Programming Languages and Compilers (CS 421)

Elsa L Gunter

2112 SC, UIUC

http://www.cs.uiuc.edu/class/fa06/cs421/

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

Where We Are Going

- We want to turn strings (code) into computer instructions
- Done in phases
- Turn strings into abstract syntax trees (parse)
- Translate abstract syntax trees into executable instructions (interpret or compile

Lexing and Parsing

- Converting strings to abstract syntax trees done in two phases
  - **Lexing:** Converting string (or streams of characters) into lists (or streams) of tokens (the "words" of the language)
  - **Parsing:** Convert a list of tokens into an abstract syntax tree

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]

Lexing

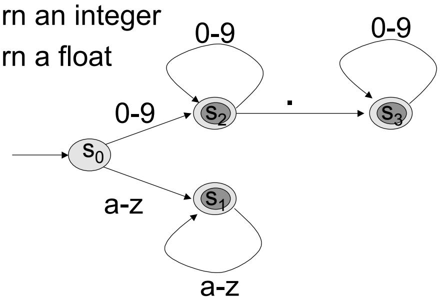
- Each category described by regular expression (with extended syntax)
- Words recognized by (encoding of) corresponding finite state automaton
- Problem: we want to pull words out of a string; not just recognize a single word

Lexing

- Modify behavior of DFA
- When we encounter a character in a state for which there is no transaction
  - Stop processing the string
  - If in an accepting state, return the token that corresponds to the state, and the remainder of the string
  - If not, fail
- Add recursive layer to get sequence

Example

- s_1 return a string
- s_2 return an integer
- s_3 return a float



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 accept 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 Caml 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
}
```

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

Ocamllex Input

- <*filename*>.ml contains one lexing function per *entrypoint*
  - Name of function is name given for *entrypoint*
  - Each entry point becomes a Caml function that takes *n*+1 arguments, the extra implicit last argument being of type Lexing.lexbuf
- *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
- $e_1 \mid e_2$: choice - what was $e_1 \vee e_2$

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 $e_1 \vee \varepsilon$

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 : 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>
val __ocaml_lex_main_rec : Lexing.lexbuf -> int
   -> 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: *action* has to tell it to -- recursive calls
- 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

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 }
```


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 }
```