Outils pour la Manipulation et l'Extraction de Données

Notes de cours

Cours 3

12 mars 2021

L'outil `ocamllex`

Syntaxe

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 ...
}
```

L'outil `ocamllex`

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 -> tau1
val f2 : Lexing.lexbuf -> tau2
...
val fn : Lexing.lexbuf -> taun
```

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

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

val from_string : string -> lexbuf

val from_function : (string -> int -> int) -> 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

Exemples

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']+
    { ... }
```

Raccourcis

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)  { ... }
```

Ambiguïtés

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 }
  ...
```

Récupérer le lexème

on peut nommer la chaîne reconnue, ou les 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)  { ... }
```

Traitement des blancs

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 la variable `lexbuf`

il est ainsi très facile de traiter les blancs :

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

Commentaires

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

Commentaires imbriqués

autre intérêt : on traite facilement les *commentaires imbriqués*

avec un compteur

```
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" }
```

Commentaires imbriqués

...ou 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

Un exemple complet

on se donne un type Caml 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" }
```

Les quatre règles

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`)

Efficacité

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

Efficacité

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
    ["and", AND; "as", AS; "assert", ASSERT;
     "begin", BEGIN; ...
}
rule token = parse
  | ident as s
    { try Hashtbl.find keywords s with Not_found -> IDENT
```

(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"|"
    { ASSERT }
| ...
```

mais plutôt

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

Compilation et dépendances

pour compiler (ou recompiler) les modules Caml, il faut déterminer les *dépendances* entre ces modules, grâce à `ocamldep`

or `ocamldep` ne connaît pas la syntaxe `ocamllex` $\Rightarrow$ il faut donc s'assurer de la fabrication préalable du code Caml avec `ocamllex`

le `Makefile` ressemble donc à ceci :

```
lexer.ml: lexer.mll
        ocamllex lexer.mll

.depend: lexer.ml
        ocamldep *.ml *.mli > .depend

include .depend
```

alternative : utiliser `ocamlbuild`

Récapitulation

- 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 pour le code écrit à l'aide d'`ocamllex`)