

CS421 Lecture 8: Lexing and ocamllex¹

Mark Hills
mhills@cs.uiuc.edu

University of Illinois at Urbana-Champaign

June 16, 2008

¹Based on slides by Mattox Beckman, as updated by Vikram Adve, Gul Agha, and Elsa Gunter

- 1 Overview
- 2 Lexing
- 3 ocamllex
- 4 Activity

Where We Are Going

- Overall goal: we want to turn strings of characters – code – into computer instructions
- Easiest to break this down into phases
- First, turn strings into abstract syntax trees (ASTs) – this is **parsing**
- Next, turn abstract syntax trees into executable instructions – **compiling** or **interpreting**

Lexing and Parsing

Strings are converted into ASTs in two phases:

Lexing Convert strings (streams of characters) into lists (or streams) of *tokens*, representing words in the language

Parsing Convert lists of tokens into abstract syntax trees

Lexing

With lexing, we break sequences of characters into different syntactic categories, called *tokens*. As an example, we could break this:

```
asd 123 jkl 3.14
```

into this:

```
[String "asd", Int 123; String "jkl"; Float 3.14]
```

Lexing Strategy

Our strategy will be to leverage regular expressions and finite automata to recognize tokens:

- each syntactic category will be described by a regular expression (with some extended syntax)
- words will be recognized by an encoding of a corresponding finite state machine

However, this still leaves us with a problem. How do we pull multiple words out of a string, instead of just recognizing a single word?

Lexing: Multiple Tokens

To solve this, we will modify the behavior of the DFA.

- if we find a character where there is no transition from the current state, stop processing the string
- if we are in an accepting state, return the token corresponding to what we found as well as the remainder of the string
- now, use iterator or recursion to keep pulling out more tokens
- if we were not in an accepting state, fail – invalid syntax

Lexing Options

We could write a lexer by writing regular expressions, and then translating these by hand into a DFA. That sounds tedious and repetitive – perfect for a computer! Can we write a program that takes regular expressions and generates automata for us?

- Someone already did – Lex!
- OCaml version of this is ocamllex

How does it work?

We need a few core items to get this working:

- Some way to identify the input string – we'll call this the lexing buffer
- A set of regular expressions that correspond to tokens in our language
- A corresponding set of actions to take when tokens are matched

The lexer can then take the regular expressions to build state machines, which are then used to process the lexing buffer. If we reach an accept state and can take no further transitions, we can apply the actions.

Mechanics of Using ocamllex

- Lexer definitions using ocamllex are written in a file with a .mll extension. The file includes the regular expressions in a table, with associated actions for each.
- OCaml code for the lexer is generated with


```
ocamllex file.mll
```
- This generates the code for the lexer in file file.ml

Sample Lexer

```

1 rule main = parse
2 | ['0'-'9']+ '.' ['0'-'9']+ { print_string "Float\n"}
3 | ['0'-'9']+ { print_string "Int\n"}
4 | ['a'-'z']+ { print_string "String\n"}
5 | _ { main lexbuf }
6 {
7 let newlexbuf = (Lexing.from_channel stdin) in
8 print_string "Ready to lex.\n";
9 main newlexbuf
10 }

```

General Lexer Format

```

1 { header }
2 let ident = regexp ...
3 rule entrypoint [arg1... argn] = parse
4   | regexp { action }
5   | ...
6   | regexp { action }
7 and entrypoint [arg1... argn] = parse
8 ...and ...
9 { trailer }

```

ocamllex Input

- header and footer contain arbitrary OCaml code to insert into generated .ml file
- shorthands for regular expressions can be introduced with


```
let ident = regexp
```
- multiple entry points turn into multiple functions in the .ml file, with the given arguments and an additional argument for the lexing buffer

Regular Expressions in ocamllex

The regular expression format is similar to what we've seen so far, but still slightly different.

- Single quoted characters for letters: 'a'
- Underscores match any letter: _
- End-of-file marker: eof
- Concatenation of most expressions same as before
- Concatenation of character sequence shown as a string: 'while'
- Choice: instead of $e_1 + e_2$, it is $e_1 | e_2$

Regular Expressions, cont.

- Character ranges – pick any character in the range, based on character codes: $[c_1 - c_2]$
- Negative character ranges – any character *not* in the range: $[\^c_1 - c_2]$
- e^* has same meaning as we've already seen
- e^+ means *one or more*, same as ee^*
- $e?$ means *one or none*, same as $e + \epsilon$
- $e_1\#e_2$ means the characters in e_1 but not in e_2
- `ident` – shorthand for earlier definition of a regular expression using `let`
- e_1 as `id` – binds matched string to `id`

For more information...

The page for the ocamllex tool is at
<http://caml.inria.fr/pub/docs/manual-ocaml/manual026.html>

Example

```

1 {
2 type result = Int of int
3           | Float of float
4           | String of string
5 }
6 let digit = ['0'-'9']
7 let digits = digit +
8 let lower_case = ['a'-'z']
9 let upper_case = ['A'-'Z']
10 let letter = upper_case | lower_case
11 let letters = letter +

```

Example

```

1 rule main = parse
2   digits'.digits as f { Float (float_of_string f) }
3   | digits as n       { Int (int_of_string n) }
4   | letters as s      { String s }
5   | _ { main lexbuf }
6 { let newlexbuf = (Lexing.from_channel stdin) in
7   print_string "Ready to lex.";
8   print_newline ();
9   main newlexbuf

```

Example

```

1 # #use "test.ml";
2 ...
3 val main : Lexing.lexbuf -> result = <fun>
4 val __ocaml_lex_main_rec :
5   Lexing.lexbuf -> int -> result = <fun>
6 Ready to lex.
7 hi there 234 5.2
8 - : result = String "hi"

```

What happened to the rest?

What went wrong?

How do we get the lexer to look at more than one token?

- The *action* has to tell it to look for more – we need recursion
- Side benefit – we can add “state” into the calls, since we can pass information as parameters
- Note: we are already doing this with the `_` case

Example

```

1 rule main = parse
2 | digits '.' digits as f
3   { Float (float_of_string f) :: main lexbuf }
4 | digits as n      { Int (int_of_string n) :: main lexbuf }
5 | letters as s     { String s :: main lexbuf }
6 | eof              { [] }
7 | _ { main lexbuf }

```

Example Results

```

1 Ready to lex.
2 hi there 234 5.2
3 - : result list = [String "hi"; String "there";
4                   Int 234; Float 5.2]
5 #

```

Ctrl-d (end of file) can be used to exit.

Dealing with Comments, First Attempt

```

1 let open_comment = "("
2 let close_comment = ")"
3 rule main = parse
4   digits '.' digits as f { Float (float_of_string f) ::
5                             main lexbuf }
6 | digits as n      { Int (int_of_string n) :: main lexbuf }
7 | letters as s     { String s :: main lexbuf }
8 | open_comment    { comment lexbuf }
9 | eof              { [] }
10 | _ { main lexbuf }
11 and comment = parse
12   close_comment { main lexbuf }
13 | _             { comment lexbuf }

```

Dealing with Nested Comments

```

1 rule main = parse ...
2 | open_comment    { comment 1 lexbuf }
3 | eof              { [] }
4 | _ { main lexbuf }
5 and comment depth = parse
6   open_comment    { comment (depth+1) lexbuf }
7 | close_comment   { if depth = 1
8                       then main lexbuf
9                       else comment (depth - 1) lexbuf }
10 | _               { comment depth lexbuf }

```

It would probably be good to add some code to raise an error if you hit end of file reading a comment, too, since this means someone opened one that they did not close.

In total...

```

1 rule main = parse
2   digits '.' digits as f { Float (float_of_string f) ::
3     main lexbuf }
4 | digits as n           { Int (int_of_string n) :: main lexbuf }
5 | letters as s          { String s :: main lexbuf }
6 | open_comment          { comment 1 lexbuf }
7 | eof                   { [] }
8 | _ { main lexbuf }
9 and comment depth = parse
10  open_comment           { comment (depth+1) lexbuf }
11 | close_comment        { if depth = 1
12   then main lexbuf
13   else comment (depth - 1) lexbuf }
14 | _                     { comment depth lexbuf }

```

A Quick Note

This material isn't specific to ocamllex – it is more about general regular expressions. However, it should give you some practice with the enhanced regular expression syntax, such as character ranges.

Problems I

Write a regular expression for the following kinds of words

- hexadecimal numbers
- numbers in scientific notation
- file names ending in .C
- numbers between 0 and 255

Describe in English the following regular expressions

- `[a-zA-Z][a-zA-Z0-9]+`
- `[a-z]*(es|ed|ing)`
- `<[a-z0-9]+@[a-z0-9]+(\.[a-z0-9]+)+>`

Answers I

- hexadecimal numbers: `[0-9A-Fa-f]+`
- numbers in scientific notation:
`[0-9]+\.[0-9]+E(+|-)[0-9]+`
- file names ending in .C: `.*\.C`
- numbers between 0 and 255:
`25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9][0-9][0-9]`
- `[a-zA-Z][a-zA-Z0-9]+` like variable names
- `[a-z]*(es|ed|ing)` words ending in “es”, “ed”, or “ing” (verb forms)
- `<[a-z0-9]+@[a-z0-9]+(\.[a-z0-9]+)+>` email addresses

Problems II

Which of the following can be described by regular expressions?

- All the words in the English language
- All the Fibonacci numbers
- Numbers that are multiples of 4 (assume ≥ 2 digits)
- Words that have exactly as many as as they have bs
- Palindromes

Answers II

- All the words in the English language
Yes — it's huge, but it works. `(a|aardvark|abate|...)`
- All the Fibonacci numbers
No — the set is infinite and requires computation
- Numbers that are multiples of 4 (assume ≥ 2 digits)
Yes — `[0-9]*([02468][048]|[13579][26])`
- Words that have exactly as many as as they have bs
No — requires unbounded counting
- Palindromes
No — requires unbounded memory
(aibohphobia = fear of palindromes)