Analyses lexicales et syntaxiques

- L'analyse lexicale transforme une suite de caractères en une suite de lexèmes (mots).
- L'analyse syntaxique transforme une suite de lexèmes en une représentation arborescente (arbre de syntaxe abstraite).

Ces deux phases logiques sont implantées comme des fonctions OCaml : le pilote appelant la fonction d'analyse syntaxique qui appelle la fonction d'analyse lexicale qui lit le flux de caractères entrant et en retire les caractères reconnus comme constituant un lexème (il y a donc un effet de bord dû à l'intéraction avec le système de fichiers).

Enjeux des analyses lexicales et syntaxiques

- Les analyses lexicales et syntaxiques ont un domaine d'application bien plus large que celui de la compilation. On les retrouve comme première passe dans de nombreuses applications (analyses de commandes, de requêtes, de documents HTML etc.).
- Ces deux analyses emploient de façon essentielle les *automates*, que l'on retrouve donc dans de nombreux domaines de l'informatique et de la télématique.
- Les *expressions régulières* sont un langage de description d'automates ; elles sont utilisées dans de nombreux outils Unix (`emacs`, `grep` etc.) et sont fournies en bibliothèque avec la plupart des interprètes et compilateurs de langages de programmation (p.ex. Perl).

Objectifs

L'étude détaillée des automates et des grammaires formelles pourrait constituer un cours à part, nous nous contentons donc ici du minimum, avec comme but

- d'expliquer le fonctionnement des analyseurs de façon à pouvoir écrire soi-même des analyseurs lexicaux (*lexers* ou *scanners*) ou syntaxiques (*parsers*) ;
- de se familiariser aussi avec les expressions régulières et les automates, à cause de leur omniprésence.

Le but n'est donc ni d'écrire le cœur d'un analyseur, ni d'inventorier toutes les techniques d'analyse.

Langages formels

- Un *alphabet* est un ensemble fini non vide de *caractères*. On note souvent les alphabets Σ et les caractères a, b ou c.
- Un *mot* sur Σ est une suite, éventuellement vide, de caractères de Σ. Le mot vide est noté ε. Un mot non vide est noté par ses caractères séparés par un point (centré), par exemple $a \cdot b \cdot c$ avec $a, b, c \in \Sigma$. Le point dénote un opérateur dit de *concaténation*, que l'on peut généraliser simplement aux mots eux-mêmes : $x \cdot y$, où x et y sont des mots sur Σ.
- Le mot vide ε est un élément neutre pour la concaténation des mots : $x \cdot \varepsilon = \varepsilon \cdot x = x$ pour tout mot x.
- La concaténation est une opération associative : $x \cdot (y \cdot z) = (x \cdot y) \cdot z$
- On écrira simplement xy au lieu de $x \cdot y$

Langages formels (suite)

- Un *langage* L sur Σ est un ensemble de mots sur Σ. La concaténation peut s'étendre aux langages : $L_1 \cdot L_2 = \{xy \mid x \in L_1, y \in L_2\}$. Nous pouvons ainsi définir inductivement l'ensemble Σ^n des mots de longueur n sur l'alphabet Σ :

$$\begin{cases} \Sigma^0 = \{\varepsilon\} \\ \Sigma^{n+1} = \Sigma \cdot \Sigma^n \end{cases}$$

- L'ensemble de tous les mots sur Σ est alors $\Sigma^* = \bigcup_{n \geqslant 0} \Sigma^n$.
- L'ensemble de tous les mots *non vides* sur Σ est $\Sigma^+ = \bigcup_{n \geqslant 1} \Sigma^n$.
- On vérifie aisément que $\Sigma^* = \{\epsilon\} \cup \Sigma \cdot \Sigma^* = \{\epsilon\} \cup \Sigma^* \cdot \Sigma$
- Soient x, y et w trois mots de Σ^* tels que $w = xy$. Alors x est un *préfixe* de w et y est un *suffixe* de w. Si $x, y \in \Sigma^+$, alors x est un *préfixe propre* de w, et y est un *suffixe propre* de w.

Langages réguliers

L'ensemble $\mathcal{R}(\Sigma^*)$ des *langages réguliers* (ou *rationnels*) sur Σ^* est défini inductivement comme étant la plus petite famille de parties de Σ^* (par définition, Σ^* est le plus grand langage sur Σ) vérifiant les propriétés

- $\varnothing \in \mathcal{R}(\Sigma^*)$
- $\{\varepsilon\} \in \mathcal{R}(\Sigma^*)$
- $\forall a \in \Sigma.\{a\} \in \mathcal{R}(\Sigma^*)$
- $\forall R_1, R_2 \in \mathcal{R}(\Sigma^*).R_1 \cup R_2 \in \mathcal{R}(\Sigma^*)$
- $\forall R_1, R_2 \in \mathcal{R}(\Sigma^*).R_1 \cdot R_2 \in \mathcal{R}(\Sigma^*)$
- $\forall R \in \mathcal{R}(\Sigma^*).R^* \in \mathcal{R}(\Sigma^*)$

Expressions régulières

Une *expression régulière* est une notation compacte et simplifiée pour représenter un langage régulier. Par exemple :

Expression régulière	**Langage régulier**	**Mots du langage**
$a \mid b$ ou $a + b$	$\{a, b\}$	a, b
ab^*a	$\{a\}\{b\}^*\{a\}$	aa, aba, $abba$ etc.
$(ab)^*$	$\{ab\}^*$	ε, ab, $abab$ etc.
$abba$	$\{abba\}$	$abba$

Expressions régulières (suite)

L'ensemble $\mathcal{E}(\Sigma)$ des expressions régulières sur un alphabet Σ est le plus petit ensemble vérifiant

- $\varnothing \in \mathcal{E}(\Sigma)$
- $\{\varepsilon\} \in \mathcal{E}(\Sigma)$
- $\forall a \in \Sigma.\{a\} \in \mathcal{E}(\Sigma)$
- $\forall e_1, e_2 \in \mathcal{E}(\Sigma).e_1 + e_2 \in \mathcal{E}(\Sigma)$
- $\forall e_1, e_2 \in \mathcal{E}(\Sigma).e_1 \cdot e_2 \in \mathcal{E}(\Sigma)$
- $\forall e \in \mathcal{E}(\Sigma).e^* \in \mathcal{E}(\Sigma)$

Des expressions régulières aux langages réguliers

Le passage des expressions régulières aux langages réguliers se fait simplement par la fonction $\mathrm{L} : \mathcal{E}(\Sigma) \to \mathcal{R}$ définie inductivement par

$$\begin{aligned}
\mathrm{L}(\epsilon) &= \varnothing \\
\mathrm{L}(a) &= \{a\} \\
\mathrm{L}(e_1 + e_2) &= \mathrm{L}(e_1) \cup \mathrm{L}(e_2) \\
\mathrm{L}(e_1 \cdot e_2) &= \mathrm{L}(e_1) \cdot \mathrm{L}(e_2) \\
\mathrm{L}(e^*) &= \mathrm{L}(e)^*
\end{aligned}$$

Extensions syntaxiques

Par commodité, les outils qui emploient les expressions régulières étendent la syntaxe donnée précédemment, sans augmenter la puissance d'expression. Par exemple :

- [abc] pour (a | b | c)
- [a-f] pour (a | b | c | d | e | f)
- [^abc] pour le complémentaire de (a | b | c) dans l'ensemble des caractères
- a ? pour a | ϵ
- _ ou . pour n'importe quel caractère.

Remarque

- Le symbole + n'est pas employé dans son sens de disjonction mais de répétition non vide.

Exemples d'expressions régulières étendues

- **Entiers décimaux**

  [0-9]+

- **Entiers hexadécimaux**

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

- **Nombres à la Pascal**

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

- **Sources OCaml**

```
bash$ ls *.ml{,[ily]}
```

Application à la production d'analyseurs lexicaux

On spécifie chaque sorte de lexème par une expression régulière, comme par exemple

- les mots-clés **let**, **in** etc.
- les variables [a-z]+ [a-zA-Z0-9_]*
- les entiers [0-9]+
- les symboles : ( ) + * = etc.

mais aussi le texte à oublier :

- les espaces ('␣' | '\n' | '\t')
- et les commentaires.

Aucun lexème n'est alors associé à ces expressions régulières.

Application à la production d'analyseurs lexicaux (suite)

Un logiciel prend une telle spécification et produit un programme qui implante l'analyseur lexical correspondant dans le langage source de l'application. L'analyseur est alors compilé normalement et son code objet (cible) est lié au reste de l'application. Il prend un flux de caractères et tente d'y reconnaître un lexème. S'il réussit, il renvoie celui-ci et se déplace d'autant dans le flux, sinon il signale une erreur ou un flux vide.

En langage C, les générateurs d'analyseurs lexicaux connus sont flex et lex (dans la distribution Red Hat Linux ce dernier est en fait un lien symbolique vers le premier). En Java, il y a jlex et javacc, par exemple. Pour un catalogue, cf. `http://catalog.compilertools.net/` et, autour du forum comp.compilers, cf. `http://compilers.iecc.com/`

ocamllex

Le générateur d'analyseurs lexicaux du système OCaml est ocamllex.
Les expressions régulières définissant les lexèmes ont une forme habituelle, mais les caractères sont entourés par des apostrophes (conventions de OCaml), p.ex. ['a'-'z']+ ['a'-'z' 'A'-'Z' '0'-'9' '_']* au lieu de [a-z]+ [a-zA-Z0-9_]*
Le type OCaml représentant les lexèmes n'est généralement pas défini dans la spécification (qui possède une extension `.mll`). Par exemple, ce type peut être

type *token* = INT **of** *int* | IDENT **of** *string* | TRUE | FALSE
| PLUS | MINUS | TIMES | SLASH | EQUAL | ARROW
| LPAR | RPAR LET | IN | REC
| FUN | IF | THEN | ELSE | AND | OR | NOT | EOF

L'expérience recommande d'associer la fin de fichier à un lexème (ici EOF), en particulier en conjonction avec un analyseur syntaxique.

Spécification d'analyseurs lexicaux avec ocamllex

Une spécification d'analyseur lexical pour ocamllex a la forme :

{ *Code OCaml optionnel en prologue* }
let r = *regexp*
...
rule $entrée_1$ = **parse**
  $regexp_{1,1}$ { *Code OCaml, dit action* }
| ...
| $regexp_{1,n}$ { *Code OCaml, dit action* }
and $entrée_2$ = **parse**
  ...
and ...
{ *Code OCaml optionnel en épilogue* }

Un exemple de spécification pour ocamllex

```
{ open Parser
  exception Illegal_char of string }
let ident = ['a'-'z'] ['_' 'A'-'Z' 'a'-'z' '0'-'9']*
rule token = parse
       [' ' '\n' '\t' '\r'] { token lexbuf }
     | "let"                { LET }
     | "rec"                { REC }
     | "="                  { EQUAL }
     ...
     | ident                { IDENT (Lexing.lexeme lexbuf)}
     | ['0'-'9']+           { INT (int_of_string (Lexing.lexeme lexbuf)) }
     | eof                  { EOF }
     | _                    { raise (Illegal_char (Lexing.lexeme lexbuf)) }
```

Un exemple de spécification pour ocamllex (suite)

- Le prologue ouvre le module Parser car celui-ci contient la définition du type token dont les constructeurs sont appliqués dans les actions (LET, REC etc.). C'est le style d'organisation quand on utilise conjointement un analyseur syntaxique produit par ocamlyacc (c'est la même configuration en langage C si on utilise lex avec yacc). Si on spécifie un analyseur lexical autonome (ne serait-ce que pour faire du test unitaire), on aurait alors probablement un module Token contenant la définition des lexèmes.
- On déclare les exceptions lexicales dans le prologue, ici simplement Illegal_char, qui doivent être filtrées au niveau du pilote de l'application.
- Une expression régulière nommée ident est définie, ainsi qu'une unique entrée token. Dans les actions, *les entrées sont des fonctions* dont le premier argument est toujours le flux de caractères entrant, toujours nommé lexbuf.

Un exemple de spécification pour ocamllex (suite et fin)

- Le module standard Lexing contient un certain nombre de fonctions qui servent à manipuler le flux de caractères. Par exemple Lexing.lexeme prend le flux et retourne le lexème qui a été reconnu par l'expression régulière *associée à l'action*.
- Notez l'appel récursif à token lorsque l'on veut ignorer certains caractères. Cela fonctionne car dans l'action, les caractères reconnus par l'expression régulière associée ont été ôté du flux.
- Il existe une pseudo-expression régulière eof qui sert à filtrer la fin de fichier. Il est recommandé de s'en servir pour produire un pseudo-lexème EOF, car les comportements implicites des applications vis-à-vis des fins de fichier peuvent varier d'un système d'exploitation à l'autre.
- Il existe une pseudo-expression régulière _ qui filtre n'importe quel caractère. *L'ordre des expressions est significatif*, donc cette expression devrait être la dernière.

Mise en œuvre de ocamllex

- Si la spécification ocamllex a pour nom lexer.mll, alors la compilation se fait en deux temps :

  1. `ocamllex lexer.mll`, qui produit soit une erreur soit lexer.ml, puis
  2. `ocamlc -c lexer.ml`, qui produit soit une erreur soit lexer.cmo et lexer.cmi (ce dernier, en l'absence de lexer.mli).

  En théorie, les actions associées aux expressions régulières ne sont pas tenues de renvoyer un lexème, car le programmeur est libre et peut, par exemple, écrire un préprocesseur, c.-à-d. une réécriture de fichiers, plutôt qu'un analyseur lexical.
- Pour créer un flux entrant de caractères à partir de l'entrée standard il faut **let** char_flow = Lexing.from_channel (stdin) **in** ...

Sous le capot

Le fichier OCaml résultant de la spécification a la forme

Prologue ...
let rec *entrée*$_1$ = **fun** lexbuf $\rightarrow$
    ... **match** ... **with**
            ... $\rightarrow$ *action*
        | ...
        | ... $\rightarrow$ *action*
and *entrée*$_2$ = **fun** lexbuf $\rightarrow$
    ...
and ...
Épilogue

où `lexbuf` est de type Lexing.lexbuf

Analyse des commentaires sur une ligne

Les commentaires sont reconnus durant l'analyse lexicale mais rien n'en est fait. Certains analyseurs analysent le contenu des commentaires et signalent donc des erreurs *à l'intérieur* de ceux-ci (ce qui peut être gênant si on y place des méta-données).
Le type le plus simple de commentaires est celui de C++ qui porte sur une ligne.

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

L'expression régulière reconnaît l'ouverture du commentaire, puis laisse passer tout caractère différent d'une fin de ligne et termine par une fin de ligne optionnelle (on suppose que le système d'exploitation est Unix et qu'une fin de ligne peut terminer le fichier).

Analyse des commentaires en blocs non imbriqués

L'entrée token reconnaît l'ouverture du commentaire, et *son action* appelle l'entrée supplémentaire in_comment qui saute tous les caractères jusqu'à la fermeture du bloc et signale une erreur si celle-ci manque (commentaire ouvert). Quand le bloc est fermé, puisqu'un commentaire ne produit pas de lexème, il faut faire un appel récursif à token pour en renvoyer un.

```
{ ... exception Open_comment }
rule token = parse
        ...
      | "/*" { in_comment lexbuf }
and in_comment = parse
        "*/" { token lexbuf }
      | eof  { raise Open_comment }
      | _    { in_comment lexbuf }
```

Analyse des commentaires en blocs imbriqués

Les commentaires du langage C peuvent être imbriqués pour isoler temporairement une partie du code qui est déjà commentée.

S'ils n'étaient pas imbriqués, on aurait pu écrire une seule expression régulière (nous ne l'avons pas fait pour des raisons de lisibilité et pour signaler facilement une absence de fermeture). Dans le cas imbriqué il n'existe pas une telle expression *pour des raisons théoriques*. On dit que les langages réguliers ne peuvent être bien parenthèsés.

L'idée est que les expressions régulières ne peuvent « garder la mémoire » du degré d'imbrication courant. Pour y parvenir on se sert donc de l'expressivité du code *des actions*. **Ainsi c'est abusivement que l'on réduit l'analyse lexicale aux seuls langages réguliers, spécifiés par des expressions régulières.**

Analyse des commentaires en blocs imbriqués (suite)

La technique consiste à modifier l'entrée in_comment de sorte que les actions soient *des fonctions dont l'argument est la profondeur d'imbrication courante.*

```
rule token = parse
        ...
      | "/*" { in_comment lexbuf 1 }
and in_comment = parse
        "*/" { fun depth → if depth = 1 then token lexbuf
                           else in_comment lexbuf (depth-1) }
      | "/*" { fun depth → in_comment lexbuf (depth+1) }
      | eof  { raise Open_comment }
      | _    { in_comment lexbuf }
```

Notez que in_comment lexbuf *est équivalent à* **fun** depth → in_comment lexbuf depth *et que* **fun** depth → raise Open_comment *serait moins efficace.*

Automates finis et expressions régulières

Les générateurs d'analyseurs lexicaux doivent combiner les expressions régulières de la spécification et les traduire vers du code source. Pour cela, elles sont d'abord traduites dans un formalisme de même expressivité, mais plus intuitif : les *automates finis*. Finalement, l'automate résultant des traitements du générateur est compilé en code source.

Commençons par présenter un cas particulier d'automate fini, dit *déterministe* (AFD). Un AFD $\mathcal{A}$ est un quintuplet $(\Sigma, \mathcal{Q}, \delta, q_0, F)$ où

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

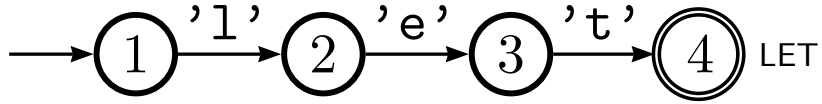
Automates finis déterministes

- Dans ce contexte, on parle aussi d'*étiquette* pour les éléments de Σ.
- Une transition est un triplet sur $\mathcal{Q} \times \Sigma \times \mathcal{Q}$
- On peut étendre δ sur $\mathcal{Q} \times \Sigma^* \to \mathcal{Q}$ par $\begin{cases} \delta(q, \epsilon) = q \\ \delta(q, aw) = \delta(\delta(q, a), w) \end{cases}$
- Le langage $\mathrm{L}(\mathcal{A})$ *reconnu* par l'automate $\mathcal{A}$ 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.
- On pourrait considérer qu'il y a plusieurs états initiaux possibles au lieu d'un seul, mais cela n'apporterait rien quant à l'analyse lexicale (qui est une application particulière de la théorie des automates finis).
- Un automate est *complet* si pour tout état q et toute étiquette a, $\delta(q, a)$ est défini.

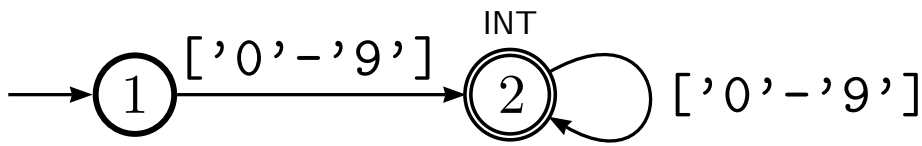
Exemples d'automates

Les automates permettent de reconnaître les lexèmes.

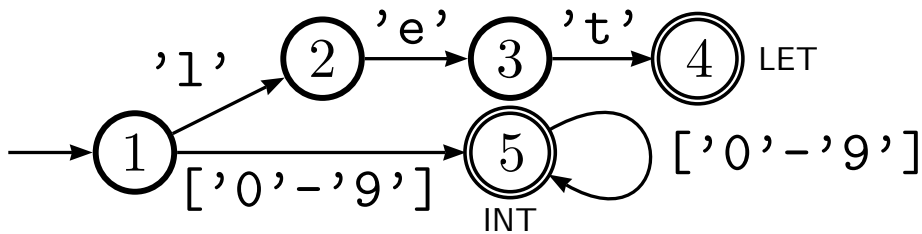
- les mots-clés :



- les entiers :



- l'un ou l'autre :



Si un état final (double cerclage) est atteint à partir de l'état initial (flèche entrante), un lexème est identifié (ici LET ou INT).

Analyse lexicale avec des automates

L'analyseur lexical considère deux informations :

- l'état courant dans l'automate spécifié,
- le caractère en tête du flux entrant.

Puis

- s'il existe une transition pour le caractère dans l'automate, alors
  - il est retiré du flux (et jeté) ;
  - l'état courant devient celui indiqué par la transition ;
  - on recommence à considérer les nouveaux état et caractère.
- s'il n'y a pas de transition (état bloquant), alors
  - si l'état courant est final alors le lexème associé est émis.
  - sinon il y a erreur (caractère illégal).

Ambiguïtés lexicales

Les problèmes qui peuvent se poser sont :

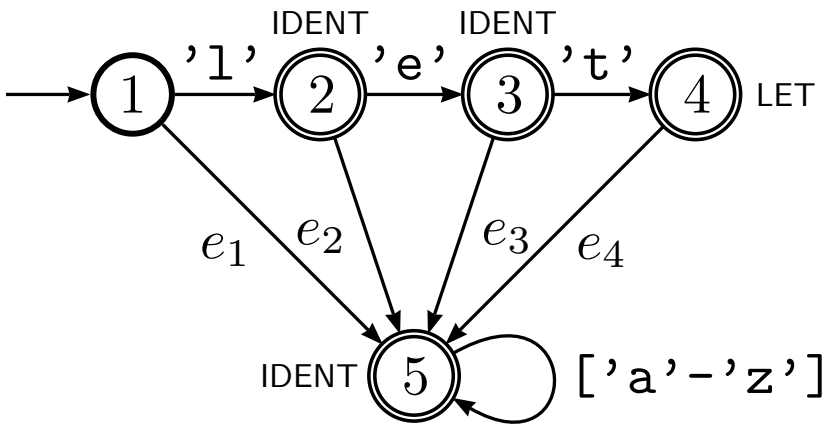
- la chaîne "let" pourrait être reconnue comme une variable et non tel mot-clé ;
- la chaîne "letrec" pourrait être reconnue comme la liste de lexèmes [LET ; IDENT "rec"] ou [LET ; REC] ou [IDENT "letrec"] etc.

La solution générale consiste à établir des règles de priorité :

- lorsque plusieurs lexèmes sont des préfixes possibles, retenir le plus long ;
- sinon suivre l'ordre de définition des sortes de lexèmes (p.ex. dans la spécification page 114 l'expression régulière "let" est écrite *avant* ident).

Ainsi la phrase `let letrec = 3 in 1 + funny` est reconnue comme la liste [LET ; IDENT "letrec" ; EQUAL ; INT 3 ; IN ; INT 1 ; PLUS ; IDENT "funny"].

Réalisation de la règle du lexème le plus long



e_1 = ['a'-'k' 'n'-'z']

e_2 = ['a'-'d' 'f'-'z']

e_3 = ['a'-'s' 'u'-'z']

e_4 = ['a'-'z]

Pour réaliser la règle du lexème le plus long, il faut ajouter une structure : une file de caractères (initialement vide) et reprendre l'algorithme page 126. Lorsque l'état courant est final et qu'une transition est possible, au lieu de jeter le caractère correspondant, il faut le conserver dans la file jusqu'à un état bloquant. Si cet état est final on renvoie le lexème associé, **sinon on retourne le lexème du dernier état final rencontré, les caractères de la file sont remis dans le flux entrant et on revient à l'état initial**.

Automates finis non-déterministes asynchrones

Pour construire de façon intuitive des automates à partir d'expressions régulières et leur appliquer des transformations, nous avons besoin d'une classe d'automates un peu différente : les automates finis *non-déterministes asynchrones* (AFNA). Un AFNA diffère sur deux points des AFD :

- On étend les étiquettes par le mot vide ε (on parle de transitions spontanées), c.-à-d. qu'on remplace Σ par $\Sigma \cup \{\varepsilon\}$: c'est l'asynchronisme.
- Il peut y avoir plusieurs transitions de même étiquette à partir d'un même état. Techniquement, δ est alors une *relation* (ternaire) sur $\mathcal{Q} \times (\Sigma \cup \{\varepsilon\}) \times \mathcal{Q}$ et non plus une fonction partielle : c'est le non-déterminisme.