INF 431

F. Morain

Analyse lexicale

14 février 2007

À propos du projet

- La page est ouverte. Tout sera fait avec le web.
- Calendrier:
  - Remise de la feuille de choix avant le 7 mars. Dans la foulée, acceptation ou non, négotiations, etc. ⇒ ne commencez pas avant cette date!.
  - Remise du projet: 29 mai.
  - Soutenances: 11 au 22 juin.

Projet == Autonomie

⇒ les TD seront de moins en moins scolaires pour vous habituer. . .

Où en est-on ?

Amphi 1: introduction.

Amphi 2: génie logiciel avec Java.

Amphi 3: analyse lexicale.

Amphi 4: analyse syntaxique.

Amphi 5: graphes I.

Amphi 6: graphes II (parcours).

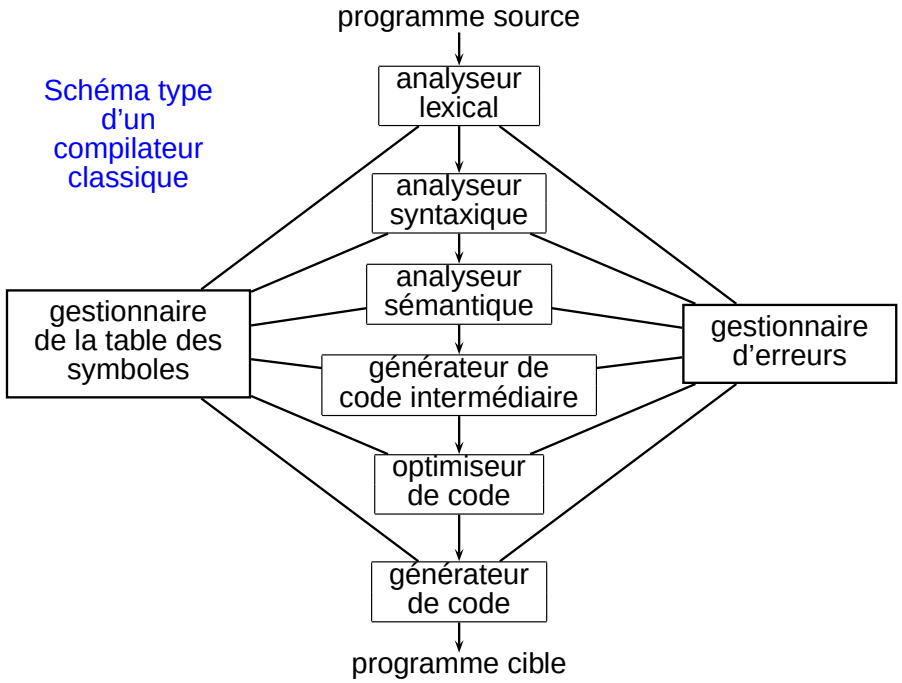
Amphi 7: graphes III (optimisation combinatoire).

Amphi 8: graphes IV (topologie).

Plan

I. Introduction

Schéma type d'un compilateur classique



II. Recherche de motifs dans un texte

Soit Σ un ensemble de lettres, qu'on appelle alphabet.

Un mot est une suite finie de lettres de Σ.

La longueur de $w = w_0 w_1 \dots w_{k-1}$ est $|w| = k$.

Il existe un unique mot de longueur 0, le mot vide ε.

L'ensemble des mots sur un alphabet Σ est noté Σ^*.

Concaténation:

$$w = uv = u|v = (u_0 u_1 \dots u_{k-1})(v_0 v_1 \dots v_{\ell-1}) = u_0 u_1 \dots u_{k-1} v_0 v_1 \dots v_{\ell-1}$$

de longueur $|w| = |u| + |v|$.

Le problème

Pb. on cherche si le mot $M = m_0 m_1 \dots m_{k-1}$ se trouve dans le texte $T = t_0 t_1 \dots t_{\ell-1}$.
Rem. Google, emacs, etc.; génôme.

Approche naïve: on fait glisser le motif M sous le texte T.

```
T1 = aaaaaaaaaaaaaaaaaaab
 M = aaaaaab
      aaaaaab
       ...
                  aaaaaab
```

Le nombre de comparaisons est $|M||T_1|$. Avec

```
T2 = bbbbbbbbbbbbbbbbbbb
```

le nombre de comparaisons n'est que $|T_2|$.
On aimerait avoir une méthode plus insensible aux propriétés du texte et du motif.

L'algorithme de Knuth-Morris-Pratt (KMP)

Idée: dans le cas de la méthode naïve, on oublie ce qui a été lu jusqu'à présent, ce qui est dommage.
Soit $M =$`abaabbab`, $T =$`ababbabaabaabb...`:

```
T = ababbabaabaabbab...
M = abaabbab
    ***#    reprendre après le dernier ab
      abaabbab
        #   reprendre au début
         abaabbab
         *****#    reprendre après le dernier aba
            abaabbab
               *****
```

À la première différence entre M et T, on a gardé en mémoire que les deux dernières lettres lues dans T coïncident avec les deux premières lettres du motif M; donc on peut poursuivre la lecture sur la troisième lettre du motif sans revenir en arrière sur le texte.

Il y a alors échec immédiat et comme aucun suffixe des cinq premières lettres du texte ne correspond à un préfixe du motif, on en déduit qu'on peut décaler entièrement le motif.

Idée de KMP: en cas d'échec, on calcule le plus long suffixe de T jusqu'à l'indice actuel qui soit préfixe pour $M \Rightarrow$ on s'avance dans la recherche suivante.

Déf. Soit $w \in \Sigma^*$ non vide. Pour toute décomposition $w = f_0 f_1 \dots f_{n-1}$ avec $f_i \neq \varepsilon$, on dira que f_i est un facteur de w; f_0 est un préfixe (ou facteur gauche), f_{n-1} un suffixe (ou facteur droit).

Un algorithme grossier

```
i <- 0; j <- 0;
tantque i < |T|-|M|+1
  // ici: 0 ⩽ j ⩽ |M| et T[i..i+j-1] = M[0..j-1]
  si j == |M| alors
    afficher "coïncidence sur T[i..i+j-1]";
    j <- 0; i++;
  sinon
    si M[j] == T[i+j] alors
      j++;
    sinon
      déterminer le plus long suffixe
      T[i+j-k+1..i+j] qui soit un préfixe
      M[0..k-1] de M, avec 0 ⩽ k < |M|-1;
      i <- i+j-k+1; j <- k;
```

Si on veut que cet algorithme soit efficace, on doit trouver un moyen d'effectuer l'étape 3 rapidement.

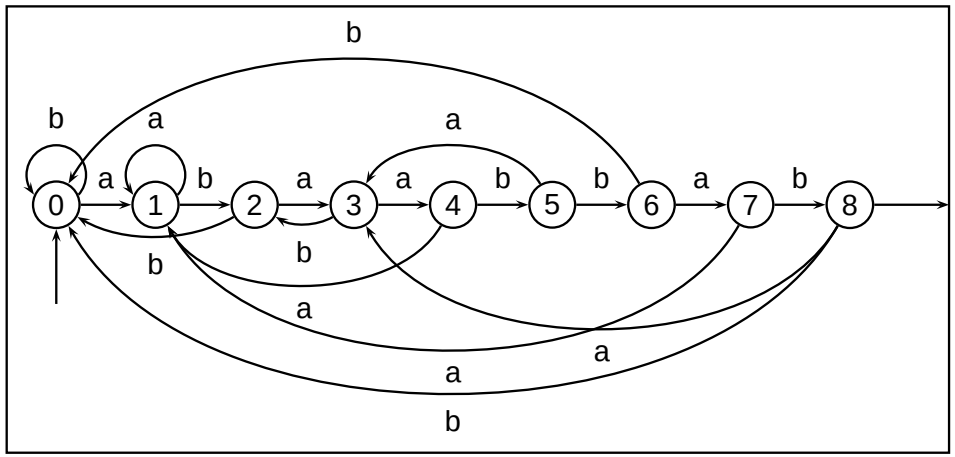
Supposons qu'on stocke les préfixes de $M =$abaabbab dans le tableau $\mathscr{P}$:

k	$M[0..k-1]$	a	b
0	ε	**1**	0
1	a	1	**2**
2	ab	**3**	0
3	aba	**4**	2
4	abaa	1	**5**
5	abaab	3	**6**
6	abaabb	**7**	0
7	abaabba	1	**8**
8	abaabbab	3	0

À chaque préfixe $\mu = M[0..k-1]$, on peut associer deux mots $\mu \cdot a$ et $\mu \cdot b$. L'un des deux est le mot suivant dans $\mathscr{P}$. Pour l'autre, on détermine son plus long suffixe qui est aussi préfixe (donc dans $\mathscr{P}$ et d'indice plus petit).
À chaque lettre qui arrive, on sait tout de suite quelle est l'action suivante à accomplir. En fait on vient de construire un automate de reconnaissance de M.

L'automate

On prend comme états les entiers k entre 0 et 8. L'entrée correspond à l'état 0, la sortie (motif reconnu), à l'état 8. Il y a $|\Sigma|$ transitions possibles d'un état à l'autre, en fonction de la lettre suivante.



Construction de la table de transition

```
static int[][] automate(String M){
  String[] prefixes = new String[M.length()+1];
  int[][] trans = new int[M.length()+1][2];

  prefixes[0] = "";
  for(int i = 1; i <= M.length(); i++)
    prefixes[i] = M.substring(0, i);
  for(int i = 1; i <= M.length(); i++){
    String mu;
    mu = prefixes[i] + "a";
    trans[i][0] = PlusLongSuffixe(prefixes, mu);
    mu = prefixes[i] + "b";
    trans[i][1] = PlusLongSuffixe(prefixes, mu);
  }
  // cas particulier de epsilon
  if(M.charAt(0) == 'a') trans[0][0] = 1;
  else trans[0][1] = 1;
  return trans;
} // coût: |M| x coût(PlusLongSuffixe)
```



```
static int PlusLongSuffixe(String[] prefixes,
                           String mu){
  int jmax = 0, lg = mu.length();

  for(int j=1; (j<prefixes.length) && (j<=lg); j++)
    if(prefixes[j].equals(mu.substring(lg-j, lg)))
      jmax = j;
  return jmax; // coût = O(|M|^2)
}
static void KMP(String T, String M){
  int[][] transition = automate(M);
  int etat = 0, ind, k = M.length();

  for(int i = 0; i < T.length(); i++){
    ind = (T.charAt(i) == 'a' ? 0 : 1);
    etat = transition[etat][ind];
    if(etat == k)
      P("Occurrence T["+i+".."+(i+k-1)+"]\n");
  }}
```

Coût: $O(|M|^3)+O(|T|)$.

Compléments

Thm. (Knuth, Morris, Pratt) il existe un algorithme de construction de la table de transition en temps $O(|M|)$.

Coro. Le coût de KMP est $O(|T|+|M|)$.

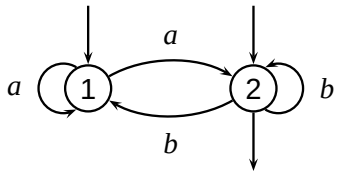
Rem. Quand $|\Sigma|$ est grand, la table est trop grande et est rarement utilisée. On peut remplir l'automate à la demande.

Rem. On peut généraliser à des multi-motifs (ensemble fini de motifs): cf. poly.

Plus rapide: Boyer-Moore (comparaison de motifs par la fin).

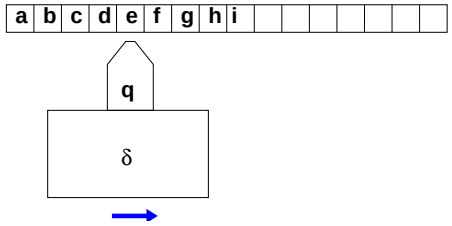
III. Rappels sur les automates fi nis

Un automate fini A est un quintuplet $A=(Q,\Sigma,\Delta,I,F)$ où:
Σ est un alphabet donné,
Q est un ensemble fini d'états,
$I\subset Q$ est l'ensemble des états initiaux,
$F\subset Q$ est l'ensemble des états finaux,
Δ est un sous-ensemble de $Q\times\Sigma\times Q$, ensemble des transitions.
Il est commode de représenter un automate par un graphe, c'est-à-dire un ensemble de nœuds (sommets) reliés par des flèches (arcs).
Ex.



$\Sigma=\{a,b\}$, $Q=\{1,2\}$, $I=\{1,2\}$, $F=\{2\}$,
$\Delta=\{(1,a,1),(1,a,2),(2,b,2),(2,b,1)\}$.

Illustration graphique



- La bande de lecture n'est pas modifiable (*read-only*).
- La tête de lecture avance d'une case vers la droite à chaque transition (repérée par une étiquette).
- L'état q évolue vers l'état q' si le caractère e est sous la tête de lecture.

Défi nitions

Soit (p,a,q) une transition: p est l'origine, a l'étiquette, q l'extrémité; on note $p \xrightarrow{a} q$.
Un chemin est une suite de transitions successives:

$$c = (q_0,a_1,q_1)(q_1,a_2,q_2)\cdots(q_{n-1},a_n,q_n).$$

On le note

$$c : q_0 \xrightarrow{a_1} q_1 \cdots q_{n-1} \xrightarrow{a_n} q_n.$$

Un chemin est réussi (ou acceptant) si $q_0 \in I$, $q_n \in F$.

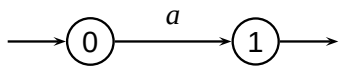
Un mot est accepté s'il est l'étiquette d'au moins un chemin réussi.

Le langage $L(A)$ reconnu par A est l'ensemble des mots reconnus par A.

Deux automates sont équivalents ssi ils reconnaissent le même langage.

Langages reconnaissables

Déf. Un langage $L \subset \Sigma^*$ est reconnaissable ssi il existe un automate A qui le reconnaisse.

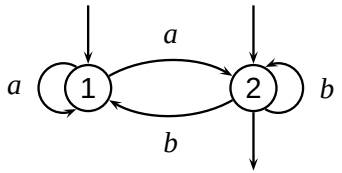
Ex. $L = \{a\}$ est reconnaissable par 

Th. (exercice) L'ensemble des langages reconnaissables est fermé par union, intersection, complément, produit, étoile.

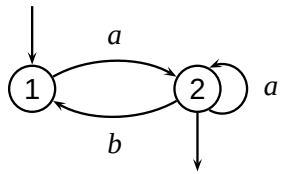
Déterminisme et non-déterminisme

Déf. Un automate est déterministe si $I = \{q_0\}$ et si pour tout état $q \in Q$, tout $a \in \Sigma$, il existe au plus une transition $q \xrightarrow{a} q'$.

Ex. L'automate suivant n'est pas déterministe.



Ex. L'automate suivant est déterministe.



Propriétés des automates déterministes

Prop. Δ est la relation fonctionnelle de

$$\begin{array}{rccc} \delta: & Q\times\Sigma & \to & Q \\ & (q,a) & \mapsto & q' \text{ tq } q \xrightarrow{a} q' \end{array}$$

Prop. On teste la reconnaissance de w par A en temps $O(|w|)$.

Ex. cf. automate de KMP.

Th. **[Rabin - Scott]** Tout langage reconnu par un automate fini non déterministe est reconnu par un automate fini déterministe.

Th. **[Myhill - Nerode]** Tout langage reconnu par un automate fini est reconnu par un automate déterministe minimal unique à un isomorphisme près sur le nom des états.

Représentations d'automate déterministe

- En dur, dans le contrôle fini du programme

```
char c = in.read(); int q = q0;
switch(q){
case 0:
  if (c == 'a') ... else if (c == 'b')  ...;
  break;
case 1:
  if (c == 'b') ... else if (c == 'c') ... ;
  break;
...
case 10:
  if ...  break;
}
```

- matrice $|Q|\times|\Sigma|$ pour représenter la fonction de transition.
- vecteur de $|Q|$ listes d'association (c',q') (graphe dont les sommets sont les états et les arcs sont les transitions étiquetées par les lettres; cf. cours sur les graphes).

IV. Expressions régulières (rationnelles)

A) Défi nitions

Déf. L'ensemble des langages rationnels sur Σ^* est le plus petit ensemble de langages $\mathrm{Rat}(\Sigma^*)$ tq:
(a) $\mathrm{Rat}(\Sigma^*)$ contient $\emptyset$ et $\{a\}$ pour $a\in\Sigma$.
(b) $\mathrm{Rat}(\Sigma^*)$ est fermé pour l'union finie, produit et étoile (si L et L' sont éléments de $\mathrm{Rat}(\Sigma^*)$, alors $L\cup L'$, $L\cap L'$ et L^* sont aussi dans $\mathrm{Rat}(\Sigma^*)$).
Prop. Soit $\mathrm{Rat}(\Sigma^*)_0$ l'ensemble des langages finis de Σ^* et $\mathrm{Rat}(\Sigma^*)_{n+1}$ l'ensemble des langages s'écrivant sous la forme $K\cup K'$, $K\cap K'$ ou K^*, avec K et K' dans $\mathrm{Rat}(\Sigma^*)_n$. Alors:

$$\mathrm{Rat}(\Sigma^*)_0 \subset \mathrm{Rat}(\Sigma^*)_1 \subset \cdots$$

et la réunion de tous ces ensembles forme $\mathrm{Rat}(\Sigma^*)$.
Ex. ensemble des mots contenant u, car $\Sigma^* u \Sigma^*$; mots de longueur paire $(\Sigma^2)^*$.

Expressions rationnelles (régulières)

On définit des expressions rationnelles formellement par:

(1) 0 et 1 sont des expressions rationnelles;

(2) toute lettre $a\in\Sigma$ est une expression rationnelle;

(3) si e et e' sont des expressions rationnelles, alors l'union $(e+e')$, (ee') et e^* sont des expressions rationnelles.

La valeur d'une expression e, notée $v(e)$ est le langage représenté par e.

Règles:

$$\begin{aligned} v(0) &= \emptyset, \\ v(1) &= \{\varepsilon\}, \\ v(a) &= \{a\} \text{ pour } a \in \Sigma, \\ v((e+e')) &= v(e) \cup v(e'), \text{(alternative, union)} \\ v((ee')) &= v(e)v(e'), \text{(concaténation, intersection)} \\ v(e^*) &= v(e)^*. \end{aligned}$$

Rem. Parfois, on note $e|e'$ au lieu de $e+e'$.
Ex.

- $v((a+b)^*) = \{a,b\}^*$;
- $v(a(ab)^*) =$ toutes les chaînes commençant par a et suivies d'un nombre indéterminé de ab;
- $v((i+j)(0+1)^*) =$ mot commençant par i ou j et suivi de nombres écrits avec des 0 ou des 1.

Algèbre de Kleene

Posons $e \leq e'$ pour $e+e' = e'$.
Les 13 lois suivantes sont vérifiées et définissent une algèbre de Kleene.

(1) $e+f = f+e$
(2) $e+(f+g) = (e+f)+g$
(3) $e+0 = e$
(4) $e(fg) = (ef)g$
(5) $1e = e1 = e$
(6) $e(f+g) = ef+eg$
(7) $(e+f)g = eg+fg$
(8) $0e = e0 = 0$ demi-anneau

(9) $e+e = e$

(10) $1+ee^* = e^*$
(11) $1+e^*e = e^*$

(12) $f+eg \leq g \Rightarrow e^*f \leq g$ (12') $eg \leq g \Rightarrow e^*g \leq g$
(13) $f+ge \leq g \Rightarrow fe^* \leq g$ (13') $ge \leq g \Rightarrow ge^* \leq g$

12-12' et 13-13' sont équivalents.

Rem. Deux expressions régulières peuvent représenter le même langage ($(a^*b)^*a^*$ représente également $\{a,b\}^*$).

Rem. Il n'y a pas d'écriture canonique des expressions régulières (formes normales), même sur une *base* finie de celles-ci.

Th. **[Kleene]** Les langages reconnus par un automate fini sont exactement ceux décrits par les expressions rationnelles (régulières).

Dém. on va donner une démonstration algorithmique de la réciproque.

B) De l'expression régulière à l'automate

Stratégie générale: construire un automate fini non déterministe, puis le déterminiser, puis le minimiser si nécessaire.

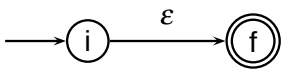
Algorithme de Thompson: on construit l'automate par combinaisons d'automates correspondant à la décomposition de l'expression régulière en sous-expressions.

À chaque étape:

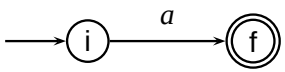
- au plus deux états nouveaux sont créés $\Rightarrow$ l'automate final a au plus deux fois plus d'états que de symboles dans l'expression régulière;
- chaque sous-automate a exactement un état final; aucun arc n'entre dans l'état de départ, aucun arc ne quitte l'état final.

Rem. Il est commode d'utiliser des transitions étiquetées par ε.

Le cas de ε: on crée deux nouveaux états

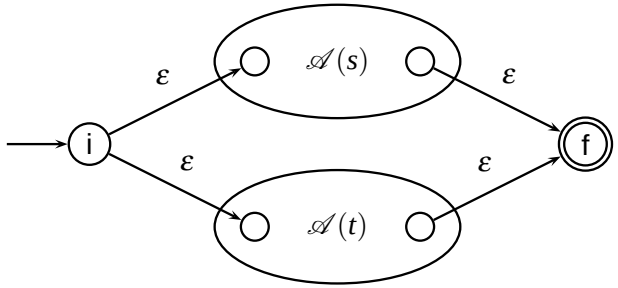


Le cas de a: on crée deux nouveaux états



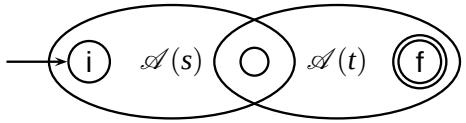
Rem. Si on rencontre plusieurs fois le même symbole, on crée autant d'automates différents, avec des états nouveaux à chaque fois.

Soient $\mathscr{A}(s)$ et $\mathscr{A}(t)$ les automates reconnaissant les expressions régulières s et t.
Le cas de $s+t$: on crée deux nouveaux états



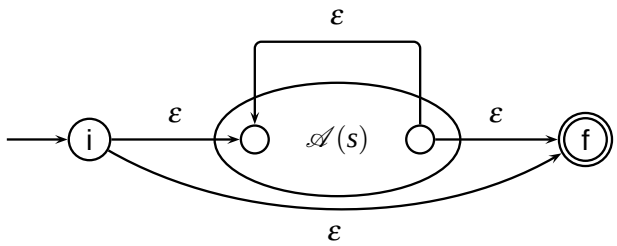
Transitions de i vers l'état de départ de $\mathscr{A}(s)$; transition de l'état final de $\mathscr{A}(s)$ vers f; idem pour $\mathscr{A}(t)$.
L'automate reconnaît bien $L(s) \cup L(t)$.

Le cas de st: on crée deux nouveaux états



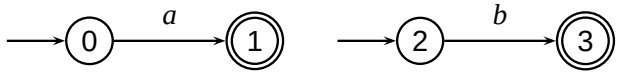
On fusionne l'état final de $\mathscr{A}(s)$ et l'état initial de $\mathscr{A}(t)$.
L'automate reconnaît bien $L(s)L(t)$.

Le cas de s^*: on crée deux nouveaux états

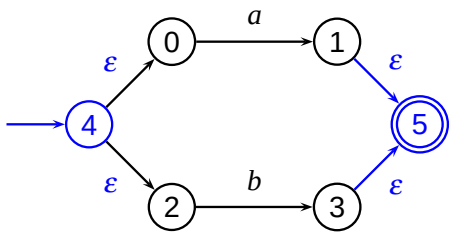


Exemple

On considère $r = (a+b)^*ab$.
On commence par $a+b$ et on assemble:

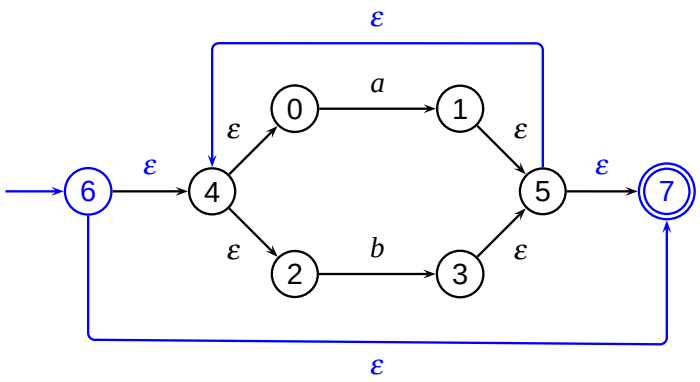


pour construire:

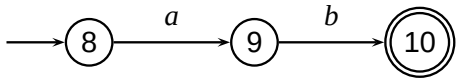


C'est le même automate pour $(a+b)$.

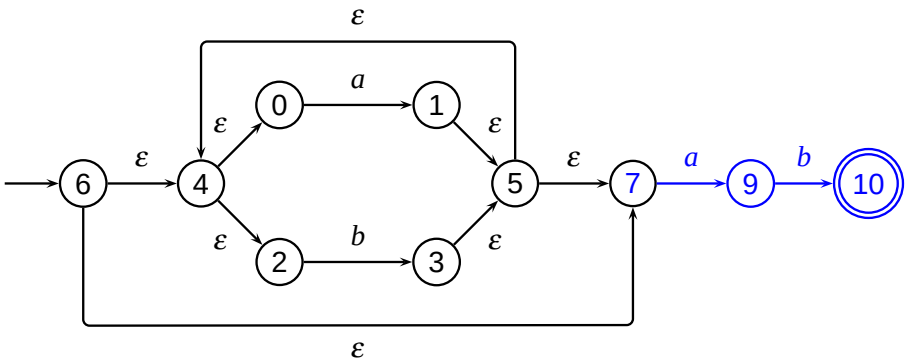
On passe maintenant à l'automate pour $(a+b)^*$:



Il ne reste plus qu'à ajouter bout à bout l'automate pour ab:



Automate final reconnaissant $(a+b)^*ab$:



Rem. Il existe un algorithme de complexité $O(N|x|)$ qui teste si un automate construit par la méthode précédente reconnaît le mot x (avec N le nombre d'états). Cet algorithme est implanté dans l'utilitaire Unix appelé **`grep`**.

Les expressions régulières dans la vraie vie

Les commandes d'Unix contiennent souvent des expressions régulières.

- **`/bin/sh`** [Bourne] pour les noms de fichiers,
- **`sed, ed`** Stream editor, Editor
- **`grep`** Get Regular ExPression
- Emacs **`ESC-x re-search-forward`**
- **`awk, perl`**
- **`lex`** **[Lesk]** Méta-compilateur d'expressions régulières pour C

Toutes ont leur syntaxe propre. En général, elles essaient d'avoir les expressions régulières les plus déterministes possible, sauf **`awk`** et **`perl`**. Par exemple,

`% ls *.java .??*`

liste les fichiers de suffixe **`.java`** ou de plus de 3 caractères commençant par un point (.).

`% sed -e 's/^  *$//'`

remplace les lignes blanches par des lignes vides sur l'entrée standard.

`% grep 'Contrôle' *`

imprime toutes les lignes contenant le mot `Contrôle` dans tous les fichiers du répertoire courant.

`% grep -i 'analyse.*lexicale' *.tex`

imprime toutes les lignes contenant le mot `analyse` suivi du mot `lexicale` dans tous les fichiers de suffixe **`.tex`** (L'option **`-i`** signifie que majuscules et minuscules sont confondues).

`% grep '^ *$' a3.tex | wc -l`

compte le nombre de lignes blanches dans le fichier **`a3.tex`**

La commande Unix la plus puissante de l'univers

```
% find ~ -name '*.tex' -exec grep poly {} \; -print
```

permet de rechercher tous les fichiers dont le suffixe est **`tex`** contenant la chaîne **`poly`**, et ce, dans **tous** ses répertoires.

Rem. on peut utiliser aussi **`egrep`** (expressions régulières plus puissantes) ou **`fgrep`**.

V. Compléments sur les chaînes de caractères

- Les caractères ont un type primitif

```
char c = 'a';
```

  - Caractères spéciaux:
    `'\n'` (*newline*) **`'\r'`** (retour charriot) **`'\t'`** (tabulation)
    `'\\'` (*backslash*) **`'\''`** (apostrophe) **`'\"'`** (guillemet)
  - Fonctions statiques de la classe **`Character`**:
    `Character.isLetter(c)`, **`Character.isDigit(c)`**, **`Character.isLetterOrDigit(c)`**, ...

- Les chaînes de caractères sont des objets de la classe **`String`**

```
String s = "Vive l'informatique!"
```

  - Méthodes de la classe **`String`**:
    `s.length()` longueur de s,
    `s.equals(t)` égalité de s et t,
    `s.indexOf(c)` première occurence de c dans s,
    `s.charAt(i)` i-ème caractère de s, ...

Chaînes modifi ables

- Les chaînes de la classe **`String`** ne sont pas modifiables.
- Les chaînes de caractères modifiables sont des objets de la classe **`StringBuffer`**

```
StringBuffer s = new StringBuffer();
```

  Méthodes de la classe **`StringBuffer`**:
  - **`s.insert(i,t)`** insère la chaîne t après le i-ème caractère de s;
  - **`s.append(t)`** met t au bout de s; etc.

- Par exemple

```
String x = "a" + 4 + "c";
```

  équivaut à

```
String x =
 new String(new StringBuffer().\\
            append("a").append(4).append("c"));
```


Conversion des caractères en entiers

- Les caractères ont un code Unicode sur 16 bits (non signés), encore appelé ISO 10646 (سلام, お早う,你好)
- En Java: **`char`** $\subset$ **`int`**, mais **`char`** $\not\subset$ **`short`**.
- Les 256 premières valeurs de l'Unicode sont l'ISO-latin-1 (ISO 8859-1).
- Les 128 premières valeurs sont l'ASCII (*American Standard Codes for Information Interchange*).

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht	nl	vt	np	cr	so	si
10	dle	dc1	dc2	dc3	dc4	nak	syn	etb	can	em	sub	esc	fs	gs	rs	us
20	sp	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
30	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
40	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
50	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
60	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
70	p	q	r	s	t	u	v	w	x	y	z	{	&	}	~	del

Attention: les vieux logiciels ne sont souvent prévus que pour les caractères de code ASCII.

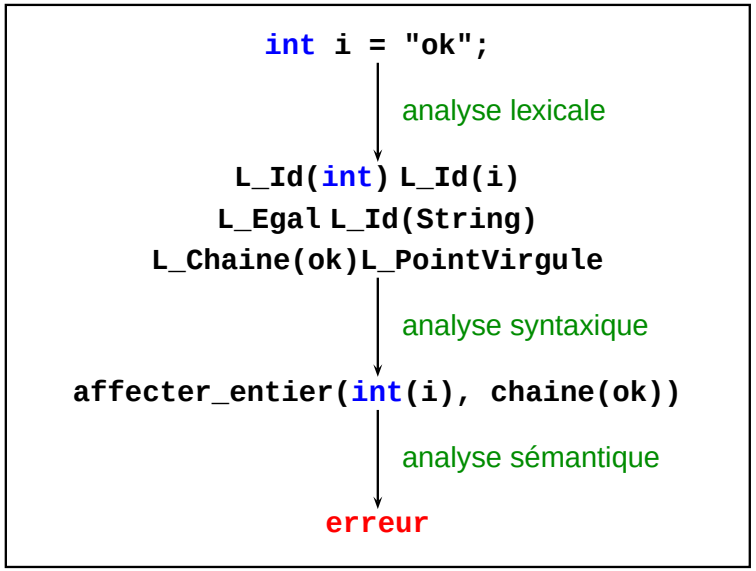
UTF-8

Compatibilité ascendante du logiciel $\Rightarrow$ UTF-8
UTF-8 (*Unicode Transformation Format* - 8 bits)
$=$ code préfixe en longueur variable.

Unicode	Code UTF-8
0000 – 007f	0xxxxxxx
0080 – 07ff	110xxxxx 10xxxxxx
0800 – ffff	1110xxxx 10xxxxxx 10xxxxxx

Cf. Cours de majeure 1 (info): Codes et théorie de l'information.

VI. Analyse lexicale



À quoi sert l'analyse lexicale?

But: isoler les lexèmes dans une chaîne de caractères.

Déf. Lexème: entité importante pour une phase ultérieure de calcul (en anglais: *token*).

Ex. De

```
public static void main(String[] args){
  int n = 2;
}
```

on récupère les mots clefs **public**, **static**, etc., des variables **args**, **n** et une constante 2.

Plus généralement: un identificateur, une constante numérique, un opérateur, une chaîne de caractères, une constante caractère, etc.

Rem. D'habitude, les espaces, tabulations, retours à la ligne, commentaires ne sont pas des lexèmes.

Déroulement

- Entrée: la chaîne de caractères suivante

  " expression = 3 * x + 2 ; "

- Résultat: la suite de lexèmes suivante

  L_Id(expression) L_Egal L_Nombre(3) L_Mul L_Id(x)
  L_Plus L_Nombre(2) L_PointVirgule L_EOF

- Même résultat si la chaîne d'entrée est

```
" expression =
          3 * x
        + 2 ; "
```

Un exemple plus compliqué

- Entrée: la chaîne de caractères suivante

```
"  class PremierProg{
     public static void main(String[] args){
       System.out.println(\"Bonjour!\");
     }
   } "
```

- Résultat: la suite de lexèmes suivante

```
L_Id(class) L_Id(PremierProg) L_AccG L_Id(public)
L_Id(static) L_Id(void) L_Id(main)
L_ParG L_Id(String) L_CroG L_CroD L_Id(args) L_ParD
L_AccG L_Id(System) L_Point L_Id(out) L_Point
L_Id(println) L_ParG L_Chaine(Bonjour!) L_ParD
L_PointVirgule L_AccD L_AccD L_EOF
```


Grammaire du langage simplifi é

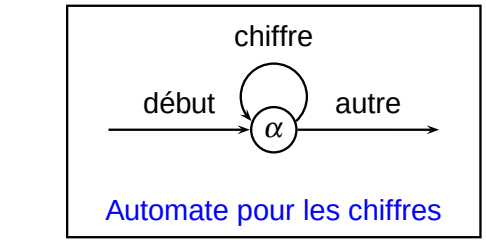
Trois lexèmes (*tokens*) définis par des expressions régulières:

Identificateur $=$ Lettre $($Lettre $|$ Chiffre$)^*$
Entier $=$ Chiffre Chiffre*
Operateur $= + \mid - \mid * \mid /$

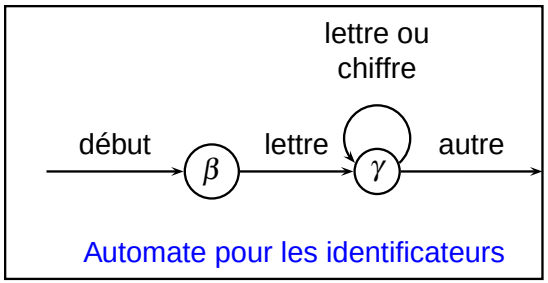
Lettre $= a \mid b \ldots \mid z \mid A \mid B \ldots \mid Z$
Chiffre $= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$
Blancs $=$ (`' '` | `'\t'` | `'\r'` | `'\n'`)*

À partir de cette description, on construit l'analyseur lexical.

Diagrammes de transition



Automate pour les chiffres

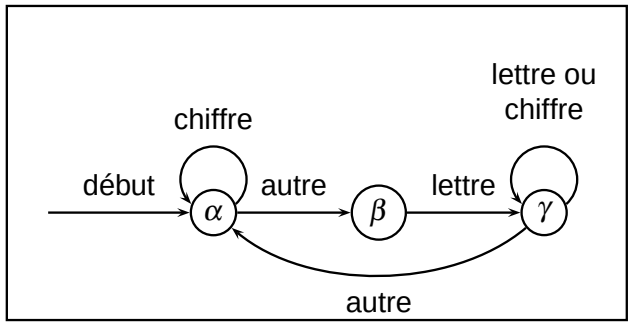


Automate pour les identificateurs

À chaque tour de boucle, on accumule les caractères lus. Si on sort avec une chaîne non nulle, on a trouvé un lexème et son type.

Un peu de LEGO

On assemble les diagrammes de transition:



On a l'embryon du programme qui reconnaît un entier ou un identificateur.

Rappel sur les entrées-sorties standards

Impression: **System.out.print**, **System.err.print**

```
public static void main(String[] args){
  BufferedReader in =
      new BufferedReader(
            new InputStreamReader(System.in));
  try{
    String s;
    while((s = in.readLine()) != null){
      int n = Integer.parseInt(s);
      System.out.println(n);
    }
  } catch (IOException e) {
    System.err.println(e);
  }
}
```

L'INEXPLICABLE IDIOME EN JAVA		
	System.in	flot de **byte**s
	InputStreamReader	flot de caractères
	BufferedReader	flot de caractères tamponné

Le programme en Java

```
class Lexeme{
  final static int L_Nombre = 0, L_Id = 1,
                   L_Plus = '+', L_Moins = '-',
                   L_Mul = '*', L_Div = '/',
                   L_ParG = '(', L_ParD = ')',
                   L_CroG = '[', L_CroD = ']',
                   L_EOF = -1;
  int nature;
  int valeur;
  String nom;

  Lexeme(int t, int i) { nature = t; valeur = i; }
  Lexeme(int t, String s) { nature = t; nom = s; }
  Lexeme(int t) { nature = t; }

  // aller chercher le lexème suivant
  static Lexeme suivant() { ... }
```


Gestion du caractère courant

```
static int c;    // caractère courant

static void avancer(){
  try{
    c = in.read();
  } catch (IOException e) { }
}

static void sauterLesBlancs(){
  while(Character.isWhitespace(c))
    avancer();
}
```



```
static Lexeme suivant(){
  sauterLesBlancs();
  if(Character.isLetter(c))
    return new Lexeme(L_Id, identificateur());
  else if(Character.isDigit(c))
    return new Lexeme(L_Nombre, nombre());
  else switch(c){
    case '+': case '-': case '*': case '/':
    case '(': case ')':
      char c1 = (char)c;
      avancer();
      return new Lexeme(c1);
    default:
      throw new Error ("Caractère illégal");
  }
}
```

Attention: revoir la fi n de fi chier.

Les deux expressions régulières simples

```
static String identificateur(){
  StringBuffer r;
  while(Character.isLetterOrDigit(c)){
    r = r.append(c);
    avancer();
  }
  return r;
}
static int nombre(){
  int r = 0;
  while(Character.isDigit(c)){
    r = r * 10 + c - '0';
    avancer();
  }
  return r;
}
```

Quand il y a de nombreux mots-clé, on leur donne (comme pour les opérateurs) des numéros de lexème distincts. Chaque lexème identificateur est comparé à une table des mots-clé.

Résumé du cours

- Recherche de motifs.
- Automates et expressions régulières.
- Caractères – chaînes de caractères.
- Analyse lexicale.
- Certains outils (**lex**) (JLex) peuvent faire produire automatiquement des analyseurs lexicaux.
- Lectures complémentaires: Kozen, *Automata and Computability*; le dragon (Aho/Sethi/Ullman – *Compilers: Principles, Techniques, and Tools. Addison–Wesley, 1986*, traduit en français).

Prochains rendez-vous: TD cet après-midi; amphi 04 mercredi 28/02.

Bonnes vacances!

Rappel sur les tables

Table: Clés $\mapsto$ Valeurs
Clés $= \{$**"int"**, **"static"**, **"if"**, **"while"**, $\ldots\}$
Valeurs $= \{$**L_Int**, **L_Static**, **L_If**, **L_While**,$\ldots\}$

Représentations statiques:

- tableau de n Paires (c_i, v_i) $(0 \leq i < n)$ ou 2 tableaux parallèles Clés et Valeurs de taille n;
- tableau de n Paires (c_i, v_i) $(0 \leq i < n)$ ordonné sur les clés; recherche dichotomique ou par interpolation;
- tableau de p listes d'associations avec hachage: Clés $\mapsto [0, p-1]$.

Représentations dynamiques:

- liste d'associations $\langle (c_0, v_0), (c_1, v_1), \ldots (c_{n-1}, v_{n-1}) \rangle$;
- arbre de recherche avec nœuds étiquetés par (c_i, v_i) $(0 \leq i < n)$ ordonné sur les clés.