Ocamllex Tutorial

Ocamllex Adaptation: SooHyoung Oh

Ocamllex Tutorial

by Ocamllex Adaptation: SooHyoung Oh

This is a tutorial on how to use `ocamllex` which is distributed with Ocaml language.

This document borrowed lots of part from flex manual.

Please mail all comments and suggestions to `<shoh at compiler dot kaist dot ac dot kr>`

This tutorial is work-in-progress. The latest version can be found at http://pllab.kaist.ac.kr/~shoh/ocaml/ocamllex-ocamlyacc/ocamllex-tutorial/index.html.

The companion tutorial for `ocamlyacc` is available at http://pllab.kaist.ac.kr/~shoh/ocaml/ocamllex-ocamlyacc/ocamlyacc-tutorial/index.html.

You can find the source of this document in ocamllex-tutorial-src.tar.gz. For printing, pdf (A4 size) is presented, and there is html (tar.gz).

You can download the source of examples used in this document from ocamllex-examples.tar.gz.

Last updated: 2004-11-16

Table of Contents

Chapter 1. Introduction

`ocamllex` is a tool for generating *scanners*: programs which recognized lexical patterns in text. `ocamllex` reads the given input files, for a description of a scanner to generate. The description is in the form of pairs of regular expressions and Ocaml code, called *rules*. `ocamllex` generates as output a Ocaml source file which defines lexical analyzer functions. This file is compiled and linked to produce an executable. When the executable is run, it analyzes its input for occurrences of the regular expressions. Whenever it finds one, it executes the corresponding Ocaml code.

If you execute the following command with the input file named *lexer*`.mll`

```
ocamllex lexer.mll
```

you will get Caml code for a lexical analyzer in file *lexer*`.ml`.

Chapter 2. Some simple examples

First some simple examples to get the flavor of how one uses `ocamllex`. The following `ocamllex` input specifies a scanner which whenever it encounters the string "current_directory" will replace it with the current directory:

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

In the first rule, "current_directory" is the *pattern* and the expression between braces is the *action*. By this rule, when the scanner matches the string "current_directory", it executes the corresponding action which prints the current directory name and call the scanner again. Recursive calling itself is necessary to do the other job.

Any text not matched by a *ocamllex* scanner generates exception `Failure "lexing: empty token"`, so you have to supply the last two rules. The second rule copies any character to its output which is not matched by the first rule, and it calls itself again. By the third rule, the program exits when it meets `end of file`. So the net effect of this scanner is to copy its input file to its output with each occurrence of "current_directory" expanded. The "{ }" in the first line delimits the `header` section from the rest.

Here's another simple example:

```
{
  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 ()
}
```

This scanner counts the number of characters and the number of lines in its input (it produces no output other than the final report on the counts). The first `header` section declares two globals, "num_lines" and "num_chars", which are accessible both inside scanner function `count` and in the `trailer` section which is the last part enclosed by braces. There are three rules, one which matches a newline ("\n") and increments both the line count and the character count, and one which matches any character other than a newline (indicated by the "_" regular expression). At the end of file, the scanner function `count` returns `unit`.

A somewhat more complicated example:

```
(* 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 ()
}
```

This is the beginnings of a simple scanner for a language. It identifies different types of *tokens* and reports on what it has seen.

The details of this example will be explained in the following sections.

Chapter 3. Format of the input file

The *ocamllex* input file consists of four sections; *header*, *definitions*, *rules* and *trailer* section:

```
(* 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 }
```

Comments are delimited by (* and *), as in Caml.

The *header* and *rules* sections are necessary while *definitions* and *trailer* sections are optional.

The *header* and *trailer* sections are enclosed in curly braces and they contain arbitrary Caml code. At the beginning of the output file, the header text is copied as is while the trailer text is copied at the end of the output file. For example, you can code open directives and some auxiliary funtions in the header section.

The *definitions* section contains declarations of simple *ident* definitions to simplify the scanner specification. Ident definitions have the form:

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

The "ident" must be valid identifiers for Caml values (starting with a lowercase letter). For example,

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

defines "digit" to be a regular expression which matches a single digit, and "id" to be a regular expression which matches a letter followed by zero-or-more letters-or-digits. A subsequent reference to

```
digit+ "." digit*
```

is identical to

```
['0'-'9']+ "." ['0'-'9']*
```

and matches one-or-more digits followed by a '.' followed by zero-or-more digits.

The *rules* section of the `ocamllex` input contains a series of entrypoints of the form:

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

The first `|` (bar) after `parse` is optional.

Each entrypoint consists of a series of pattern-action:

```
  | pattern { action }
```

where the action must be enclosed in curly braces.

See below for a further description of patterns and actions.

Chapter 4. Patterns

The patterns in the input are written using regular expressions in the style of lex, with a more Caml-like syntax. These are:

- `'c'`
  match the character 'c'. The character constant is the same syntax as Objective Caml character.

- `_`
  (underscore) match any character.

- `eof`
  match an end-of-file .

- `"foo"`
  the literal string "foo". The syntax is the same syntax as Objective Caml string constants.

- `['x' 'y' 'z']`
  character set; in this case, the pattern matches either an 'x', a 'y', or a 'z' .

- `['a' 'b' 'j'-'o' 'Z']`
  character set with a range in it; ranges of characters 'c1' - 'c2' (all characters between c1 and c2, inclusive); in this case, the pattern matches an 'a', a 'b', any letter from 'j' through 'o', or a 'Z'.

- `[^ 'A'-'Z']`
  a "negated character set", i.e., any character but those in the class. In this case, any character EXCEPT an uppercase letter.

- `[^ 'A'-'Z' '\n']`
  any character EXCEPT an uppercase letter or a newline

- `r*`
  zero or more r's, where r is any regular expression

- `r+`
  one or more r's, where r is any regular expression

- `r?`
  zero or one r's, where r is any regular expression (that is, "an optional r")

- `ident`
  the expansion of the "ident" defined by an earlier `let ident = regexp` definition.

- `(r)`
  match an r; parentheses are used to override precedence (see below)

- `rs`

  the regular expression r followed by the regular expression s; called "concatenation"

- `r|s`

  either an r or an s

- `r#s`

  match the difference of the two specified character sets.

- `r as ident`

  bind the string matched by `r` to identifier `ident`.

The regular expressions listed above are grouped according to precedence, from highest precedence at the top to lowest at the bottom; '*' and '+' have highest precedence, followed by '?', 'concatenation', `'|'`, and then `'as'`. For example,

```
"foo" | "bar"*
```

is the same as

```
("foo")|("bar"*)
```

since the '*' operator has higher precedence than than alternation (`'|'`). This pattern therefore matches *either* the string "foo" *or* zero-or-more of the string "bar".

To match zero-or-more "foo"'s-or-"bar"'s:

```
("foo"|"bar")*
```

A negated character set such as the example "[^ 'A'-'Z']" above *will match a newline* unless "\n" (or an equivalent escape sequence) is one of the characters explicitly present in the negated character set (e.g., "[^ 'A'-'Z' '\n']"). This is unlike how many other regular expression tools treat negated character set, but unfortunately the inconsistency is historically entrenched. Matching newlines means that a pattern like [^"]* can match the entire input unless there's another quote in the input.

Chapter 5. How the input is matched

When the generated scanner is run, it analyzes its input looking for strings which match any of its patterns. If it finds more than one match, it takes the one matching the most text (the "longest match" principle). If it finds two or more matches of the same length, the rule listed first in the `ocamllex` input file is chosen (the "first match" principle).

Once the match is determined, the text corresponding to the match (called the *token*) is made available in the form of a string. The *action* corresponding to the matched pattern is then executed (a more detailed description of actions follows), and then the remaining input is scanned for another match.

If no match is found, the scanner raises the `Failure "lexing: empty token"` exception.

Now, let's see the examples which shows how the patterns are applied.

```
rule token = parse
  | "ding" { print_endline "Ding" }  (* "ding" pattern *)
  | ['a'-'z']+ as word    (* "word" pattern *)
    { print_endline ("Word: " ^ word) }
  ...
```

When "ding" is given as an input, the `ding` and `word` pattern can be matched. `ding` pattern is selected because it comes before `word` pattern. So if you code like this:

```
rule token = parse
  | ['a'-'z']+ as word    (* "word" pattern *)
    { print_endline ("Word: " ^ word) }
  | "ding" { print_endline "Ding" }  (* "ding" pattern *)
  | ...
```

`ding` pattern will be useless.

In the following example, there are three patterns: `ding`, `dong` and `dingdong`.

```
rule token = parse
  | "ding" { print_endline "Ding" }  (* "ding" pattern *)
  | "dong" { print_endline "Dong" }  (* "dong" pattern *)
  | "dingdong" { print_endline "Ding-Dong" } (* "dingdong" pattern *)
  ...
```

When "dingdong" is given as an input, there are two choices: `ding` + `dong` pattern or `dingdong` pattern. But by the "longest match" principle, `dingdong` pattern will be selected.

Though the "shortest match" principle is not used so frequently, `ocamllex` supports it. If you want to select the shortest prefix of the input, use `shortest` keyword instead of the `parse` keyword. The "first match" principle holds still with the "shortest match" principle.

Chapter 6. Actions

Each pattern in a rule has a corresponding action, which can be any arbitrary Ocaml expression. For example, here is the specification for a program which deletes all occurrences of "zap me" from its input:

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

Here is a program which compresses multiple blanks and tabs down to a single blank, and throws away whitespace found at the end of a line:

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

Actions can include arbitrary Ocaml code which returns a value. Each time the lexical analyzer function is called it continues processing tokens from where it last left off until it either reaches the end of the file.

Actions are evaluated after the `lexbuf` is bound to the current lexer buffer and the identifer following the keyword `as` to the matched string. The usage of `lexbuf` is provided by the Lexing standard library module;

- `Lexing.lexeme lexbuf`

  Return the matched string.

- `Lexing.lexeme_char lexbuf` *n*

  Return the *n*th character in the matched string. The index number of the first character starts from 0.

- `Lexing.lexeme_start lexbuf`

  `Lexing.lexeme_end lexbuf`

  Return the absolute position in the input text of the beginning/end of the matched string. The position of the first character is 0.

- `Lexing.lexeme_start_p lexbuf`

  `Lexing.lexeme_end_p lexbuf`

  (Since Ocaml 3.08) Return the position of type `position` (See Position).

- *entrypoint [exp1... expn]* `lexbuf`

  Call the other lexer on the given entry point. Notice that `lexbuf` is the last argument.

6.1. Position

* *Since Ocaml 3.08*

The position information on scanning the input text is recorded in the `lexbuf` which has a field `lex_curr_p` of the type `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 *)
}
```

The value of `pos_bol` field is the number of characters between the beginning of the file and the beginning of the line while the value of `pos_cnum` field is the number of characters between the beginning of the file and the position.

The lexing engine manages only the `pos_cnum` field of `lexbuf.lex_curr_p` with the number of characters read from the start of `lexbuf`. So you are reponsible for the other fields to be accurate. Typically, whenever the lexer meets a newline character, the action contains a call to the following function:

```
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;
  }
;;
```

Chapter 7. The generated scanner

The output of `ocamllex` is the file *lex*.ml when it is invoked as `ocamllex` *lex*`.mll`. The generated file contains the *scanning functions*, a number of tables used by it for matching tokens, and a number of auxiliary routines. The *scanning functions* are declared as followings:

```
let entrypoint [arg1... argn] lexbuf =
  ...
and ...
```

where the fuction *entrypoint* has `n + 1` arguments. `n` arguments come from the definition of the *rules* secton. And the resulting scanning function has one more argument named `lexbuf` of `Lexing.lexbuf` type as the last one.

Whenever *entrypoint* is called, it scans tokens from the `lexbuf` argument. When it finds a match in patterns, it executes the corresponding action and returns. If you want to continue the lexical analyze after evaluating of the action, you must call the *scanning function* recursively.

Chapter 8. Start conditions

`ocamllex` provides a mechanism for conditionally activating rules. When you want do activate the other rule, just call the other *entrypoint* function. For example, the following has two rules, one for finding tokens and one for skipping comments.

```
{}
rule token = parse
  | [' ' '\t' '\n']+
   (* skip spaces *)
   { token lexbuf }
  | "(*"
 (* activate "comment" rule *)
   { comment lexbuf }
  ...
and comment = parse
  | "*)"
   (* go to the "token" rule *)
   { token lexbuf }
  | _
 (* skip comments *)
   { comment lexbuf }
  ...
```

When the generated scanner meets comment start token `"(*"` at the `token` rule, it activates the other rule `comment`. When it meets the end of comment token `"*)"` at the `comment` rule. it returns to the scanning `token` rule.

Chapter 9. Interfacing with `ocamlyacc`

One of the main uses of `ocamllex` is as a companion to the `ocamlyacc` parser-generator. `ocamlyacc` parsers call one of the *scanning functions* to find the next input token. The routine is supposed to return the type of the next token with an associated value. To use `ocamllex` with `ocamlyacc`, scanner functions should use parser module to refer token types, which are defined in '%tokens' attributes appearing in the `ocamlyacc` input. For example, if input filename of `ocamlyacc` is *parse.mly* and one of the tokens is "NUMBER", part of the scanner might look like:

```
{
  open Parse
}

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

Chapter 10. Options

`ocamllex` has the following options:

- `-o` *output-file*

  By default, `ocamllex` produces *lexer*`.ml`, when `ocamllex` is invoked as "`ocamllex` *lexer*`.mll`". You can change the name of the output file using `-o` option.

- `-ml`

  By default, `ocamllex` produces code that uses the Caml built-in automata interpreter. Using this option, the automaton is coded as Caml functions. This option is useful for debugging `ocamllex`, but it's not recommended for production lexers.

- `-q`

  By default, `ocamllex` outputs informational messages to standard output. If you use `-q` option, they are suppressed.

Chapter 11. Usage Tips

11.1. Keyword Hashtable

The number of status transitions generated by `ocamllex` are limited to at most 32767. If you use too many transitions, for example, too many keywords, `ocamllex` generates the following error message:

```
camllex: transition table overflow, automaton is too big
```

It tells that your lexer definition is too complex. To make the generated automata small, you have to encode using keyword table:

```
{
  let keyword_table = Hashtbl.create 72
  let _ =
    List.iter (fun (kwd, tok) -> Hashtbl.add keyword_table kwd tok)
              [ ("keyword1", KEYWORD1);
                ("keyword2", KEYWORD2);
  ...
              ]
}

rule token = parse
  | ...
  | ['A'-'Z' 'a'-'z'] ['A'-'Z' 'a'-'z' '0'-'9' '_']* as id
              { try
          Hashtbl.find keyword_table id
                with
    Not_found -> IDENT id
  }
  | ...
```

For a complete example, see Toy Language program.

11.2. Nested Comments

Some language such as Ocaml support nested comment. It can be implemented like this:

```
{ }

rule token = parse
  | "(*"  { print_endline "comments start"; comments 0 lexbuf }
  | [' ' '\t' '\n'] { token lexbuf }
  | ['a'-'z']+ as word
    { Printf.printf "word: %s\n" word; token lexbuf }
  | _ as c { Printf.printf "char %c\n" c; token lexbuf }
  | eof  { raise End_of_file }

and comments level = parse
  | "*)" { Printf.printf "comments (%d) end\n" level;
      if level = 0 then token lexbuf
    else comments (level-1) lexbuf
  }
  | "(*" { Printf.printf "comments (%d) start\n" (level+1);
      comments (level+1) lexbuf
  }
  | _  { comments level lexbuf }
  | eof  { print_endline "comments are not closed";
      raise End_of_file
  }
```

When the scanner function meets *comments start* token `"(*"` in evaluating `token` rule, it enters `comments` rule with level of 0. `token` rule is invoked again when all comments are closed. Comments nesting level is increased whenever there is *comment start* token `"(*"` in the input text.

If the scanner function meets *end of comments* token `"*)"`, it tests the comments nesting level. When the nesting level is not zero, it decrements the level by one and continues to scan comments. It returns to `token` rule when all the comments are closed i.e., the nesting level is zero.

Chapter 12. Examples

This chapter includes examples in complete form. Some are revised from the code fragments of the previous chapters.

12.1. Translate

This example translates the text "current_directory" to the *current directory*.

```
{ }

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 ()
}
```

12.2. Word Count

This example shows the number of lines, words and characters of the given file if the filename is given, or of the standard input if no command arguments are given.

```
{ }

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 ()
}
```

12.3. Toy Language

In this example, the scanner function `toy_lang` returns a value of `token` type, but the `main` function does nothing with it.

```
{
  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 ()
}
```

Chapter 13. License

13.1. License in flex manual

13.2. Ocamllex Adaptation Copyright and Permissions Notice