon\} \times E)$. La définition de la reconnaissance est la même.

## 4.2 Tout langage régulier est reconnaissable par un AFND

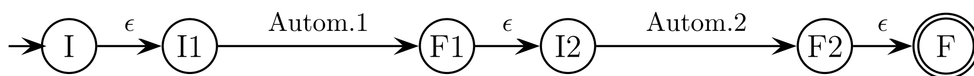
### 4.2.1 Reconnaissance par un automate avec $\epsilon$-transitions

Dans un premier temps, on démontre la forme la plus faible (et la plus facile à démontrer) : tout langage régulier est reconnaissable par un automate non déterministe avec $\epsilon$-transitions. Pour cela, étant donnée un langage régulier $\mathcal{L}$ , on va exhiber un automate qui le reconnaît. La construction est récursive, en suivant la définition des langages réguliers. Dans la démonstration, l'hypothèse de récurrence est que $\mathcal{L}_1$ est reconnu par l'automate Autom.1 et $\mathcal{L}_2$ par Autom.2. Les états initiaux et finaux de ces automates sont, avec des notations évidentes, I1, I2, F1, F2.

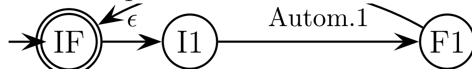
- Si le langage $\mathcal{L}$ est $\{\epsilon\}$ ou un langage singleton (étape initiale de la récurence), l'automate est trivial : $\xrightarrow{\epsilon}$ ou $\xrightarrow{a}$ suffit.
- Si $\mathcal{L} = \mathcal{L}_1 + \mathcal{L}_2$, $\mathcal{L}$ est reconnu par l'automate




- Si $\mathcal{L} = \mathcal{L}_1 \,.\, \mathcal{L}_2$, $\mathcal{L}$ est reconnu par l'automate




- Si $\mathcal{L} = \mathcal{L}_1^*$, $\mathcal{L}$ est reconnu par l'automate




Donc tous les langages appartenant au plus petit sous-ensemble contenant les langages singletons dont $\{\epsilon\}$ et clos par $+$, . et * – autrement dit les langages réguliers – sont reconnaissables par un automate avec $\epsilon$-transitions

### 4.2.2 Supprimer les $\epsilon$-transitions

Tout automate avec $\epsilon$-transitions peut se transformer en un automate sans $\epsilon$-transition qui reconnaît le même langage.

- Pour chaque $\epsilon$-transition $\delta(s, \epsilon) = s1$ (s $\xrightarrow{\epsilon}$ s1)
  - pour chaque transition $\delta(s1, a) = s2$ créer une transition $\delta(s, a) = s2$ si elle n'existe pas déja
  - si s1 est final, marquer s 'final'
  - supprimer la transition $\delta(s, \epsilon) = s1$

– Quand toutes les $\epsilon$-transitions ont été supprimées, supprimer les états inatteignables : répéter [si un s n'est pas initial et n'a plus de transition entrante (telle que $\delta(s0, x) = s$), supprimer s et ses transitions sortantes]tant qu'il existe un s satisfaisant la condition.

Pour un algorithme efficace de suppression des états inateignables par marquage de graphe, se reporter au dernier exercice du cours précédent.

## 4.3 Déterminisation

Notons que l'automate construit dans le paragraphe précédent a un seul état initial. A tout automate non déterministe à un seul état initial sans $\epsilon$-transition on peut faire correspondre un automate déterministe qui reconnaît le même langage. Une conséquence immédiate de cette construction est que tout langage régulier est reconnaissable par un automate déterministe.

### 4.3.1 Analyse

La construction de l'automate déterministe associé est plus élaborée que celles que nous avons vues jusqu'à présent, parce que c'est un automate sur un nouvel ensemble d'états. Soit $\mathcal{T}^N$=(E$^N$, $\mathcal{A}$, $\Delta^N$, e$_0^N$,F$^N$) l'automate non déterministe en question. On va construire l'automate associé $\mathcal{T}^D$=(E$^D$, $\mathcal{A}$, $\delta^D$, e$_0^D$,F$^D$). T$^D$ est un automate sur les *parties* de E$^N$ autrement dit sur $2^{(E^N)}$.

La remarque essentielle qui permet la construction est la suivante : étant donnée la relation de transition non déterministe $\Delta^N$, on peut étendre cette relation aux ensembles d'états de E$^N$. Pour chaque symbole a de l'alphabet A, on **définit** de façon unique l'image d'une partie S de E$^N$ par a selon $\Delta^N$ de la façon suivante : Im(S,a,$\Delta^N$) = $\{s'|\exists s \in St.q.\Delta^N(s, a, s')\}$.

On pose $\delta^D(s, a)$ = Im(S,a,$\Delta^N$) : la transition déterministe va d'un état de T$^D$ à son image par l'extention de $\Delta^N$. L'état initial de l'automate déterministe, quand on n'a encore rien lu, est l'ensemble singleton $\{e_0^N\}$. On veut que ses états finaux soient ceux qui reconnaissent les mots qui ont un chemin conduisant à un état final dans l'automate non déterministe (pour que les deux reconnaissent les mêmes mots). On a besoin pour formuler le critère de démontrer une propriété.

**Propriété 2** *S'il existe un chemin $C^N = e_0^N, e_1^N, \ldots, e_k^N$, étiqueté par le mot $a_0$, $a_1$, ..., $a_{k-1}$ dans $\mathcal{T}^N$, alors il existe un chemin $C^D = e_0^D, e_1^D, \ldots, e_k^D$, étiqueté par le même mot $a_0$, $a_1$, ..., $a_{k-1}$ dans $\mathcal{T}^D$, et pour tout i on a $e_i^N \in e_i^D$.*

La preuve se fait par récurence sur la longueur k du chemin.

– Si k est 0, l'étiquette est vide et on a $e_0^N \in e_0^D = \{e_0^N\}$.
– Supposons maintenant que la propriété est vraie jusqu'à k-1. On a en particulier $e_{k-1}^N \in e_{k-1}^D$. Or on a $\Delta^N(e_{k-1}^N, a_{k-1}, e_k^N)$ par définition de C$^N$, donc $e_k^N \in \delta^D(e_{k-1}^D, a_{k-1})$ par définition de $\delta^D$. Donc $\delta^D(e_{k-1}^D, a_{k-1})$ est bien définie et il suffit de choisir $e_k^D = \delta^D(e_{k-1}^D, a_{k-1})$ : le chemin e$_0^D$,e$_1^D$,...,e$_k^D$ de $\mathcal{T}^D$ reconnaît le mot a$_0$, a$_1$, ..., a$_{k-1}$.

La définition des états finaux en résulte immédiatement : un état e$^D$ de $\mathcal{T}^D$ est final ssi un de ses éléments est un état final de $\mathcal{T}^N$, autrement dit ssi $e^D \cup F^N \neq \emptyset$. La propriété ci-dessus montre que les deux automates reconnaissent les mêmes mots.

Une dernière remarque pour conclure cette section : nous n'avons pas démontré que toutes les parties de E$^N$ sont des états atteignables et productifs de $\mathcal{T}^D$, et ce n'est pas en général le cas. Autrement dit que, l'ensemble des états utiles de E$^D$ n'est en général qu'une partie de $2^{(E^N)}$, et on a intérêt à simplifier l'automate obtenu. Qui plus est, rien ne garantit que cet automate soit le seul ou le meilleur - nous reverrons la question un peu plus loin.

### 4.3.2 Algorithme

L'algorithme construit un par un les états atteignables de $\mathcal{T}^D$.

– Initialement, on crée une liste qui ne contient que l'état initial $\{e_0\}$ et un tableau des transitions qui ne contient que la colonne $\{e_0\}$.

– Pour chaque état S de la liste et chaque symbole a de l'alphabet, on calcule S' = Im(S,a,$\Delta^N$).
  + S'il n'y a pas une colonne S' dans le tableau, on ajoute une nouvelle colonne dont S' est la tête et on ajoute S' à la fin de la liste.
  + Dans tous les cas, on note S' dans la case (a,S) du tableau

### 4.3.3 Méthode directe

On peut éviter de passer par l'étape de suppression des $\epsilon$-transition. Il suffit de rajouter la règle suivante dans la définition des états de $\mathcal{T}^D$ : si $e^N \in S$ et il y a une $\epsilon$-transition de $e^N$ à $e1^N$, alors $e1^N \in S$. Un sous-ensemble de E$^N$ qui a cette propriété est dir *clos par $\epsilon$-transition*. Tout sous-ensemble est contenu dans un unique sous-ensemble clos par $\epsilon$-transition (sa cloture). L'automate déterministe est construit sur les sous-ensembles clos par $\epsilon$-transition de E$^N$. Son état initial est la cloture de $\{e_0\}$. La transition a est définie pour S si S' = Im(S, a, $\Delta^N$) est non vide, et elle relie S à la cloture de S'.

L'algorithme reste le même à une différence près : Pour chaque état S de la liste et chaque symbole a de l'alphabet, on calcule S' = Cloture(Im(S,a,$\Delta^N$)).

## 4.4 Minimisation

### 4.4.1 Automate des résiduels

Soit $\mathcal{L}$ un langage sur l'alphabet $\mathcal{A}$. On considère $\mathcal{A}$* l'ensemble de tous les mots sur $\mathcal{A}$. $\mathcal{L}$ permet de définit une relation d'équivalence $R_L$ sur $\mathcal{A}$* : $m_1$ et $m_2$ sont équivalents ssi $\forall m \in \mathcal{A} * m_1 m \in L \Leftrightarrow m_2 m \in L$.

$R_L$ est *régulière à droite* : si $m_1 R m_2$, alors pour tout m on a $m_1 m R m_2 m$ (c'est une conséquence immédiate de la définition - raisonner par l'absurde en supposant que $m_1 m$ et $m_2 m$ ne sont pas dans la même classe d'équivalence). Ceci permet de définir l'image d'une classe d'équivalence S de $R_L$ par un symbole a de $\mathcal{A}$ (c'est la classe S' qui contient tous les $m_1 a$ pour $m_1$ dans S). On a donc un ensemble d'états et de transitions sur $\mathcal{A}$.

Dernier point, si un mot d'une classe de $R_L$ est dans $\mathcal{L}$, alors tous les mots de cette classe sont dans $\mathcal{L}$ : il suffit d'utiliser la définition de $R_L$ avec $m = \epsilon$. On peut donc définir un automate $\mathcal{T}_L$ sur $\mathcal{A}$ qui reconnait $\mathcal{L}$, en prenant comme état initial la classe de $\epsilon$et comme états finaux ceux qui ne contiennent que des mots de $\mathcal{L}$. Mais il reste une difficulté : rien ne dit que le nombre d'états de cet automate est fini.

Supposons en plus que $\mathcal{L}$ est reconnaissable par un automate déterministe fini complet $\mathcal{T}$, ce qui, nous l'avons vu ci-dessus, est le cas des langages réguliers. Alors chaque mot $m = a_0 a_1 \ldots a_{n-1}$ correspond à un chemin $e_0, e_1, \ldots, e_n$ dans $\mathcal{T}$. De plus, deux mots $m_1$ et $m_2$ dont les chemins ont le même dernier état sont dans la même classe de $R_L$ ($m_1 m$ conduit dans un état final ssi $m_2 m$ conduit dans le même état ; donc les deux sont acceptés ou aucun ne l'est). Donc le nombre de classes de $R_L$ est inférieur au nombre d'états de $\mathcal{T}$, et $\mathcal{T}_L$ est le plus petit automate qui reconnaisse $\mathcal{L}$.

### 4.4.2 Réduction à l'automate minimal

Etand donné un ADFC qui reconnait $\mathcal{L}$, on donne ici un algorithme qui construit l'automate minimal ayant la même propriété (l'automate des résiduels). Chaque état sépare les mots en deux ensembles : ceux qui, ajoutés à son chemin, sont dans le langage L, et les autres. Le principe de l'algorithme est de partager progressivement les ensembles d'états qui ne produisent pas la même séparation : à la fin, tous les états dans le même ensemble généreront la même séparation, donc pouront être fusionnés.

On part de l'ensemble E des états. Initialement, on le partage entre F et E - F (epsi est reconnu par le premier ensemble et pas par le second. Si à un moment E est partagé entre S1, . . . ,Sn, pour chaque symbole a et chaque Si, on regarde si tous les états de Si ont leur image par a dans le même Sj. Sinon, on partage Si en autant de parties disjointes que de Sj (et les états dont l'image

est dans le même Sj sont dans le même état !). L'algorithme s'arète quand aucun état ne peut plus être partagé. Il est facile de montrer que chaque état obtenu a la propriété caractéristique de l'automate des résiduels, donc que c'est l'automate minimal unique recherché.

## 4.5 Réciproque

La réciproque consiste bien sûr à montrer que tout langage reconnaissable est régulier. Comme on a déja montré qu'un langage reconnaissable par automate non déterministe avec $\epsilon$-transition l'est aussi par un automate déterministe (et que un automate déterministe est un cas particulier d'automate non déterministe), il suffit de montrer que tout langage reconnaissable par un automate déterministe est régulier. Pour cela, on va partir d'un automate déterministe et construire une expression régulière qui décrit exactement le langage reconnu par l'automate.

Pour faire cette construction, on va utiliser un *automate étendu* : sa structure est la même qu'un automate déterministe habituel, mais les étiquettes des transitions peuvent être une expression régulière (au lieu d'être limitées à un symbole de l'alphabet).

Soit $\mathcal{A}$ un automate déterministe. L'étape initiale de l'algorithme construit un automate étendu $\mathcal{AE}$. Celui-ci est une copie de $\mathcal{A}$ à laquelle on a ajouté deux états, $\alpha$ et $\omega$. On a de plus des transitions sans label de $\alpha$ vers l'état initial $\mathrm{e}_0$ de $\mathcal{A}$ et de chaque état final de $\mathcal{A}$ vers $\omega$. Dans $\mathcal{AE}$, $\alpha$ est initial et seul $\omega$ est final.

Ensuite, chaque étape de l'algorithme consiste à supprimer un état de $\mathcal{AE}$ à condition que cet état ait au moins une transition entrante et une transition sortante.

# Cours 5

# Logiciels utilisant les E.R.

La syntaxe des expressions régulières implémentées est légèrement différente de celle qui a été utilisée pour la théorie, et surtout elle varie d'un logiciel à l'autre et suivant l'ancienneté de la version. Les différentes variantes syntaxiques sont récapitulées dans un document séparé. Pour l'essentiel, la disjonction (correspondant au + entre langages) est le plus souvent notée '|' au lieu de '+', + désigne la répétition au moins une fois ($r+$ est une abréviation de $(r|rr*)$), et la syntaxe étendue introduit des compteurs de répétition {m,n} (entre m et n répétitions). Les exemples donnés ont été testés sous Linux.

## 5.1 Grep

Grep est le plus simple des logiciels utilisant des expressions régulières. Il sélectionne dans un fichier toutes les lignes filtrées par une expression régulière $ER$ - plus précisément toutes les lignes qui contiennent une chaîne acceptée par $ER$ (cf. le Pb2 de l'introduction à la reconnaissance des mots). On peut paramétrer la recherche pour qu'elle renvoie les lignes voisines de celle reconnue (son contexte), ou ne fasse pas de différence entre majuscules et minuscules, ou filtre par une disjonction d'expressions régulières. Les options sont en ligne dans le manuel Unix (`man grep`).

Deux exemples pour illustrer la variété des usages de grep. Le premier extrait les définitions de fonction des fichiers C et donne leur numéro de ligne. Il est assez rudimentaire : il dépend d'une convention d'écriture et laisse passer des parasites (les instructions de contrôle portant sur un bloc).

```
 F102 ~/code> grep  "[A-Za-z_]\+(.*)[ \t]*[{]" *.c
clavier.c:23:void tt_handler(int sig){
clavier.c:24:  switch(sig) {
clavier.c:34:int main(void) {
clavier.c:56:  while ((c=getchar())!= 'a')  {
horloge.c:6:void alarm_handler(int i)  {
horloge.c:10:int main(void) {
```

Le second exemple effectue une recherche de tournures syntaxiques dans un fichier de texte en français (les phrases indiquant une répétition).

```
 F102 ~/code> grep  "[E|e]ncore|[A|à] nouveau|re(fai|commenc|pris)"
```

## 5.2 Éditeurs de texte interactifs

Aussi bien vi que emacs utilisent des expressions régulières pour des recherche ou recherche-remplace. On peut utiliser une partie de l'expression reconnue lors du remplacement : si l'on met une sous-expression entre parenthèses, celle-ci peut être citée dans le texte de remplacement : `\&`

cite toute l'expression reconnue, et `\n` (n = chiffre de 0 à 9) la nième sous-expression (dans l'ordre d'apparition des parenthèses ouvrantes). De même, `\<` et et `\>` marquent le début et la fin d'un mot (le passage d'un espace à un caractère alphabétique ou d'un caractère alphabétique à un espace). Par exemple, la recherche de `\(\<[^ ]*\>\) \1` donnera les mots répétés deux fois.

Vi utilise le / pour chercher une expression régulière (comme more ou less qui en dérivent) . On peut donner une expression de substitution identiques à celles de sed (cf. section suivante) pour substituer une expression régulière.

## 5.3 sed

## 5.4 awk

# Cours 6

# perl

## 6.1 Préambule

Une documentation très complète sur perl est disponible en ligne sur le manuel unix (command `man`). Voici une première liste de rubriques déja bien suffisante :

| | |
|---|---|
| perl(1) | Practical Extraction and Report Language |
| perlbook(1) | Perl book information ; |
| perlboot(1) | Beginner's Object-Oriented Tutorial ; |
| perldata(1) | Perl data types |
| perldbmfilter(1) | Perl DBM Filters |
| perldebtut(1) | Perl debugging tutorial |
| perldebug(1) | Perl debugging |
| perldsc(1) | Perl Data Structures Cookbook |
| perlform(1) | Perl formats |
| perlfunc(1) | Perl builtin functions |
| perlintro(1) | a brief introduction and overview of Perl |
| perllexwarn(1) | Perl Lexical Warnings |
| perllol(1) | Manipulating Arrays of Arrays in Perl |
| perlop(1) | Perl operators and precedence |
| perlopentut(1) | tutorial on opening things in Perl |
| perlport(1) | Writing portable Perl |
| perlre(1) | Perl regular expressions |
| perlref(1) | Perl references and nested data structures |
| perlreftut(1) | Mark's very short tutorial about references |
| perlrequick(1) | Perl regular expressions quick start |
| perlretut(1) | Perl regular expressions tutorial |
| perlrun(1) | how to execute the Perl interpreter |
| perlsub(1) | Perl subroutines |
| perlsyn(1) | Perl syntax |
| perltoc(1) | perl documentation table of contents |
| perlutil(1) | utilities packaged with the Perl distribution |
| perlvar(1) | Perl predefined variables |
| perldoc(1) | Look up Perl documentation in Pod format |

Pour lancer Perl, utilisez de préférence un fichier exécutable, par exemple `monprog.pl`:
- La première ligne est `#!  /usr/bin/perl -w`
- Commencez par `use strict ;`
- Écrivez votre programme à partir de là
- Rendez monprog exécutable (`chmod u+x monprog.pl`)

– lancez `./monprog.pl`

Dernière recommandation, déclarez toutes vos variables par `my` en prévoyant leur espace de validité (tout le reste du fichier si vous mettez `my $var` en dehors de tout block, tout le reste du bloc si vous placez la déclaration dans un bloc). Use strict vous y contraindra, et vous perdrez moins de temps à debuger.

## 6.2 Variables et fonctions

### 6.2.1 Les trois types

Le type des variables est indiqué par la première lettre de leur nom. Il y en a trois :

**scalaire** recouvre toutes les variables ayant une valeur unique, que celle-ci soit utilisée comme chaîne de caractères ou comme nombre (entier ou à virgule). Il est noté par `$` (`$a`, `$machin`, etc.)

**tableau** recouvre les ensembles de valeurs indicés par un entier. Ce type est noté `@` (`@x`, `@tab`, etc.). Comme ne C, les indices de tableaux commencent à 0, et on accède à un élément en mettant son indice entre crochets (`$tab[5]`, etc.). Notez le $ : `$tab[5]` est un scalaire, ce sont les crochets qui indiquent qu'il est extrait d'un tableau.

**hash** recouvre les ensembles de valeurs indicés par des chaînes : comme les tableaux, il fait correspondre une valeur à un indice, mais au lieu que les indices soient des entiers consécutifs, dans les hashs ce sont des scalaires quelconques. C'est beaucoup plus souple, mais en contrepartie on n'a pas la notion d'entrée suivante (c'est pour cela qu'on a besoin d'une structure de donnée différente pour retrouver quelle valeur correspond à quel indice).

Un hash est noté par `%` (`%a`, `%machin`, etc). On accède aux éléments du hash par un mécanisme analogue à celui des tableaux : `$machin{jean}` est l'élément du hash `%machin` dont l'indice est `jean`. Notez que le dollar `$` marque qu'on accède à un scalaire, et l'accolade `{}` qu'on y accède par l'intermédiaire du hash `%machin`. A ne pas confondre avec `$machin` sans accolade, qui désigne une autre variable.

Remarquez que les espaces de noms sont distincts : `$machin`, `@machin` et `%machin` sont trois variables différentes, et on peut employer dans le même programme sans perturber l'interpréteur (mais le lecteur ...).

### 6.2.2 Renseigner les hashs

Un hash non initialisé est vide. On l'initialise avec une forme de liste complexe qui peut utiliser deux écritures :

– `my %tab = (jean => ''1,83 m'', jacques => ''1,54 m'') ;` ou
– `my %tab = (jean , ''1,83 m'', jacques , ''1,54 m'') ;`

On lit donc une valeur dans le hash en utilisant les accolades, par exemple `print (''$tab{jean}'');`
On ajoute ou on modifie un couple (indice, valeur) en affectant l'indice : `$tab{louis}=''1,75 m''`.

### 6.2.3 Les fonctions

On déclaare une fonction par le mot clef `sub`. Suit une accolade dans laquelle est le corps de la fonction (les instructions qu'elle exécute). Remarquez qu'*une déclaration de fonction n'indique pas ses paramètres*. On les récupère à l'exécution dans la variable implicite @_ (donc un tableau de nom _ ). Ce qui signifie que :

– rien n'indique combien de variables la fonction attend, et de quel type.
– on peut utiliser des fonctions avec un nombre variable d'arguments

A l'appel, les paramètres sont dans une liste, donc entre parenthèses. Notez que, lorsqu'on ne passe rien, la parenthèse est inutile.

Exemples :

```
sub essai {
my @ltab = @_ ; # on  recupere les variables dans un tableau
}

sub essai {
my @ltab = (@_) ; # La même chose, à cause de l'applatissement des listes
}

sub essai {
my ($ldeb, @lrest) = @_ ; # La premiere dans un scalaire, le reste
}                           # dans un tableau
```

### 6.2.4 Type et contexte

Perl est un langage qui essaye de donner un sens à ce qu'a écrit le programmeur. Pour cela, il utilise abondamment la conversion de type. On appelle *contexte* le type de variable attendu ; si le type reçu ne convient pas, Perl essaye de le convertir.

Par exemple, `$var = ...` attend un scalaire. `@tab` est un tableau. Si vous écrivez `$var = @tab`, Perl 'comprend' que vous voulez un scalaire en rapport avec le tableau, et renvoie son nombre d'éléments. Ci-dessous deux applications de cette conversion :

```
sub essai {
my $nvar = @_ ;# on recupere le nombre d'arguments dans une liste
}

# pour vérifier le contenu d'un tableau
for (my $var = 0 ; $var < @tab ; $var++) {
print("$var : $tab[$var]\n");
}
```

Le jeu des conversions devient particulièrement raffiné quand une fonction renvoie un résultat qui diffère selon le contexte. Par exemple, la fontion `localtime` qui lit l'heure locale du système renvoie un tableau de nombres en contexte tableau, et sa traduction en chaîne de caractères en contexte scalaire. Ci-dessous un exemple de programme pour illustrer ce comportement :

```
> cat essai.pl
#! /usr/bin/perl -w
use strict ;

my $var1 = localtime ; # résultat dans une variable scalaire
print($var1, "\n") ;

my @tab =  localtime ; # résultat dans un tableau
   for (my $var = 0 ; $var < @tab ; $var++) {
   print("$var : $tab[$var]\n");
   }

print(localtime,"\n"); # utilisation directe

> ./essai.pl
Sat Jan 28 13:37:03 2006
0 : 3
1 : 37
2 : 13
```

```
3 : 28
4 : 0
5 : 106
6 : 6
7 : 27
8 : 0
337132801066270
```

L'affectation à $var1 a produit une chaîne. Les lignes numérotées donnent le contenu du tableau (au début, on reconnaît facilement *secondes, minutes,heure,quantième du mois*. La ligne `print(localtime,"\n")` place l'appel de fontion en contexte de liste (la liste des arguments de print), et donc passe le tableau en argument.

## 6.3 Expressions régulières

Les expressions régulières s'utilisent avec = et l'un des deux opérateurs décrits ci-dessous. = s'appelle un attachement (Bind en anglais), parce qu'il sert à attacher la variable à l'opérateur qu'on lui applique.

L'opérateur de filtrage s'appelle m et, en contexte scalaire, il renvoie un booléen. Par ex, `if($lig =~ m/^C.*b$/)` teste si la ligne (ou la variable) $lig commence par C et finit par b. Il n'est pas toujours commode d'utiliser / pour marquer les limites de l'ER. m prend le premier caractère qui suit, donc on pouvait aussi bien écrire `if($lig =~ m|^C.*b$|)` ou `if($lig =~ mx^C.*b$x)`. Les groupements sont marqués par des parenthèses sans antislash. Ils peuvent être cités par \1, etc., dans le motif cherché. Quand on utilise le séparateur /, on peut ommettre m – c'est l'opérateur par défaut. Notez aussi que la variable n'est pas modifiée après filtrage, donc que si l'on ne récupère pas la valeur de retour le filtrage ne sert à rien.

L'opérateur de substitution s'appelle s et fonctionne comme pour sed ou vi. Les groupements sont marqués par des parenthèses sans antislash. Ils peuvent être cités par \1, etc., dans le motif cherché, et par $1, etc ; dans le texte de substitution. Ainsi pour substituer des guillemets simples à des guillemets doubles dans $var, on utilisera `$var =~ s/"(.*)"/'$1'`]]

\d abbrège un chiffre, \D un non-chiffre. \w abrège un alphanumérique, \W un non alphanumérique. Enfin \s abrège un espace (de tout type : blanc, tabulation, etc), et \S un non-espace.

On peut récupérer les sous-expressions lors d'un filtrage, à condition de se placer en contexte de liste. C'est une syntaxe un peu longue, mais on s'en sort :

```
if((\$nom,\$val)=(\$lig =~ m/^NOM : (\w+)\s*VALEUR : (\d+)/))
```

Si la ligne ne correspond pas, le résultat est faux ; si il correspond, il est vrai, $nom et $val sont affectées par les groupements reconnus. Vous avez déja vu qu'on peut aussi récupérer $1 et $2 dans le if.

Les mêmes mécanismes de récupération sont possibles avec une substitution.

Les quantificateurs non-gourmands sont obtenus en ajoutant un point d'interrogation après un quantificateur (*, +, ?, {n,m}). Ils cherchent **la plus petite chaîne** qui matche l'ER. Ils sont très utiles en particulier pour la récupération.

les options de s ou de m sont

- g : global. Effectue tous les remplacements au lieu du premier. Avec m//g, on fait en général une boucle while pour poursuivre la recherche dans le même élément.
- i : insensible à la casse. pas de différence entre majuscules et minuscules.
- e : evaluer. Dans une substitution, permet de *calculer* le texte de remplacement en utilisant une *fonction* dans la seconde partie de l'ER.
- s : ligne unique (single). Les changements de lignes (\ n) inclus dans la chaîne sont traités comme des caractèress normaux : ils sont filtrés par . et n'acceptent pas $ ou ^ en milieu de chaîne.
- m : ligne multiple. les changements de ligne inclus dans la chaîne acceptent $ et ^ et ne filtrent pas .

Notez que sans l'option s ou m, le fonctionnement de s/// est intermédiaire entre les deux : les changements de ligne inclus dans la chaîne n'acceptent pas $ et ^ et ne filtrent pas non plus .

# Cours 7

# E.R. pour programmer en C

## 7.1 Lex : un analyseur lexical

## 7.2 Utiliser des E.R. dans un programme : la bibliothèque regex