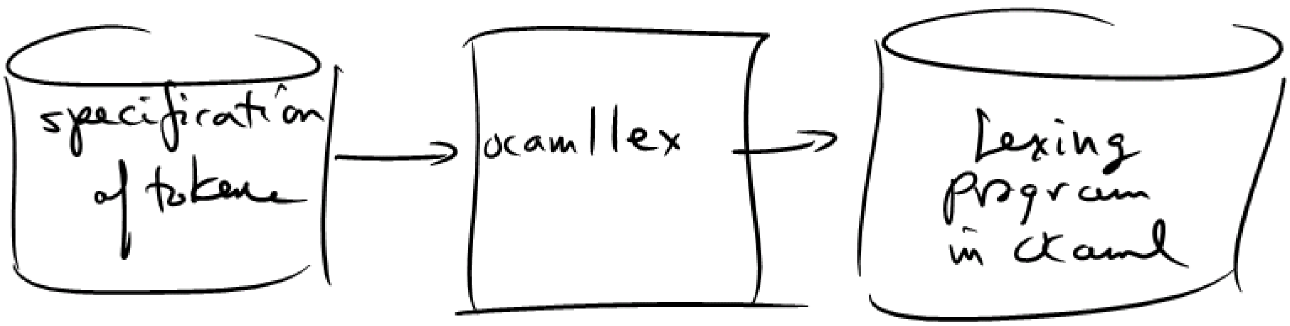
CS421 Lecture 6

- 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 regular expr r defines a "language" – i.e. a set of strings – denoted $\mathcal{L}(r)$.

Regular Expression Examples

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

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

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

Regular Expression Examples

- Keywords

  IF : 'i' 'f'

  ⋮

- Operators

  < : '<'

  << : '<' '<'

  ⋮

- Identifiers

  ('a' | 'b' | ... | 'z' | 'A' | ... | 'Z') ('a' | 'b' | ... | 'Z' | '0' | ... | '9')*

- Int literals

  ('0' | ... | '9') ('0' | ... | '9')*

Abbreviations

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

['a' - 'z'] ⇒ 'a' | 'b' | ... | 'z'

['a' - 'z' '_' 'A' - 'Z'] ⇒ 'a' | ... | 'z' | '_' | 'A' | ... | 'Z'

r+ ⇒ r r*

r? ⇒ r | ""

[^ 'a' - 'z'] ⇒ chars not in ['a' - 'z']

_ ⇒ any single char, is 'a' | 'b' | ...

Regular Expression Example

- Float-point Literal

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

Regular Expression Example

- ~~New~~ C++-Style Comments (//)

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

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

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

Implementing Reg Expr

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

Lexing with Reg Exprs

- Create one large RE:

  List all r.e's for every token
  and separate by '|'

- Then add actions

(cont.)

- Ambiguous cases:
- Two tokens found, one longer
- Two tokens found, the same length

General Input

{ *header* }
let *ident* = *regexp* ...
rule *entrypoint* [*arg1*... *argn*] = parse
      regexp { *action* }
   | ...
   | *regexp* { *action* }
and *entrypoint* [*arg1*... *argn*] = parse ...and ...
{ *trailer* }

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

Example 1: Get token from start of input

```
(* Ex. 1: Return a string giving the type
 *        of the token at the start of the input *)

rule main = parse
   ['0'-'9']+                  { "Int" }
 | ['0'-'9']+'.'['0'-'9']+     { "Float" }
 | ['a'-'z']+                  { "String" }

{ let get_token s =
      let b = Lexing.from_string (s)
      in main b }
```

```
> ocaml
# #use "ex1.ml";;
# get_token
     "71+19";;
> "Int" : string
# get_token
  " 71+19";;
error
```

Example 2: Get token from start of input, return element of data type

```
{   type token = Int | Float | Ident   }

rule main = parse
   ['0'-'9']+                  { Int }
 | ['0'-'9']+'.'['0'-'9']+     { Float }
 | ['a'-'z']+                  { Ident }

{  let get_token s =
      let b = Lexing.from_string (s)
      in main b }
```

get_token "71+9";;
Int : token

Example 3: Get first token in input, after skipping other characters

```
{ type token = Int | Float | Ident }

rule main = parse
   ['0'-'9']+                 { Int }
 | ['0'-'9']+'.'['0'-'9']+    { Float }
 | ['a'-'z']+                 { Ident }
 | _                          { main lexbuf }

{ let get_token s =  ... same as above ... }
```

get_token " 73.9, .. "
Float : token

Example 4: Get first token, and its value, after skipping other characters

```
{ type token = Int of int | Float of float | Ident of
   string }

rule main = parse
    ['0'-'9']+ as s                { Int (int_of_string s) }
  | ['0'-'9']+'.'['0'-'9']+  as s  { Float (float_of_string s) }
  | ['a'-'z']+ as s                { Ident s }
  | _                              { main lexbuf }

{  let get_token s =  ... same as above ... }
```

(['0'-'9']+ as i) '.' (['0'-'9']+ as d)

{ .. i .. d }

Example 5: Get all tokens in input

```
{ type token = Int of int | Float of float | Ident of
   string | EOF }

rule main = parse
   ['0'-'9']+ as s                  { Int (int_of_string s) }
 | ['0'-'9']+'.'['0'-'9']+  as s    { Float (float_of_string s) }
 | ['a'-'z']+  as s                 { Ident s }
 | _                                { main lexbuf }
 | eof                              { EOF }

{  let get_token s = ... same as above ...
```

get_all_tokens " 71.3% x1 " ;;
[Float 71.3 , Ident "x" , Int 1] : token list

continued...

Example 5 (cont.): Get all tokens in input

```
let get_all_tokens s =
    let b = Lexing.from_string (s)
    in let rec get_tokens () =
            match main b with
               EOF -> []
             | t -> t :: get_tokens ()
       in get_tokens ()
}
```

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*

Ocamllex Manual

- More details can be found at

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

Example 6: example 5 using abbreviations

```
{ type token = Int of int | Float of float | Ident of
   string | EOF }
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 +
```

continued...

Example 6 (cont.): example 5 using abbreviations

```
rule main = parse
   digits as s                { Int (int_of_string s) }
 | digits '.' digits as s     { Float (float_of_string s) }
 | letters as s               { Ident s }
 | _                          { main lexbuf }
 | eof                        { EOF }
```

C-style comments

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

continued…

C-style comments (cont.)

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

OCaml-style 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 }
```