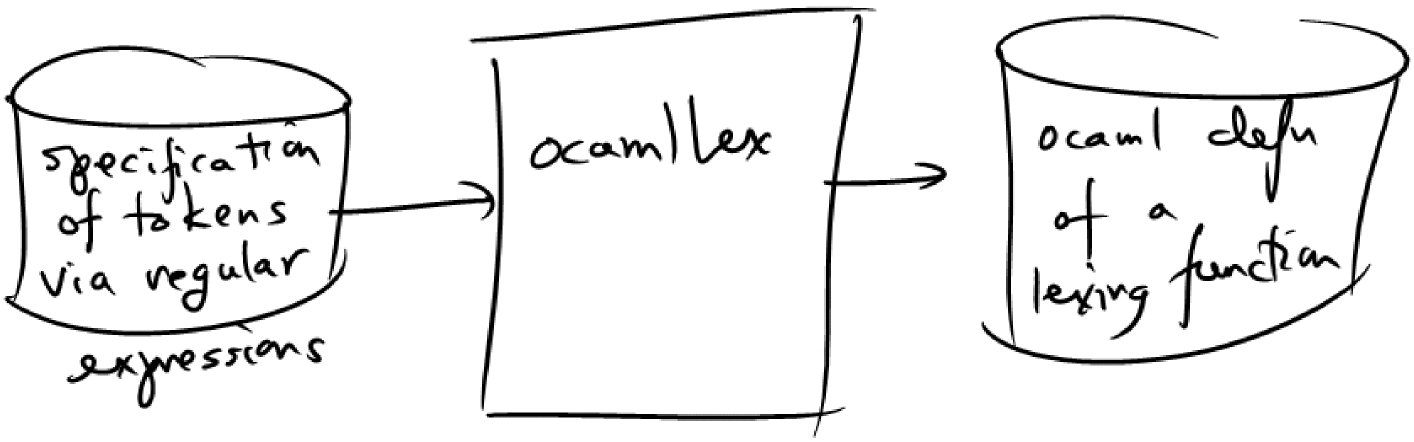
CS421 Lecture 6

- Today's class
  - Regular Expressions
  - Ocamllex

- These slides are based on slides by Elsa Gunter, Mattox Beckman

Overview of Ocamllex



Regular Expressions

- A regular expression is one of
  - ϵ, aka “”
  - ‘a’ for any character a
  - $r_1 r_2$, where r_1 and r_2 are regular expr’s
  - $r_1 \mid r_2$, where r_1 and r_2 are regular expr’s
  - r^*, where r is a reg expr’s
  - Ø

Every reg. expr. r represents a set of strings, denoted $\mathcal{L}(r)$

Regular Expression Examples

$\mathcal{L}(\text{'a' 'b' 'c'}) = \{\text{"abc"}\}$

$\mathcal{L}((\text{'a'} \mid \text{'b'})\ \text{'c'}) = \{\text{"ac"}, \text{"bc"}\}$

$\mathcal{L}((\text{'a'} \mid \text{'b'})^*\ \text{'c'}) = \{\text{"c"}, \text{"ac"}, \text{"bc"}, \text{"aac"}, \text{"abc"}, \ldots\}$

Regular Expression Examples

- Keywords 'c' 'a' 's' 'e'
  | 'c' 'l' 'a' 's' 's' | ....
- Operators
  '<' | '<' '<' | '<' '=' | ....
- Identifiers ('a' | 'b' | ... | 'z' | 'A' | ... | 'Z')
  ('a' | ... | 'Z' | '0' | '1' | ... | '9')*
- Int literals

Abbreviations

"$c_1 c_2 \ldots c_n$" ⇒ 'c_1' 'c_2' ... 'c_n'

['a' - 'z' '#'] ⇒ 'a' | 'b' | ... | 'z' | '#'

['a' 'w' '#'] ⇒ 'a' | 'w' | '#'

r+ ⇒ r(r*)

r? ⇒ r | ""

[^ 'a' - 'z'] ⇒ all char's <u>except</u> 'a' - 'z'

(complement of ['a' - 'z'])

_ ⇒ any single char

Regular Expression Example

- Float-point Literal

$$['0'-'9']+ . ['0'-'9']+ (['e''E'] ['+''-'] ? ['0'-'9']+) ?$$

$$r* = (r+)?$$

Regular Expression Example

- New-Style Comments (//)

  `"//" [^ '\n']* '\n'`

- Old-Style Comments (/* ... */)

  `"/*" ([^ '*'] | '*'+ [^ '*' '/'])* "*/"`

Implementing Reg Expr

- Translate RE's to NFA's, then to DFA's

Lexing with Reg Exprs

- Create one large RE:

```
  RE for case            {action for case}
| RE for class           {action for class}
| :
| RE for ident's         {action for idents}
| RE for f.p. constants        :
| RE for int consts            :
  :
```

- Then add actions

(cont.)

- Ambiguous cases:
- Two tokens found, one longer

  Choose longer one

- Two tokens found, the same length

  Choose earlier reg. expr.

General Input

{ *header* } — ocaml defns

let *ident* = *regexp* ... — abbrev's for reg. expr's

rule *entrypoint* [*arg1*... *argn*] = parse

    regexp { *action* }

  | ...

  | *regexp* { *action* }

(*entrypoint*: name of gen'd function, with arg's arg1, ..., argn, lexbuf)

and *entrypoint* [*arg1*... *argn*] = parse ...and ...

{ *trailer* } — ocaml defns

Ocamllex Input

- *header* and *trailer* contain arbitrary ocaml code put at top an bottom of *<filename>*.ml
- let *ident* = *regexp* ... Introduces *ident* for use in later regular expressions

Mechanics

- Put table of regular expressions and corresponding actions (written in ocaml) into a file
  <filename>.mll
- Call
  ocamllex <filename>.mll
- Produces Ocaml code for a lexical analyzer in file <filename>.ml

Sample Input

```
rule main = parse
   ['0'-'9']+                { print_string "Int\n"}
 | ['0'-'9']+'.'['0'-'9']+  { print_string "Float\n"}
 | ['a'-'z']+               { print_string "String\n"}
 | _                        { main lexbuf }
 {
 let newlexbuf = (Lexing.from_channel stdin) in
       print_string "Ready to lex.\n";
       main newlexbuf
}
```

Ocamllex Input

- *<filename>*.ml contains one lexing function per *entrypoint*
  - Name of function is name given for *entrypoint*
  - Each entry point becomes an Ocaml function that takes *n*+1 arguments, the extra implicit last argument being of type Lexing.lexbuf
- *arg1*... a*rgn* are for use in *action*

Ocamllex Regular Expression

- Single quoted characters for letters: 'a'
- _: (underscore) matches any character
- eof: special "end_of_file" marker
- Concatenation: concatenation
- "*string*": concatenation of sequence of characters
- $e_1 \mid e_2$: choice

Ocamllex Regular Expression

- $[c_1 - c_2]$: choice of any character between first and second inclusive, as determined by character codes
- $[\hat{} c_1 - c_2]$: choice of any character NOT in set
- e^*: same as before
- $e+$: same as $e\ e^*$
- $e?$: option - was $e_1 \mid \varepsilon$

Ocamllex Regular Expression

- e_1# e_2: the characters in e_1 but not in e_2; e_1 and e_2 must describe just sets of characters
- *ident*: abbreviation for earlier reg exp in let *ident* = *regexp*
- e_1 as *id*: binds the result of e_1 to *id* to be used in the associated *action*

(['0'-'9']+ as decpart) '.' (['0'-'9']+ as fracpart) ....

Ocamllex Manual

- More details can be found at

http://caml.inria.fr/pub/docs/manual-ocaml/manual026.html

Example: test.mll

```
{ type result = Int of int | Float of float | String
  of string }
let digit = ['0'-'9']
let digits = digit +
let lower_case = ['a'-'z']
let upper_case = ['A'-'Z']
let letter = upper_case | lower_case
let letters = letter +
```

Example: test.mll

```
rule main = parse
  digits'.'digits as f  { Float (float_of_string f) }
 | digits as n            { Int (int_of_string n) }
 | letters as s           { String s}
 | _ { main lexbuf }
 { let newlexbuf = (Lexing.from_channel stdin) in
      print_string "Ready to lex.";
      print_newline ();
      main newlexbuf  }
```

Example

```
# #use "test.ml";;
...
val main : Lexing.lexbuf -> result = <fun>
Ready to lex.
hi there 234 5.2
- : result = String "hi"
```

What happened to the rest?!?

Example

```
# let b = Lexing.from_channel stdin;;
# main b;;
hi 673 there
- : result = String "hi"
# main b;;
- : result = Int 673
# main b;;
- : result = String "there"
```

Problem

- How to get lexer to look at more than the first token?
- Answer 1: repeatedly call lexing function
- Answer 2: *action* has to tell it to -- recursive calls. Value of action is token list instead of token.
- Note: already used this with the _ case

Example

```
rule main = parse
  digits '.' digits as f { Float (float_of_string f)
                                 :: main lexbuf}
 | digits as n           { Int (int_of_string n) ::
                                 main lexbuf }
 | letters as s          { String s :: main lexbuf}
 | eof                   { [] }
 | _                     { main lexbuf }
```

Example Results

Ready to lex.

hi there 234 5.2

- : result list = [String "hi"; String "there"; Int 234; Float 5.2]

#

Used Ctrl-d to send the end-of-file signal

Dealing with Comments

First Attempt

```
let open_comment = "(*"
let close_comment = "*)"
rule main = parse
    digits '.' digits as f { Float (float_of_string f)
                                   :: main lexbuf}
  | digits as n             { Int (int_of_string n) ::
                                    main lexbuf }
  | letters as s            { String s :: main lexbuf}
```

Dealing with Comments

```
  | open_comment        { comment  lexbuf}
  | eof                 { [] }
  | _                   { main lexbuf }
and comment = parse
    close_comment       { main lexbuf }
  | _                   { comment lexbuf }
```

Dealing with Nested Comments

```
rule main = parse ...
  | open_comment        { comment 1 lexbuf}
  | eof                 { [] }
  | _                   { main lexbuf }
and comment depth = parse
    open_comment        { comment (depth+1) lexbuf }
  | close_comment       { if depth = 1
                            then main lexbuf
                            else comment (depth - 1)
                                      lexbuf }
  | _                   { comment depth lexbuf }
```