Traduction et Sémantique
Analyse lexicale

Luigi Santocanale
LIF, Université de Provence
Marseille, FRANCE

12 février 2010

Plan

Plan

Expressions regulières : syntaxe

Soit Σ un alphabet et Σ^* l'ensemble de mots sur Σ.

On définit $\mathscr{R}(\Sigma)$, l'ensemble des expressions régulières sur Σ, par induction :

- $\varepsilon \in \mathscr{R}(\Sigma)$,
- si $a \in \Sigma$, alors $a \in \mathscr{R}(\Sigma)$,
- si $r_1, r_2 \in \mathscr{R}(\Sigma)$ alors
  - $r_1 r_2 \in \mathscr{R}(\Sigma)$,
  - $r_1 | r_2 \in \mathscr{R}(\Sigma)$,
- si $r \in \mathscr{R}(\Sigma)$ alors $r^* \in \mathscr{R}(\Sigma)$.

Exemple : pour $\Sigma = \{a, b\}$,

$$(ab)^*abb\,,\ (ab)^*ba(ab)^*\,,\ (ab^*ba)^* \in \mathscr{R}(\Sigma)\,.$$

Expressions regulières : sémantique

Un langage sur Σ est un sous-ensemble de Σ^*.
Pour $r \in \mathscr{R}(\Sigma)$ on definit $L(r)$ (le langage de r) :

$$\begin{aligned}
L(\varepsilon) &= \{\varepsilon\}, \qquad \text{le singleton contenant le mot vide}\\
L(a) &= \{a\},\\
L(r_1 r_2) &= \{w_1 \cdot w_2 \mid w_1 \in \mathscr{R}(r_1) \text{ et } w_2 \in \mathscr{R}(r_2)\},\\
&\qquad \text{où } \cdot \text{ denote la concatenation de mots}\\
L(r_1|r_2) &= L(r_1) \cup L(r_2),\\
L(r^*) &= \bigcup_{n \geq 0} \underbrace{L(r) \bullet \ldots \bullet L(r)}_{n \text{ fois}},
\end{aligned}$$

c'est-à-dire :

$$w \in L(r^*) \quad \text{ssi} \quad \exists n \geq 0,\ \exists\, w_i \in L(r),\ i = 1, \ldots, n, \ \text{tels que} \quad w = w_1 \cdot \ldots \cdot w_n.$$

Des Exemples

$$\begin{aligned}
L(a^*) &= \{w \in \Sigma^* \mid \text{seulement des } a\text{, longueur arbitraire}\},\\
L((ab)^*) &= \{\text{les suites de } ab\},\\
L(b?(ab)^*a?) &= \{w \in \Sigma^* \mid \text{tout } a \text{ est suivi par } b \text{ et}\\
&\qquad \text{tout } b \text{ est suivi par } a\}.
\end{aligned}$$

Soit $r \in \mathscr{R}(\Sigma)$ et $w \in \Sigma^*$. On dit que :

- le modèle r reconnaît le mot w,
- le motif r filtre w,

si

$$w \in L(r).$$

Des raccourcis

$$\begin{aligned}
r^+ &= r(r)^*\\
L(r^+) &= \{w \in \Sigma^* \mid \text{une ou plusieurs fois un mot filtrée par } r\}
\end{aligned}$$

$$\begin{aligned}
r? &= \varepsilon | r\\
L(r?) &= \{\varepsilon\} \cup L(r).
\end{aligned}$$

En Unix :
si $\Sigma = \{a_1, \ldots, a_n\}$ et $J = \{a_{j_1}, a_{j_2}, \ldots, a_{j_k}\} \subseteq \Sigma$:

$$\begin{aligned}
[\mathtt{a}_{\mathtt{i}_1}\mathtt{a}_{\mathtt{i}_2}\ldots\mathtt{a}_{\mathtt{j}_k}] &= \mathtt{a}_{\mathtt{j}_1}|\mathtt{a}_{\mathtt{j}_2}|\ldots|\mathtt{a}_{\mathtt{j}_k}\\
[\mathtt{a}_\mathtt{i} - \mathtt{a}_\mathtt{j}] &= \mathtt{a}_\mathtt{i}|\mathtt{a}_{\mathtt{i}+1}|\ldots|\mathtt{a}_\mathtt{j}.
\end{aligned}$$

Définitions régulières : un exemple

Les nombres non signées en Pascal :

$$\begin{aligned}
\mathtt{chiffre} &\to [0-9]\\
\mathtt{chiffres} &\to \mathtt{chiffre}^+\\
\mathtt{fraction_opt} &\to (.\mathtt{chiffres})?\\
\mathtt{exposant_opt} &\to (\mathtt{E}(+|-|\varepsilon)\mathtt{chiffres})?\\
\mathtt{nb} &\to \mathtt{chiffres}\ \mathtt{fraction_opt}\ \mathtt{exposant_opt}
\end{aligned}$$

On pose :

$$\begin{aligned}
r(\mathtt{chiffre}) &= [0-9]\\
r(\mathtt{chiffres}) &= [0-9]^+\\
r(\mathtt{fraction_opt}) &= (.[0-9]^+)?\\
r(\mathtt{exposant_opt}) &= (\mathtt{E}(+|-|\varepsilon)[0-9]^+)?\\
r(\mathtt{nb}) &= [0-9]^+(.[0-9]^+)?(\mathtt{E}(+|-|\varepsilon)[0-9]^+)?
\end{aligned}$$

Définitions régulières

De la forme

$$\begin{aligned} D_1 &\to r_1 \\ D_2 &\to r_2 \\ &\vdots \\ D_n &\to r_n \end{aligned}$$

où $r_i \in \mathscr{R}(\Sigma \cup \{D_1, \ldots, D_{i-1}\})$, pour $i = 1, \ldots, n$.

On reconstruit des expressions régulières $r(D_i) \in \mathscr{R}(\Sigma)$:

$$r(D_i) = r_i[r(D_1)/D_1, \ldots, r(D_{i-1})/D_{i-1}].$$

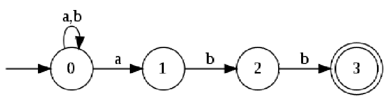
Les AFN

Un Automate Fini Non-déterministe (AFN) est un tuplet $\mathscr{A} = \langle Q, \Sigma, \Delta, i, F \rangle$ où

- Q est un ensemble fini d'états ,
- Σ est son alphabet,
- $\Delta : Q \times \Sigma \longrightarrow \mathscr{P}(Q)$ est la fonction de transition,
- $i \in Q$ est l'état initial,
- $F \subseteq Q$ sont les états finaux.

Exemple

Représentation graphique :



C'est à dire :

- $Q = \{0, 1, 2, 3\}$,
- $\Sigma = \{a, b\}$,
-

Δ	a	b
0	$\{0,1\}$	$\{0\}$
1	$\emptyset$	$\{2\}$
2	$\emptyset$	$\{3\}$
3	$\emptyset$	$\emptyset$

- $i = 0$, $F = \{3\}$.

Acceptation par AFN

On dit qu'un mot

$$w = a_1 a_2 \ldots a_n$$

est accepté par l'AFN $\mathscr{A}$ s'il existe $n \geq 0$ et $e_0, \ldots, e_n$ tels que

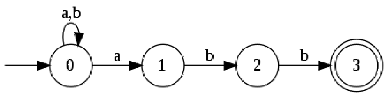
- $e_0 = i$ et $e_n \in F$,
- $e_i \in \Delta(e_{i-1}, a_i)$, pour $i = 1, \ldots, n$.

On pose :

$$L(\mathscr{A}) = \{ w \in \Sigma^* \mid w \text{ est accepté par } \mathscr{A} \}.$$

Exemple

Pour $\mathscr{A}$ l'automate



on a

$$L(\mathscr{A}) = \{\, w \in \{\, a, b \,\}^* \mid w \text{ se termine par } abb \,\}.$$

Le Théorème de Kleene

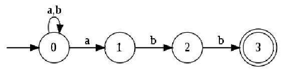
Théorème. Pour $L \subseteq \Sigma^*$, les faits suivants sont équivalents :

- il existe une expression régulière r tel que $L = L(r)$,
- il existe un AFN $\mathscr{A}$ tel que $L = L(\mathscr{G})$.

S. C. Kleene.
Representation of events in nerve nets and finite automata.
In *Automata studies*, Annals of mathematics studies, no. 34, pages 3–41. Princeton University Press, Princeton, N. J., 1956.

Par exemple,

$$L((a|b)^* abb) = L(\mathscr{A})$$

où $\mathscr{A}$ est l'automate  .

Le problème $w \in L(\mathscr{A})$ (I)

- le non déterminisme est un problème,
- ...qu'on peut se résoudre si on parcours tous les chemins de i en parallèle.

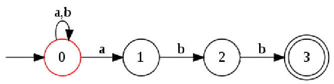
```
S_courant = { i }
tant que w ≠ ε faire
    a = tete(w) (a ∈ Σ),  S_next = ∅

    pout tout q ∈ S_courant faire
        S_next = S_next ∪ Δ(q, a)

    w = queue(w),  S_courant = S_next

si S_courant ∩ F ≠ ∅ accepter
sinon refuser
```

Complexité : $O(|w| \times |Q|)$.

Un calcul parallèle

...avec l'automate $\mathscr{A}$ et le mot *ababb*.

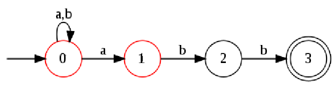
ababb



Un calcul parallèle

...avec l'automate $\mathscr{A}$ et le mot *ababb*.

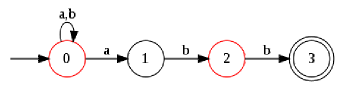
ababb



Un calcul parallèle

...avec l'automate $\mathscr{A}$ et le mot *ababb*.

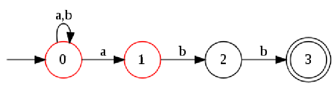
ababb



Un calcul parallèle

...avec l'automate $\mathscr{A}$ et le mot *ababb*.

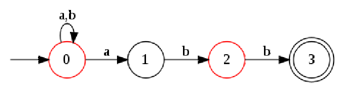
ababb



Un calcul parallèle

...avec l'automate $\mathscr{A}$ et le mot *ababb*.

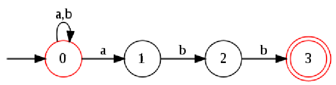
ababb



Un calcul parallèle

. . . avec l'automate $\mathscr{A}$ et le mot *ababb*.

ababb



Les AFD

- Un Automate Fini Déterministe (AFD) est un AFN $\langle Q, \Sigma, \Delta, i, F \rangle$ tel que $\Delta(q, a)$ est vide ou un singleton.
- La table d'un AFD contient au plus un seul élément dans chaque case.
- $S_{courant}$ est toujours un singleton (ou l'ensemble vide)
- La calcul de $w \in L(\mathscr{A})$ se fait en temps $O(|w|)$

Le Théorème de Rabin-Scott

Théorème. Pour $L \subseteq \Sigma^*$, les faits suivants sont équivalents :

- il existe un AFN $\mathscr{A}$ tel que $L = L(\mathscr{A})$.
- il existe un AFD $\mathscr{D}$ tel que $L = L(\mathscr{D})$.

M. O. Rabin and D. Scott.
Finite automata and their decision problems.
IBM J. Res. Develop., 3 :114–125, 1959.

Idée de la preuve (AFN -> AFD)

- Étant donné $\mathscr{A}$ et pour tout mot w,
  on se pose la question si $w \in (\mathscr{A})$
  Comme avant, on fait ces calculs par en *parallèle*.
- Cela revient à construire/explorer l'automate des sous-ensembles

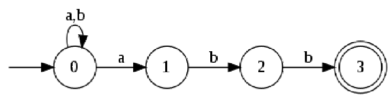
$$\mathscr{P}(\mathscr{A}) = \langle Q', \Sigma, \Delta', i', F' \rangle$$

où :

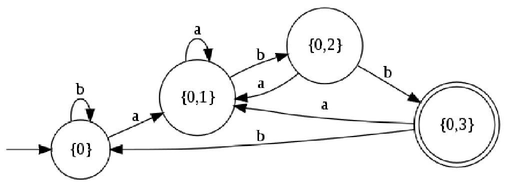
- $Q' = \mathscr{P}(Q)$,
- $\Delta'(S, a) = \bigcup_{s \in S} \Delta(s, a)$,
- $i' = \{i\}$ et $S \in F'$ ssi $S \cap F \neq \emptyset$.

Notre exemple

Pour l'automate $\mathscr{A}$



on obtient :



Un algorithme pour construire $\mathscr{P}(\mathscr{A})$

$a\_explorer = \{\{i\}\}$
$visites = \emptyset$

`tant que` $a\_explorer \neq \emptyset$
  `choisir` $S_{courant} \in a\_explorer$

  `pour tout` $a \in \Sigma$ `faire`
    $S_{next} = \emptyset$
    `pour tout` $s \in S_{courant}$
      $S_{next} = S_{next} \cup \Delta(q,a)$
    `poser` $\Delta'(S_{courant}, a) = S_{next}$

    `si` $S_{next} \notin a\_explorer \cup visites$
      `ajouter` S_{next} `à` $a\_explorer$

    `enlever` $S_{courant}$ `de` $a\_explorer$
    `ajouter` $S_{courant}$ `à` $visites$

Caveats

- L'automate $\mathscr{P}(\mathscr{A})$ peut avoir $2^{|Q|}$ états.
- Pour le calculer on peut avoir besoin de $2^{|Q|}$ étapes de calculs.
- On peut réduire (minimiser) le nombre d'états d'un AFD.
- Le langage

$$(a|b)^* a \underbrace{(a|b) \ldots (a|b)}_{n \text{fois}}$$

est le langage d'un AFN de taille $O(n)$.
Tout AFD qui reconnaît ce langage a taille au moins 2^n états.

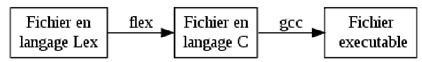
Le problème $w \in L(\mathscr{A})$ (II)

- Calculer si $w \in L(\mathscr{A})$ en parallèle revient à calculer avec $\mathscr{P}(\mathscr{A})$.
- Cet algorithme pour décider si $w \in L(\mathscr{A})$ revient à construire $\mathscr{P}(\mathscr{A})$ à la volée.
- On peut l'améliorer en mettant les calculs dans une cache. On parle alors de construction *paresseuse* de $\mathscr{P}(\mathscr{A})$.
- Une solution radicalement différente est de construire d'abord tout $\mathscr{P}(\mathscr{A})$ et vérifier ensuite si $w \in L(\mathscr{P}(\mathscr{A}))$. Cela pose des problèmes d'espace, mais est très vite.

Plan

L'outil Lex

C'est un compilateur d'analyseurs lexicaux :



Écrit par Lesk en 1975.
Utilisé par nombreux compilateurs sous Unix.

Flex (fast lexical analyzer generator) :
écrit par Vern Paxson en 1987.
Ce n'est part du projet GNU.

Le langage Lex

Trois sections (séparées par %%) :

1. Définitions régulières (options, code C en préambule, ...)
2. Suite de règles de la forme

$$\begin{array}{ll} & \vdots \\ m_i & \{ \ a_i \ \} \\ & \vdots \end{array}$$

   où m_i est un motif et a_i est l'action associée à ce motif.
   - Chaque m_i est une expression régulière sur l'alphabet ASCII + définitions,
   - Chaque a_i est un morceau de code C (à déclencher si le motif m_i est reconnu).
3. Code C : cette section définit des fonctions, à inclure dans le code C, dont on peut avoir besoin (`main`, `yywrap`, ...)

La loi du plus long lexème (leftmost longmost POSIX)

Soit w un prefixe de u :

$$u = w \cdot w' \, .$$

Suppossons que

$$w \in L(m_i) \, , \qquad\qquad u \in L(m_j) \, .$$

Filtrer w ou u ?

On filtre la chaîne de caractères la plus longue,
on applique la règle lui associée :

$$\mathtt{m}_j \qquad \{ \ \mathtt{a}_j \ \}$$

Un exemple

Considérons les deux règles

```
if|let                  {printf("Mot clé\n");}
[a-zA-z]([a-zA-z0-9])*  {printf("Identificateur\n");}
```

Si le flot d'entrée est de la forme

```
let ifin = 4 in ...
```

La chaîne `ifin` est reconnue comme un identificateur,
(et non pas comme mot clé).
"Identificateur" est affiché à l'écran.

La loi de priorité des réglès

Considérons des règles

$$\begin{array}{ll} \mathtt{m}_i & \{\ \mathtt{a}_i\ \} \\ \vdots & \\ \mathtt{m}_j & \{\ \mathtt{a}_j\ \} \end{array}$$

et supposons que

$$w \in L(m_i) \cap L(m_j).$$

Faut il déclencher l'action a_i où a_j ?

On déclenche l'action plus prioritaire, c'est-à-dire a_i.

Exemple

Considérons les deux règles

```
if|let                  {printf("Mot clé\n");}
[a-zA-z]([a-zA-z0-9])*  {printf("Identificateur\n");}
```

Si le flot d'entrée est de la forme

```
if in = 4 then ...
```

La chaîne `if` est reconnue comme comme mot clé
(et non pas comme identificateur).
"Mot clé" est affiché à l'écran.

Remarque

La loi du plus long lexème est plus forte que
la loi des priorités entre règles.

Un erreur qui se produit si on oublie ce fait :

```
if|then|else {printf("Mot clé"\n);}
     .
     .
     .
.+           {printf("Un erreur trop long\n");}
```

La chaîne

```
if x := 3 then
4 else 4
```

est divisé ainsi :

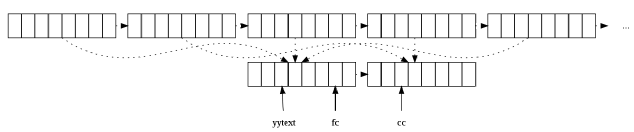
$$\underbrace{\text{if x := 3 then}}_{\text{erreur}}\ \underbrace{\text{4 else 4}}_{\text{erreur}} \ldots$$

Plan

Le problème des I/O

- Le flot de caractères en entrée est
  – le plus souvent – un fichier.
- Les transferts du disque à la mémoire vive sont coûteux.
- Transférer 100 fois un caractères est beaucoup plus coûteux que transférer 1 fois 100 caractères.

Le double tampon circulaire



- Circularité : les deplacements se font modulo $2 * \texttt{TAILLETAMPON}$
- Trois pointeurs :
  - `yytext` : debut du lexème,
  - `fc` : fin courante du lexème,
  - `cc` : caractere courant.
- Quand `cc` s'approche à la fin d'un tampon,
  un nouveau tampon est transféré du flot vers la mémoire.
- Limite de la taille des lexèmes :
  - si $cc - yytext > 0$ alors $cc - yytext < \texttt{TAILLETAMPON}$,
  - si $cc - yytext < 0$ alors $yytext - cc > \texttt{TAILLETAMPON}$.

Des expressions régulières aux automates

Soit

```
m1 { a1 }
   .
   .
   .
mn { an }
```

un ensemble de reglès.

1. On construit un AFD $\mathscr{D} = \langle Q, i, \Delta, F \rangle$ tel que
$$L(\mathscr{D}) = L(m_1 | \ldots | m_n).$$
2. On construit aussi une fonction
$$\lambda : F \longrightarrow \{1, \ldots, n\}$$
telle que si $w = a_1, \ldots, a_k$ et
$$i \xrightarrow{a_1} q_1 \ldots q_{k-1} \xrightarrow{a_k} q_k \in F$$
et $\lambda(q_k) = i$, alors $w \in L(m_i)$ et $w \notin L(m_j)$ pour $j < i$.

```
repeter

   etat_courant = i
   yylex = cc = fc
   priorite = 1
```



```
   repeter
      etat_courant = Δ(etat_courant,*cc)

      si etat_courant=erreur
         si yylex < fc
            excuter l'action a_priorite
         sinon
            /* yylex = fc, on a pas progressé */
            excuter l'action par defaut
            fc++
         sortir de la boucle
```



```
      sinon
         si etat_courant ∈ F
            fc=cc
            priorite = λ(etat_courant)

      cc++
```