Analyse lexicale

I] INTRODUCTION

L'**analsye lexicale** est le découpage d'un texte en "mots". On parle de *lexèmes* (tokens).
Cette analyse facilite l'étape suivante : *analyse syntaxique*.

source = suite de caractères.
L'analyse lexicale la transforme en une suite de tokens
L'analyse syntaxique la transforme en une syntaxe abstraite (arbre, etc.)

Les *blancs* permettent de séparer les tokens. Ils sont supprimés lors de l'analyse lexicale

/! Dans certains langages, certains caractères blancs peuvent être significatifs :

- retours chariot et début de lignes lorsque l'indentation détermine la structure
  Ex. : Python et Haskell
- tabulations en make
- retours chariot transformés en points-virgules en Go ou Julia

Les commentaires jouent le rôle de blancs.

Note : les outils de documentation peuvent exploiter les commentaires

Pour réaliser l'analyse lexicale, on utilise deux outils :

- **regexp** pour définir les tok€ens
- **automates** finis pour les reconnaître

II] REGEXP

On se donne un alphabet A

```
let r =
  | ∅
  | epsilon (mot vide)
  | a    pour caractère a dans A
  | rr   concaténation
  | r|r  alternative
  | r*   étoile
```

Par exemple :

- Constantes décimales : `(1.9)+`
- Identificateurs : `(a.z|A.Z)(a.z|A.Z|_|0.9)*`

À partir du type `regexp`, voilà une fonction prenant un mot en argument et renvoyant si le mot appartient au langage
dénoté par la regexp

```
let rec null = function (* Détermine si le mot vide appartient au langage de la regexp *)
  | Vide | Caractere _ -> false
  | Epsilon | Etoile _ -> false
  | Union (r1, r2) -> null r1 || null r2
  | Produit (r1, r2) -> null r1 && null r2
;;
```

Le **résidu** d'une regexp r et d'un caractère c est :
`Res(r, c) = {w | cw appartient à 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 the Union (r', residu r2 c) else r'
    | Etoile r1 ->
      Produit (residu r1 c, r)
  ;;

  let rec reconnait r mot =
    match mot with
    | [] -> null r
    | c :: w -> reconnait (residu r c) w
;;
```

III] AUTOMATES FINIS

On se donne un alphabet (le même)
Un **automate** est un (Q, T, I, F)

- *Q* ensemble fini d'états
- *T* ensemble de transitions
- *I* ensemble d'états initiaux
- *F* ensemble d'états terminaux

Un mot est reconnu par un automate si blablabla.
Le langage défini par l'automate est l'ensemble des mots reconnus

Théorème de Kleene :

Les regexp et les automates finis définissent les **mêmes langages**.

IV] ANALYSEUR LEXICAL

On va se servir de ce qu'on vient de voir, mais qq *ambiguités* :

- "funx" est reconnu par la regexp des *identificateurs*
  mais contient un préfixe reconnu par la regexp de *"fun"*...
  --> On fait le choix de reconnaître le **token le plus long possible**
- "fun" est reconnu par les deux
  --> on classe les tokens par **ordre de priorité**

L'analyseur doit mémoriser le *dernier état final rencontré*.
Lorsqu'il n'y a plus de transition possible :

- Si aucune position finale mémorisée -> *échec*
- On a lu le préfixe wv de l'entrée, avec w token reconnu par le dernier état final rencontré :
  -> on renvoie **w** et on **redémarre** avec v préfixé au reste de l'entrée

```
type automaton = {
  initial : int;
  trans : int Cmap.t array;
  action : action array;
};;
```

avec

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

et

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

La table de transitions est pleine pour les états et creuse poru les caractères
On se donne une fonction :

```
let transition autom s c =
  try Cmap.dinc c automa.trans(s)
  with Not_Found -> -1
;;

let analyser autom input =
  let n = String.length input in
  let current_pos = ref 0 in (* position courante *)
  fun () ->
    let rec scan last state pos =
      let state' =
        if pos = n then -1
        else transition autom atate input.[pos]
      in
      if state' >= 0 then (* Transition possible *)
        let last = match autom.action.(state') with
          | NoAction -> last
          | Action a -> Some (pos + 1, a)
        in
        scan last state' (pos + 1)
      else    (* Pas de transition possible *)
        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)   (* On renvoie la nature et le text
      in
      scan None autom.initial !current_pos
;;
```

En pratique, on a des outils qui construisent les automates à partir des regexp
-> la famille lex

V] CONSTRUCTION DE L'AUTOMATE

Algorithme de Thompson :

On construit d'abord un **epsilon-automate fini non-déterministe**. Ensuite, on *détermine* et *minimise* cet automate.

Mais lex procède d'une autre manière :

Algorithme de Berry & Sethi :

Idée : on met en correspondance les lettres d'un mot reconnu avec celles apparaissant dans la regexp.

On *distingue* les différentes *occurences* d'une même letrte dans la regexp.

Puis on construit un **automate** donc les états sont les ensembles de telles lettres.

Pour construire les transitions, il faut déterminer les lettres qui peuvent apparaître après une autre dans un mot reconnu, la fonction `follow`

On se donne une fonction `first`, renvoyant l'ensemble des premières lettres des mots reconnus par la regexp
De même la fonction `last`.

```
let follow x r =
  match r with
  | ∅ -> ∅
  | epsilon -> ∅
  | a -> ∅
  | Produit (r1, r2) ->
    if c in last r1
      then (follow c r1) U (follow c r2) U (first r2)
    else
      (follow c r1) U (follow c r2)
  | Union (r1, r2) ->
    (follow c r1) U (follow c r2)
  | Etoile r1 ->
    if c in (last r)
      then (follow c r1) U (first c r1)
    else
      follow c r1
;;
```

Construction :
On ajoute un caractère spécial `#` à la fin de la regexp `r`.
On suit la procédure :

1. L'état initial est `first(r#)`
2. tant qu'il existe un *état* `s` dont il faut calculer les transitions et pour chaque caractère `c` :
    - soit `s'` l'état `Union_{ci dans s}{follow(ci, r#)}` (les états sont des ensembles de caractères)
    - Ajouter la transition `s -c> s'`
3. les états *acceptants* sont les états contenant #

VI] OCAMLLEX

C'est un outil d'**analyse lexicale** pour OCaml.

Fichiers : `.mll`

Forme générale :

```
(* Prélude *)
{
  (* Code OCaml arbitraire*)
}

(* Règles d'analyse lexicale *)
rule f1 = parse
| regexp1 {action1} (* Les actions sont du code OCaml arbitraire *)
| regexp2 {action2}
| ...
and f2 = parse

...
and fn = parse
...

{
  (* Code OCaml arbitraire *)
}
```

Une fois le `.mll` écrit, on le compile avec `ocamllex ***.mll`
Cela produit un fichier OCaml `***.ml` qui définit une **fonction** pour chaque analyseur `f1`, ...,

`fn` :

```
val f1 : Lexing.lexbuf -> type1
...
```

`Lexing.lexbuf` est un type de la **bibliothèque standard**.

Regexp :

- _ -> *n'importe quel caractère*
- 'a' le caractère a
- "foobar" -> la chaîne foobar
- [caractères] -> ensemble de caractères. ex : ['a'-'z' 'A'-'Z']
- [^caractères] -> complémentaire. ex : [^""]
- r1|r1 -> alternative
- r1r2 -> concaténation
- r* -> étoile
- r+ -> r(r*)
- r? -> r|"" (* ie. une ou zéro occurences de r *)
- eof -> *fin de l'entrée*

On peut définir des *raccourcis* :

```
let letter = ['a'-'z' 'A'-'Z'];;
```

Pour les analyseurs définis avec le mot-clé `parse`, c'est la règle du **plus long token reconnu** qui s'applique.
À *longueur égale*, c'est **la première règle qui apparaît** qui l'emporte

-> mettre "fun" avant *ident*

Si on remplace `parse` par `shortest`, on va avoir le **token le plus court**.

On peut *nommer la chaîne reconnue*, ou des sous-expressions, à l'aide de la construction `as`.

Dans une action, on peut appeler *récursivement* l'analyseur lexical.

Pour traiter les blancs :

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

Ou alors :

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

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

Encore mieux : gestion des commentaires imbriqués :

```
{
  level = ref 0;
}
rule token = parse
  | "(*" {level := 1; comment lexbuf; token lexbuf}
  | ...

and comment = parse
  | "*)" {decr level; if !level > 0 then comment lexbuf}
  | "(*" {incr level; comment lexbuf}
  | _ {comment lexbuf}
  | eof {failwith "Commentaire non terminé"}
```

Exemple sur un micro-langage :

On se donne des *commentaires* comme en OCaml, des entiers, variables, construcions `fun x -> e` et des `e+e`.

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

```
  def issou(a):
    return 2*a
```

javascript

```
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}
  | eof {Teof}
  | _ as c {failwith ("caractère illégal : " ^ (String.make 1 c))}

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

Par défaut, `ocamllex` encode l'automate dans une **table**, interprétée à l'exécution.
L'option `-ml` permet de produire du **code OCaml pur**, où l'automate est encodé par des fonctions.

Remarque :
Même en utilisant une table, l'automate peut prendre *beaucoup de place*.
Il est préférable d'utiliser *une seule expression* pour les **identificateurs** et les **mots-clé** puis de les séparer avec une *table de hachage*.

De même si on veut faire un langage **insensible à la casse**, il faut préférer faire du `String.lowercase` après avor match un identificateur.

/! Pour compiler, il faut déterminer les *dépendances* entre les modules.

VII] APPLICATIONS D'OCAMLLEX

Cet outil n'est pas limité à l'*analyse lexicale*.

On peut l'utiliser dès qu'on souhaite analyser un texte sur la base de **regexp**.
--> filtres

Exemples :

1. Réunir les lignes vides en une seule.

```
rule scan = parse
  | '\n' '\n'+ {print_string '\n\n'; scan lexbuf}
  | eof {()}
  | _ as c {print_char c; scan lexbuf}
{let () = scan (Lexing.from_channel stdin)}
```

2. Compter les occurences d'un mot dans un texte

```
{
  let word = Sys.argv.(1);
}
rule scan c = parse
  | ['a'-'z' 'A'-'Z']+ as w {scan (fi 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 occurence(s)\n" n
}
```

3. Traducteur OCaml vers HTML
   --> mots-clé en vert, commentaire en rouge
   --> numéroter les lignes

```
{
  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
  let file = Sys.argv.(1)
  let cout = open_out (file ^ ".html")
  let print s = Printf.fprintf cout s
  let () = print "<html><head><title>%s</title></head><body>\n<pre>" file
  let count = ref 0
  let newline () = incr count; print "\n%3d: " !count
  let () = newline ()
  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
}
let ident =
  ['A'-'Z' 'a'-'z' '_'] ['A'-'Z' 'a'-'z' '0'-'9' '_']*
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 }
  | "\n"
    { newline (); scan lexbuf }
  | "(*"
    { print "<font color=\"990000\">(*";
    comment lexbuf;
    print "</font>";
    scan lexbuf }
  | "<" { print "&lt;"; scan lexbuf }
  | "&" { print "&amp;"; scan lexbuf }
  | '"' { print "\""; string lexbuf; scan lexbuf }
  | "'\"'"
  | '"' { print "\""; string lexbuf; comment lexbuf }
```

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

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

and string = parse
  | '"' { print "\"" }
  | "<" { print "&lt;"; string lexbuf }
  | "&" { print "&amp;"; string lexbuf }
  | "\\" _
  | _ as s { print "%s" s; string lexbuf }
  {
    let () =
      scan (Lexing.from_channel (open_in file));
      print "</pre>\n</body></html>\n";
      close_out cout
  }
```

4. Indentation automatique de C

```
{
  open Printf
  let margin = ref 0
  let print_margin () =
    printf "\n%s" (String.make (2 * !margin) ' ')
}
let space = [' ' '\t']
rule scan = parse
  | '\n' space* { print_margin (); scan lexbuf }
  | "{" { incr margin; printf "{"; scan lexbuf }
  | "}" { decr margin; printf "}"; scan lexbuf }
  | '\n' space* "}" { decr margin; print_margin (); printf "}"; scan lexbuf }
  | '"' ([^ '\\' '"'] | '\\' _)* '"' as s { printf "%s" s; scan lexbuf }
  | "//" [^ '\n']* as s { printf "%s" s; scan lexbuf }
  | "/*" { printf "/*"; comment lexbuf; scan lexbuf }
  | _ as c { printf "%c" c; scan lexbuf }
  | eof { () }

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

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