OCAML-LEX
I.Savnik

2008/10/04
2012/10/11 (popravki)

ENOSTAVNI PRIMERI

Enostaven primer programa, ki bere vhodno datoteko znak za znakom.
V primeru, da naleti na niz znakov "current_directory" ga zamenja s
trenutnim direktorijem.

```
{ }
rule translate = parse
  | "current_directory" { print_string (Sys.getcwd ()); translate lexbuf }
  | _ as c { print_char c; translate lexbuf }
  | eof   { exit 0 }
```

Prvo pravilo pravi, da vse vzorce "current_directory" spremenimo v
rezultat ovrednotenja izraza v oklepajih, ki ga imenujemo akcija.

Drugo pravilo prepiše vse znake iz vhoda na izhod in rekurzivno spet pokliče
izvajanje pravil.

Tretje pravilo definira zaključitev izvajanja pravil. V primeru, da
se na vhodu pojavi eof ustavi izvajanje bralca.

Še en enostaven primer.

```
{
  let num_lines = ref 0
  let num_chars = ref 0
}
rule count = parse
  | '\n'        { incr num_lines; incr num_chars; count lexbuf }
  | _           { incr num_chars; count lexbuf }
  | eof         { () }
{
  let main () =
    let lexbuf = Lexing.from_channel stdin in
    count lexbuf;
    Printf.printf "# of lines = %d, # of chars = %d\n" !num_lines !num_chars
  let _ = Printexc.print main ()
}
```

Ta bralec prešteje znake in vrstice v vhodni datoteki.

Najprej imamo sekcijo, ki jo imenujemo *glava*. V glavi so
definirane spremenljivke.

Glavi sledi preostanek kode bralca obdan z oklepaji, ki ga imenujemo
dodatek.

V dodatku so tri pravila od katerih prvo ujame konec vrstice ('\n'),
drugo sprejme katerikoli znak in zadnje čaka na konec datoteke.

Prvo pravilo poveča števec vrstic in znakov. Drugo poveča samo
število znakov in tretje zaključi izvajanje ob koncu datoteke.

Poglejmo si še malce bolj kompleksen primer bralca izrazov enostavnega
jezika.

```
(* scanner for a toy language *)
{
   open Printf
}
let digit = ’0’-’9’]
let id = ’a’-’z’] ’a’-’z’ ’0’-’9’]*
rule toy_lang = parse
   | digit+ as inum
    { printf "integer: %s (%d)\n" inum (int_of_string inum);
    toy_lang lexbuf
  }
   | digit+ ’.’ digit* as fnum
    { printf "float: %s (%f)\n" fnum (float_of_string fnum);
    toy_lang lexbuf
  }
   | "if"
   |  "then"
   |  "begin"
   |  "end"
   |  "let"
   |  "in"
   |  "function" as word
     { printf "keyword: %s\n" word;
     toy_lang lexbuf
 }
   | id as text
     { printf "identifier: %s\n" text;
     toy_lang lexbuf
 }
   |  ’+’
   |  ’-’
   |  ’*’
   |  ’/’ as op
     { printf "operator: %c\n" op;
     toy_lang lexbuf
 }
   | ’{’ [^ ’\n’]* ’}’ { toy_lang lexbuf } (* eat up one-line comments *)
   | [’ ’ ’\t’ ’\n’] { toy_lang lexbuf } (* eat up whitespace *)
   | _ as c
     { printf "Unrecognized character: %c\n" c;
     toy_lang lexbuf
 }
   | eof    { }
{
   let main () =
      let cin =
         if Array.length Sys.argv > 1
         then open_in Sys.argv.(1)
         else stdin
      in
      let lexbuf = Lexing.from_channel cin in
      toy_lang lexbuf
   let _ = Printexc.print main ()
}
```

Bralec prepozna različne tipe žetonov in sproti javlja kaj
je sprejel.

OBLIKA VHODNE DATOTEKE

Vhodna datoteka ocamllex je sestavljena iz: glave, definicij, pravil
in preostanka.

```
(* header section *)
```

```
{ header }
(* definitions section *)
let ident = regexp
let ...
(* rules section *)
rule entrypoint [arg1... argn] = parse
  | pattern { action }
   | ...
   | pattern { action }
and entrypoint [arg1... argn] = parse
    ...
and ...
(* trailer section *)
{ trailer }
```

Sekciji glava in pravila sta obvezni.

Sekciji glava in preostanek sta obdani z zavitimi oklepaji in lahko
vsebujeta poljubno ML kodo.

Sekcija z definicijami lahko vsebuje poljubne definicije identifikatorjev,
ki imajo obliko:

```
let ident = regexp
let ...
```

Identifikator zadošča pravilom za Caml vrednosti.

```
let digit = ['0'-'9']
let id = ['a'-'z']['a'-'z' '0'-'9']*
```

Zgornji primer definira identifikator digit, ki je regularni izraz
za eno samo cifro, in identifikator id za črko, ki ji lahko sledi poljubno cifer.

Izraz digit+ "." digit* je enakovreden ['0'-'9']+ "." ['0'-'9']* .
Oba se ujemata z eno ali več cifer, ki ji sledi pika ter nič ali več
cifer.

Sekcija pravil ima naslednjo obliko.

```
rule entrypoint [arg1... argn] = parse
  | pattern { action }
   | ...
   | pattern { action }
and ...
```

Vsak vzorec ima sledečo obliko (znak | v prvem pravilu je opcijski).

```
| pattern { action }
```

VZORCI

Vzorci zapišemo v obliki regularnih izrazov, ki imajo Caml sintakso.

- 'c'
  ujami znak 'c'. .
- _
  ujami katerikoli znak.
- eof
  ujami znak eof.
- "foo"

  ujami niz znakov "foo".
• [’x’ ’y’ ’z’]
  ujami enega izmed naštetih znakov.
• [’a’ ’b’ ’j’-’o’ ’Z’]
  množica znakov vsebuje območje znakov 'j'-'o'. Izraz se lahko
  ujame z enim izmed znakov v območju oz. preostalimi esplicitno
  naštetimi.
[^ ’A’-’Z’]
  negirana množica znakov ujame vse znake, ki niso v množici.
• r*
  nič ali več ponovitev r-ja, kjer je r regularni izraz.
• r+
  ena ali več ponovitev r-ja, kjer je r regularni izraz.
• r?
  nič ali ena pojavitev r-ja.
• ident
  "ident" je lahko eden izmed prej definiranih regularnih izrazov.
• (r)
  oklepaji so uporabljeni za definicijo prioritet.
• rs
  konkatenacija regularnih izrazov.
• r|s
  disjunkcija regularnih izrazov.
• r#s
  ujame razliko med dvema množicama znakov. .
• r as ident
  poveži niz, ki se ujame z r z ident.

Zgornji regularni izrazi so urejeni po prioriteti (najvišji ima
najvišjo prioriteto).

AKCIJE

Vsak vzorec ima pripadajočo akcijo, ki je lahko poljuben Ocaml izraz.
Naslednji primer izpusti vse nize "zap me" iz vhodne datoteke.

```
{}
rule token = parse
  | "zap me" { token lexbuf } (* ignore this token: no processing and continue *)
  | _ as c { print_char c; token lexbuf }
```

Naslednji primer združi večkratno pojavitev presledkov in tabulatorejev v en
sam presledek ter vrže proč bele znake na koncu vrstice.

```
{}
rule token = parse
  | [’ ’ ’\t’]+ { print_char ’ ’; token lexbuf }
  | [’ ’ ’\t’]+ ’\n’ { token lexbuf }     (* ignore this token *)
```

Akcije vsebujejo poljubno Ocaml kodo. Vsakič ko pokličemo funkcijo
leksikalnega analizatorja se nadaljuje branje na mestu, kjer smo se prej
ustavili.

Akcije se ovrednotijo po tem, ko je lexbuf povezan s trenutnim leksikalni
vmesnikom. lexbuf je definiran v modulu Lexing, ki je del standardne
knjižnice.

• Lexing.lexeme lexbuf
  Vrni niz, ki se je ujel.

• Lexing.lexeme_char lexbuf n
  Vrni n-ti znak v nizu, ki se je ujel (indeks se začne pri 0).

• Lexing.lexeme_start lexbuf
  Lexing.lexeme_end lexbuf
  Vrne absolutno pozicijo v vhodnem tekstu začetka/konca niza, ki se je ujel.

• Lexing.lexeme_start_p lexbuf
  Lexing.lexeme_end_p lexbuf
  Vrni pozicijo pozicije tipa (glej POZICIJA).

• entrypoint [exp1... expn] lexbuf
  Pokliči drugi lekser z dano vstopno točko. lexbuf je zadnji argument.

POZICIJA

Podatki o poziciji branja vhodnega teksta so shranjeni v lexbuf ,
ki vsebuje polje tipa position.

```
type position = {
   pos_fname : string; (* file name *)
   pos_lnum : int; (* line number *)
   pos_bol : int; (* the offset of the beginning of the line *)
   pos_cnum : int; (* the offset of the position *)
}
```

Vrednost polja pos_bol je število znakov od začetka datoteke do
začetka vrstice, ki se trenutno obdeluje. Vrednost polja pos_cnum
je številovod začetka datoteke do trenutne pozicije.

Lekser skrbi samo za polje pos_cnum zapisa lexbuf.lex_curr_p,
medtem ko mora za preostala polja poskrbeti uporabnik.

Naslednja funkcija poskrbi za števce v primeru, da lekser naleti na
novo vrstico.

```
let incr_linenum lexbuf =
   let pos = lexbuf.Lexing.lex_curr_p in
   lexbuf.Lexing.lex_curr_p <- { pos with
     Lexing.pos_lnum = pos.Lexing.pos_lnum + 1;
     Lexing.pos_bol = pos.Lexing.pos_cnum;
   }
;;
```

VMESNIK DO ocamlyacc

Ena od najpomembnejših uporab ocamllex je v povezavi z ocamlyacc.

Razčlenjevalniki narejeni z ocamlyacc kličejo lekser narejen z
ocamllex. Lekser vrne tip naslednjega simbola in njeovo vrednost.

Lekser mora uporabljati tipe simbolov definirane z ocamlyacc v
okviru spremenljivke %tokens.

Recimo, da ima vhodna datoteka ocamlyacc definiran simbol NUMBER,
potem bi lekser lahko izgledal takole.

```
{
  open Parse
}
rule token = parse
  ...
  | ['0'-'9']+ as num { NUMBER (int_of_string num) }
```

...

PRIMERI

Naslednji primeri so podani kompletno.

Naslednji primer prevede tekst "current_directory" v trenutni direktorij.

```
{ }
rule translate = parse
   | "current_directory" { print_string (Sys.getcwd ()) }
   | _ as c { print_char c }
   | eof     { exit 0 }
{
   let main () =
     let lexbuf = Lexing.from_channel stdin in
     while true do
         translate lexbuf
     done
   let _ = Printexc.print main ()
}
```

Naslednji enostaven primer prešteje vrstice, besede in znake v vhodni datoteki.
Če datoteka ni podana uporabi standarden vhod.

```
{ }
rule count lines words chars = parse
   | '\n' { count (lines+1) words (chars+1) lexbuf }
   | [^ ' ' '\t' '\n']+ as word
     { count lines (words+1) (chars+ String.length word) lexbuf }
   | _ { count lines words (chars+1) lexbuf }
   | eof { (lines, words, chars) }
{
   let main () =
     let cin =
         if Array.length Sys.argv > 1
         then open_in Sys.argv.(1)
         else stdin
     in
     let lexbuf = Lexing.from_channel cin in
     let (lines, words, chars) = count 0 0 0 lexbuf in
     Printf.printf "%d lines, %d words, %d chars\n" lines words chars
   let _ = Printexc.print main ()
}
```

Naslednji primer implementira lekser za enostavnen jezik.
Funkcija toy_lang vrne vrednost tipa token vendar funkcija
main ne naredi nič z vrnjeno vrednostjo.

```
{
   open Printf
   let create_hashtable size init =
     let tbl = Hashtbl.create size in
     List.iter (fun (key, data) -> Hashtbl.add tbl key data) init;
     tbl
   type token =
     | IF
     | THEN
     | ELSE
     | BEGIN
     | END
```

```
     | FUNCTION
     | ID of string
     | OP of char
     | INT of int
     | FLOAT of float
     | CHAR of char
   let keyword_table =
     create_hashtable 8 [
        ("if", IF);
        ("then", THEN);
        ("else", ELSE);
        ("begin", BEGIN);
        ("end", END);
        ("function", FUNCTION)
     ]
}
let digit = ['0'-'9']
let id = ['a'-'z' 'A'-'Z']['a'-'z' '0'-'9']*
rule toy_lang = parse
   | digit+ as inum
    { let num = int_of_string inum in
    printf "integer: %s (%d)\n" inum num;
    INT num
  }
   | digit+ '.' digit* as fnum
    { let num = float_of_string fnum in
    printf "float: %s (%f)\n" fnum num;
    FLOAT num
 }
   | id as word
    { try
       let token = Hashtbl.find keyword_table word in
       printf "keyword: %s\n" word;
       token
    with Not_found ->
       printf "identifier: %s\n" word;
       ID word
 }
   | '+'
   | '-'
   | '*'
   | '/' as op
    { printf "operator: %c\n" op;
    OP op
 }
   | '{' [^ '\n']* '}' (* eat up one-line comments *)
   | [' ' '\t' '\n'] (* eat up whitespace *)
    { toy_lang lexbuf }
   | _ as c
    { printf "Unrecognized character: %c\n" c;
    CHAR c
 }
   | eof
    { raise End_of_file }
{
   let rec parse lexbuf =
       let token = toy_lang lexbuf in
       (* do nothing in this example *)
     parse lexbuf
  let main () =
    let cin =
       if Array.length Sys.argv > 1
       then open_in Sys.argv.(1)
       else stdin
    in
    let lexbuf = Lexing.from_channel cin in
    try parse lexbuf
    with End_of_file -> ()
  let _ = Printexc.print main ()
}
```

LITERATURA

1. SooHyoung Oh, Ocamllex Tutorial, 2004.
2. SooHyoung Oh, Ocamlyacc Tutorial, 2004.
3. C.Donnelly, R.Stallman, Bison, The Yacc-compatible Parser Generator, 2012.
4. The Lex & Yacc Page, http://dinosaur.compilertools.net/, 2012.