CS 421 Lecture 6: Regular expressions

- Announcements

- Lecture outline
  - Regular expressions
  - Ocamllex
Announcements

- MP2 extension and update
  - New due date 1:00pm Friday, June 12
  - Problem 10 has been updated
  - “Valid” old solutions will get full credit

- MP3 has been posted
  - Due 1:00pm Wed, June 17
  - Warning: more work than the first MPs
  - Collaboration is allowed in two-person teams
Overview of Ocamllex

- Automatic OCaml lexer generator

- Specification of tokens via regular expressions

- Ocamllex

- OCaml definition of a lexing function
Regular expressions

- A regular expression is one of
  - $\varepsilon$, a.k.a. ""
  - 'a' for any character a
  - $r_1r_2$, where $r_1$ and $r_2$ are regular expr’s
  - $r_1|r_2$, where $r_1$ and $r_2$ are regular expr’s
  - $r^*$, where r is a regular expr
  - $\emptyset$

- Every regular expr r represents a set of strings, denoted $L(r)$
  - Language of r
Regular expression examples

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

- \( L(\{'a' | 'b'\} 'c') = \{\text{"ac", "bc"}\} \)

- \( L(\{'a' | 'b'\}* 'c') = \{\text{"c", "ac", "bc", "aac", "abc", ...}\} \)
Regular expression examples

- Keywords:
  - ‘c’ ‘a’ ‘s’ ‘e’ | ‘c’ ‘l’ ‘a’ ‘s’ ‘s’ | ...

- Operators
  - ‘<’ | ‘<’ ‘<’ | ‘<’ ‘=’ | ...

- Identifiers
  - (‘a’ | ‘b’ | ... | ‘z’ | ‘A’ | ... | ‘Z’)
    (‘a’ | ‘b’ | ... | ‘z’ | ‘A’ | ... | ‘Z’ | ‘0’ | ‘1’ | ... | ‘9’)*

- Int literals
  - ??
Abbreviations

- "c₁c₂ ... cₙ" => 'c₁' 'c₂' ... 'cₙ'
- ['a'–'z'`#'] => 'a' | 'b' | ... | 'z' | `#'
- ['a' 'w' `#'] => 'a' | 'w' | `#'
- r+ => r(r*)
- r? => r | ""
- [^`a'–'z'] => all chars except 'a' – 'z'
  (complement of 'a' – 'z')
- _ => any single char
Regular expressions examples

- Floating-point literal
  
  \[
  \text{[`0`-`9`]+ . [`0`-`9`]+ ([`e`E] [`+`-`-`]? [`0`-`9`]+)?}
  \]

- Note: \( r^* = (r+) \)?
Regular expression examples

- **C++ style comments (// ...)**
  
  ```
  "//" [^ \n]* \n
  
  ```

- **C style comments (/* ... */)**
  
  ```
  "/*" ([^ *] | *+[^ */])* */
  ```
Implementing regular expressions

- Translate REs to NFAs
- Translate NFAs to DFAs
Lexing with regular expressions

- Create one large RE:
  
  \[
  
  \text{RE for case} \quad \{\text{action for case}\}
  
  | \text{RE for class} \quad \{\text{action for class}\}
  
  | \text{...}
  
  | \text{RE for idents} \quad \{\text{action for idents}\}
  
  | \text{RE for FP lits} \quad \{\text{action for FP lits}\}
  
  | \text{RE for Int lits} \quad \{\text{action for int lits}\}
  
- Then add some actions
Lexing with regular expressions (cont.)

- Ambiguous cases:
  - Two tokens found, one longer
    - Choose the longer one
  - Two tokens found, the same length
    - Choose the earlier reg. expr.
Ocamlllex mechanics

- Put table of regular expressions and corresponding actions (written in Ocaml) into a file
  <filename>.mll

- Call
  ocamlllex <filename.mll>

- Produces Ocaml code for a lexical analyzer in
  <filename>.ml
Ocamlllex input

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

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

header – ocaml defns
Entrypoint – name of gen’d function with args arg1, ..., argn, lexbuf
trailer – ocaml defns
Ocamlllex input

- *header* and *trailer* contain arbitrary Ocaml code put at top and bottom of `<filename>.ml`
- let *ident* = *regexp* ... introduces *ident* for use in later regular expressions
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
}
Ocamlllex output

- `<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 argument being of type `Lexing.lexbuf`
  - `arg1 \ldots argn` are for use in `action`
Ocamllex regular expressions

- ‘a’ : single quoted characters for letters
- _ : matches any character
- eof : special end_of_file marker
- e₁e₂ : concatenation
- “string” : concatenation of a sequence of characters
- e₁|e₂ : choice
Ocamllex regular expressions

- \([c_1-c_2]\) : choice of any character between first and second, inclusive, as determined by character codes
- \([^c_1-c_2]\) : choice of any character NOT in the set
- \(e^*\) : same as before
- \(e+\) : same as \(e\ e^*\)
- \(e?\) : option – was \(e_1|\varepsilon\)
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
  - Example
    
    ```
    ([‘0’–’9’]+ as decpart ‘.’ ([‘0’–’9’]+ as fracpart ...```

Ocamllex manual

- More details can be found at
Example: test.mll

```ml
{ 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 = lower_case | upper_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.\n";
 main newlexbuf }
Example

> ocamllex test.mll
> ocaml
# #use "test.ml"
...
val main : Lexing.lexbuf -> result = <fun>
Ready to lex.
hi there 234 5.6
- : result = String "hi"
#

- What happened to the rest?
Example

```ocaml
# let b = Lexing.from_channel stdin;;
# main b;;
hi 789 there
- : result = String "hi"
# main b;;
- : result = Int 789
# main b;;
- : result = String "there"
```
Problem

- How to get the 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 5.6
- : result list = [String “hi”; String “there”; Int 243; Float 5.6]
#

- Use Ctrl-D to send the end_of_file character
Example: dealing with comments

- **First attempt**

  ```
  let open_comment = "(*)"
  let close_comment = "*)"
  rule main = parse
    ...
    | open_comment { comment lexbuf }
    | eof { [] }
    | _ { main lexbuf }
  and comment = parse
    close_comment { main lexbuf }
    | _ { comment lexbuf }
  ```
Example: dealing with comments

- Second attempt – 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 } |