École Normale Supérieure

Langages de programmation et compilation

Jean-Christophe Filliâtre

analyse lexicale

l'analyse lexicale est le découpage du texte source en « mots »

de même que dans les langues naturelles, ce découpage en mots facilite le travail de la phase suivante, l'analyse syntaxique

ces mots sont appelés des **lexèmes** (*tokens*)

analyse lexicale : exemple

source = suite de caractères

```
fun x -> (* ma fonction *)
    x+1
```

↓

analyse lexicale

↓

suite de lexèmes

⋮

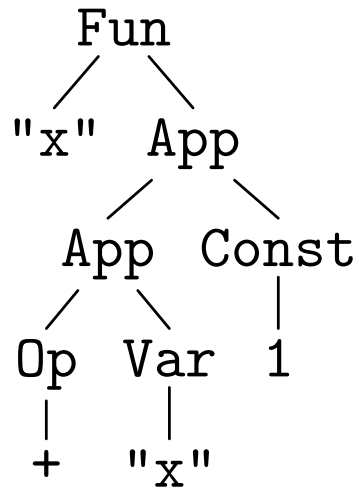
↓

analyse syntaxique

(prochain cours)

↓

syntaxe abstraite



les blancs

les blancs (espace, retour chariot, tabulation, etc.) jouent un rôle dans l'analyse lexicale ; ils permettent notamment de séparer deux lexèmes

ainsi `funx` est compris comme un seul lexème (l'identificateur `funx`) et `fun x` est compris comme deux lexèmes (le mot clé `fun` et l'identificateur `x`)

de nombreux blancs sont néanmoins inutiles (comme dans `x + 1`) et simplement ignorés

les blancs n'apparaissent pas dans le flot de lexèmes renvoyé

les blancs

les conventions diffèrent selon les langages,
et certains des caractères « blancs » peuvent être significatifs

exemples :

- les tabulations pour `make`
- retours chariot et espaces de début de ligne lorsque l'indentation détermine la structure des blocs (Python, Haskell, etc.)
- retours chariot parfois transformés en points-virgules (Go, Julia, etc.)

les commentaires jouent le rôle de blancs

```
fun(* et hop *)x -> x + (* j'ajoute un *) 1
```

ici le commentaire `(* et hop *)` joue le rôle d'un blanc significatif (sépare deux lexèmes) et le commentaire `(* j'ajoute un *)` celui d'un blanc inutile

note : les commentaires sont parfois exploités par d'autres outils (`ocamldoc`, `javadoc`, etc.), qui les traitent alors différemment dans leur propre analyse lexicale

```
val length : 'a list -> int
  (** Return the length (number of elements) of ...
```

pour réaliser l'analyse lexicale, on va utiliser

- des **expressions régulières** pour décrire les lexèmes
- des **automates finis** pour les reconnaître

on exploite notamment la capacité à construire automatiquement un automate fini déterministe reconnaissant le langage décrit par une expression régulière

expressions régulières

on se donne un alphabet A

$$
\begin{array}{rcll}
r & ::= & \emptyset & \text{langage vide} \\
 & | & \epsilon & \text{mot vide} \\
 & | & a & \text{caractère } a \in A \\
 & | & r\,r & \text{concaténation} \\
 & | & r \mid r & \text{alternative} \\
 & | & r\star & \text{étoile}
\end{array}
$$

conventions : l'étoile a la priorité la plus forte, puis la concaténation, puis enfin l'alternative

le **langage** défini par l'expression régulière r est l'ensemble de mots $L(r)$ défini par

$$
\begin{aligned}
L(\emptyset) &= \emptyset \\
L(\epsilon) &= \{\epsilon\} \\
L(a) &= \{a\} \\
L(r_1\, r_2) &= \{w_1 w_2 \mid w_1 \in L(r_1) \wedge w_2 \in L(r_2)\} \\
L(r_1 \mid r_2) &= L(r_1) \cup L(r_2) \\
L(r\star) &= \bigcup_{n \geq 0} L(r^n) \quad \text{où } r^0 = \epsilon,\ r^{n+1} = r\, r^n
\end{aligned}
$$

sur l'alphabet $\{a, b\}$

- mots de trois lettres

$$(a|b)(a|b)(a|b)$$

- mots se terminant par un a

$$(a|b) \star a$$

- mots alternant a et b

$$(b|\epsilon)(ab) \star (a|\epsilon)$$

constantes entières décimales, éventuellement précédées de zéros

$$(0|1|2|3|4|5|6|7|8|9)\,(0|1|2|3|4|5|6|7|8|9)\star$$

identificateurs composés de lettres, de chiffres et du souligné, et commençant par une lettre

$$(a|b|\ldots|z|A|B|\ldots|Z)\,(a|b|\ldots|z|A|B|\ldots|Z|\_|0|1|\ldots|9)\star$$

constantes flottantes d'OCaml (`3.14` `2.` `1e-12` `6.02e23` etc.)

$$d\,d\star\,(.d\star\,|\,(\epsilon\,|\,.d\star)(e|E)\,(\epsilon|+|-)d\,d\star)$$

avec $d = 0|1|\ldots|9$

les commentaires de la forme `(* ... *)`, mais **non imbriqués**, peuvent également être définis de cette manière

$$\boxed{(}\boxed{*}\ \left(\boxed{*}\star r_1 \mid r_2\right)\star\ \boxed{*}\boxed{*}\star\ \boxed{)}$$

où r_1 = tous les caractères sauf $*$ et)
et r_2 = tous les caractères sauf $*$

commentaires imbriqués

les expressions régulières ne sont pas assez expressives pour définir les commentaires **imbriqués** (le langage des mots bien parenthésés n'est pas régulier)

on expliquera plus loin comment contourner ce problème

on se donne le type

```
type expreg =
  | Vide
  | Epsilon
  | Caractere of char
  | Union     of expreg * expreg
  | Produit   of expreg * expreg
  | Etoile    of expreg
```

écrire une fonction

```
val reconnait: expreg -> char list -> bool
```

le plus simplement possible

soit r une expression régulière ; est-ce que le mot vide ϵ appartient à $L(r)$?

```
let rec null = function
  | Vide | Caractere _ -> false
  | Epsilon | Etoile _ -> true
  | Union   (r1, r2)   -> null r1 || null r2
  | Produit (r1, r2)   -> null r1 && null r2
```

résidu (dérivée de Brzozowski)

pour une expression régulière r et un caractère c on définit

$$Res(r, c) = \{w \mid cw \in L(r)\}$$

```
let rec residu r c = match r with
  | Vide | Epsilon ->
      Vide
  | Caractere d ->
      if c = d then Epsilon else Vide
  | Union (r1, r2) ->
      Union (residu r1 c, residu r2 c)
  | Produit (r1, r2) ->
      let r' = Produit (residu r1 c, r2) in
      if null r1 then Union (r', residu r2 c) else r'
  | Etoile r1 ->
      Produit (residu r1 c, r)
```

fin de la récréation

```
let rec reconnait r = function
  | []     -> null r
  | c :: w -> reconnait (residu r c) w
```

(20 lignes)

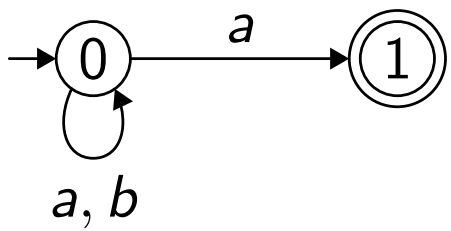
automates finis

Définition

Un automate fini sur un alphabet A est un quadruplet (Q, T, I, F) où

- *Q est un ensemble fini d'états*
- *$T \subseteq Q \times A \times Q$ un ensemble de transitions*
- *$I \subseteq Q$ un ensemble d'états initiaux*
- *$F \subseteq Q$ un ensemble d'états terminaux*

exemple : $Q = \{0, 1\}$, $T = \{(0, a, 0), (0, b, 0), (0, a, 1)\}$, $I = \{0\}$, $F = \{1\}$



un mot $a_1 a_2 \dots a_n \in A^\star$ est **reconnu** par un automate (Q, T, I, F) ssi

$$s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \cdots s_{n-1} \xrightarrow{a_n} s_n$$

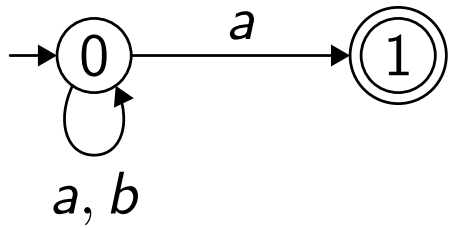
avec $s_0 \in I$, $(s_{i-1}, a_i, s_i) \in T$ pour tout i, et $s_n \in F$

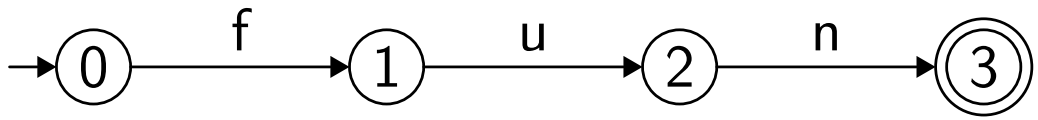
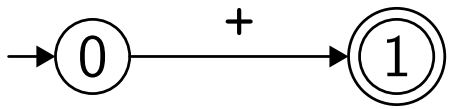
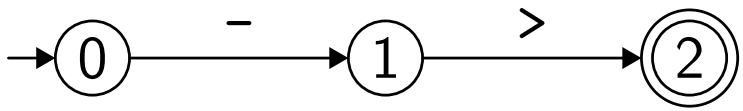
le **langage** défini par un automate est l'ensemble des mots reconnus

Théorème (de Kleene)

Les expressions régulières et les automates finis définissent les mêmes langages.

$(a|b) \star a$

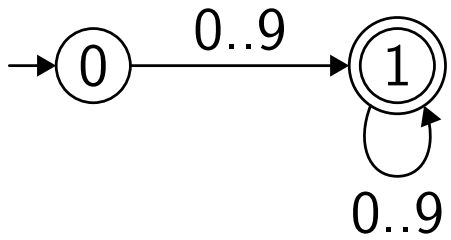


lexème	expression régulière	automate
mot clé `fun`	*f u n*	
symbole +	+	
symbole ->	- >	

expression régulière

$$(0|1|2|3|4|5|6|7|8|9)\,(0|1|2|3|4|5|6|7|8|9)\star$$

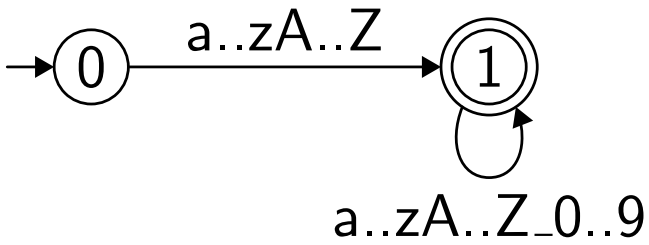
automate



expression régulière

$$(a|b|\ldots|z|A|B|\ldots|Z)\,(a|b|\ldots|z|A|B|\ldots|Z|\_|0|1|\ldots|9)\star$$

automate

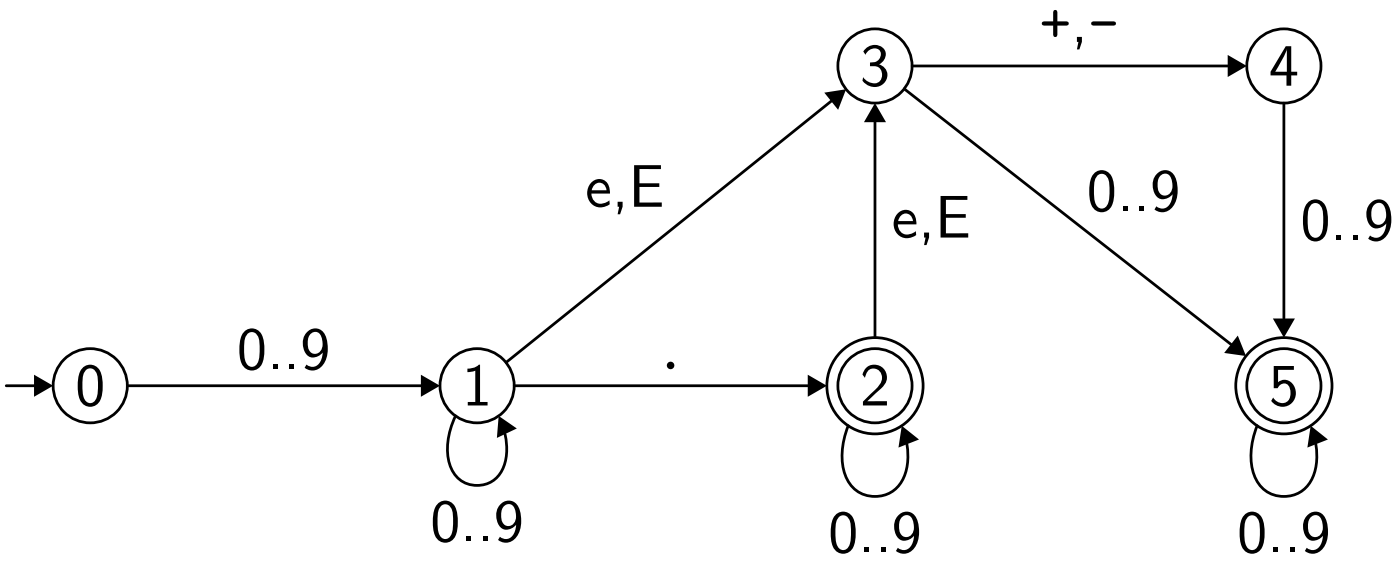


expression régulière

$$d\,d\star\,(.d\star\,|\,(\epsilon\,|\,.d\star)(e|E)\,(\epsilon|+|-)d\,d\star)$$

où $d = 0|1|\ldots|9$

automate

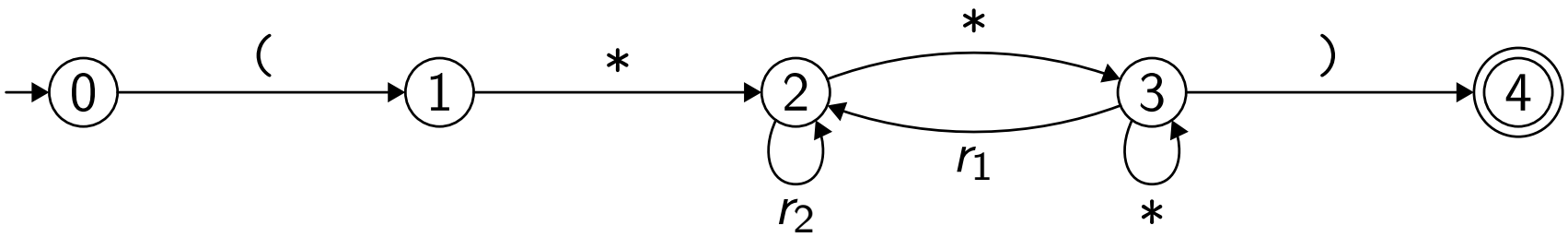


expression régulière

$$\boxed{(}\ \boxed{*}\ \left(\boxed{*}\star r_1 \mid r_2\right)\star\ \boxed{*}\ \boxed{*}\star\ \boxed{)}$$

où r_1 = tous les caractères sauf $*$ et)
et r_2 = tous les caractères sauf $*$

automate fini



analyseur lexical

un **analyseur lexical** est un automate fini pour la ≪ réunion ≫ de toutes les expressions régulières définissant les lexèmes

le fonctionnement de l'analyseur lexical, cependant, est différent de la simple reconnaissance d'un mot par un automate, car

- il faut décomposer un mot (le source) en une **suite** de mots reconnus
- il peut y avoir des **ambiguïtés**
- il faut construire les lexèmes (les états finaux contiennent des **actions**)

le mot `funx` est reconnu par l'expression régulière des identificateurs, mais contient un préfixe reconnu par une autre expression régulière (`fun`)

$\Rightarrow$ on fait le choix de reconnaître le lexème le plus long possible

le mot `fun` est reconnu par l'expression régulière du mot clé `fun` mais aussi par celle des identificateurs

$\Rightarrow$ on classe les lexèmes par ordre de priorité

avec les trois expressions régulières

$$a, \quad ab, \quad bc$$

un analyseur lexical va **échouer** sur l'entrée

$$abc$$

(ab est reconnu, comme plus long, puis échec sur c)

pourtant le mot abc appartient au langage $(a|ab|bc)\star$

l'analyseur lexical doit donc mémoriser le dernier état final rencontré, le cas échéant

lorsqu'il n'y a plus de transition possible, de deux choses l'une :

- aucune position finale n'a été mémorisée $\Rightarrow$ échec de l'analyse lexicale
- on a lu le préfixe wv de l'entrée, avec w le lexème reconnu par le dernier état final rencontré $\Rightarrow$ on renvoie le lexème w, et l'analyse redémarre avec v préfixé au reste de l'entrée

programmons un analyseur lexical

on introduit un type d'automates finis déterministes

```
type automaton = {
  initial : int;                  (* état = entier *)
  trans   : int Cmap.t array;     (* état -> char -> état *)
  action  : action array;         (* état -> action *)
}
```

avec

```
type action =
  | NoAction          (* pas d'action = pas un état final *)
  | Action of string  (* nature du lexème *)
```

et

```
module Cmap = Map.Make(Char)
```

la table de transitions est pleine pour les états (tableau) et creuse pour les caractères (AVL)

on se donne

```
let transition autom s c =
  try Cmap.find c autom.trans.(s) with Not_found -> -1
```

l'objectif est d'écrire une fonction `analyzer` qui prend un automate et une chaîne à analyser, et renvoie une fonction de calcul du prochain lexème

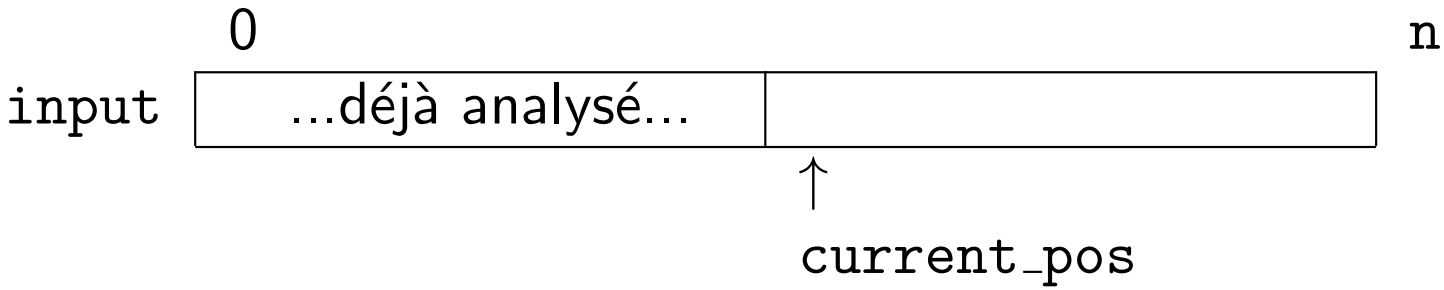
soit

```
val analyzer :
  automaton -> string -> (unit -> string * string)
```

note : on pourrait également renvoyer la liste des lexèmes, mais on adopte ici la méthodologie qui est utilisée en pratique dans l'interaction entre analyse lexicale et analyse syntaxique

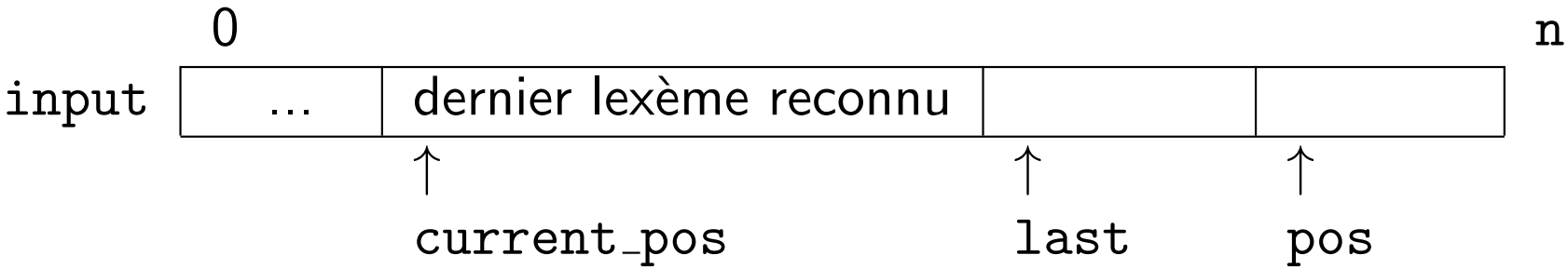
on mémorise la position courante dans l'entrée à l'aide d'une référence `current_pos`

```
let analyzer autom input =
  let n = String.length input in
  let current_pos = ref 0 in (* position courante *)
  fun () ->
    ...
```



note : l'application partielle de `analyzer` est cruciale

```
let analyzer autom input =
  let n = String.length input in
  let current_pos = ref 0 in
  fun () ->
    let rec scan last state pos =
      (* on s'apprête à examiner le caractère pos *)
      ...
    in
    scan None autom.initial !current_pos
```



on détermine alors si une transition est possible

```
let rec scan last state pos =
  let state' =
    if pos = n then -1
    else transition autom state input.[pos]
  in
  if state' >= 0 then
    (* une transition vers state' *) ...
  else
    (* pas de transition possible *) ...
```

si oui, on met à jour `last`, le cas échéant, et on appelle `scan` récursivement sur `state'`

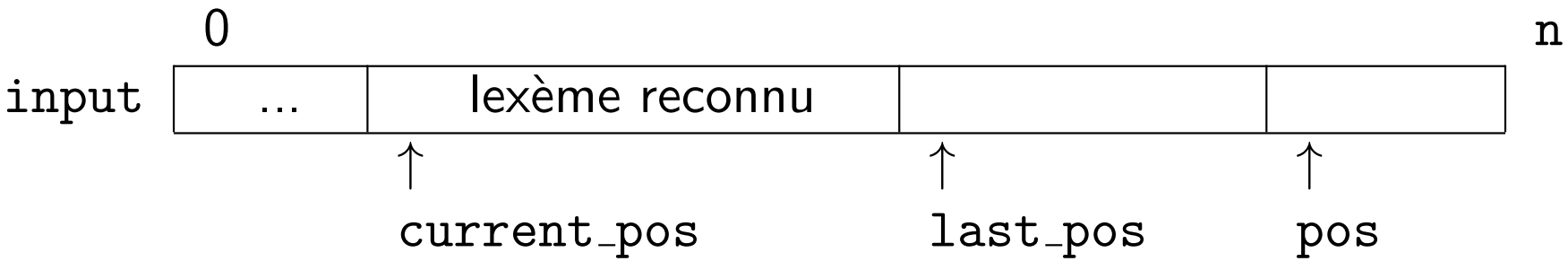
```
    if state' >= 0 then
      let last = match autom.action.(state') with
        | NoAction -> last
        | Action a -> Some (pos + 1, a)
      in
      scan last state' (pos + 1)
```

`last` a la forme `Some (p, a)` où

- `p` est la position qui suit le lexème reconnu
- `a` est l'action (le type du lexème)

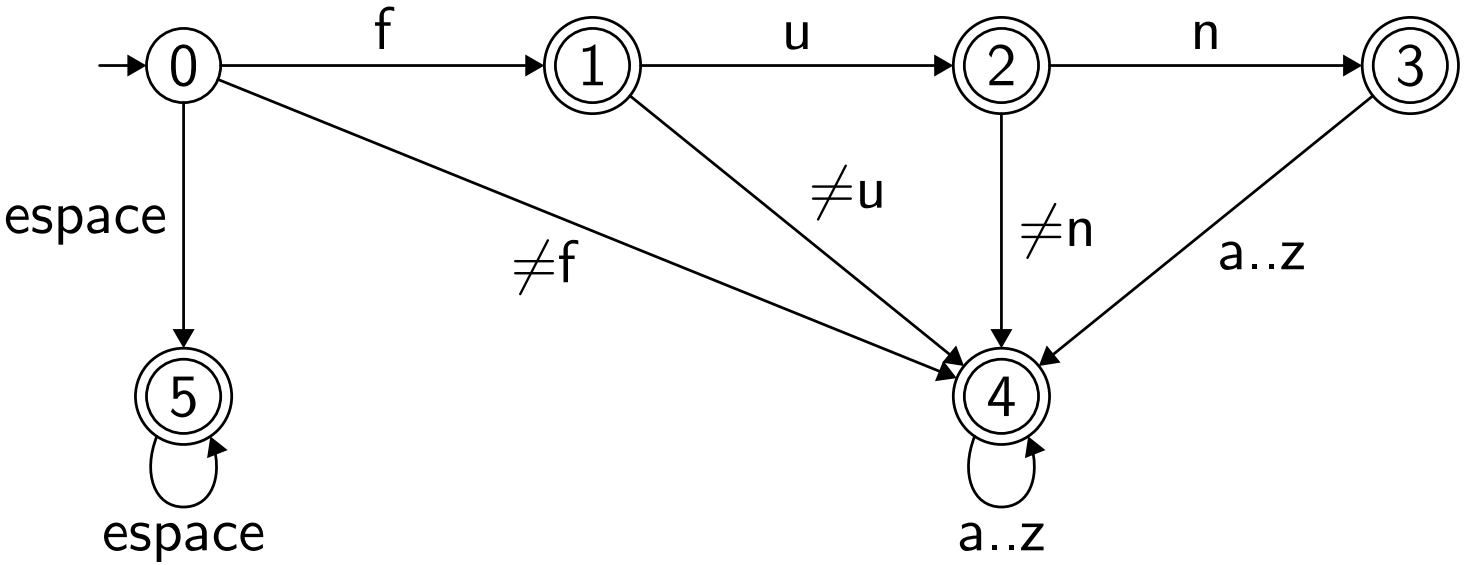
si au contraire aucune transition n'est possible, on examine `last` pour déterminer le résultat

```
      else match last with
       | None ->
           failwith "échec"
       | Some (last_pos, action) ->
           let start = !current_pos in
           current_pos := last_pos;
           action, String.sub input start (last_pos - start)
```



testons avec

- un mot clé : *fun*
- des identificateurs : $(a..z)(a..z)\star$
- des blancs : *espace espace*$\star$



```
let autom = {
  initial = 0;
  trans = [| ... |];
  action = [|
    (*0*) NoAction;
    (*1*) Action "ident";
    (*2*) Action "ident";
    (*3*) Action "keyword";
    (*4*) Action "ident";
    (*5*) Action "space";
  |]
}
```

```
# let next_token = analyzer autom "fun funx";;
# next_token ();;

- : string * string = ("keyword", "fun")

# next_token ();;

- : string * string = ("space", " ")

# next_token ();;

- : string * string = ("ident", "funx")

# next_token ();;

Exception: Failure "échec".
```

il y a bien sûr d'autres possibilités pour programmer un analyseur lexical

exemple : *n* fonctions mutuellement récursives, une par état de l'automate

```
let rec state0 pos = match input.[pos] with
  | 'f'                  -> state1 (pos + 1)
  | 'a'..'e' | 'g'..'z' -> state4 (pos + 1)
  | ' '                  -> state5 (pos + 1)
  | _                    -> failwith "échec"

and state1 pos = match input.[pos] with
  | ...
```

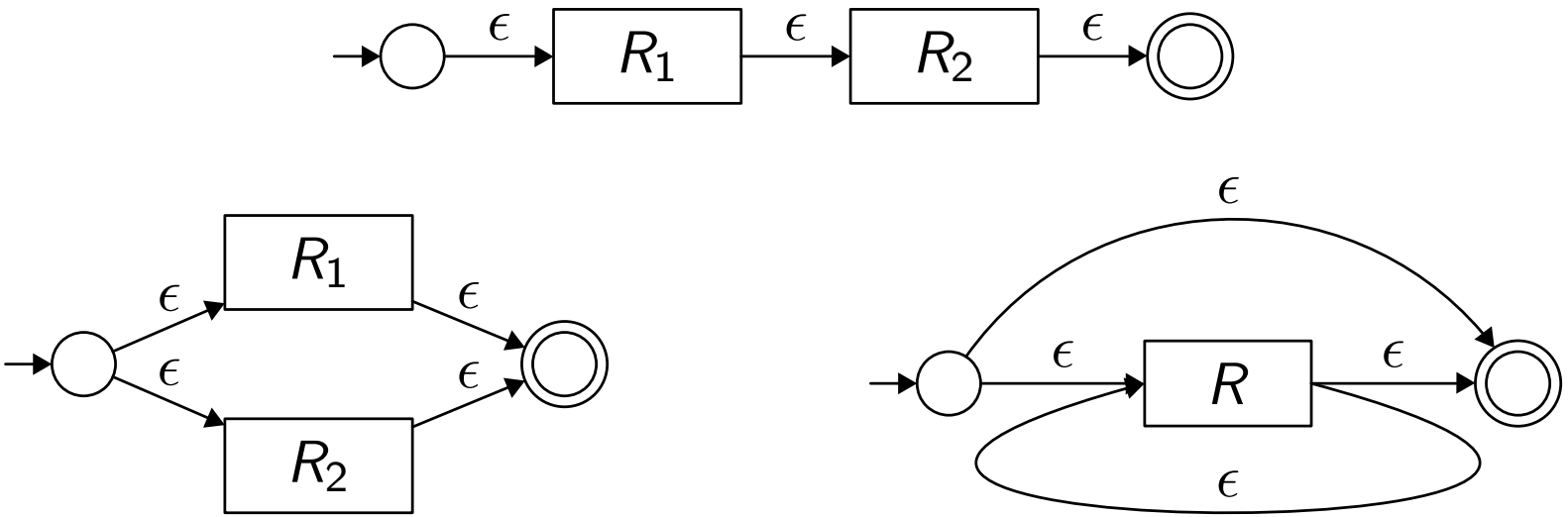
(voir TD de cette semaine)

en pratique, on dispose d'outils qui construisent les analyseurs lexicaux à partir de leur description par des expressions régulières et des actions

c'est la grande famille de `lex` : `lex`, `flex`, `jflex`, `ocamllex`, etc.

construction de l'automate

on peut construire l'automate fini correspondant à une expression régulière en passant par l'intermédiaire d'un automate non déterministe



on peut ensuite déterminiser, voire minimiser cet automate
(cf. un cours de langages formels)

mais...

construction directe (Berry & Sethi, 1986)

. . . on peut aussi construire directement un automate déterministe

idée de départ : on peut mettre en correspondance les lettres d'un mot reconnu et celles apparaissant dans l'expression régulière

exemple : le mot *aabaab* est reconnu par $(a|b) \star a(a|b)$, de la manière suivante

$$
\begin{array}{ll}
a & (\mathbf{a}|b) \star a(a|b) \\
a & (\mathbf{a}|b) \star a(a|b) \\
b & (a|\mathbf{b}) \star a(a|b) \\
a & (\mathbf{a}|b) \star a(a|b) \\
a & (a|b) \star \mathbf{a}(a|b) \\
b & (a|b) \star a(a|\mathbf{b})
\end{array}
$$

distinguons les différentes lettres de l'expression régulière :

$$(a_1|b_1) \star a_2(a_3|b_2)$$

on va construire un automate dont les états sont des ensembles de lettres

l'état s reconnaît les mots dont la première lettre appartient à s

exemple : l'état $\{a_1, a_2, b_1\}$ reconnaît les mots dont la première lettre est soit un a correspondant à a_1 ou a_2, soit un b correspondant à b_1

pour construire les transitions de s_1 à s_2 il faut déterminer les lettres qui peuvent apparaître après une autre dans un mot reconnu (*follow*)

exemple : toujours avec $r = (a_1|b_1) \star a_2(a_3|b_2)$, on a

$$follow(a_1, r) = \{a_1, a_2, b_1\}$$

pour calculer *follow*, on a besoin de calculer les premières (resp. dernières) lettres possibles d'un mot reconnu (*first*, resp. *last*)

exemple : toujours avec $r = (a_1|b_1) \star a_2(a_3|b_2)$, on a

$$\begin{aligned} first(r) &= \{a_1, a_2, b_1\} \\ last(r) &= \{a_3, b_2\} \end{aligned}$$

pour calculer *first* et *last*, on a besoin d'une dernière notion : est-ce que le mot vide appartient au langage reconnu ? (*null*)

$$
\begin{aligned}
null(\emptyset) &= \texttt{false} \\
null(\epsilon) &= \texttt{true} \\
null(a) &= \texttt{false} \\
null(r_1\, r_2) &= null(r_1) \wedge null(r_2) \\
null(r_1 \mid r_2) &= null(r_1) \vee null(r_2) \\
null(r\star) &= \texttt{true}
\end{aligned}
$$

(déjà fait plus haut en OCaml)

on peut alors définir *first*

$$
\begin{aligned}
first(\emptyset) &= \emptyset \\
first(\epsilon) &= \emptyset \\
first(a) &= \{a\} \\
first(r_1\, r_2) &= first(r_1) \cup first(r_2) \quad \text{si } null(r_1) \\
&= first(r_1) \quad \text{sinon} \\
first(r_1 \mid r_2) &= first(r_1) \cup first(r_2) \\
first(r\star) &= first(r)
\end{aligned}
$$

la définition de *last* est similaire

puis *follow*

$$
\begin{aligned}
follow(c, \emptyset) &= \emptyset \\
follow(c, \epsilon) &= \emptyset \\
follow(c, a) &= \emptyset \\
follow(c, r_1\, r_2) &= follow(c, r_1) \cup follow(c, r_2) \cup first(r_2) \quad \text{si } c \in last(r_1) \\
&= follow(c, r_1) \cup follow(c, r_2) \quad \text{sinon} \\
follow(c, r_1 \mid r_2) &= follow(c, r_1) \cup follow(c, r_2) \\
follow(c, r\star) &= follow(c, r) \cup first(r) \quad \text{si } c \in last(r) \\
&= follow(c, r) \quad \text{sinon}
\end{aligned}
$$

enfin, on peut construire l'automate pour l'expression régulière r

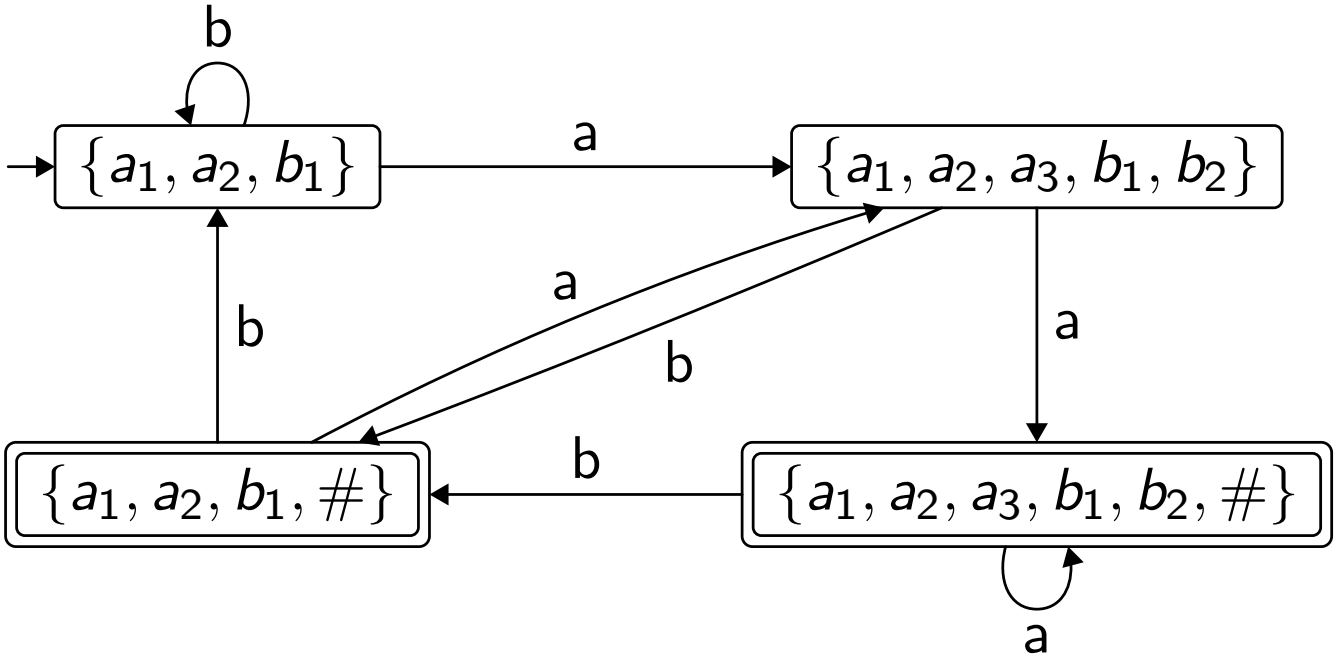
on commence par ajouter un caractère $\#$ à la fin de r
l'algorithme est alors le suivant :

1. l'état initial est l'ensemble $first(r\#)$

2. tant qu'il existe un état s dont il faut calculer les transitions
   pour chaque caractère c de l'alphabet
   soit s' l'état $\bigcup_{c_i \in s} follow(c_i, r\#)$
   ajouter la transition $s \xrightarrow{c} s'$

3. les états acceptant sont les états contenant $\#$

pour $(a|b) \star a(a|b)$ on obtient



TD de cette semaine : programmer cette construction

référence : Gérard Berry & Ravi Sethi,
From regular expressions to deterministic automata, 1986

l'outil ocamllex

un fichier `ocamllex` porte le suffixe `.mll` et a la forme suivante

```
{
  ... code OCaml arbitraire ...
}
rule f1 = parse
| regexp1 { action1 }
| regexp2 { action2 }
| ...
and f2 = parse
  ...
and fn = parse
  ...
{
  ... code OCaml arbitraire ...
}
```

on compile le fichier `lexer.mll` avec `ocamllex`

```
% ocamllex lexer.mll
```

ce qui produit un fichier OCaml `lexer.ml` qui définit une fonction pour chaque analyseur `f1`, ..., `fn` :

```
val f1 : Lexing.lexbuf -> type1
val f2 : Lexing.lexbuf -> type2
...
val fn : Lexing.lexbuf -> typen
```

le type `Lexing.lexbuf`

le type `Lexing.lexbuf` est celui de la structure de données qui contient l'état d'un analyseur lexical

le module `Lexing` de la bibliothèque standard fournit plusieurs moyens de construire une valeur de ce type, dont

```
val from_channel : in_channel -> lexbuf
```

```
val from_string : string -> lexbuf
```

les expressions régulières d'ocamllex

`_`	n'importe quel caractère
`'a'`	le caractère `'a'`
`"foobar"`	la chaîne `"foobar"` (en particulier ϵ = `""`)
`[`*caractères*`]`	ensemble de caractères (par ex. `['a'-'z' 'A'-'Z']`)
`[^`*caractères*`]`	complémentaire (par ex. `[^ '"']`)
r_1 `|` r_2	l'alternative
$r_1\, r_2$	la concaténation
r `*`	l'étoile
r `+`	une ou plusieurs répétitions de r ($\stackrel{\text{def}}{=} r\, r\star$)
r `?`	une ou zéro occurrence de r ($\stackrel{\text{def}}{=} \epsilon \mid r$)
`eof`	la fin de l'entrée

identificateurs

```
  | ['a'-'z' 'A'-'Z'] ['a'-'z' 'A'-'Z' '_' '0'-'9']* { ... }
```

constantes entières

```
  | ['0'-'9']+ { ...}
```

constantes flottantes

```
  | ['0'-'9']+
      ( '.' ['0'-'9']*
      | ('.' ['0'-'9']*)? ['e' 'E'] ['+' '-']?  ['0'-'9']+ )
    { ... }
```

on peut définir des raccourcis pour des expressions régulières

```
let letter = ['a'-'z' 'A'-'Z']
let digit = ['0'-'9']
let decimals = '.' digit*
let exponent = ['e' 'E'] ['+' '-']?  digit+

rule token = parse
  | letter (letter | digit | '_')*          { ... }

  | digit+                                  { ... }

  | digit+ (decimals | decimals? exponent)  { ... }
```

pour les analyseurs définis avec le mot clé `parse`, la règle du plus long lexème reconnu s'applique

à longueur égale, c'est la règle qui apparaît en premier qui l'emporte

```
| "fun"              { Tfun }
| ['a'-'z']+ as s    { Tident s }
```

pour le plus court, il suffit d'utiliser `shortest` à la place de `parse`

```
rule scan = shortest
  | regexp1 { action1 }
  | regexp2 { action2 }
  ...
```

on peut nommer la chaîne reconnue, ou des sous-chaînes reconnues par des sous-expressions régulières, à l'aide de la construction `as`

```
| ['a'-'z']+ as s  { ... }

| (['+' '-']? as sign) (['0'-'9']+ as num)  { ... }
```

dans une action, il est possible de rappeler récursivement l'analyseur lexical, ou l'un des autres analyseurs simultanément définis

le tampon d'analyse lexical doit être passé en argument ;
il est contenu dans une variable appelée `lexbuf`

il est ainsi très facile de traiter les blancs :

```
rule token = parse
  | [' ' '\t' '\n']+ { token lexbuf }
  | ...
```

pour traiter les commentaires, on peut utiliser une expression régulière

... ou un analyseur dédié :

```
rule token = parse
  | "(*"  { comment lexbuf }
  | ...

and comment = parse
  | "*)"  { token lexbuf }
  | _     { comment lexbuf }
  | eof   { failwith "commentaire non terminé" }
```

avantage :
on traite correctement l'erreur liée à un commentaire non fermé

autre intérêt : on traite facilement les **commentaires imbriqués**

avec un compteur

```
rule token = parse
  | "(*"  { comment 1 lexbuf; token lexbuf }
  | ...

and comment level = parse
  | "*)"  { if level > 1 then comment (level - 1) lexbuf }
  | "(*"  { comment (level + 1) lexbuf }
  | _     { comment level lexbuf }
  | eof   { failwith "commentaire non terminé" }
```

... ou même sans compteur !

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

and comment = parse
  | "*)"  { () }
  | "(*"  { comment lexbuf; comment lexbuf }
  | _     { comment lexbuf }
  | eof   { failwith "commentaire non terminé" }
```

note : on a donc dépassé la puissance des expressions régulières

prenons l'exemple d'un micro langage avec des entiers, des variables, des constructions `fun` *x* `->` *e* et *e* `+` *e* et des commentaires de la forme `(*...*)`

on se donne un type OCaml pour les lexèmes

```
type token =
  | Tident of string
  | Tconst of int
  | Tfun
  | Tarrow
  | Tplus
  | Teof
```

un exemple complet

```
rule token = parse
  | [' ' '\t' '\n']+ { token lexbuf }
  | "(*"             { comment lexbuf }
  | "fun"            { Tfun }
  | ['a'-'z']+ as s  { Tident s }
  | ['0'-'9']+ as s  { Tconst (int_of_string s) }
  | "+"              { Tplus }
  | "->"             { Tarrow }
  | _ as c           { failwith ("caractère illégal : " ^
                                 String.make 1 c) }
  | eof              { Teof }

and comment = parse
  | "*)"  { token lexbuf }
  | _     { comment lexbuf }
  | eof   { failwith "commentaire non terminé" }
```

quatre ≪ règles ≫ à ne pas oublier quand on écrit un analyseur lexical

1. traiter les **blancs**
2. les règles **les plus prioritaires en premier** (par ex. mots clés avant identificateurs)
3. signaler les **erreurs lexicales** (caractères illégaux, mais aussi commentaires ou chaînes non fermés, séquences d'échappement illégales, etc.)
4. traiter la **fin de l'entrée** (`eof`)

par défaut, `ocamllex` encode l'automate dans une **table**, qui est interprétée à l'exécution

l'option `-ml` permet de produire du code OCaml pur, où l'automate est encodé par des fonctions ; ce n'est pas recommandé en pratique cependant

même en utilisant une table, l'automate peut prendre beaucoup de place, en particulier s'il y a de nombreux mots clés dans le langage

il est préférable d'utiliser une seule expression régulière pour les identificateurs et les mots clés, et de les séparer ensuite grâce à une table des mots clés

```
{
  let keywords = Hashtbl.create 97
  let () = List.iter (fun (s,t) -> Hashtbl.add keywords s t)
    ["and", AND; "as", AS; "assert", ASSERT;
     "begin", BEGIN; ...
}
rule token = parse
| ident as s
  { try Hashtbl.find keywords s
    with Not_found -> IDENT s }
```

(in)sensibilité à la casse

si on souhaite un analyseur lexical qui ne soit pas sensible à la casse, surtout ne pas écrire

```
| ("a"|"A") ("n"|"N") ("d"|"D")
    { AND }
| ("a"|"A") ("s"|"S")
    { AS }
| ("a"|"A") ("s"|"S") ("s"|"S") ("e"|"E") ("r"|"R") ("t"|"T")
    { ASSERT }
| ...
```

mais plutôt

```
rule token = parse
| ident as s
  { let s = String.lowercase s in
    try Hashtbl.find keywords s
    with Not_found -> IDENT s }
```

pour compiler (ou recompiler) les modules OCaml, il faut déterminer les **dépendances** entre ces modules et donc s'assurer de la fabrication préalable du code OCaml avec `ocamllex`

si on utilise `dune`, on indique comme ceci la présence d'un fichier `lexer.mll` qui doit être traité avec `ocamllex`

```
(ocamllex
 (modules lexer))

(executable
 (name minilang)
 ...
```

(cf le code fourni au TD 3 par exemple)

applications d'ocamllex

(à d'autres fins que l'analyse lexicale)

applications d'`ocamllex`

l'utilisation d'`ocamllex` n'est pas limitée à l'analyse lexicale

dès que l'on souhaite analyser un texte (chaîne, fichier, flux) sur la base d'expressions régulières, `ocamllex` est un outil de choix

en particulier pour écrire des **filtres**, *i.e.* des programmes traduisant un langage dans un autre par des modifications locales et relativement simples

exemple 1 : réunir plusieurs lignes vides consécutives en une seule

aussi simple que

```
rule scan = parse
  | '\n' '\n'+ { print_string "\n\n"; scan lexbuf }
  | _ as c     { print_char c; scan lexbuf }
  | eof        { () }

{ let () = scan (Lexing.from_channel stdin) }
```

on fabrique un exécutable avec

```
% ocamllex mbl.mll
% ocamlopt -o mbl mbl.ml
```

on l'utilise ainsi

```
% ./mbl < infile > outfile
```

exemple 2 : compter les occurrences d'un mot dans un texte

```
{
  let word = Sys.argv.(1)
}
rule scan c = parse
  | ['a'-'z' 'A'-'Z']+ as w
     { scan (if word = w then c+1 else c) lexbuf }
  | _
     { scan c lexbuf }
  | eof
     { c }
{
  let c = open_in Sys.argv.(2)
  let n = scan 0 (Lexing.from_channel c)
  let () = Printf.printf "%d occurrence(s)\n" n
}
```

exemple 3 : un petit traducteur OCaml vers HTML,
pour embellir le source mis en ligne

objectif

- usage : `caml2html file.ml`, qui produit `file.ml.html`
- mots clés en vert, commentaires en rouge
- numéroter les lignes
- le tout en moins de 100 lignes de code

on écrit tout dans un unique fichier `caml2html.mll`

on commence par vérifier la ligne de commande

```
{
  let () =
    if Array.length Sys.argv <> 2
    || not (Sys.file_exists Sys.argv.(1)) then begin
      Printf.eprintf "usage: caml2html file\n";
      exit 1
    end
```

puis on ouvre le fichier HTML en écriture et on écrit dedans avec `fprintf`

```
  let file = Sys.argv.(1)
  let cout = open_out (file ^ ".html")
  let print s = Printf.fprintf cout s
```

on écrit le début du fichier HTML avec comme titre le nom du fichier
on utilise la balise HTML <pre> pour formatter le code

```
let () =
  print "<html><head><title>%s</title></head><body>\n<pre>" file
```

on introduit une fonction pour numéroter chaque ligne,
et on l'invoque immédiatement pour la première ligne

```
let count = ref 0
let newline () = incr count; print "\n%3d: " !count
let () = newline ()
```

on définit une table des mots clés (comme pour un analyseur lexical)

```
  let is_keyword =
    let ht = Hashtbl.create 97 in
    List.iter
      (fun s -> Hashtbl.add ht s ())
      [ "and"; "as"; "assert"; "asr"; "begin"; "class";
        ... ];
    fun s -> Hashtbl.mem ht s
}
```

on introduit une expression régulière pour les identificateurs

```
let ident =
  ['A'-'Z' 'a'-'z' '_'] ['A'-'Z' 'a'-'z' '0'-'9' '_']*
```

on peut attaquer l'analyseur proprement dit

pour un identificateur, on teste s'il s'agit d'un mot clé

```
rule scan = parse
  | ident as s
      { if is_keyword s then begin
          print "<font color=\"green\">%s</font>" s
        end else
          print "%s" s;
        scan lexbuf }
```

à chaque saut de ligne, on invoque la fonction `newline` :

```
  | "\n"
      { newline (); scan lexbuf }
```

pour un commentaire, on change de couleur (rouge) et on invoque un autre analyseur `comment` ; à son retour, on repasse dans la couleur par défaut et on reprend l'analyse de `scan`

```
| "(*"
    { print "<font color=\"990000\">(*";
      comment lexbuf;
      print "</font>";
      scan lexbuf }
```

tout autre caractère est imprimé tel quel

```
| _ as s { print "%s" s; scan lexbuf }
```

quand c'est fini, c'est fini !

```
| eof    { () }
```

pour les commentaires, il suffit de ne pas oublier `newline` :

```
and comment = parse
  | "(*"   { print "(*"; comment lexbuf; comment lexbuf }
  | "*)"   { print "*)" }
  | eof    { () }
  | "\n"   { newline (); comment lexbuf }
  | _ as c { print "%c" c; comment lexbuf }
```

on termine avec l'application de `scan` sur le fichier d'entrée

```
{
  let () =
    scan (Lexing.from_channel (open_in file));
    print "</pre>\n</body></html>\n";
    close_out cout
}
```

c'est presque correct :

- le caractère < est significatif en HTML et doit être écrit `<`
- de même il faut échapper & avec `&`

- une chaîne OCaml peut contenir `(*`
  (ce propre code en contient !)

rectifions

on ajoute une règle pour < et un analyseur pour les chaînes

```
| "<"    { print "&lt;"; scan lexbuf }
| "&"    { print "&amp;"; scan lexbuf }
| '"'    { print "\""; string lexbuf; scan lexbuf }
```

il faut faire attention au caractère '"', où " ne marque pas le début d'une chaîne

```
| "'\"'"
| _ as s { print "%s" s; scan lexbuf }
```

on applique le même traitement dans les commentaires (petite coquetterie d'OCaml : on peut commenter du code contenant "*)")

```
  | '"'     { print "\""; string lexbuf; comment lexbuf }
  | "'\"'"
  | _ as s { print "%s" s; comment lexbuf }
```

enfin les chaînes sont traitées par l'analyseur `string`, sans oublier les séquences d'échappement (telles que \" par exemple)

```
and string = parse
  | '"'     { print "\"" }
  | "<"     { print "&lt;"; string lexbuf }
  | "&"     { print "&amp;"; string lexbuf }
  | "\\" _
  | _ as s { print "%s" s; string lexbuf }
```

maintenant, ça fonctionne correctement
(un bon test consiste à essayer sur `caml2html.mll` lui-même)

pour bien faire, il faudrait également convertir les tabulations de début de ligne (typiquement insérées par l'éditeur) en espaces

laissé en exercice...

exemple 4 : indentation automatique de programmes C

idée :

- à chaque accolade ouvrante, on augmente la marge
- à chaque accolade fermante, on la diminue
- à chaque retour chariot, on imprime la marge courante
- sans oublier de traiter les chaînes et les commentaires

on se donne de quoi maintenir la marge et l'afficher

```
{
  open Printf

  let margin = ref 0
  let print_margin () =
    printf "\n%s" (String.make (2 * !margin) ' ')
}
```

indentation automatique de programmes C

à chaque retour chariot, on ignore les espaces de début de ligne et on affiche la marge courante

```
let space = [' ' '\t']

rule scan = parse
  | '\n' space*
      { print_margin (); scan lexbuf }
```

les accolades modifient la marge courante

```
  | "{"
      { incr margin; printf "{"; scan lexbuf }
  | "}"
      { decr margin; printf "}"; scan lexbuf }
```

petit cas particulier : une accolade fermante en début de ligne doit diminuer la marge avant qu'elle ne soit affichée

```
  | '\n' space* "}"
      { decr margin; print_margin (); printf "}";
        scan lexbuf }
```

on n'oublie pas les chaînes de caractères

```
  | '"' ([^ '\\' '"'] | '\\' _)* '"' as s
      { printf "%s" s; scan lexbuf }
```

ni les commentaires de fin de ligne

```
  | "//" [^ '\n']* as s
      { printf "%s" s; scan lexbuf }
```

ni les commentaires de la forme /* ... */

```
  | "/*"
      { printf "/*"; comment lexbuf; scan lexbuf }
```

où

```
and comment = parse
  | "*/"
      { printf "*/" }
  | eof
      { () }
  | _ as c
      { printf "%c" c; comment lexbuf }
```

enfin, tout le reste est affiché tel quel, et `eof` termine l'analyse

```
rule scan = parse
  | ...
  | _ as c
      { printf "%c" c; scan lexbuf }
  | eof
      { () }
```

le programme principal tient en deux lignes

```
{
  let c = open_in Sys.argv.(1)
  let () = scan (Lexing.from_channel c); close_in c
}
```

ce n'est pas parfait, bien entendu

en particulier, un corps de boucle ou de conditionnelle réduit à une seule instruction ne sera pas indenté :

```
if (x==1)
    continue;
```

exercice : faire la même chose pour OCaml (c'est beaucoup plus difficile, et on pourra se limiter à quelques cas simples)

- les **expressions régulières** sont à la base de l'analyse lexicale
- le travail est grandement automatisé par des outils tels que `ocamllex`
- `ocamllex` est plus expressif que les expressions régulières, et peut être utilisé bien au delà de l'analyse lexicale

  (note : ces propres transparents sont réalisés avec un préprocesseur LaTeX écrit à l'aide d'`ocamllex`)

- TD 4
  - compilation expression régulière vers automate fini

- prochain cours
  - analyse syntaxique (1/2)