Programming Languages and Compilers (CS 421)

Elsa L Gunter
2112 SC, UIUC

https://courses.engr.illinois.edu/cs421/sp2023

Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha
Regular Expressions - Review

- Start with a given character set – a, b, c...
- \( \mathcal{L}(\varepsilon) = \{\varepsilon\} \)

Each character is a regular expression

- It represents the set of one string containing just that character
- \( \mathcal{L}(a) = \{a\} \)
Regular Expressions

- If \( x \) and \( y \) are regular expressions, then \( xy \) is a regular expression
  - It represents the set of all strings made from first a string described by \( x \) then a string described by \( y \)

If \( L(x) = \{a, ab\} \) and \( L(y) = \{c, d\} \), then \( L(xy) = \{ac, ad, abc, abd\} \)
If $x$ and $y$ are regular expressions, then $x \lor y$ is a regular expression

- It represents the set of strings described by either $x$ or $y$

If $L(x) = \{a, ab\}$ and $L(y) = \{c, d\}$

then $L(x \lor y) = \{a, ab, c, d\}$
Regular Expressions

- If \( x \) is a regular expression, then so is \( (x) \)
  - It represents the same thing as \( x \)
- If \( x \) is a regular expression, then so is \( x^* \)
  - It represents strings made from concatenating zero or more strings from \( x \)
  - If \( L(x) = \{a,ab\} \) then \( L(x^*) = \{“”, a, ab, aa, aab, abab, …\} \)
- \( \varepsilon \)
  - It represents \{“”\}, set containing the empty string
- \( \Phi \)
  - It represents \{ \} , the empty set
Example Regular Expressions

- \((0 ∨ 1)^*1\)
  - The set of all strings of 0’s and 1’s ending in 1, \{1, 01, 11,\ldots\}

- \(a^*b(a^*)\)
  - The set of all strings of a’s and b’s with exactly one b

- \(((01) ∨ (10))^*\)
  - You tell me

Regular expressions (equivalently, regular grammars) important for lexing, breaking strings into recognized words
Right Regular Grammars

- Subclass of BNF (covered in detail soon)
- Only rules of form
  \(<\text{nonterminal}> ::= <\text{terminal}> <\text{nonterminal}>\) or
  \(<\text{nonterminal}> ::= <\text{terminal}>\) or
  \(<\text{nonterminal}> ::= \epsilon\)
- Defines same class of languages as regular expressions
- Important for writing lexers (programs that convert strings of characters into strings of tokens)
- Close connection to nondeterministic finite state automata – nonterminals \(\cong\) states; rule \(\cong\) edge
Example

- Right regular grammar:
  \[
  <\text{Balanced}> ::= \varepsilon \\
  <\text{Balanced}> ::= 0<\text{OneAndMore}> \\
  <\text{Balanced}> ::= 1<\text{ZeroAndMore}> \\
  <\text{OneAndMore}> ::= 1<\text{Balanced}> \\
  <\text{ZeroAndMore}> ::= 0<\text{Balanced}>
  \]

- Generates even length strings where every initial substring of even length has same number of 0’s as 1’s
Implementing Regular Expressions

- Regular expressions reasonable way to generate strings in language
- Not so good for recognizing when a string is in language
- Problems with Regular Expressions
  - which option to choose,
  - how many repetitions to make
- Answer: finite state automata
- Should have seen in CS374
Example: Lexing

- Regular expressions good for describing lexemes (words) in a programming language
  - Identifier = (a ∨ b ∨ ... ∨ z ∨ A ∨ B ∨ ... ∨ Z) (a ∨ b ∨ ... ∨ z ∨ A ∨ B ∨ ... ∨ Z ∨ 0 ∨ 1 ∨ ... ∨ 9)*
  - Digit = (0 ∨ 1 ∨ ... ∨ 9)
  - Number = 0 ∨ (1 ∨ ... ∨ 9)(0 ∨ ... ∨ 9)* ∨ ~ (1 ∨ ... ∨ 9)(0 ∨ ... ∨ 9)*
  - Keywords: if = if, while = while,...
Lexing

- Different syntactic categories of “words”: tokens

Example:

- Convert sequence of characters into sequence of strings, integers, and floating point numbers.

"asd 123 jkl 3.14" will become:
[String "asd"; Int 123; String "jkl"; Float 3.14]
Lex, ocamllex

- Could write the reg exp, then translate to DFA by hand
  - A lot of work
- Better: Write program to take reg exp as input and automatically generates automata
- Lex is such a program
- ocamllex version for ocaml
How to do it

To use regular expressions to parse our input we need:

- Some way to identify the input string — call it a lexing buffer
- Set of regular expressions,
- Corresponding set of actions to take when they are matched.
How to do it

- The lexer will take the regular expressions and generate a state machine.
- The state machine will take our lexing buffer and apply the transitions...
- If we reach an accepting state from which we can go no further, the machine will perform the appropriate action.
Mechanics

- Put table of reg exp 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`
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
main newlexbuf
}
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 and bottom of `<filename>.ml`

- `let ident = regexp ...` Introduces *ident* for use in later regular expressions
<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

- \( \text{arg1... argn} \) are for use in *action*
Ocamllex Regular Expression

- Single quoted characters for letters: ‘a’
- _: (underscore) matches any letter
- Eof: special “end_of_file” marker
- Concatenation same as usual
- “string”: concatenation of sequence of characters
- \( e_1 / e_2 \): choice - what was \( e_1 \lor e_2 \)
Ocamllex Regular Expression

- \([c_1 - c_2]\): choice of any character between first and second inclusive, as determined by character codes
- \(^{^c_1 - c_2}\): choice of any character NOT in set
- \(e^*\): same as before
- \(e+\): same as \(e \ e^*\)
- \(e?\): option - was \(e \lor \varepsilon\)
Ocamllex Regular Expression

- $e_1 \neq e_2$: 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*
More details can be found at

Version for ocaml 4.07:
https://v2.ocaml.org/releases/4.07/htmlman/lexyacc.html

Current version (ocaml 4.14)
https://v2.ocaml.org/releases/4.14/htmlman/lexyacc.html

(same, except formatting, I think)
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 = upper_case | lower_case
let letters = letter +
```
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_newline ();
  main newlexbuf   }
Example

# use "test.ml";;
...
val main : Lexing.lexbuf -> result = <fun>
val __ocaml_lex_main_rec : Lexing.lexbuf -> int ->
  result = <fun>
hi there 234 5.2
- : result = String "hi"

What happened to the rest?!?
Example

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

- Work on MP8
  - Add a few keywords
  - Implement booleans and unit
  - Implement Ints and Floats
  - Implement identifiers
Problem

- How to get lexer to look at more than the first token at one time?

Answer: *action* has to tell it to -- recursive calls
  - Not what you want to sew this together with ocamlyacc

- Side Benefit: can add “state” into lexing

- 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

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

<table>
<thead>
<tr>
<th>open_comment</th>
<th>{ comment lexbuf }</th>
</tr>
</thead>
<tbody>
<tr>
<td>eof</td>
<td>{ [] }</td>
</tr>
<tr>
<td>_</td>
<td>{ main lexbuf }</td>
</tr>
<tr>
<td>_</td>
<td>{ main lexbuf }</td>
</tr>
</tbody>
</table>

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 }
Dealing with nested comments

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 }
| open_comment { (comment 1 lexbuf}
| eof { [] }
| _ { main lexbuf }
Dealing with nested comments

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 }