Licence STS, semestre 4 — 2009–2010
Langages et Génie Logiciel (Info 223) — 2 avril 2010
`http://www.lri.fr/~paulin/LGL`

Cours 9 : Analyse lexicale en Caml

Le générateur d'analyseur lexicaux `ocamllex` permet de programmer rapidement des outils puissants pour analyser des textes.

- Traitement de différentes *unités lexicales* :
  suite de caractères qui appartiennent à des langages réguliers L_i décrits par des expressions régulières r_i.
- À chaque expression régulière r_i est associée une action a_i :
  du code à exécuter qui peut dépendre de la chaîne $s \in L_i$ reconnue.
- Le programme que l'on veut construire se comporte ainsi : il reconnait une chaîne s au début du texte t qui est dans un des langages L_i et il exécute alors l'action associée a_i.

Expressions régulières

La syntaxe autorisée par ocamllex pour les expressions régulières :

regexp ::= `"`*chaîne*`"`
| *regexp* `?`
| *regexp* `+`
| `[` *caractères* `]`
| `[^` *caractères* `]`
| `eof`
| `_`
| `(` *regexp* `)`
| *regexp* `as` *ident*

caractères ::= (`'`*car*`'` | `'`*car*`'-'`*car*`'`)+

Précédences : plus hautes `+` et `*`, ensuite `?`, puis la concaténation puis le choix `|`.

Syntaxe des fichiers

Un fichier `ocamllex` se présente de la manière suivante :

```
  { header }
let ident = regexp ...
rule entrypoint [arg1... argn] =
  parse regexp { action }
      | ...
      | regexp { action }
and entrypoint [arg1... argn] =
  parse ...
and ...
{ trailer }
```

La commande

`ocamllex` *file.mll*

produit un fichier *file.ml* qui doit ensuite être compilé par ocaml.
Elle affiche la taille de l'automate et de la table stockant les transitions.

- Chaque *entrypoint* définit une fonction ocaml.
- Ces fonctions peuvent être paramétrées et ont toujours un argument supplémentaire `lexbuf` qui correspond à la suite de caractères à traiter.
- Ces fonctions peuvent aussi être appelées récursivement dans les actions.
- Ces fonctions cherchent à reconnaitre l'unité lexicale **la plus longue**.
- Si l'unité correspond à plusieurs expressions régulières, on choisit l'action associée à la première.

Exemple

```
{ let chars = ref 0 and words = ref 0 and lines = ref 0
  let inword = ref false }
let sep = [ ' ' | '\t' ]
rule count = parse
  | '\n' {incr chars;incr lines;inword:= false;count lexbuf }
  | sep  {incr chars;inword:= false;count lexbuf }
  | _    {incr chars;
  if not !inword then (incr words; inword := true);
    count lexbuf }
  | eof  { () }
{ let process_file f =
    let c = open_in f in count (Lexing.from_channel c);
    Printf.printf "%8d %8d %8d %s\n" !lines !words !chars f;
    close_in c
  let _ = if Array.length Sys.argv <> 2
          then Printf.printf "wc requires 1 argument"
          else process_file Sys.argv.(1)
}
```

Cas particuliers

On peut traiter avec `ocamllex` des expressions qui ne sont pas régulières comme les commentaires imbriqués.

Solution à l'aide d'un compteur

```
{ let com = ref 0 }
rule nexttoken = parse
   "(*"   {com:=1; comment lexbuf; nexttoken lexbuf}
and comment = parse
   "*)"   {if !com > 1 then begin decr com; comment lexbuf end}
 | "(*"   {incr com; comment lexbuf}
 | eof    {raise Error "commentaire non terminé"}
 | _      {comment lexbuf}
```

Solution récursive

```
rule nexttoken = parse
   "(*"   {comment lexbuf; nexttoken lexbuf}
and comment = parse
   "*)"   {}
 | "(*"   {comment lexbuf; comment lexbuf}
 | eof    {raise Error "commentaire non terminé"}
 | _      {comment lexbuf}
```

Reconnaissance de mots-clés Les expressions régulières ne sont pas toujours la meilleure solution. Par exemple distinguer des mots-clés à l'aide d'automates est peu efficace. On utilise plutôt une table de hachage pour stocker les mots-clés et une expression régulière qui reconnait tous les identificateurs.