Langages
Analyse lexicale
Expressions régulières
Automates

Didier Rémy

2000 - 2001

http://cristal.inria.fr/~remy/poly/compil/7/
http://w3.edu.polytechnique.fr/profs/informatique/Didier.Remy/compil/7/

En amont de la chaîne de compilation

$$\mathrm{SC}\ \boxed{\begin{array}{c}\text{Analyses}\\ \xrightarrow{\text{Lex.}}\ \mathrm{SL}\ \xrightarrow{\text{Gram.}}\end{array}}\ \mathrm{SA}\ \overbrace{\xrightarrow[\text{}]{\substack{\text{Génér.}\\ \text{Canon.}}}}^{(1)}\ \mathrm{CI}\ \overbrace{\xrightarrow{\substack{\text{Sélect.}\\ \text{instr.}}}\ \mathrm{CA}\ \xrightarrow{\substack{\text{Alloc.}\\ \text{regist.}}}}^{(2)}\ \mathrm{CA}$$

Analyse en deux passes :

1. Analyse lexicale : transforme une suite de caractères en une suite de lexèmes (mots).
2. Analyse grammaticale : transforme une suite de lexèmes en une représentation arborescente (syntaxe abstraite).

Enjeux

Les analyses lexicales et grammaticales ont un domaine d'application bien plus large que celui de la compilation. On les retrouve comme première passe dans de nombreuses applications (analyses des commandes, des requêtes, etc,).

Ces deux analyses utilisent de façon essentielle les automates, mais on retrouve aussi les automates dans de nombreux domaines de l'informatique.

Les expressions régulières sont un langage de description d'automates ; elles sont utilisées dans de nombreux outils Unix, et fournies en bibliothèque dans la plupart des langages de programmation.

Note

L'étude détaillée des automates et des grammaires formelles pourrait constituer un cours à part entière.

Nous nous contentons ici de la présentation formelle minimale, avec comme but :
- d'expliquer le fonctionnement des analyseurs de façon à pouvoir écrire soi-même des analyseurs lexicaux ou grammaticaux.
- de se familiariser aussi avec les expressions régulières et les automates, car on les retrouve ensuite fréquemment en informatique.

Le but du cours n'est pas d'écrire le moteur d'un analyseur, ni de répertorier toutes les techniques d'analyses.

Les langages formels

On se donne un ensemble Σ appelé alphabet, dont les éléments sont appelés caractères.

Un mot (sur Σ) est une séquence de caractères (de Σ).

On note ϵ le mot vide, uv la concaténation des mots u et v (la concaténation est associative avec ϵ pour élément neutre).

Σ^* est l'ensemble des mots sur Σ

Σ^+ est l'ensemble des mots non vides.

Un langage sur Σ est un sous-ensemble L de Σ^*.

Si U et V sont des langages sur Σ, on note UV l'ensemble des mots obtenus par la concaténation d'un mot de U et d'un mot de V ; U^* (resp. U^+), l'ensemble des mots obtenus par la concaténation d'un nombre arbitraire, éventuellement nul (resp. non nul) de mots de U.

Exemples

1. Σ_1 est l'alphabet français et L_1 l'ensemble des mots du dictionnaire français avec toutes leurs déclinaisons.
2. Σ_2 est L_1 et L_2 est l'ensemble des phrases grammaticalement correctes de la langue française.

   Ou bien L'_2 le sous-ensemble des palindromes de L_2.
3. Σ_3 est l'ensemble des caractères ASCII, et L_3 est composé de tous les mots clés de pseudo pascal, de l'ensemble des symboles, de l'ensemble des identificateurs et de l'ensemble des entiers.
4. Σ_4 est L_3 et L_4 est l'ensemble des programmes pseudo pascal.
5. Σ est $\{a, b\}$ et L est l'ensemble $\{a^n b^n \mid n \in I\!N\}$ (sous ensemble des expressions bien parenthésées).

Expressions régulières

Les expressions régulières sont des opérateurs permettant de décrire certains langages (les langages réguliers) définis sur un même alphabet Σ.

On note a, b, *etc.* des lettres de Σ, M et N des expressions régulières, $[\![M]\!]$ le langage associé à M.

- Une lettre de l'alphabet a désigne le langage $\{a\}$.
- Epsilon : ϵ désigne le langage $\{\epsilon\}$.
- Concaténation : $M \cdot N$ désigne le langage $[\![M]\!][\![N]\!]$.
- Alternative : $M \mid N$ désigne le langage $[\![M]\!] \cup [\![N]\!]$.
- Répétition : M^* désigne le langage $[\![M]\!]^*$.

On ajoute également du sucre syntaxique (i.e. sans changer l'expressivité) :

- $[abc]$ pour $(a \mid b \mid c)$ et $[a_1{-}a_2]$ pour $\{c \in \Sigma, a_1 \leq c \wedge c \leq a_2\}$, (on suppose ici que l'alphabet est ordonné)
- $M?$ pour $M \mid \epsilon$ et M^+ pour MM^*.

Analyse lexicale

On veut couper une phrase (suite de caractères) en lexèmes.

On décrit les lexèmes par des expressions régulières. Exemple :

1. Les mots clés : `"let"`, `"in"`
2. Les variables : `['a'-'z']+ ['0'-'9']*`
3. Les entiers : `['0'-'9']+`
4. Les symboles : `'('`, `')'`, `'+'`, `'*'`, `'='`
5. Le lexème vide : `(' ' | '\n')`

On peut représenter les lexèmes par un type concret :

```
type token =
    LET | IN | ..                       (* mots clés *)
  | VAR of string                       (* variables *)
  | INT of int                          (* entiers *)
  | LPAREN | RPAREN | PLUS | TIMES | EQUAL (* symboles *)
```

Analyse lexicale (suite)

Principe C'est un algorithme qui
- prend une suite de caractère
- la transforme en une suite de lexème, et retourne les caractères non reconnus.

Problème Il y a des ambiguités. Par exemple :
- `let` pourrait être reconnu comme une variable.
- `lettre` pourrait aussi être reconnu comme la séquence `LET; VAR "tre"` ou encore `VAR "let"; VAR "tre"`

Rappel

mots clés	`"let","in"`
variables	`['a'-'z']+ ['0'-'9']*`
entiers	`['0'-'9']+`
symboles	`'(',')','+','*','='`
lexème vide	`(' ' | '\n')`

Règles de priorité

On choisit par priorité :

1. Le lexème le plus long,
2. L'ordre de définition.

Ainsi la phrase `let lettre = 3 in 1 + fin` produit la suite de lexèmes :

```
LET; VAR "lettre"; EQUAL; IN; INT 1; PLUS; VAR "fin"
```

ocamllex

On écrit un fichier `lexeur.mll` qui est compilé en un fichier source `lexeur.ml` par le programme `ocamllex`.

```
{ (* prelude : code Ocaml reproduit verbatim *) }
rule entree_1 = parse
  regexp_1  { (* code ocaml qui retourne un lexème *) }
| regexp_2  { (* code ocaml qui retourne un lexème *) }
| ...
and entree_2
...
{ (* postlude : code Ocaml reproduit verbatim *) }
```

Pour construire les lexèmes, on récupére la chaîne reconnue par `Lexing.lexeme lexbuf`. Attention : le nom de la variable `lexbuf` est fixé par convention.

Exemple

Fichier essai.mll

```
{ open Lexing
  type token = IDENT of string | INT of int |  LET | IN
      |  PLUS | TIMES | EQUAL | LPAREN | RPAREN
  exception Eof  exception Illégal   }
rule token = parse
   [' ' '\ t' '\ n']           { token lexbuf }   (* skip *)
|  "let"                       { LET }
|  "in"                        { IN }
|  ['a'-'z'] + ['0'-'9'] *     { IDENT (lexeme lexbuf)}
|  ['0'-'9']+                  { INT(int_of_string
                                        (lexeme lexbuf)) }
|  '=' { EQUAL } | '+' { PLUS } | '*' { TIMES }
|  eof { raise  Eof }  | _ { raise  Illégal   }
```

Exemple (suite)

Fichier essai.mll

```
{ let lex buf =
    let rec lex () =
      try let h = token buf in h :: lex () with Eof -> [] in
    lex ()
  let lex_string s = lex (Lexing.from_string s)
  let lex_chan c = lex (Lexing.from_channel c) }
```

Compilation en deux étapes :

1. Fabrication du fichier essai.ml

   ocamllex essai . mll

2. Compilation normale du fichier essai.ml

   ocamlc −c essai.ml

Remarques

Utilisation de la récursion `token lexbuf` pour une production vide.

L'utilisation de plusieurs règles revient à combiner plusieurs lexeurs indépendants mais qui travaillent sur la même entrée. Application à l'analyse des chaînes de caractères (inefficace) :

```
rule token = parse
    [' ' '\t' '\n']   { token lexbuf }
  | '"'             { STRING(String.concat "" (string lexbuf)) }
  | ['0'-'9']+       { INT(int_of_string (lexeme lexbuf)) }
  | eof              { raise EOF }
and string = parse
  | '"'              { [] }
  | '\\' '"'         { "\"" :: string lexbuf }
  | eof              { raise Unterminated_string }
  | _                { let c = lexeme_char lexbuf 0 in
                       String.make 1 c :: string lexbuf }
```

Récupération des erreurs

En cas d'erreur, il est nécessaire d'indiquer un minimum d'informations, telles que la position de l'erreur, et/ou la dernière séquence non reconnue :

- La position du dernier lexème dans le flux d'entrée est déterminée par `Lexing.lexeme_start lexbuf` et `Lexing.lexeme_end lexbuf`.
- La sous-chaîne correspondante est (`Lexing.lexeme lexbuf`).

Comment fonctionnne Ocamllex ?

Un exemple de compilation :

1. Chaque expression régulière est compilée en un automate,
2. L'ensemble des automates sont fusionnés en un seul,
3. L'automate résultant est determinisé.
4. L'automate est minimisé.

Automates finis déterministes

Un automate fini déterministe M est un quintuple $(Q, \Sigma, \delta, q_0, F)$ où

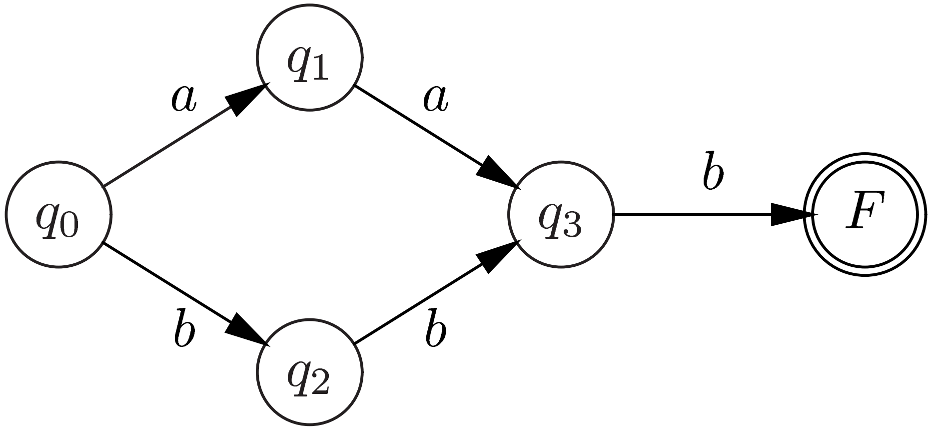
- Σ est un alphabet ;
- Q est un ensemble fini d'états ;
- $\delta : Q \times \Sigma \to Q$ est la fonction (partielle) de transition ;
- q_0 est l'état initial ;
- F est un ensemble d'états finaux.

On peut étendre δ sur $Q \times \Sigma^* \to Q$ par $\begin{cases} \delta(q, \epsilon) = q \\ \delta(q, aw) = \delta(\delta(q, a), w) \end{cases}$

Le langage $L(M)$ reconnu par l'automate M est l'ensemble $\{w \mid \delta(q_0, w) \in F\}$ des mots permettant d'atteindre un état final à partir de l'état initial.

Exemple

Automate déterministe :

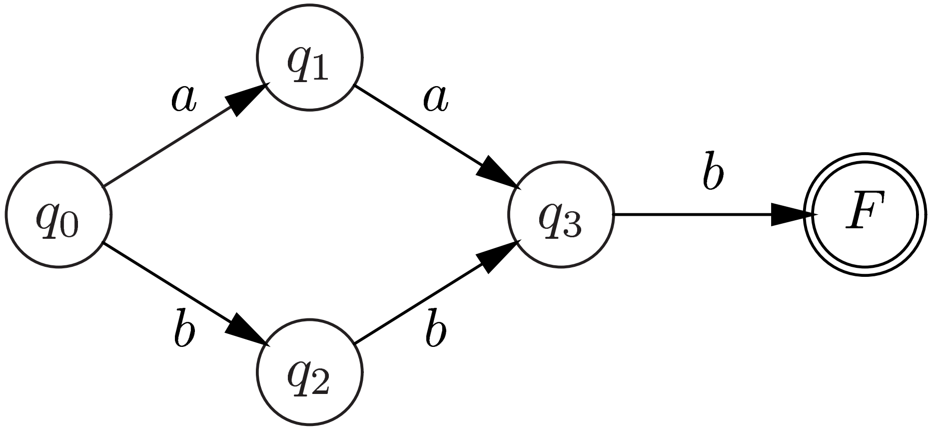


- $\Sigma = \{a, b\}$
- $Q = \{q_0, q_1, q_2, q_3, F\}$
- $\delta = \{(q_0, a) \mapsto q_1, (q_0, b) \mapsto q_2, (q_1, a) \mapsto q_3, (q_2, b) \mapsto q_3, (q_3, b) \mapsto F\}$
- État initial q_0 et un seul état final F.

Quel est le langage reconnu ?

Exemple

Automate déterministe :



- $\Sigma = \{a, b\}$
- $Q = \{q_0, q_1, q_2, q_3, F\}$
- $\delta = \{(q_0, a) \mapsto q_1, (q_0, b) \mapsto q_2, (q_1, a) \mapsto q_3, (q_2, b) \mapsto q_3, (q_3, b) \mapsto F\}$
- État initial q_0 et un seul état final F.

Quel est le langage reconnu ? $L = \{aab, bbb\}$

Automates finis non-déterministes

Il existe plusieurs transitions avec la même étiquette, mais vers des états différents. On autorise aussi des ϵ-transitions.

Formellement, comme ci-dessus, excepté $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to 2^Q$.

On étend δ sur $Q \times \Sigma^* \to Q$ par $\begin{cases} q \in \delta(q, \epsilon) & \text{(en plus)} \\ \delta(q, aw) = \bigcup_{q' \in \delta(q,a)} \in \delta(q', w) \end{cases}$

Le langage $L(M)$ reconnu par un automate non déterministe est $\{w \mid \delta(q_0, w) \cap F \neq \emptyset\}$

Automate d'une expression régulière

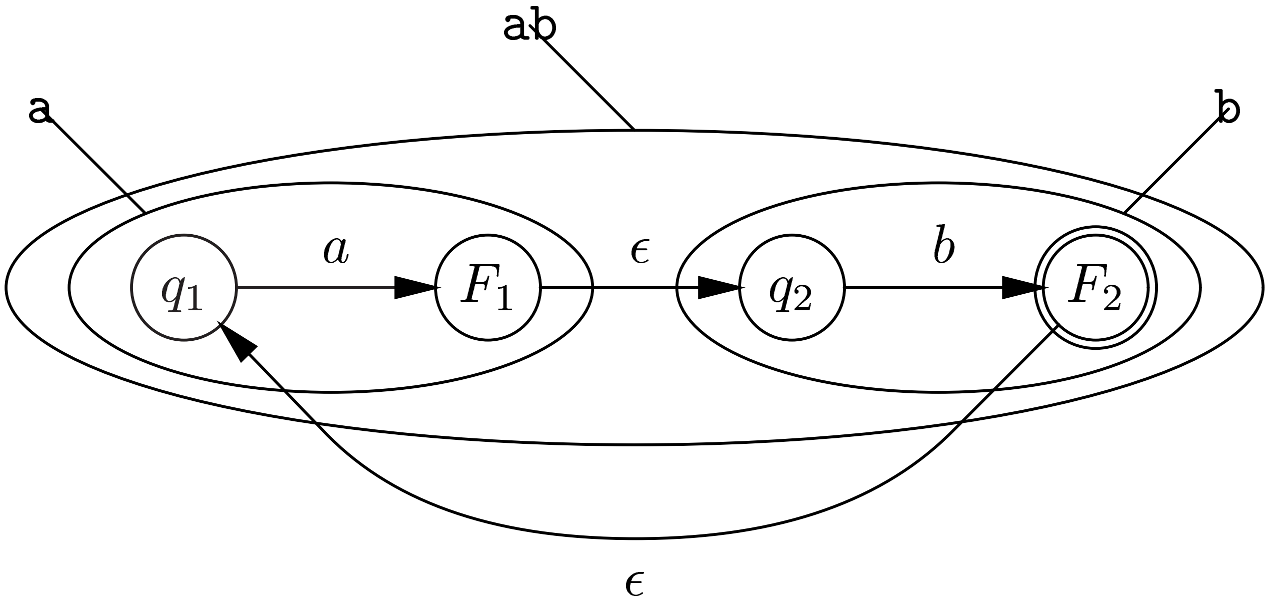
L'alphabet Σ est fixé.

On associe à une expression régulière M un automate non déterministe (Q, δ, s, F) défini récursivement par :

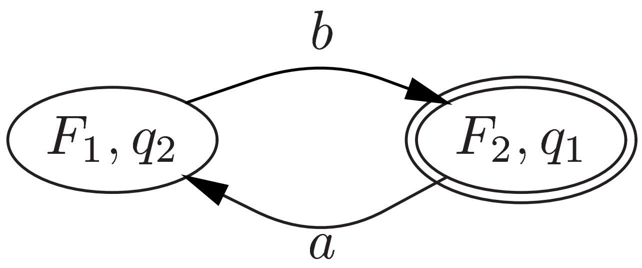
- $[\![a]\!] = (\{s, f\}, \{s \overset{a}{\mapsto} f\}, s, \{f\})$
- $[\![\epsilon]\!] = (\{s, f\}, \{s \overset{\epsilon}{\mapsto} f\}, s, \{f\})$
- $[\![M \mid M']\!] = (Q \cup Q' \cup \{s''\}, \delta \cup \delta' \cup \{s'' \overset{\epsilon}{\mapsto} s, s'' \overset{\epsilon}{\mapsto} s'\}, s'', F \cup F')$
- $[\![M \cdot M']\!] = (Q \cup Q', \delta \cup \delta' \cup \{f \overset{\epsilon}{\mapsto} s', f \in F\}, s, F')$
- $[\![M^*]\!] = (Q, \delta \cup \{f \overset{\epsilon}{\mapsto} s, f \in F\}, s, \{s\})$

Exemple

Automate non-déterministe associé à l'expression régulière $(\texttt{ab})*$.



Déterminisation de l'automate (egale à l'ϵ-fermeture, ici) :



Déterminisation d'un automate

Pour tout automate non déterministe $M = (Q, \Sigma, \delta, q_0, F)$, il existe un automate déterministe M' qui reconnaît le même langage.

Une solution est $M' = (2^Q, \Sigma, \delta', \{q_0\}, \{F\})$ où

$$\delta'(q', a) = \texttt{fermeture}\left(\bigcup_{q \in q'} \delta(q, a)\right)$$

où $\texttt{fermeture}\,(q')$ est le plus petit ensemble contenant q' fermé par ϵ-transition :

$$q \in \texttt{fermeture}\,(q') \Longrightarrow \delta(q, \epsilon) \subset \texttt{fermeture}\,(q')$$

Se calcule par point fixe !

L'automate déterministe associé à Q a $2^{|Q|}$ états, mais en général, seulement les états atteignables depuis $\{q_0\}$ nous intéressent. Heureusement, en pratique[a], cet ensemble est

[a] avec quelques précautions

relativement petit, souvent de l'ordre de $|Q|$.

Minimisation de l'automate

Tout automate admet un automate équivalent (qui reconnaît le même langage) avec un nombre d'états minimum.

Principe On peut le calculer à partir d'un automate déterministe par superposition d'états. Deux états sont équivalents si les suffixes de L reconnus par l'automate à partir de ces états sont égaux.

En particulier, $\forall q \in Q, \forall w \in \Sigma^*, \begin{cases} q \in F \wedge q' \notin F \Rightarrow q \not\mathcal{R}\, q' \\ \delta(q,w) \not\mathcal{R}\, \delta(q',w) \Rightarrow q \not\mathcal{R}\, q' \end{cases}$

L'automate minimal est obtenu en prenant l'équivalence la plus large. C'est le plus grossier raffinement de la partition initiale $\mathcal{P}_0 \triangleq \{F, Q \setminus F\}$ qui soit stable par image inverse $A \mapsto \delta^{-1}(A,a)$ pour tout $A \in \mathcal{P}$ et $a \in \Sigma$.

Calcul par une variation sur divide and conquer

Quelques indications

Étant donné un langage L sur Σ, on définit la relation d'équivalence $\mathcal{R}$ dans Σ^* par $x \mathcal{R} y$ ssi $\forall z \in \Sigma^*, xz \in L \Leftrightarrow yz \in L$. Cette relation est fermée par suffixe, i.e. si $x \mathcal{R} y$ alors $\forall z \in \Sigma^*, xz \mathcal{R} yz$.

On montre que les trois propriétés suivantes sont équivalentes (par exemple, dans l'ordre $(1) \Rightarrow (2) \Rightarrow (3) \Rightarrow (1)$).

(1) Le langage L défini sur Σ est reconnaissable par un automate déterministe.

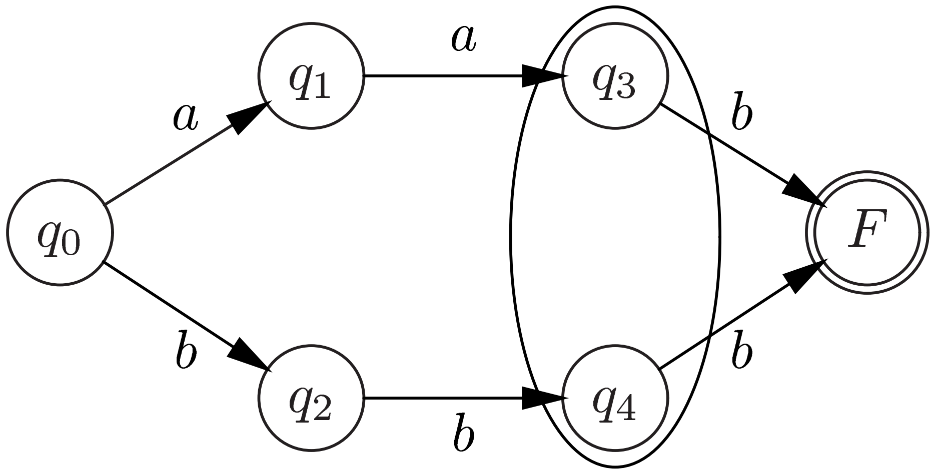
(2) L est une partition finie d'une relation d'équivalence fermée par suffixe.

(3) La partition de la relation $\mathcal{R}$ ci-dessus est finie.

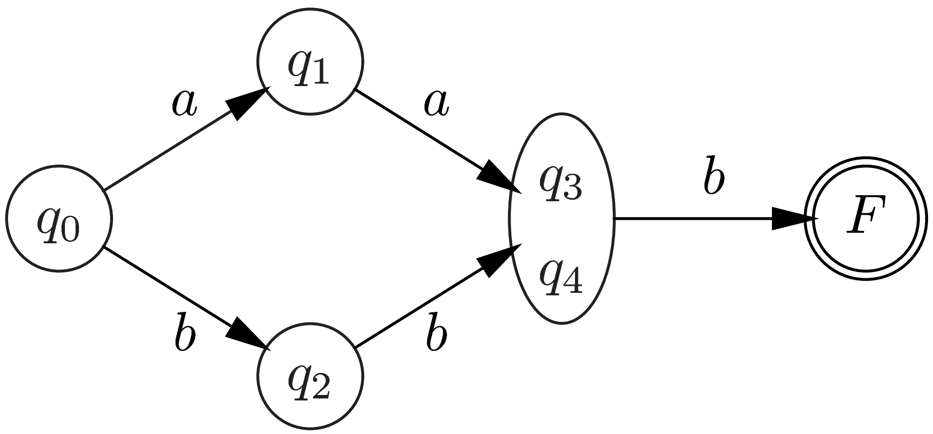
Pour montrer $(3) \Rightarrow (1)$ on considère l'automate dont les états sont les classes d'équivalence pour $\mathcal{R}$, la fonction de transition est définie par $\delta(\bar{x}, a) = \overline{xa}$, l'état initial est le mot vide et les états finaux sont $\{\bar{x}, x \in L\}$.

Cet automate est minimal, car la relation $\mathcal{R}$ ne peut pas avoir plus de classes qu'un automate reconnaissant L.

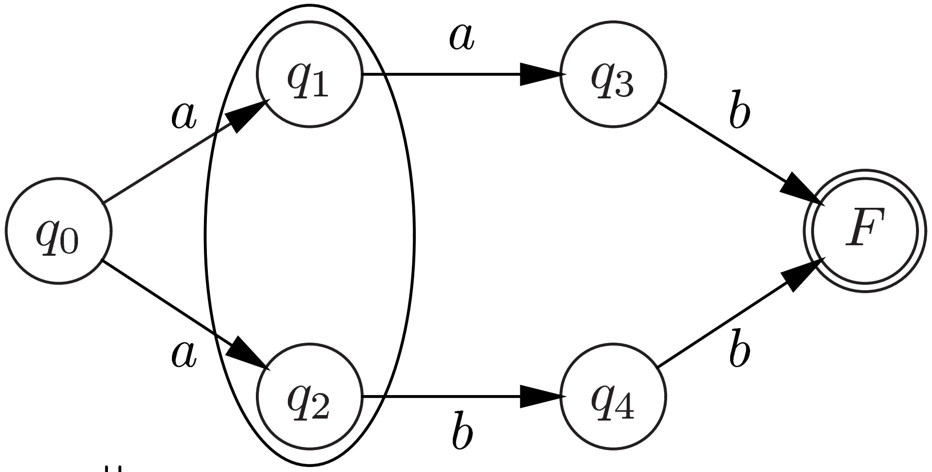
Exemple de minimisation



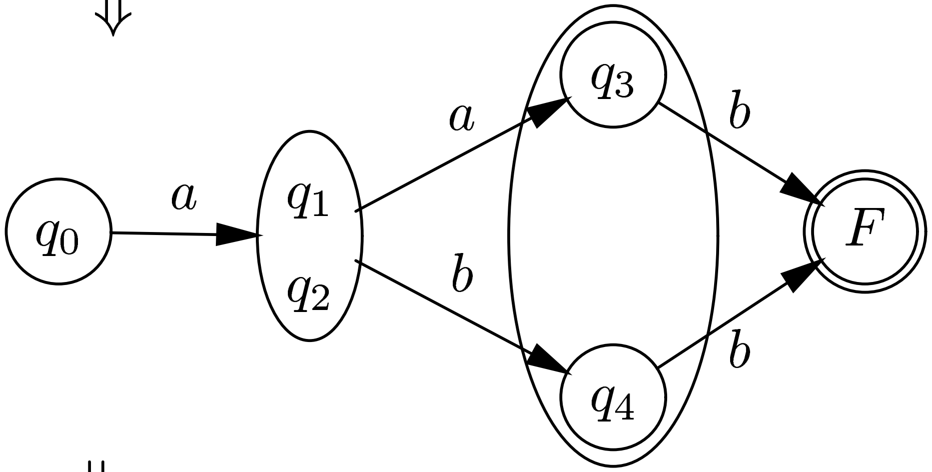
$$\Downarrow$$



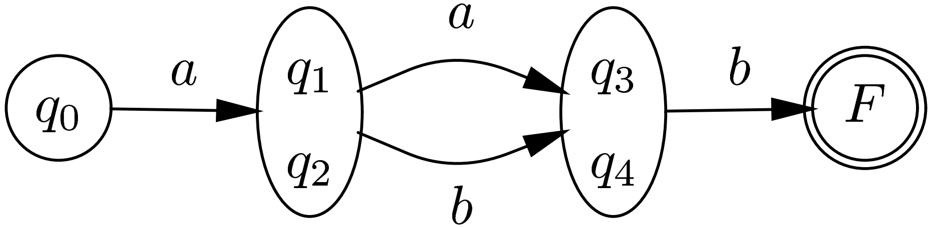
Déterminisation + minimisation



Déterminisation $\Downarrow$



Minimisation $\Downarrow$



Calcul de l'automate minimal

1. On considère un raffinement $\mathcal{P}$ de la partition $\mathcal{P}_0$. Un élément A de $\mathcal{P}$ est dit stabilisé si $\mathcal{P}$ est stable par les images inverses $\delta^{-1}(A, a)$ pour tout a.
2. On choisit un élément A non stabilisé de $\mathcal{P}$.
3. Pour chaque $a \in \Sigma$, on raffine la partition $\mathcal{P}$ afin qu'elle soit stable par les $\delta^{-1}(A, a)$.

   Pour cela, il faut diviser chaque composante $B \in \mathcal{P}$ en $B_1 = B \cap \delta^{-1}(A, a)$ et $B_2 = B \setminus \delta^{-1}(A, a)$.
4. Si B est non stabilisé, alors il faut stabiliser B_1 et B_2. Sinon, il suffit de stabiliser B_1 pour que B_2 le soit, ou inversement. On choisit de stabiliser la composante de plus petit cardinal.

Avec de (très) bonnes structures de donnée, la complexité est en $m \log(n)$ où m et n sont les nombres d'arcs et de nœuds de l'automate initial.

Représentation d'un automate

La fonction δ de domaine fini $Q \times \Sigma$ peut être représentée par une matrice de dimension 2 dont les éléments sont les états (pour un automate déterministe) ou ensembles d'états (pour un automate non-déterministe) définissant δ.

On peut choisir une représentation pleine (tableau de tableaux) ou creuse (tableau de listes) selon la situation. La première est bien sûr plus efficace en temps mais plus gourmande en espace.

Dans le cas d'une matrice creuse, on peut aussi économiser de l'espace en superposant les tableaux de domaines disjoints (astuce souvent utilisée en pratique qui a l'efficacité en temps de la représentation pleine et souvent l'efficacité en espace de la représentation creuse).

Plutôt que d'interpréter l'automate, on peut le représenter directement par une définition récursive avec filtrage.

Exemple

Automate minimal représentant l'expression régulière $(\mathtt{ab})^*$:

Les états sont des entiers. Les étiquettes sont aussi des entiers 0 et 1 représentant les transitions `a` et `b`. Par convention, l'état initial est 0. Dans la table -1 représente une transition interdite.

```
let sigma = function
    'a' -> 0 | 'b' -> 1 | _ -> failwith "sigma"
type automat= {final : bool array; delta : int array array}
let automat =
  { final = [| true; false ; |];
    delta = [| (* étiquettes :     a     b    *)
               (* état 0 *)  [|   1;   -1   |];
               (* état 1 *)  [|  -1;    0   |]; |] }
exception Exit;;
```

Interprétation des tables

```
let run automat input =
  let i = ref 0 in
  let matched = ref (-1) in
  if automat.final.(0) then matched := 0;
  let q = ref 0 in
  (try while !i < String.length input do
    let a = sigma (input.[!i]) in
    let q' = automat.delta.(!q).(a) in
    if q' < 0 then raise Exit;
    q := q';
    incr i;
    if automat.final.(q') then matched := !i;
  done with Exit -> ());
  !matched;; (* retourne le dernier caractère reconnu *)
run automat "ababaabb";; (* retourne 4 *)
```

Automate pour lex

La question n'est plus *est-ce la chaîne est reconnaissable* en entier *par une expression régulière*, mais trouver la chaîne la plus longue reconnue et la première règle (dans l'ordre de définition) la reconnaissant.

On veut construire le lexeur défini par les deux règles $(e_i\{a_i\})_{i\in I}$ (lire reconnaître l'expression régulière la plus longue parmi les e_i et exécuter l'action a_i en privilégiant l'ordre.)

Pour cela on enrichit on enrichie à la fois la construction de l'automate et son interprétation.

Fabrication de l'automate pour lex

On construit les automates non déterministes $(Q_i, \Sigma, \delta_i, q_i, F_i)_{i \in I}$.

L'automate initial non déterministe pour lex $(Q, \Sigma, \delta, q_0, \{q_F\})$ est défini par :

$$Q \stackrel{\triangle}{=} \bigcup_{i \in I} Q_i \cup q_0 \cup \{p_i, i \in I\}$$
$$\delta \stackrel{\triangle}{=} \bigcup_{i \in I} \delta_i \cup \{q_0 \stackrel{\epsilon}{\mapsto} q_i \mid i \in I\} \cup \{q \stackrel{p_i}{\mapsto} q_F \mid q \in F_i, i \in I\}$$

On déterminise cet automate, puis on retire, sur l'automate déterminisé, les transitions de sortie $q \mapsto p_i q_F$ lorsqu'il existe une transition $q \mapsto p_j q_F$ avec $i > j$.

En pratique, on représente les transitions de sortie à la place des états finaux, et on construit une structure de la forme :

```
type état = int  (* on numérote les règles *)
type rule = int   (* on numérote les états  *)
type automat=
  { final  :  rule  option array ;  delta  :  état  array  array ; };;
```

Interprétation de l'automate pour lex

On intepréte l'automate se souvenant de la dernière sortie possible

À chaque pas, on regarde s'il existe une transition possible vers la sortie, auquel cas, on indique la règle p_i permettant cette transition, ainsi que le nombre de lettres lus depuis le début.

Lorsqu'on échoue (plus de transitions possibles) on retourne exécute l'action de la dernière sortie possible (en lui passant le nombre de caractères lus), sinon on échoue.

En Ocaml

Les états finaux sont les transitions

```
let run automat input =
  let i = ref 0 in
  let matched = ref None in
  let q = ref 0 in
  (try while !i < String.length input do
    let a = sigma (input.[!i]) in
    let q' = automat.delta.(!q).(a) in
    if q' < 0 then raise Exit;
    q := q';
    incr i;
    begin match automat.final.(q') with
    | Some r -> matched := Some (r, !i)
    | None -> ()
    end;
  done with Exit -> ());
  match !matched with
    Some (r, k) ->
         Some (r, String.sub input 0 k),
         String.sub input k (String.length input - k)
  | None -> None, input;;
```

Langage reconnu par un automate

Le langage reconnu par un automate est rationnel (nombre fini de suffixes, voir aussi le cours suivant), en particulier, un automate ne peut pas reconnaître le langage composé des expressions bien parenthésées.

En effet un automate a un nombre fini d'états. Or la reconnaissance des expressions bien parenthésées demande de se souvenir du nombre de parenthèses déjà reconnues qui peut être arbitraire.

Pour définir un langage de programmation, il nous faut donc des outils plus puissants que les expressions régulières.

Ocamllex

Les actions sont des expressions arbitraires, donc Ocamllex est un langage complet, et la question du langage reconnu par Ocamllex n'as pas de sens si les actions peuvent interagir avec le flux d'entrée.

Toutefois avec une seule règle et sans interaction entre les actifs et le flux d'entrée, on reste à l'intérieur du formalisme précédent.

Reconnaissance des parenthèses On utilise simplement un compteur pour le nombre de parenthèses ouvertes.

```
{ let c = ref 0 }
rule token = parse
     '('   { incr c; token lexbuf }
 |   ')'   { if !c > 0 then (decr c; token lexbuf) else false }
 |   eof   { (!c = 0) }
{ let test s = token (Lexing.from_string s) }
```

Élimination des commentaires

Une variante toute simple devient un programme intéressant...

```
{ let depth = ref 0 }
rule uncomment = parse
  | "(*<*)"   { incr depth; uncomment lexbuf }
  | "(*>*)"   { if !depth > 0 then decr depth else exit 1;
                uncomment lexbuf }
  | eof       { if !depth > 0 then exit 1 }
  | _         { if !depth = 0 then
                   print_string (Lexing.lexeme lexbuf);
                uncomment lexbuf; }
{ let lexbuf = Lexing.from_channel stdin in
  uncomment lexbuf }
```

Exercice 1 *Parser aussi les chaînes de caractères.* □

Exercices

Exercice 2 *Écrire un lexeur pour le langage Pseudo-Pascal. Les lexèmes sont*

- *Les blancs (blanc, tabulation, retour à la ligne)*
- *Les mots clés :* **var**, *alloc*, **false**, **true**, **read**, **write** , **writeln**, **array**, **of**, **do**, **begin** , **end**, **if**, **then**, **else**, **while**, **type** , **function**, **procedure**, **integer**, **boolean**, **program**.
- *les entiers*
- *les identificateurs (composés de caractères alphabétiques (mininuscule ou majuscules et terminant éventellement par des chiffres.*
- *les lexèmes :* $;; \quad := \quad <> \quad <= \quad >= \quad < \quad > \quad ; \quad \tilde{} \quad : \quad = \quad - \quad + \quad * \quad / \quad (\quad) \quad [\quad]$.

Exercices (suite)

La liste de mots clés, qui peut être assez longue, au risque de conduire à une explostion du nombre d'états de l'automate minimal. Pour éviter cela, on confond les mots clés avec les identificateurs au niveau du lexeur, et on les différencie dans le langage hôte. Écrire une nouvelle version du lexeur, équivalente à la précédente, qui procède ainsi. □

En travaux dirigé, suivre l'énoncé détaillé de Luc Maranget.

Extraire les URL d'un document HTML

Il s'agit d'écrire un lexeur qui extrait d'un document HTML une liste de chaîne de caractères, représentant l'ensemble des liens vers d'autres documents.

Un document HTML contient un ensemble de balises séparées par du texte. Les balises sont entre les caractères < et >). On pourra considérer que ces caractères n'apparaissent nulle part ailleurs.

1. En s'inspirant du lexeur qui élimine les commentaires, écrire un lexeur qui ne retient que les balises HTML.
2. Raffiner le lexeur pour n'imprimer que les URL des balises de type `A`.
3. Pour simplifier on pourra d'abord considérer que
   – Les chaînes ne contiennent pas la clé `href`.
   – Les chaînes ne contiennent pas le caractère `'"'`.
4. Relacher les contraintes précédentes.
5. On pourra aussi considérer les liens ”src” dans les ancres de type `IMG`.

Analyse des URL

Une URL est elle-même structurée : elle commmence par le nom d'un protocole (optionnel), l'adresse d'un serveur (optionnel), un chemin absolu ou relatif, et des suffixes (arguments ou étiquettes).

Déstructurer une URL en ses composantes de base.

– en utilisant un lexeur,
– en utilisant la librairie des str des expressions régulières.