CISS 445 Programming Languages

41. Lexing with OCAML 1

Dr. Liow

Big Picture

- Recall that there are several steps in translating a program as a string of characters into executable code.
  - Build abstract syntax tree
  - Etc.
- The abstract syntax tree is constructed in two steps:
  - Lexing: Generating tokens (words) of the language. We will focus on this.
  - Parsing: Building the abstract syntax tree.

Lexical Analyzer

- You can write down regex, translate to DFSM (maybe through a NFSM first) by hand and then write code.
- A lot of work.
- So …

Tools

- Compiler construction is a mature area in Computer Science. There are many tools to help you generate code.
- In particular there are many lexers which are programs that accept regexs and generate lexical analyzers.
  - For C there are lex and flex.
  - For OCAML there is ocamlex
  - Etc.

Lexer

- Note that the lexical analyzer must do more than just recognizing substrings.
- Once a substring is recognized, an action must be performed.
- The lexer allows you to specify action for each regex. The lexical analyzer generated is able to recognize substrings (matching regexs) and perform an action for each regex.

Lexer

- The lexical analyzer generated by the lexer is a state machine.
- The state machine takes a buffer and apply transitions. If we reach an accept state from which we can go no further, the state machine will perform the appropriate action.

OCAMLLEX

- You write the specification of the lexical analyzer you want into a file with the name **`<blah>.mml`**
- To generate the state machine type
  `ocamllex <blah>.mml`
- You will get an OCAML program called **`<blah>.ml`** or **`<blah>.mml.ml`**
  i.e., the state machine is an OCAML program

OCAMLLEX

- The format of the .mml file looks like this:

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

OCAMLLEX

- ***`header`***: OCAML code you want to include at the top of the lexical analyzer
- **`let`** ***`ident`*** **`=`** ***`regex`***: defines ***`ident`*** for use in later regex
- ***`trailer`***: OCAML code you want to include at the bottom of the lexical analyzer
- ***`regex`*** **`{`*****`action`*****`}`**: Regular expression and corresponding action if there is a match
- ***`regex`*** **`as`** ***`name`*** **`{`*****`action`*****`}`**: As above with ***`name`*** bound to matching string

OCAMLLEX

- Writing regex in OCAMLLEX:
  - **`'a'`** for character a
  - **`_`** matches any character
  - **`eof`** for end-of-file
  - **`"abc"`** for string of characters **`a`**, **`b`**, and **`c`**
  - Concatenation is the same
  - **`r|s`** for r U s
  - **`[`*`c1`*`-`*`c2`*`]`** match characters from *c1* to *c2* (using ASCII code). Example **`['a'-'z']`**.

OCAMLLEX

- Writing regex in OCAMLLEX:
  - **[^*c1*-*c2*]** match characters NOT in range *c1*-*c2*
  - r* same as before
  - r+ same as rr*
  - r? same as (ε U r) (i.e. optional)
  - r#s matches characters in character set r but not in s. Example **['a'-'z']#['e']**

Example

Create file test.mll:

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

Example (continuation)

```
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 }

{
  let newlexbuf =
    let x = (Lexing.from_string "hi there 123.4 5") in
    let _ = print_string "Ready to lex ...\n" in
    main x
  ;;
}
```

OCAMLLEX

- From the console:

```
> ocamllex test.mml
7 states, 412 transitions, table size 1690 bytes
```

- The file generated is **`test.mml.ml`** or **`test.ml`** (depending on platform).
- **`test.mml.ml`** or **`test.ml`** is an OCAML program.
- Now run OCAML and do #use “test.mml.ml”;;

OCAMLLEX

Now run test.mml.ml (or test.ml):

```
val main : Lexing.lexbuf -> result list = <fun>
val __ocaml_lex_main_rec : Lexing.lexbuf -> int ->
  result list = <fun>
Ready to lex ...
val newlexbuf : result list =
  [String "hi"; String "there"; Float 123.4; Int 5]
#
```

Mutual Recursion

- More information about using OCAMLLEX …

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

Mutual Recursion

- `and`: for mutual recursion. Example:

```
# let rec
    f x =
        let _ = print_string "f" in
        if x = 0 then 1
        else if x mod 2 = 0 then f (x-1)
           else g (x-1)
  and
    g x =
        let _ = print_string "g" in
        if x = 0 then 2
        else if x mod 2 = 0 then g (x-1)
             else f (x-1);;
# f 5;;
fggffg- : int = 2
```

Entry Point

- *entrypoint* [*arg1*... *argn*]
  - An entry point becomes a function. You specify n arguments *arg1*, …, *argn*. There are actually n+1. The last one has type Lexing.lexbuf for the buffer to be lexed.
  - *arg1*, …, *argn*: Used in actions

Entry Point

- Look at first example:

```
<... remove ...>
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 }
{
 let newlexbuf =
  let x = (Lexing.from_string "hi there
          123.4 5") in
  let _ = print_string
          "Ready to lex ...\n" in
  main x;;
}
```

OCAMLLEX

- To run another example just do this at the OCAML prompt:

```
# main (Lexing.from_string "hi mom 123");;
```

- The buffer to lex is the string. You can also specify other channels (Example: keyboard, etc. For keyboard use ctrl-d for eof.)
- The first example has only one entry point.

Multiple Entry Points

```
{ type result = Int of int | Float
  of float | String of string }

let digit = ['0'-'9']
let digits = digit +
let begin_comment = "(*"
let end_comment = "*)"
```

Multiple Entry Points

```
rule main = parse
     digits '.' digits as f { Float (float_of_string f) ::
                                main lexbuf }
   | digits as n            { Int (int_of_string n) ::
                                main lexbuf }
   | begin_comment          { comment lexbuf }
   | eof                    { [] }
   | _                      { main lexbuf }
and
   comment = parse
     end_comment            { main lexbuf }
   | _                      { comment lexbuf }
{
  let newlexbuf =
    let x = (Lexing.from_string "1 2 (* 3 4.5 *) 6 7 8") in
    let _ = print_string "Ready to lex ...\n" in
    main x;;
}
```