SÉANCE 8

ANALYSE LEXICALE

SUJETS

Le rôle de l'analyseur lexical

Spécification des tokens (catégories lexicales)

Expressions régulières (expression rationnelle)

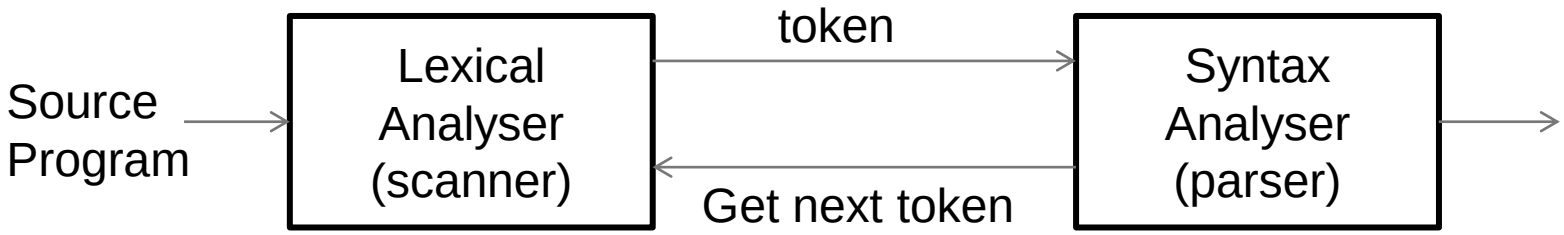
Automates finies déterministes et non déterministes

LE RÔLE DE L'ANALYSEUR LEXICAL

uOttawa
L'Université canadienne
Canada's university

L'analyse lexical est la première phase du compilateur

- Rôle: lit les caractères d'entrée et produit une séquence de catégories lexicales que l'analyseur syntaxique utilise pour l'analyse syntaxique
- Enlève les espaces blancs



ANALYSE LEXICALE

Il y a plusieurs raisons pour lesquelles on sépare la phase d'analyse de compilation en analyse lexicale et l'analyse syntaxique:

- Modèle plus simple (en couches)
- Efficacité du compilateur

Des outils spécialisés ont été développés afin d'aider à automatiser la construction des deux séparément

LEXÈMES

Lexème (entité lexicale): séquence de caractères dans le programme source qui est matché par le motif du token

- Les lexèmes d'un langage de programmation incluent:
  - **Identificateurs**: noms de variables, méthodes, classes, package et interfaces…
  - **Littérales**: valeurs fixes (ex. "1", "17.56", "0xFFE" …)
  - **Opérateurs**: pour opérations mathématique, Booléennes et logiques (ex. "+", "-", "&&", "|" …)
  - **Mots spéciaux**: mots-clés (ex. "if", "for", "public" …)

TOKENS, MOTIFS, LEXÈMES

uOttawa
L'Université canadienne
Canada's university

Token: catégorie de lexèmes

Un motif est une règle qui décrit l'ensemble de lexèmes qui représentent un token (catégorie lexicale) particulier dans le programme source

EXEMPLES DE TOKENS

uOttawa
L'Université canadienne
Canada's university

```
double pi = 3.1416;
```

La chaine des caractère `pi` est un lexème pour le token "identificateur"

Token	Exemple de Lexèmes	Description informelle du motif
type	double, int, byte, long, float	double or int or byte or long or float
if	if	if
booelan_operator	<. <=, ==, >, >=	< or <= or == or > or >=
id	pi, count, d2	Lettre suivie de lettres et chiffres
literal	3.1414, "test"	Toute chaîne alphanumérique de caractères

LEXÈMES ET TOKEN (CATÉGORIES PLUS DÉTAILLÉS)


```
Index = 2 * count +17;
```

Lexèmes	Tokens
Index	variable
=	equal_sign
2	int_literal
*	multi_op
Count	variable
+	plus_op
17	int_literal
;	semicolon

ERREURS LEXICALES

Peu d'erreurs sont perceptibles au niveau lexical

- L'analyseur lexical possède une vue localisée du programme source

On laisse les autres phases du compilateur s'occuper des erreurs

SPÉCIFICATION DES TOKENS

On a besoin d'une notation puissante afin de spécifier les motifs des tokens

- L'expression régulière au secours !!

Dans le processus d'étude des expressions régulières, on visite les sujets suivants:

- Opération sur les langages
- Définitions régulières
- Notation abrégée (shorthand notion) des expressions régulières

RAPPEL: LANGAGES

Σ: alphabet, c'est un ensemble fini qui consiste de tous les caractères d'entrée ou symboles

Σ^*: fermeture de l'alphabet, l'ensemble de toutes les chaînes possibles dans Σ, incluant la chaîne vide ε

Un langage (formel) est un sous-ensemble spécifique de Σ^*

OPÉRATIONS SUR LES LANGAGES

uOttawa
L'Université canadienne
Canada's university

Operation	Definition
union of L and M written $L \cup M$	$L \cup M = \{s \mid s \in L \text{ or } s \in M\}$
concatenation of L and M written LM	$LM = \{st \mid s \in L \text{ and } t \in M\}$
Kleene closure of L written L^*	$L^* = \bigcup_{i=0}^{\infty} L^i$
positive closure of L written L^+	$L^+ = \bigcup_{i=1}^{\infty} L^i$

OPÉRATIONS SUR LES LANGAGES

uOttawa
L'Université canadienne
Canada's university

Format non-mathématique:

- **L'union entre les langages L et M:** l'ensemble des chaînes qui appartiennent à au moins un des deux langages
- **Concaténation des langages L et M:** l'ensemble de toutes les chaînes de forme **st** où **s** est une chaîne de **L** et **t** est une chaîne de **M**
- **Intersection entre les langages L et M:** l'ensemble de toutes les chaînes qui se trouvent dans les deux langages
- **Fermeture Kleene (d'après Stephen Kleene)**: l'esemble de toutes les chaînes qui sont des concaténations de **0 ou plus** de chaînes du langage original
- **Fermeture positive** : l'ensemble de toutes les chaînes qui sont des concaténations de **1 ou plus** de chaînes du langage original

EXPRESSIONS RÉGULIÈRES

uOttawa
L'Université canadienne
Canada's university

Une expression régulière est une notation compacte pour décrire une chaîne.

Typiquement, un identificateur est une lettre suivie d'un zéro ou plus de lettres ou de chiffres→ lettre (lettre | chiffre)*

|: ou

***: zéro ou plus de**

RÈGLES

ε Est une expression régulière qui dénote {ε}, l'ensemble contenant une chaîne vide

Si *a* est un caractère dans Σ, donc *a* est une expression régulière qui dénote {*a*}, l'ensemble contenant la chaîne *a*

Supposons *r* et *s* sont des expressions régulières qui dénotent les langages L et M, donc

- (*r*) |(*s*) est une expression régulière qui dénote L∪M.
- (*r*)(*s*) est une expression régulière qui dénote LM.
- (*r*) * est une expression régulière qui dénote L*.

CONVENTIONS DE PRÉCÉDENCE

L'opérateur unaire * possède la plus haute précédence et il est associative à gauche.

La concaténation possède la deuxième plus haute précédence et il est associative à gauche.

| possède la plus basse précédence et est associative à gauche.

(a)|(b)*(c)→a|b*c

PROPRIÉTÉS D'UNE EXPRESSION RÉGULIÈRE

uOttawa
L'Université canadienne
Canada's university

Axiom	Description
$r\|s = s\|r$	$\|$ is commutative
$r\|(s\|t) = (r\|s)\|t$	$\|$ is associative
$(rs)t = r(st)$	concatenation is associative
$r(s\|t) = rs\|rt$ $(s\|t)r = sr\|tr$	concatenation distributes over $\|$
$\varepsilon r = r$ $r\varepsilon = r$	ε is the identity for concatenation
$r^* = (r\|\varepsilon)^*$	relation between * and ε
$r^{**} = r^*$	* is idempotent

EXEMPLES D'EXPRESSIONS RÉGULIÈRES

uOttawa
L'Université canadienne
Canada's university

Si Σ = {a,b}

1. a dénote {a}
2. a | b dénote {a,b}
3. (a | b) (a | b) dénote {aa, ab, ba, bb}
4. a* dénote {ε, a, aa, aaa, aaaa…}
5. (a | b)* dénote l'ensemble de toutes les chaînes de a et b (incluant ε)
6. a | a*b dénote {a, b, ab, aab, aaab, aaaab, …}

DÉFINITIONS RÉGULIÈRES

uOttawa
L'Université canadienne
Canada's university

Si Σ est un alphabet de symboles de base, alors une *définition régulière* est une séquence de définitions de forme:

$d_1 \rightarrow r_1$

$d_2 \rightarrow r_2$

...

$d_n \rightarrow r_n$

Où d_i est un nom distinct, et chaque r_i est une expression régulière à travers les symboles dans $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$,

- i.e., les symboles de base et les noms pré-définis.

EXEMPLE DE DÉFINITIONS RÉGULIÈRES

uOttawa
L'Université canadienne
Canada's university

$$\textit{letter} \rightarrow (a \mid b \mid c \mid \ldots \mid z \mid A \mid B \mid C \mid \ldots \mid Z)$$

$$\textit{digit} \rightarrow (0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)$$

$$\textit{id} \rightarrow \textit{letter} \; (\; \textit{letter} \mid \textit{digit} \;)^*$$

$$\textit{integer} \rightarrow (+ \mid - \mid \varepsilon) \; (0 \mid (1 \mid 2 \mid 3 \mid \ldots \mid 9) \; \textit{digit}^*)$$

$$\textit{decimal} \rightarrow \textit{integer} \; . \; (\; \textit{digit} \;)^*$$

$$\textit{real} \rightarrow (\; \textit{integer} \mid \textit{decimal} \;) \; \mathtt{E} \; (+ \mid -) \; \textit{digit}^*$$

NOTATION ABRÉGÉE

Certaines motifs sont très fréquemment exprimés par des expressions régulières qu'il est convenable d'introduire des notations abrégées pour eux

Nous avons déjà vu quelques-unes de ces notations:

1. **Un ou plusieurs instances:** a+ dénote l'ensemble de toutes les chaînes de un ou plusieurs a's
2. **Zéro ou plusieurs instances:** a* dénote toutes les chaînes de zéro ou plusieurs a's
3. **Classes de caractères:** la notation [abc] où a, b et c dénotent l'expression régulière a | b | c
4. **Classes de caractères abréviés:** la notation [a-z] dénote l'expression régulière a | b | …. | z

NOTATION ABRÉGÉE

En utilisant les classes de caractères, on peut décrire les identificateurs comme étant des chaînes décrites par l'expression régulière suivante:

[A-Za-z][A-Za-z0-9]*

AUTOMATE FINIE

uOttawa
L'Université canadienne
Canada's university

Maintenant que nous avons appris les expressions régulières

- Comment peut-on savoir si une chaîne (ou lexème) suit un motif d'expression régulière ou non?

Nous allons de nouveau utiliser les machines d'état!

- Cette fois, elles ne sont pas des machines d'état UML ou réseaux de Petri
- Nous les appelons: Automate Finie

Le programme qui exécute cette machine d'état est appelé un *reconnaisseur (recognizer)*

AUTOMATE FINI

Un *reconnaisseur* pour un langage est un programme qui prend comme entrée une chaîne *x* et répond

- "Oui" si *x* est un lexème du langage
- "Non" si au contraire

Nous compilons une expression régulière dans un *reconnaisseur* en construisant un diagramme appelé un *automate fini*

Un automate fini peut être *déterministe* ou *non-déterministe*

- Non-déterministe veut dire que plus d'une transition d'un état est possible sur le même symbole d'entrée

AUTOMATE FINI NON-DÉTERMINISTE (AFN)

uOttawa
L'Université canadienne
Canada's university

Un ensemble d'états *S*

Un ensemble de symboles d'entrée qui appartiennent à l'alphabet Σ

Un ensemble de transitions qui sont déclenchées par le traitement d'un caractère

Un état simple s_0 qui est reconnu comme l'état de départ *(initial)*

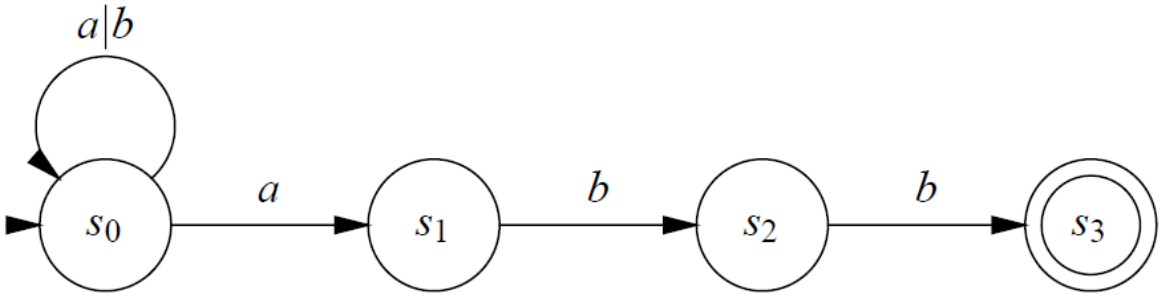
Un ensemble d'états *F* qui sont reconnus comme les états d'acceptation *(final)*.

EXEMPLE D'UN AFN

L'expression régulière suivante

(a|b)*abb

Peut être décrite en utilisant un AFN avec le diagramme suivant:



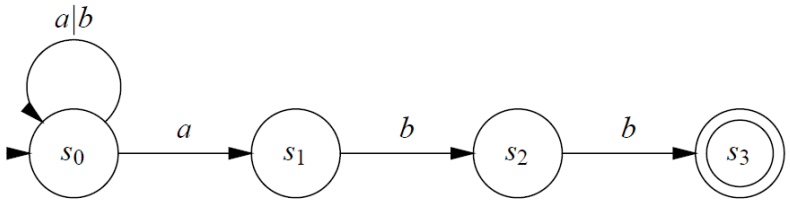
EXEMPLE D'UN AFN

Le diagramme précédent peut être décrit en utilisant le tableau suivant aussi

Souvenez-vous que l'expression régulière était:

(a|b)*abb

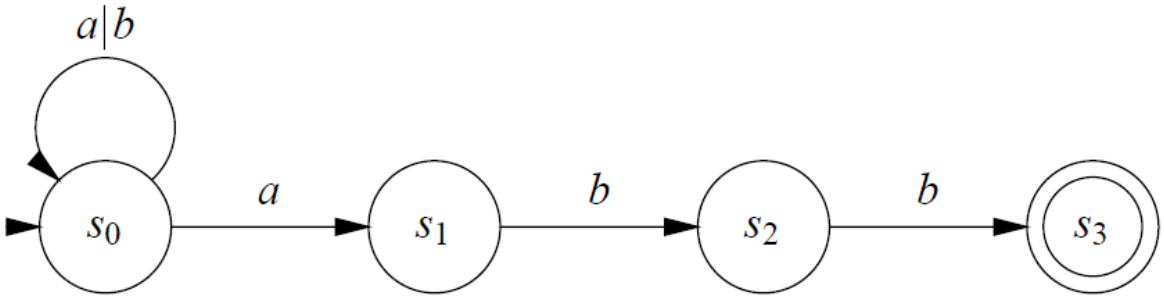
	a	b
s_0	$\{s_0, s_1\}$	$\{s_0\}$
s_1	–	$\{s_2\}$
s_2	–	$\{s_3\}$



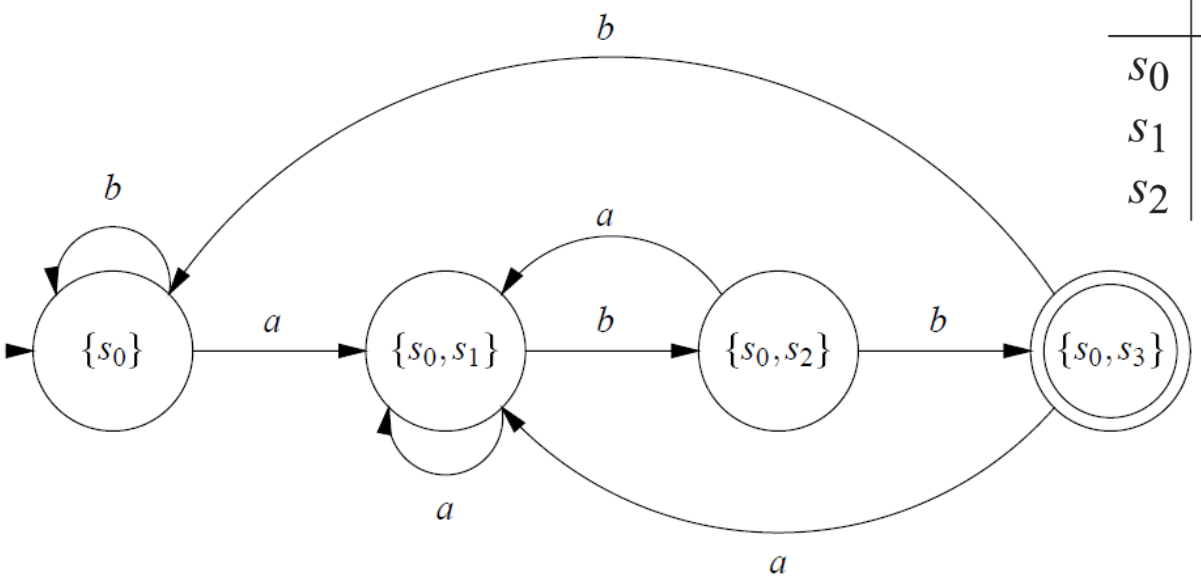
AFN VS AFD

Toujours avec l'expression régulière: (a|b)*abb

AFN:



AFD:



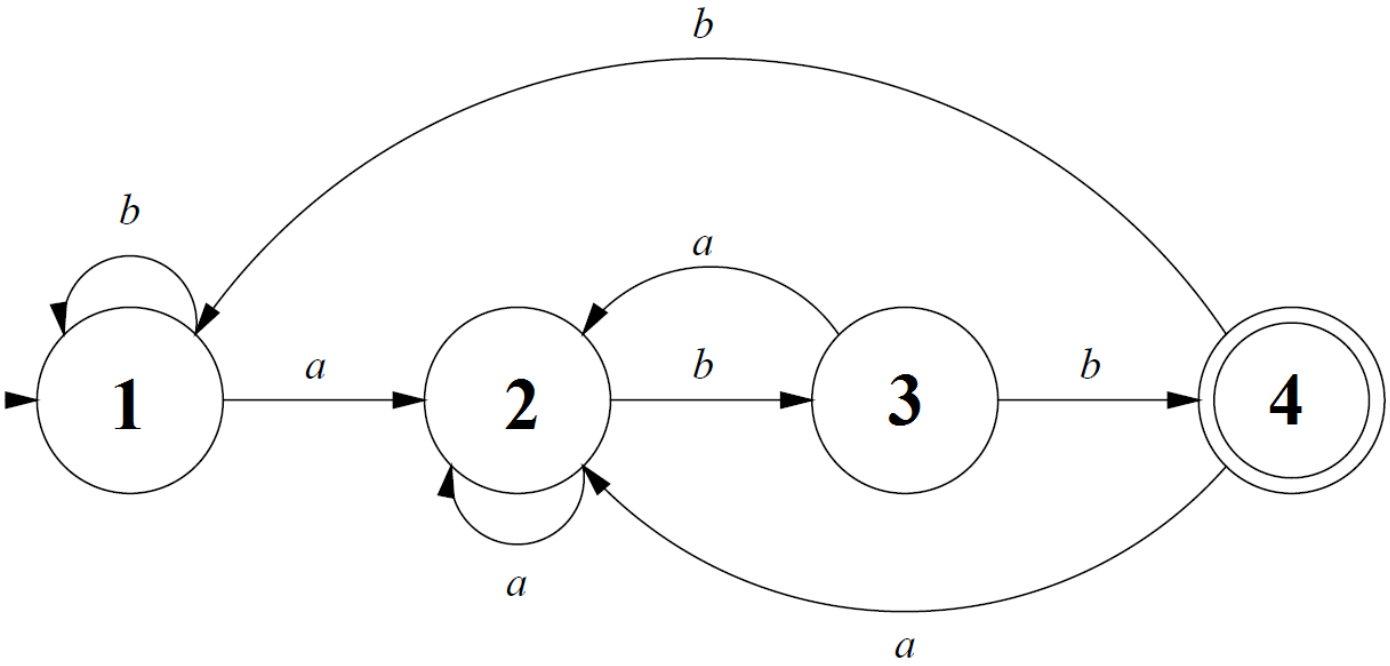
	a	b
s_0	$\{s_0, s_1\}$	$\{s_0\}$
s_1	–	$\{s_2\}$
s_2	–	$\{s_3\}$

UN AUTRE EXEMPLE DE AFD

uOttawa
L'Université canadienne
Canada's university

Pour la même expression régulière vu précédemment

(a|b)*abb

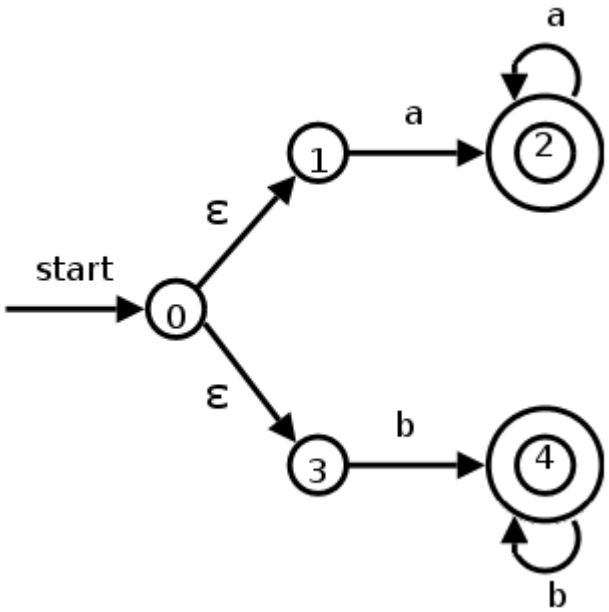


UN AUTRE EXEMPLE DE AFN

uOttawa
L'Université canadienne
Canada's university

Un AFN qui accepte l'expression régulière suivante:

aa*|bb*



AUTOMATE FINI DÉTERMINISTE (AFD)

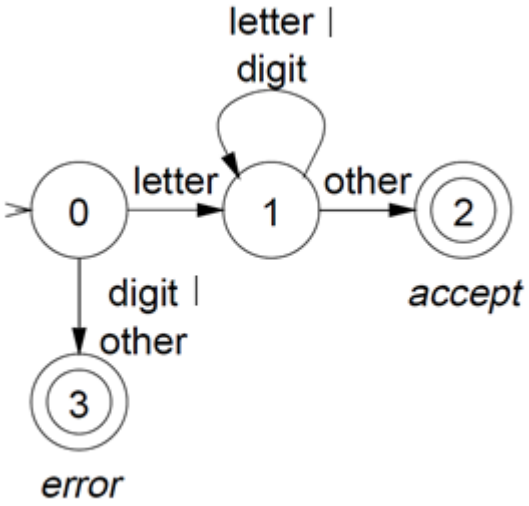
uOttawa
L'Université canadienne
Canada's university

Un AFD est un cas spécial de AFN dans lequel

- Aucun état a une transition ε
- Pour chaque état *s* et symbole d'entrée *a*, il existe au plus une transition marqué *a* qui quitte *s*

EXEMPLE DE AFD

Reconnaisseur pour un identificateur:



identifier

$$letter \rightarrow (a \mid b \mid c \mid \ldots \mid z \mid A \mid B \mid C \mid \ldots \mid Z)$$

$$digit \rightarrow (0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)$$

$$id \rightarrow letter\ (\ letter \mid digit\ )^*$$

TABLEAUX POUR LE RECONNAISSEUR

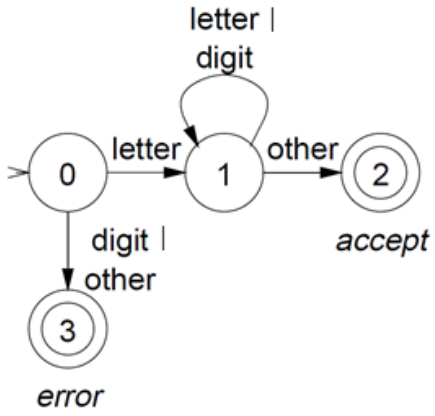
uOttawa
L'Université canadienne
Canada's university

`char_class:`

	$a-z$	$A-Z$	$0-9$	other
value	letter	letter	digit	other

`next_state:`

class	0	1	2	3
letter	1	1	—	—
digit	3	1	—	—
other	3	2	—	—



Pour changer l'expression régulière, nous pouvons simplement changer de tableaux…

CODE POUR LE RECONNAISSEUR

uOttawa
L'Université canadienne
Canada's university

Algorithme du reconnaisseur:

```
char ← next_char();
state ← 0;          /* code for state 0 */
done ← false;
token_value ← ""    /* empty string */
while( not done ) {
   class ← char_class[char];
   state ← next_state[class,state];
   switch(state) {
      case 1:       /* building an id */
         token_value ← token_value + char;
         char ← next_char();
         break;
      case 2:       /* accept state */
         token_type = identifier;
         done = true;
         break;
      case 3:       /* error */
         token_type = error;
         done = true;
         break;
   }
}
return token_type;
```

Référence:

char_class:

	$a-z$	$A-Z$	$0-9$	other
value	letter	letter	digit	other

next_state:

class	0	1	2	3
letter	1	1	—	—
digit	3	1	—	—
other	3	2	—	—

Exemple:
Pour la séquence de caractères: ***d8;***

char←d
state←0
done←false
token_value←""

CODE POUR LE RECONNAISSEUR

uOttawa
L'Université canadienne
Canada's university

Algorithme du reconnaisseur:

```
char ← next_char();
state ← 0;          /* code for state 0 */
done ← false;
token_value ← ""    /* empty string */
while( not done ) {
   class ← char_class[char];
   state ← next_state[class,state];
   switch(state) {
      case 1:       /* building an id */
         token_value ← token_value + char;
         char ← next_char();
         break;
      case 2:       /* accept state */
         token_type = identifier;
         done = true;
         break;
      case 3:       /* error */
         token_type = error;
         done = true;
         break;
   }
}
return token_type;
```

Référence:

char_class:

	$a-z$	$A-Z$	$0-9$	other
value	letter	letter	digit	other

next_state:

class	0	1	2	3
letter	1	1	—	—
digit	3	1	—	—
other	3	2	—	—

Exemple:
Pour la séquence de caractères: ***d8;***

char←d
state←0
done←false
token_value←""
class←letter

CODE POUR LE RECONNAISSEUR

Algorithme du reconnaisseur:

```
char ← next_char();
state ← 0;          /* code for state 0 */
done ← false;
token_value ← ""    /* empty string */
while( not done ) {
   class ← char_class[char];
   state ← next_state[class,state];
   switch(state) {
      case 1:       /* building an id */
         token_value ← token_value + char;
         char ← next_char();
         break;
      case 2:       /* accept state */
         token_type = identifier;
         done = true;
         break;
      case 3:       /* error */
         token_type = error;
         done = true;
         break;
   }
}
return token_type;
```

Référence:

char_class:

	$a-z$	$A-Z$	$0-9$	other
value	letter	letter	digit	other

next_state:

class	0	1	2	3
letter	1	1	—	—
digit	3	1	—	—
other	3	2	—	—

Exemple:
Pour la séquence de caractères: ***d8;***

char←d
state←~~0~~ 1
done←false
token_value←""
class←letter

CODE POUR LE RECONNAISSEUR

Algorithme du reconnaisseur:

```
char ← next_char();
state ← 0;          /* code for state 0 */
done ← false;
token_value ← ""    /* empty string */
while( not done ) {
   class ← char_class[char];
   state ← next_state[class,state];
   switch(state) {
      case 1:       /* building an id */
         token_value ← token_value + char;
         char ← next_char();
         break;
      case 2:       /* accept state */
         token_type = identifier;
         done = true;
         break;
      case 3:       /* error */
         token_type = error;
         done = true;
         break;
   }
}
return token_type;
```

Référence:

char_class:

	$a-z$	$A-Z$	$0-9$	other
value	letter	letter	digit	other

next_state:

class	0	1	2	3
letter	1	1	—	—
digit	3	1	—	—
other	3	2	—	—

Exemple:
Pour la séquence de caractères: ***d8;***

char←d
state←~~0~~ 1
done←false
token_value←"d"
class←letter

CODE POUR LE RECONNAISSEUR

Algorithme du reconnaisseur:

```
char ← next_char();
state ← 0;          /* code for state 0 */
done ← false;
token_value ← ""    /* empty string */
while( not done ) {
   class ← char_class[char];
   state ← next_state[class,state];
   switch(state) {
      case 1:       /* building an id */
         token_value ← token_value + char;
         char ← next_char();
         break;
      case 2:       /* accept state */
         token_type = identifier;
         done = true;
         break;
      case 3:       /* error */
         token_type = error;
         done = true;
         break;
   }
}
return token_type;
```

Référence:

char_class:

	$a-z$	$A-Z$	$0-9$	other
value	letter	letter	digit	other

next_state:

class	0	1	2	3
letter	1	1	—	—
digit	3	1	—	—
other	3	2	—	—

Exemple:
Pour la séquence de caractères: ***d8;***

char←~~d~~ 8
state←~~0~~ 1
done←false
token_value←"d"
class←letter

CODE POUR LE RECONNAISSEUR

Algorithme du reconnaisseur:

```
char ← next_char();
state ← 0;          /* code for state 0 */
done ← false;
token_value ← ""    /* empty string */
while( not done ) {
   class ← char_class[char];
   state ← next_state[class,state];
   switch(state) {
      case 1:       /* building an id */
         token_value ← token_value + char;
         char ← next_char();
         break;
      case 2:       /* accept state */
         token_type = identifier;
         done = true;
         break;
      case 3:       /* error */
         token_type = error;
         done = true;
         break;
   }
}
return token_type;
```

Référence:

char_class:

	$a-z$	$A-Z$	$0-9$	other
value	letter	letter	digit	other

next_state:

class	0	1	2	3
letter	1	1	—	—
digit	3	1	—	—
other	3	2	—	—

Exemple:
Pour la séquence de caractères: ***d8;***

char←~~d~~ 8
state←~~0~~ 1
done←false
token_value←"d"
class←~~letter~~ digit

CODE POUR LE RECONNAISSEUR

Algorithme du reconnaisseur:

```
char ← next_char();
state ← 0;          /* code for state 0 */
done ← false;
token_value ← ""    /* empty string */
while( not done ) {
   class ← char_class[char];
   state ← next_state[class,state];
   switch(state) {
      case 1:       /* building an id */
         token_value ← token_value + char;
         char ← next_char();
         break;
      case 2:       /* accept state */
         token_type = identifier;
         done = true;
         break;
      case 3:       /* error */
         token_type = error;
         done = true;
         break;
   }
}
return token_type;
```

Référence:

char_class:

	$a-z$	$A-Z$	$0-9$	other
value	letter	letter	digit	other

next_state:

class	0	1	2	3
letter	1	1	—	—
digit	3	1	—	—
other	3	2	—	—

Exemple:
Pour la séquence de caractères: ***d8;***

char←~~d~~ 8
state←~~0~~ ~~1~~ 1
done←false
token_value←"d"
class←~~letter~~ digit

CODE POUR LE RECONNAISSEUR

Algorithme du reconnaisseur:

```
char ← next_char();
state ← 0;          /* code for state 0 */
done ← false;
token_value ← ""    /* empty string */
while( not done ) {
   class ← char_class[char];
   state ← next_state[class,state];
   switch(state) {
      case 1:       /* building an id */
         token_value ← token_value + char;
         char ← next_char();
         break;
      case 2:       /* accept state */
         token_type = identifier;
         done = true;
         break;
      case 3:       /* error */
         token_type = error;
         done = true;
         break;
   }
}
return token_type;
```

Référence:

char_class:

	$a-z$	$A-Z$	$0-9$	other
value	letter	letter	digit	other

next_state:

class	0	1	2	3
letter	1	1	—	—
digit	3	1	—	—
other	3	2	—	—

Exemple:
Pour la séquence de caractères: ***d8;***

char←~~d~~ 8
state←~~0~~ ~~1~~ 1
done←false
token_value←"d8"
class←~~letter~~ digit

CODE POUR LE RECONNAISSEUR

Algorithme du reconnaisseur:

```
char ← next_char();
state ← 0;          /* code for state 0 */
done ← false;
token_value ← ""    /* empty string */
while( not done ) {
   class ← char_class[char];
   state ← next_state[class,state];
   switch(state) {
      case 1:       /* building an id */
         token_value ← token_value + char;
         char ← next_char();
         break;
      case 2:       /* accept state */
         token_type = identifier;
         done = true;
         break;
      case 3:       /* error */
         token_type = error;
         done = true;
         break;
   }
}
return token_type;
```

Référence:

char_class:

	$a-z$	$A-Z$	$0-9$	other
value	letter	letter	digit	other

next_state:

class	0	1	2	3
letter	1	1	—	—
digit	3	1	—	—
other	3	2	—	—

Exemple:
Pour la séquence de caractères: ***d8;***

char←~~d~~ ~~8~~ ;
state←~~0~~ ~~1~~ 1
done←false
token_value←"d8"
class←~~letter~~ digit

CODE POUR LE RECONNAISSEUR

Algorithme du reconnaisseur:

```
char ← next_char();
state ← 0;          /* code for state 0 */
done ← false;
token_value ← ""    /* empty string */
while( not done ) {
   class ← char_class[char];
   state ← next_state[class,state];
   switch(state) {
      case 1:       /* building an id */
         token_value ← token_value + char;
         char ← next_char();
         break;
      case 2:       /* accept state */
         token_type = identifier;
         done = true;
         break;
      case 3:       /* error */
         token_type = error;
         done = true;
         break;
   }
}
return token_type;
```

Référence:

char_class:

	$a-z$	$A-Z$	$0-9$	other
value	letter	letter	digit	other

next_state:

class	0	1	2	3
letter	1	1	—	—
digit	3	1	—	—
other	3	2	—	—

Exemple:
Pour la séquence de caractères: ***d8;***

char←~~d~~ ~~8~~ ;
state←~~0~~ ~~1~~ 1
done←false
token_value←"d8"
class←~~letter digit~~ other

CODE POUR LE RECONNAISSEUR

Algorithme du reconnaisseur:

```
char ← next_char();
state ← 0;          /* code for state 0 */
done ← false;
token_value ← ""    /* empty string */
while( not done ) {
   class ← char_class[char];
   state ← next_state[class,state];
   switch(state) {
      case 1:       /* building an id */
         token_value ← token_value + char;
         char ← next_char();
         break;
      case 2:       /* accept state */
         token_type = identifier;
         done = true;
         break;
      case 3:       /* error */
         token_type = error;
         done = true;
         break;
   }
}
return token_type;
```

Référence:

char_class:

	$a-z$	$A-Z$	$0-9$	other
value	letter	letter	digit	other

next_state:

class	0	1	2	3
letter	1	1	—	—
digit	3	1	—	—
other	3	2	—	—

Exemple:
Pour la séquence de caractères: ***d8;***

char←~~d~~ ~~8~~ ;
state←~~0~~ ~~1~~ ~~1~~ 2
done←false
token_value←"d8"
class←~~letter digit~~ other

CODE POUR LE RECONNAISSEUR

uOttawa
L'Université canadienne
Canada's university

Algorithme du reconnaisseur:

```
char ← next_char();
state ← 0;          /* code for state 0 */
done ← false;
token_value ← ""    /* empty string */
while( not done ) {
   class ← char_class[char];
   state ← next_state[class,state];
   switch(state) {
      case 1:       /* building an id */
         token_value ← token_value + char;
         char ← next_char();
         break;
      case 2:       /* accept state */
         token_type = identifier;
         done = true;
         break;
      case 3:       /* error */
         token_type = error;
         done = true;
         break;
   }
}
return token_type;
```

Référence:

char_class:

	$a-z$	$A-Z$	$0-9$	other
value	letter	letter	digit	other

next_state:

class	0	1	2	3
letter	1	1	—	—
digit	3	1	—	—
other	3	2	—	—

Exemple:
Pour la séquence de caractères: ***d8;***

char←~~d~~ ~~8~~ ;
state←~~0~~ ~~1~~ ~~1~~ 2
done←false
token_value←"d8"
class←~~letter digit~~ other

token_type←identifier

CODE POUR LE RECONNAISSEUR

Algorithme du reconnaisseur:

```
char ← next_char();
state ← 0;          /* code for state 0 */
done ← false;
token_value ← ""    /* empty string */
while( not done ) {
   class ← char_class[char];
   state ← next_state[class,state];
   switch(state) {
      case 1:       /* building an id */
         token_value ← token_value + char;
         char ← next_char();
         break;
      case 2:       /* accept state */
         token_type = identifier;
         done = true;
         break;
      case 3:       /* error */
         token_type = error;
         done = true;
         break;
   }
}
return token_type;
```

Référence:

char_class:

	$a-z$	$A-Z$	$0-9$	other
value	letter	letter	digit	other

next_state:

class	0	1	2	3
letter	1	1	—	—
digit	3	1	—	—
other	3	2	—	—

Exemple:
Pour la séquence de caractères: ***d8;***

char←~~d~~ ~~8~~ ;
state←~~0~~ ~~1~~ ~~1~~ 2
done←~~false~~ true
token_value←"d8"
class←other

token_type←identifier

MERCI!

QUESTIONS?