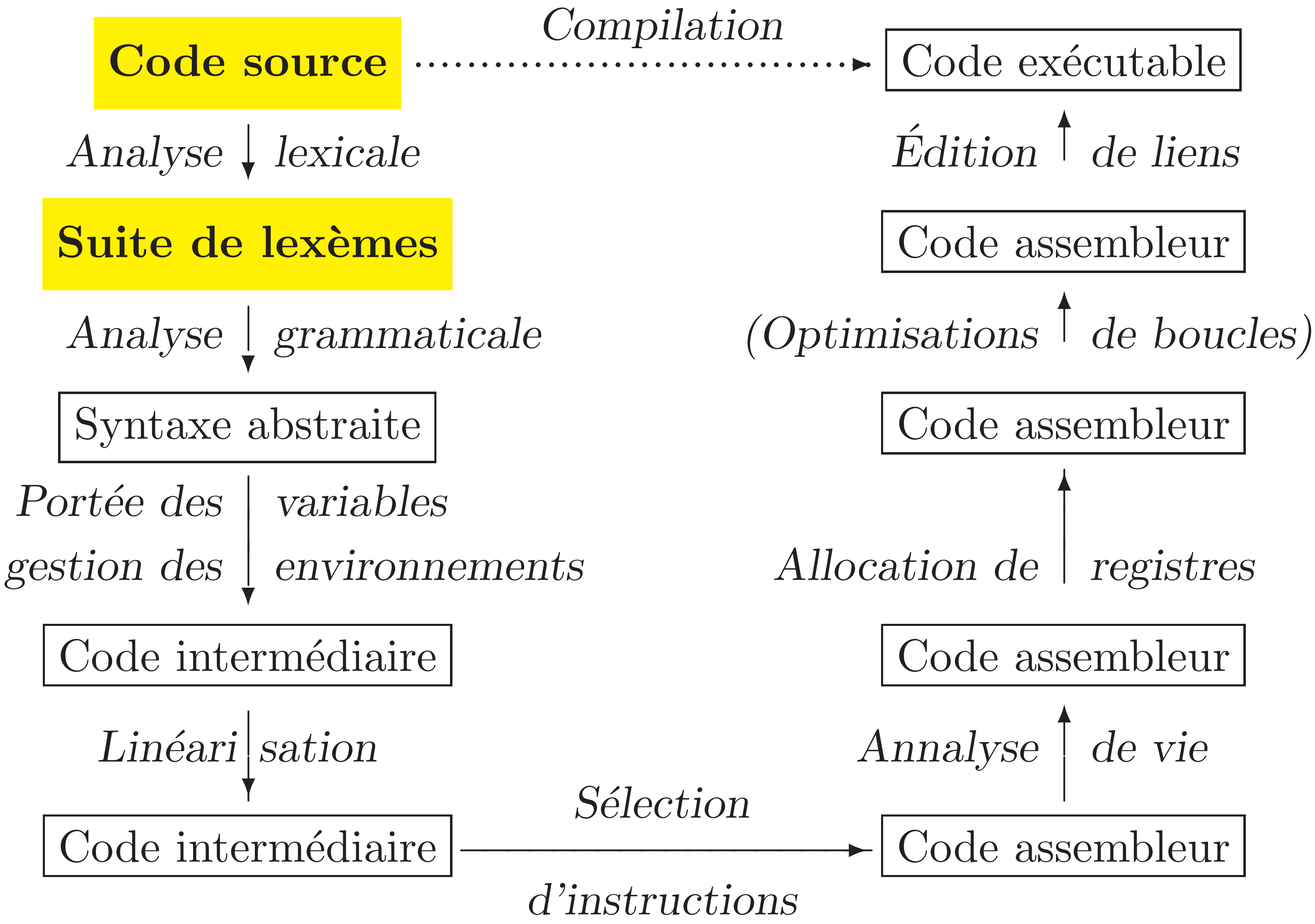
Langages formels
Analyse lexicale
Expressions régulières
Automates

Luc Maranget

`http://pauillac.inria.fr/~maranget`

2005


Code source
Compilation
Code exécutable
Analyse lexicale
Édition de liens
Suite de lexèmes
Code assembleur
Analyse grammaticale
(Optimisations de boucles)
Syntaxe abstraite
Code assembleur
Portée des variables
gestion des environnements
Allocation de registres
Code intermédiaire
Code assembleur
Linéarisation
Annalyse de vie
Code intermédiaire
Sélection
d'instructions
Code assembleur

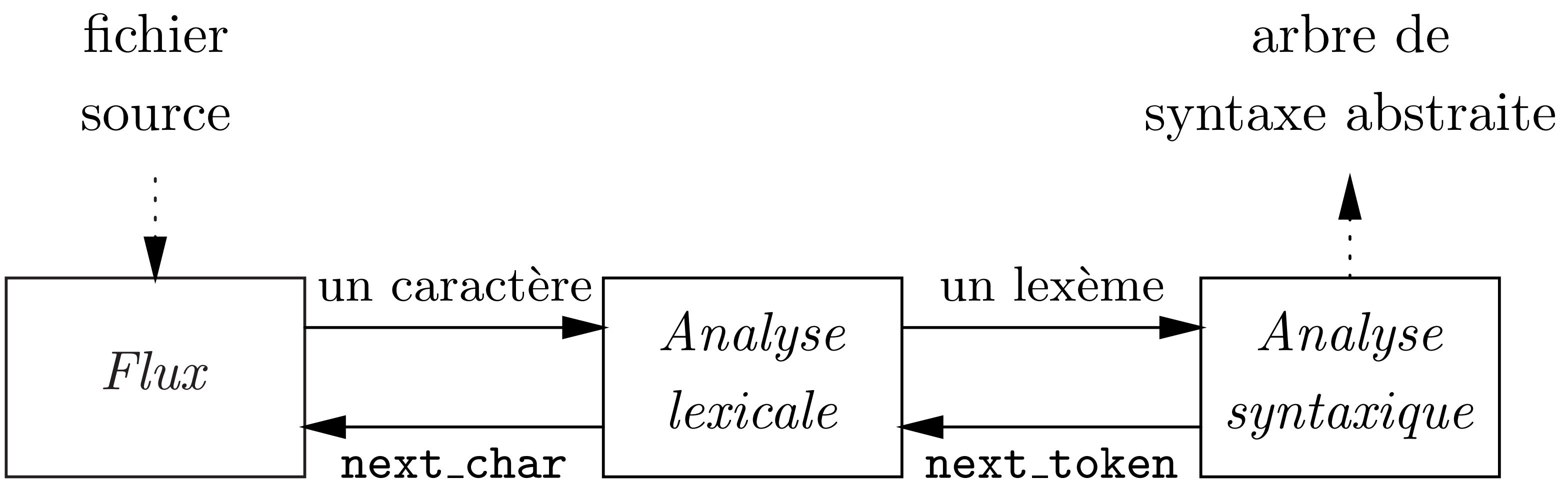
Analyse en deux phases

Conceptuellement, deux phases :

Analyse lexicale transforme une suite de caractères en une suite de lexèmes (mots).

Analyse grammaticale transforme une suite de lexèmes en une représentation arborescente (syntaxe abstraite).

Stricto-sensu, une seule passe.



Enjeux

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

Ces deux analyses utilisent de façon essentielle les automates, 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 (`emacs`, `grep`...), et fournies en bibliothèque dans la plupart des langages de programmation (cf. Perl).

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.

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 et a 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^*.

Exemples

1. Σ_1 est l'alphabet et L_1 l'ensemble des mots du dictionnaire français avec toutes leurs variations (pluriels, conjugaisons).
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, des symboles, des identificateurs, et de l'ensemble des entiers décimaux.
4. Σ_4 est L_3 et L_4 est l'ensemble des programmes pseudo pascal.
5. Σ est $\{a, b\}$ et L est $\{a^n b^n \mid n \in \mathbb{N}\}$ (expressions bien parenthésées).

Expressions régulières

Un formalisme simple permettant de décrire certains langages simples (les langages réguliers).

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

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

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

- $[abc]$ pour $(a \mid b \mid c)$ et $[a_1{-}a_2]$ pour $\{c \in \Sigma, a_1 \leq c \wedge c \leq a_2\}$ (et $[\hat{}\ \ldots]$, les complémentaires).
- $M?$ pour $M \mid \epsilon$ et M^+ pour MM^*.
- _ (ou parfois .) pour n'importe quel caractère.

Exemples

Entiers décimaux

$$[0-9]+$$

Entiers hexadécimaux

$$0x([0-9a-fA-F])+$$

Nombres (Pascal)

$$[0-9]+ \quad (.[0-9]*)? \quad ([Ee][-+]?[0-9]+)?$$

Sources Caml

« `# ls *.ml{,[ily]}` » (Avec quelques changements de syntaxe. . .)

Analyse lexicale

On décrit chaque sorte de lexème par une expression régulière :

1. Les mots clés : `"let"`, `"in"`
2. Les variables :
   `['a'-'z''A'-'Z']+ ['a'-'z''A'-'Z''0'-'9']*`
3. Les entiers : `['0'-'9']+`
4. Les symboles : `'('`, `')'`, `'+'`, `'*' '='`
5. Les espaces : `(' ' | '\n' | '\t')`, à oublier.

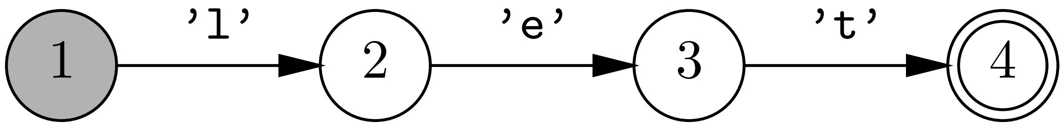
À reconnaître et transformer en :

```
type token =
  | LET | IN                                  (* mots-clés *)
  | VAR of string                             (* variables *)
  | INT of int                                (* entiers *)
  | LPAR | RPAR | ADD | SUB | MUL | DIV | EQUAL (* symboles *)
```

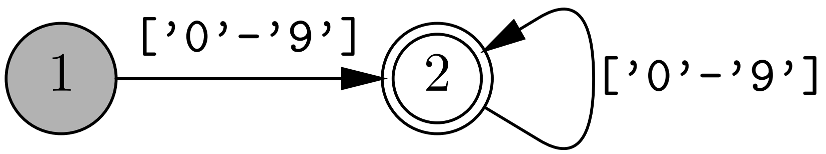
Procédure

Pour toute expression régulière M, il existe un automate qui reconnaît $[\![M]\!]$.

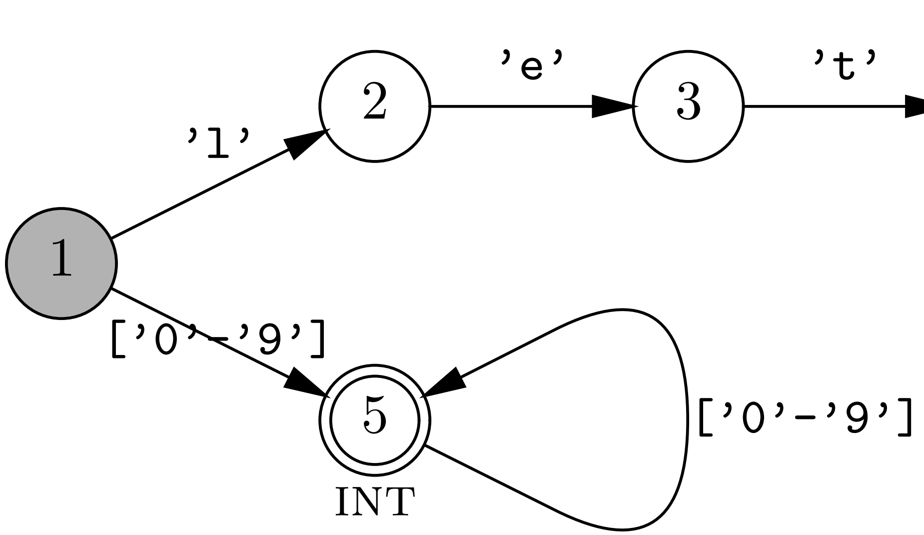
1. Les mots clefs :



2. Les entiers :



3. L'un ou l'autre :



Utilisation de l'automate, on veut un lexème…

- Appeler l'automate,
- Il renvoie le code d'un état final atteint.

Recommencer pour lire tous les lexèmes (*flux* = entrée impérative).

Ambiguïté

Problème Il y a des ambigüités :

- "let" pourrait être reconnu comme une variable.
- "lettre" pourrait aussi être reconnu comme la séquence `LET; VAR "tre"` ou encore `VAR "let"; VAR "tre"`

Solution Des règles de priorité.

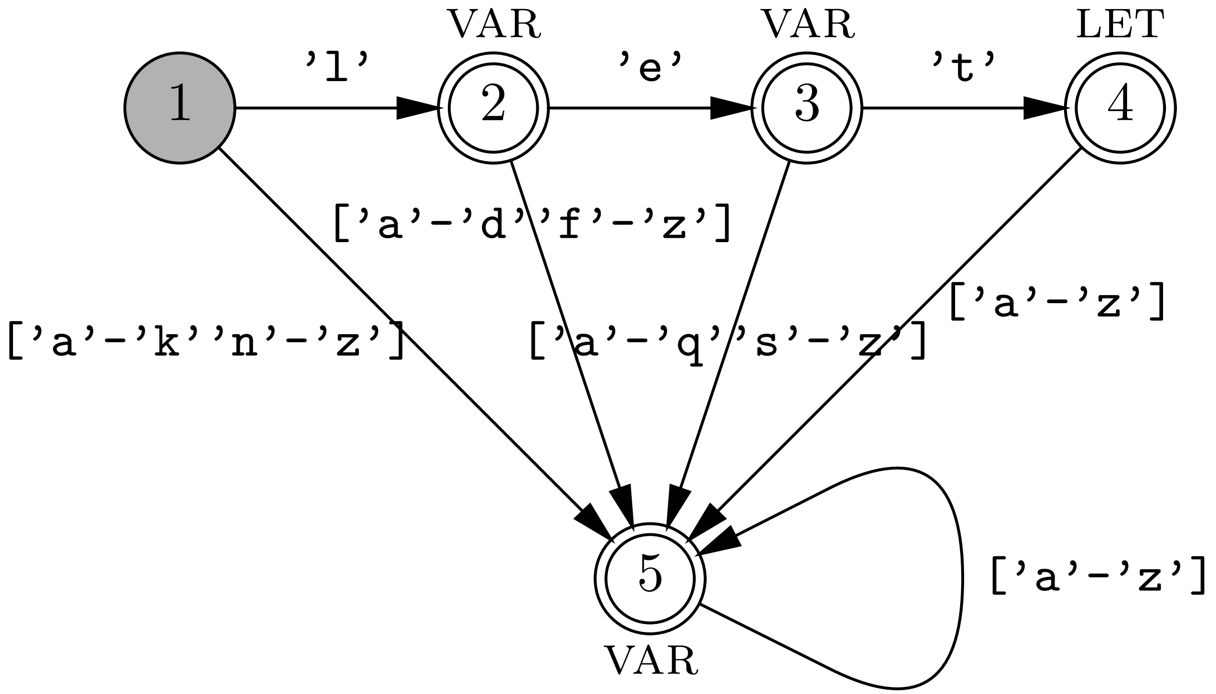
1. Entre deux lexèmes de taille différentes : le plus long.
2. Sinon, suivre l'ordre de définition des sortes de lexèmes.

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

```
LET; VAR "lettre"; EQUAL; INT 3; IN; INT 1;
PLUS; VAR "fin"
```

Réalisation des règles de priorité

1. **Le lexème le plus long** À l'exécution de l'automate : se souvenir du dernier état final rencontré, le rendre en cas de blocage.
2. **L'ordre de définition** Au moment de la construction de l'automate.



`ocamllex`, un générateur d'analyseurs syntaxiques

L'analyseur est sous forme de règles « `|` *motif* `{`*action*`}` » dans un fichier `lexer.mll`.

Compilation en deux temps :

- `ocamllex lexer.mll`
- `ocamlc -c lexer.ml`

Le fichier `lexer.ml` contient l'automate sous forme Caml. Cet automate déclenche la bonne action lorsque son entrée est filtrée par un des motifs.

- L'*action* peut être de renvoyer un lexème. Le type de l'automate est alors `Lexing.lexbuf -> Token.token`.
- On peut, dans le même fichier `.mll`, définir plusieurs automates (mutuellement récursifs) à la fois.

  `rule` $auto_1$ `= parse` ... `and` $auto_2$ `= parse` ...

Exemple de source `ocamllex`

```
{ (* prélude copié au début de lexer.ml *)
open Token
exception Error
}
(* définition de l'analyseur « token » *)
rule token = parse
(* Les lexèmes stricto-sensu *)
| '('   {LPAR} | ')'  {RPAR} ...
| "let" {LET}  | "in" {IN}
| ['A'-'Z' 'a'-'z'] ['A'-'Z' 'a'-'z' '0'-'9']* as lxm
        {VAR lxm}
| ['0'-'9']+ as i
        {INT (int_of_string i))}
```

Exemple (suite)

```
(* Règles supplémentaires *)
| eof                    {EOF}
| [' ' '\n' '\t' ]  {token lexbuf}
| ""                     {raise Error}
```

Type de l'analyseur token ? Lexing.lexbuf -> Token.token

Usage

```
let entrée = Lexing.from_channel stdin in (* Fabriquer le flux *)
let count = ref 0 in
while Lexer.token entrée <> Token.EOF do
  count := !count + 1
done
```

Première sorte de commentaire

Le commentaires sont enlevés au moment de l'analyse syntaxique.

// Je vais au bout de la ligne

Solution en une ligne de plus pour `token`

```
| "//" [^ '\n']* '\n'? {token lexbuf}
```

Deuxième sorte de commentaire

Une ouverture, une fermeture , imbrication interdite (mais pourquoi donc ?).

/ Je suis un commentaire*
*sur deux lignes */*

Solution Avec un deuxième automate :

```
rule token = parse
    ...
| "/*"  {incomment lexbuf}
and incomment = parse
| "*/"  {token lexbuf}
| _     {incomment lexbuf}
| eof   {raise Error}
```

Troisième sorte de commentaires

Les mêmes imbricables (mais pourquoi faire ?)

(∗ Je suis
(∗ un commentaire ∗)
dans le commentaire ∗)

Il y a un vrai problème théorique. Les automates finis ne savent pas reconnaître le langage des parenthèses.

Solution, dans le code des actions

```
{let depth = ref 0}
rule token = parse
| "(*"  {depth := 1 ; incomment lexbuf ; token lexbuf}
and incomment = parse
| "*)"
(* fermeture "*)" de l'ouverture "(*" à la profondeur depth *)
    {depth := !depth-1 ;
    if !depth > 0 then incomment lexbuf}
| "(*"  {depth := !depth+1 ; incomment lexbuf}
| _     {incomment lexbuf}
| eof   {raise Error}
```

Ce code est-il pleinement satisfaisant en pratique ? Pas tout à fait, voir le commentaire dans le code lui-même !

Une autre solution, sans référence

```
rule token = parse
  ...
| "(*"  {incomment 0 lexbuf ; token lexbuf}

and incomment depth = parse
| "*)"
    {if depth > 0 then incomment (depth-1) lexbuf}
| "(*"  {incomment (depth+1) lexbuf}
| _     {incomment depth lexbuf}
| eof   {raise Error}
```

Type de incomment ?

```
val incomment : int -> Lexing.lexeme -> unit
```

Une autre solution, sans compteur du tout

```
rule token = parse
| "(*"  {incomment lexbuf ; token lexbuf}

and incomment = parse
| "*)" {()}
| "(*" {incomment lexbuf ; incomment lexbuf}
| _     {incomment lexbuf}
| eof  {raise Error}
```

Les chaînes (avec citations)

"Voila \" le délimiteur des chaînes et le caractère \\ pour citer\n"

C'est très semblable au commentaires `/*... */`. Mais il faut récupérer le contenu de la chaîne.

```
let sbuff = Buffer.create 16 (* fabriquer le buffer *)

(* Mettre un caractère à la fin de sbuff *)
let put_char c = Buffer.add_char sbuff c

(* Récupérer le contenu de sbuff et le réinitialiser *)
let to_string () =
  let r = Buffer.contents sbuff in
  Buffer.clear sbuff ; r
```

Solution, avec un buffer

```
rule token = parse   ...
| '"'  {STRING (instring lexbuf)}

and instring = parse
| '"' {to_string ()} (* Fin *)
|  '\\' ('\\' | '"' as c) (* Caractères cités *)
    {put_char c ;  instring lexbuf}
| '\\' 'n'  (* Caractères spéciaux *)
    {put_char '\n' ; instring lexbuf}
| _ as c
    {put_char c ; instring lexbuf}
| eof
    {raise Error}
```

Abondance de mots-clés

Lorsqu'il y a beaucoup de mot-clés.

```
rule token = parse
    ...
| "var" { VAR } | "alloc" { ALLOC }
| "false" { BOOL false } | "true" { BOOL true }
    ...
|  "boolean" { BOOLEAN } |  "program" { PROGRAM }
| [ 'A'-'Z' 'a'-'z' ] + as lxm { Var lxm }
    ...
```

Quel problème pratique peut-il se poser : La taille de l'automate, en gros il aura N états, où N est la somme des longueurs des mot-clés.

Abondance de mot-clés

- Dans le prélude, on fabrique une association chaîne $\longrightarrow$ lexème.
- Dans, l'automate, on reconnaît variables et mot-clés ensemble, puis on discrimine.

```
(* Table de hachage, taille initiale 17 *)
let keywords = Hashtbl.create 17

(* Vérifier/ajouter un mot-clé *)
let check_keyword s =
  try Hashtbl.find keywords s with Not_found -> IDENT s

and add_keyword s k = Hashtbl.add keywords s k

(* Définir les mot-clés *)
let () =
  add_keyword "var" VAR ; add_keyword "alloc" ALLOC ;
  ...
  add_keyword "boolean" BOOLEAN ; add_keyword "program" PROGRAM ;
  ()
```

```
rule token = parse
      $\ldots$
| [ 'A'–'Z''a'–'z'] + as lxm { check_keyword lxm }
      $\ldots$
```

Comparaisons, pour deux analyseurs des lexème de Pseudo-Pascal.

```
# ocamllex sans_hash.mll
117 states, 8014 transitions, table size 32758 bytes
# ocamllex avec_hash.mll
26 states, 435 transitions, table size 1896 bytes
```

Diversion : utilisation avancée de `ocamllex`

On dispose de 1000 fichiers de style agenda, le format est assez libre :

- Chaque ligne contient un nom, suivi éventuellement d'un numéro de téléphone.
- Les noms sont composés de suites de mots, qui peuvent contenir des tirets à l'intérieur.
- Les numéros de téléphone ont 10 chiffres et peuvent contenir des espaces à peu-près n'importe où.

On souhaite (pour lire tout ça avec Excel) transformer les fichiers en un format plus strict : le CSV.

- Chaque ligne comporte deux items.
- Les items sont séparés par des virgules.
- Les items sont compris entre deux double-quotes « `"` ».

Diversion : les motifs

```
{ ... }
(* Noms de personnes *)
let blank = [' ' '\t']
let lettre = ['A'-'Z' 'a'-'z' 'À'-'Ö' 'Ù'-'ö' 'ù'-'ÿ']
let mot = lettre+
let mot2 = mot ('-' mot)*
let nom = mot2 (blank+ mot2)*
(* ex: Jean-Hugues Leroy de Pressalé *)

(* Nunéro de téléphone à 10 chiffres *)
let digit = ['0'-'9'] (* un chiffre *)
let db = digit blank*
let numero = db db db db db db db db db digit
```

Diversion : l'automate

```
let eol = '\n' (* Unix *) | ('\r' '\n') (* Windows *) | '\r' (* Mac *)
(* Saut de ligne universel (?) *)

rule line = parse
| (nom as name) (blank+ (numero as phone))? blank* eol
    {Printf.printf "\"%s\",\"%s\"\n"
      name (match phone with None -> "" | Some s -> s) ;
     line lexbuf }
| eof { () }

{ let buf = from_channel stdin

let _ = line buf ; exit 0 }
```

Retour à la réalité : les erreurs

En cas d'erreur, il est gentil d'indiquer un minimum d'information :

- une position dans le fichier d'entrée,
- le genre de l'erreur,
- le texte non compris.

Exemple Caml :

```
let x = 1
let y = "coucou
let z = 1
```

```
File "er.ml", line 2, characters 8-9:
String literal not terminated
```

Quant à tenter de réparer…

Comment fonctionnne `ocamllex` ?

Le schéma traditionnel :

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.

En fait, `ocamllex` produit directement des automates déterministes et ne les minimise pas.

Automates finis déterministes (DFA)

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

Automates finis non-déterministes (NFA)

Comme DFA, compte tenu des deux détails suivants :

1. Il peut y avoir plusieurs transition homonymes issues d'un sommet.
2. Il existe des transitions « *spontanées* » (étiquelle ϵ.)

Modèle de δ comme une relation sur $Q \times (\Sigma \cup \{\epsilon\}) \times Q$.

$$\begin{cases} & \delta(q, \epsilon, q) & \\ \\ \delta(q, \epsilon, q'') \wedge \delta(q'', w, q') & \Rightarrow & \delta(q, w, q') \\ \delta(q, a, q'') \wedge \delta(q'', w, q') & \Rightarrow & \delta(q, aw, q') \end{cases}$$

Langage reconnu : $\{w \mid \exists q_f \in F, \delta(q_0, w, q_f)\}$

Formalisation de l'exécution d'un NFA

Le plus simple est de considérer des ensembles d'états :

- la fermeture $\mathcal{F}$, comme le point fixe de $S = S \cup \{q \mid \exists q' \in S, \delta(q', \epsilon, q)\}$ (autrement dit, on suit toutes les transitions spontanées possibles issues des états de S).
- la consommation d'un caractère a, noté $\mathcal{C}_a$ comme $\mathcal{C}_a(S) = \{q \mid \exists q' \in S, \delta(q', a, q)\}$.

État initial :

$$\mathcal{F}(\{q_0\})$$

Une étape d'exécution :

$$S \xrightarrow{a} \mathcal{F}(\mathcal{C}_a(S))$$

Automate d'une expression régulière

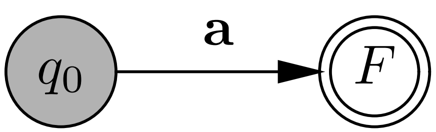
L'alphabet Σ est fixé.

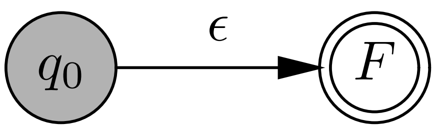
On associe à une expression régulière M un automate non déterministe (Q, δ, s, F) défini récursivement par :

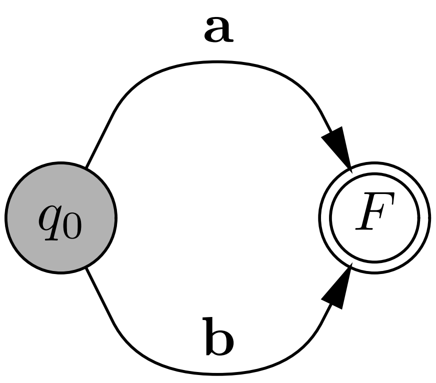
- $[\![a]\!] = (\{s, f\}, \{s \overset{a}{\mapsto} f\}, s, \{f\})$
- $[\![\epsilon]\!] = (\{s, f\}, \{s \overset{\epsilon}{\mapsto} f\}, s, \{f\})$
- $[\![M \mid M']\!] = (Q \cup Q' \cup \{s''\}, \delta \cup \delta' \cup \{s'' \overset{\epsilon}{\mapsto} s, s'' \overset{\epsilon}{\mapsto} s'\}, s'',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

La même chose en dessins

Pour **a**, ϵ et [**ab**].

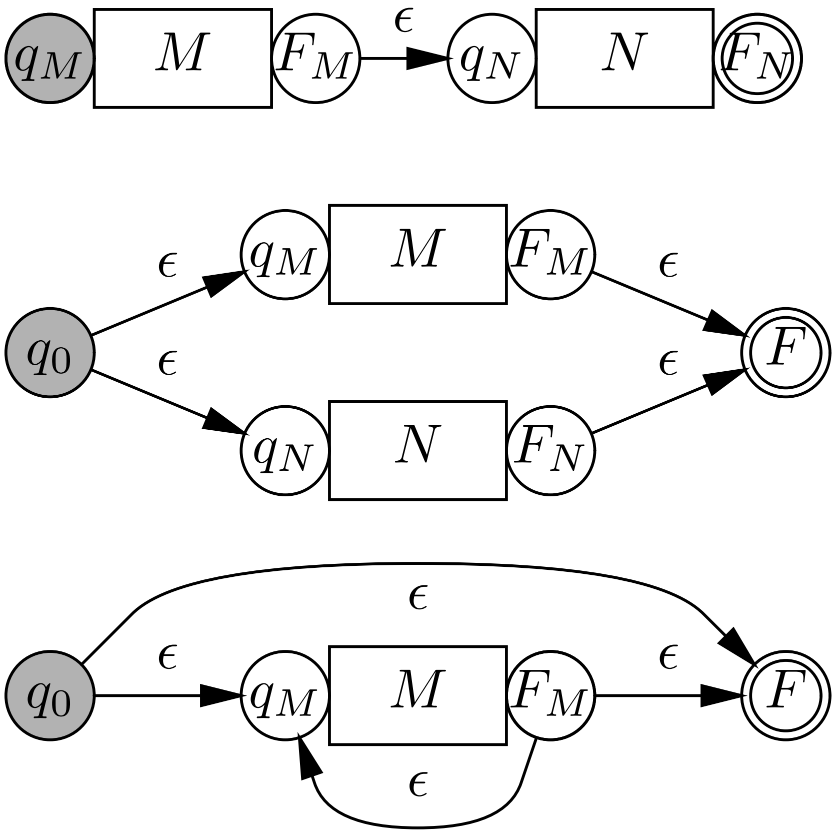






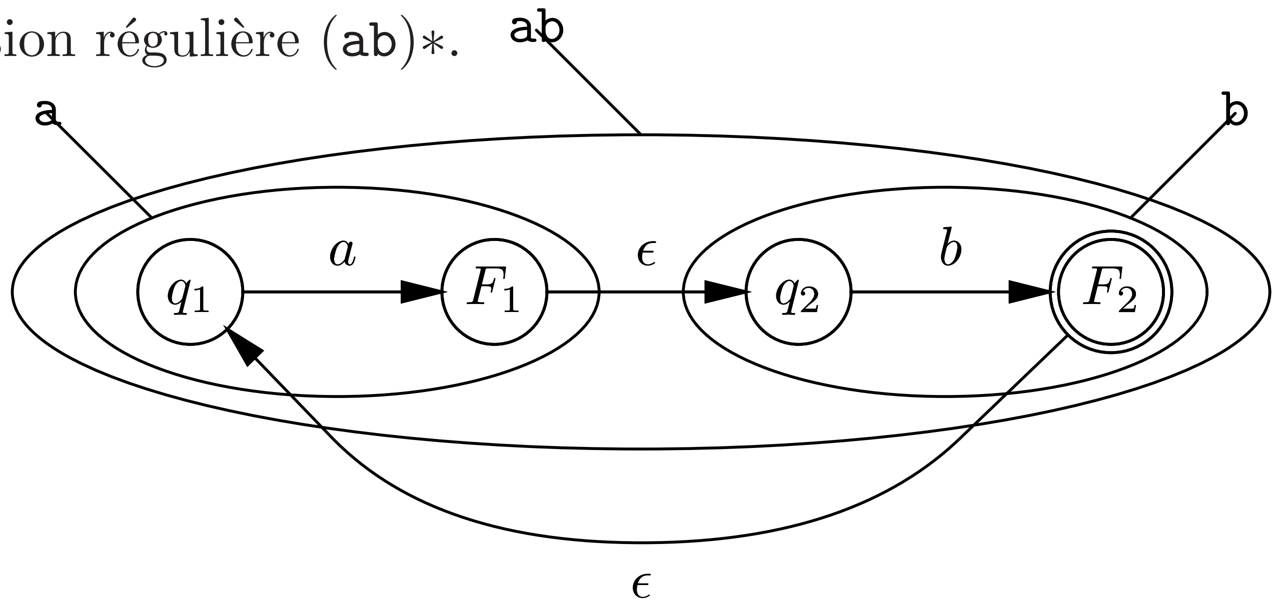
La même chose en dessins

Pour $M\ N$, $M \mid N$ et $M*$.

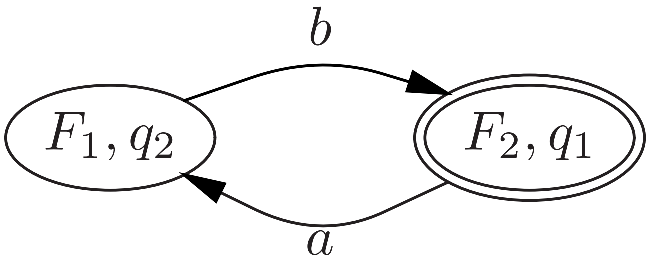


Exemple

Expression régulière (`ab`)$*$.



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



Déterminisation d'un automate

Pour tout automate non déterministe $A_n = (Q, \delta, q_0, F)$, il existe un automate déterministe $A_d = (R, \gamma, Q_0, G)$ qui reconnaît le même langage.

On pose $R \subseteq 2^Q$ et tout simplement :

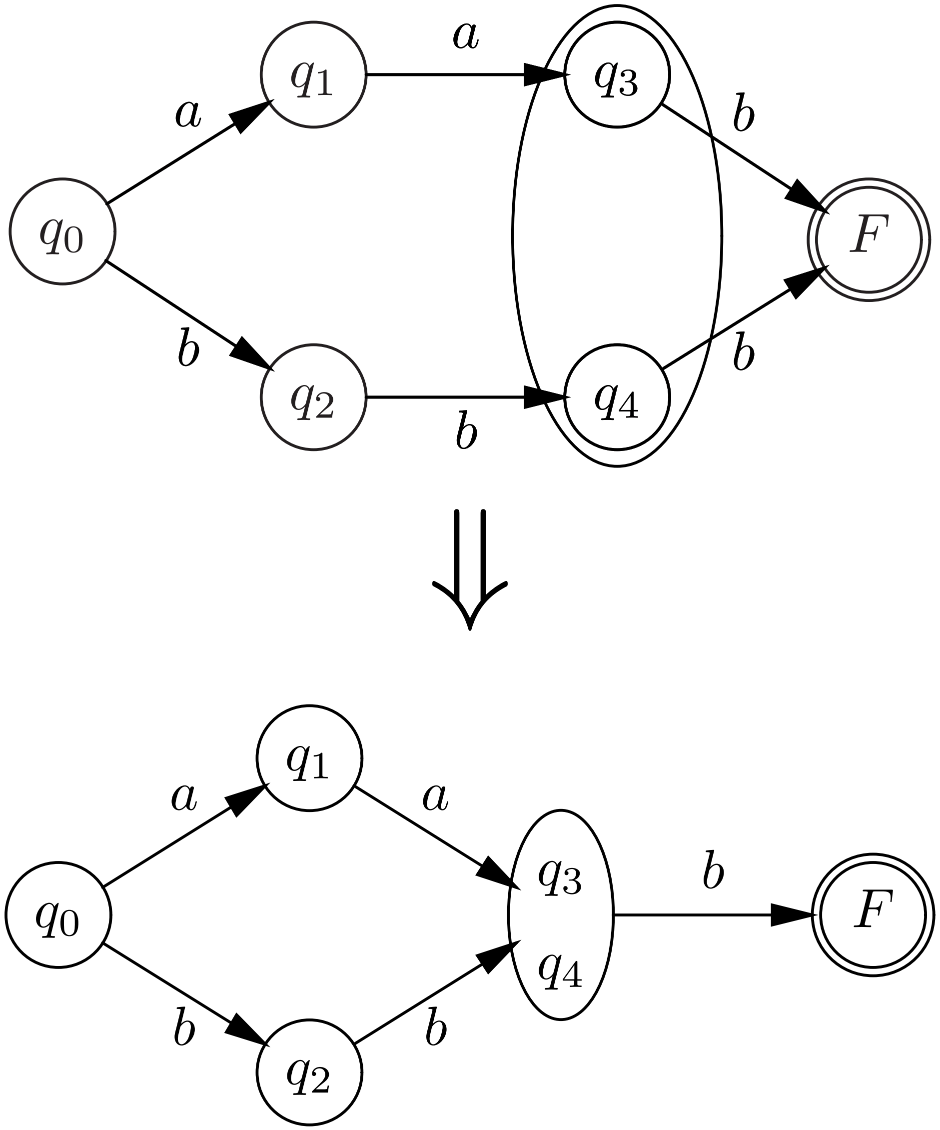
$$Q_0 = \mathcal{F}(\{q_0\})$$

$$\gamma(Q_i, a) = Q_j \quad \Leftrightarrow \quad Q_j = \mathcal{F}(\mathcal{C}_a(Q_i))$$

Cela revient à parcourir toutes les exécutions de A_n en s'arrêtant aux état déjà vus.

Minimisation de l'automate

Elle repose sur l'équivalence de langage de suffixes définis par deux états.



Un peu plus sur la minimisation

En pratique on utilise une relation d'équivalence $\cong$ plus facilement calculable (par point fixe) :

$$q_i \cong q_j \quad \Leftrightarrow \quad (q_i \in F \;\Leftrightarrow\; q_j \in F) \wedge (\forall a \in \Sigma, \delta(q_i, a) \cong \delta(q_j, a))$$

La relation $\cong$ est facile à calculer par point fixe. Le faire de façon efficace est une autre affaire (nombreux algorithmes publiés).