

Analyse Lexicale

- rappels sur les langages rationnels
- exemple OcamlLex
- extensions pour analyseurs lexicaux:
 - rappel du dernier état final
 - longest match, first defined
- Utilisation de OcamlLex
- Exemples

Rappels sur les langages rationnels

expressions rationnelles (regular expressions) :

$\epsilon, +, *, *$

automates finis non déterministes :

(Q, I, T, \mathcal{F})

automates finis déterministes :

(Q, I, T, \mathcal{F})

avec \mathcal{F} fonctionnelle et I singleton

déterminisation :

construction de l'ensemble puissance

minimalisation : construction de Moore

propriétés : lemme de l'étoile; fermeture par complémentation, intersection, union; décidabilité du langage vide, fini ou infini

Extensions pour analyseurs lexicaux

Un analyseur lexical doit séparer un flot de caractères en une série de "tokens" (unités lexicales), chacune définie par une expression régulière.

Cela est légèrement différent du problème de la reconnaissance d'un langage rationnel:

- on recherche le plus long¹ **prefixe** de l'entrée qui corresponde à une des définitions d'unités lexicales, et on doit pouvoir retourner à la fois ce **prefixe** et identifier quelle définition a été trouvée, donc...
- on augmente l'automate fini avec une information supplémentaire² sur les états finaux
- on maintient deux variables supplémentaires: `dernierEtatFinal` et `positionEntreeAuDernierEtatFinal` qu'il faut maintenir à jour.
- on retourne `dernierEtatFinal` quand l'automate se trouve bloqué.

Utilisation de OcamlLex

Un fichier source pour OcamlLex a extension `.mll` et a cette structure:

```
{ prologue }
let ident = regexp ...
rule etatinitial =
  parse regexp { action }
  | ...
  | regexp { action }
and etatinitial =
  parse ...
and ...
{ epilogue }
```

Les commentaires sont délimités par `(* et *)` comme en OCaml.

Prologue et epilogue sont des sections *optionnelles* qui contiennent du code Ocaml arbitraire

Typiquement, prologue définit des fonctions nécessaires dans les actions, alors que epilogue est plus souvent utilisé, comme dans l'exemple précédent, pour réaliser un programme "stand-alone").

Exemple OcamlLex

Voici un exemple de fichier de définition OcamlLex:

```
exception Eof;;
let count = ref 0;;
```

¹on n'échoue pas s'il reste qqe chose après

²la règle qui corresponde à l'état; en cas de ambiguïté, on choisit la première en ordre de définition

```

}
let alphanum = [^ ' ' '\t' '\n']
let mot = alphanum alphanum*

rule token = parse
  mot    {count := !count + 1;}
  | _    {}
  | eof  {raise Eof}
{
let _ = try let lexbuf = Lexing.from_channel stdin in
  while true do token lexbuf; done
  with Eof -> print_int !count; print_newline(); exit 0
}

```

Utilisation de OcamlLex: expressions rationnelles

Un expression de base est l'une des 6 formes suivantes

'char'	le caractère dénoté par char .
-	un caractère quelconque (<i>mais pas eof</i>).
eof	la fin du flot de caractères.
"string"	une chaîne de caractères.
[ens-char]	filtre tout caractère dans l'ensemble ens-char, qui peut être une constante 'c', un intervalle 'c1' - 'c2' ou l'union de deux ensembles (ex: 'a' - 'z' 'A' - 'Z')
[^ens-char]	tout caractère qui n'est pas dans ens-char

Utilisation de OcamlLex: expressions rationnelles

Les expressions de base peuvent être composées avec les opérateurs suivants

exp *	étoile de Kleene
exp +	1 ou plus occurrences de exp
exp ?	1 ou 0 occurrences de exp
exp1 exp2	une occurrence de exp1 ou une de exp2
exp1 exp2	une occurrence de exp1, puis une de exp2
(exp)	la même chose que exp
ident	l'expression abrégée de nom ident cf. suivant

Précédences:

* et + précèdent ?, qui précède la concaténation, et enfin |

Utilisation de OcamlLex: abréviations

Il est possible de donner un nom à des expressions rationnelle qui apparaissent souvent, en écrivant:

```
let ident = regexp
```

entre le prologue et les règles.

Bien entendu, ces définitions *ne peuvent pas être récursives*.

Utilisation de OcamlLex: états initiaux

Dans `rule token = parse`, `token` est un identificateur Ocaml valide (qui commence par une minuscule), et définit un état initial de l'automate, que l'on peut invoquer sur un flot de caractères. Par exemple:

```
let lexbuf = Lexing.from_channel stdin in
token lexbuf;
```

Construit un flot à partir de l'entrée standard et appelle l'automate sur le flot avec l'état initial "token".

La possibilité d'avoir plusieurs états initiaux est fort pratique pour "changer de mode", ce qui permet de traiter par exemple de façon simple les commentaires imbriqués.

Les actions

Dans les actions, on peut mettre du code Ocaml quelconque. Le système nous donne quelques fonctions pour interagir avec l'état du lecteur; si `lexbuf` est le nom de notre tampon, alors

`Lexing.lexeme lexbuf` est la chaîne de caractères reconnue

`Lexing.lexeme_char lexbuf n` est le *n*-ième caractère dans la chaîne reconnue (on commence à 0)

`Lexing.lexeme_start lexbuf` la position dans le flot d'entrée du début de la chaîne reconnue (on commence à 0)

`Lexing.lexeme_end lexbuf` la position dans le flot d'entrée de la fin de la chaîne reconnue (on commence à 0)

Compilation

1. écrire un fichier `lexer.mll` avec les définitions OcamlLex
2. appeler OcamlLex: `ocamllex lexer.mll`
3. cela produit le fichier `lexer.ml`
4. compiler `lexer.ml`:
 - s'il est un programme standalone: `ocamlc -o lexer lexer.ml` et ensuite on peut l'exécuter en tapant `./lexer`
 - s'il est un module: `ocamlc -c lexer.ml`

Un peu de pratique

Commentaires Imbriqués

```
{ exception Eof;;
  let level = ref 0;; }
let ouvrecomm = "/*"
let fermecomm = "*/"

rule token = parse
  ouvrecomm {level:=1; comment lexbuf;}
  | _ {print_string(Lexing.lexeme lexbuf);}
  | eof {raise Eof}
and comment = parse
  fermecomm {level := !level -1;
             if !level=0 then token lexbuf else comment lexbuf; }
  | ouvrecomm {level := !level + 1; comment lexbuf;}
  | _ {comment lexbuf; (* glob comments *)}
  | eof { raise Eof;}

{let _ = try let lexbuf = Lexing.from_channel stdin in
          while true do token lexbuf; done
          with Eof -> exit 0 }
```