ML 4 – A Lexer for OCaml's Type System

CS 421 – Fall 2017

Revision 1.0

Assigned October 26, 2017
Due November 2, 2017
Extension November 4, 2017

1 Change Log

1.0 Initial Release.

2 Overview

To complete this MP, make sure you are familiar with the lectures on both regular expressions and lexing.

After completing this MP, you should understand how to implement a practical lexer using a lexer generator such as Lex. Hopefully you will also gain a sense of appreciation for the availability of lexer generators, instead of having to code a lexer completely from scratch.

The language we are making a parser for is called PicoML, which is basically a subset of OCaml. It includes functions, lists, integers, strings, let expressions, etc.

3 Overview of Lexical Analysis (Lexing)

Recall from lecture that the process of transforming program code (i.e, as ASCII or Unicode text) into an *abstract syntax* tree (AST) has two parts. First, the *lexical analyzer* (lexer) scans over the text of the program and converts the text into a sequence of *tokens*, usually as values of a user-defined disjoint datatype. These tokens are then fed into the *parser*, which builds the actual AST.

Note that it is not the job of the lexer to check for correct syntax - this is done by the parser. In fact, our lexer will accept (and correctly tokenize) strings such as "`if if let let if if else`", which are not valid programs.

4 Lexer Generators

The tokens of a programming language are specified using regular expressions, and thus the lexing process involves a great deal of regular-expression matching. It would be tedious to take the specification for the tokens of our language, convert the regular expressions to a DFA, and then implement the DFA in code to actually scan the text.

Instead, most languages come with tools that automate much of the process of implementing a lexer in those languages. To implement a lexer with these tools, you simply need to define the lexing behavior in the tool's specification language. The tool will then compile your specification into source code for an actual lexer that you can use. In this MP, we will use a tool called *ocamllex* to build our lexer.

4.1 *ocamllex* specification

The lexer specification for *ocamllex* is documented here:

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

What follows below is only the short version. If it doesn't make sense, or you need more details, consult the link above. You will need to become especially familiar with *ocamllex*'s regular expression syntax.

ocamllex's lexer specification is slightly reminiscent of an OCaml `match` statement:

```
rule myrule = parse
  | regex1 { action1 }
  | regex2 { action2 }
...
```

When this specification is compiled, it creates a recursive function called `myrule` that does the lexing. Whenever `myrule` finds something that matches *regex1*, it consumes that part of the input and returns the result of evaluating the expression *action1*. In our code, the lexing function should return the token it finds.

Here is a quick example:

```
rule mylexer = parse
   | [' ' '\t' '\n']          {mylexer lexbuf}
   | ['x' 'y' 'z']+ as thingy { ... thingy ... }
```

The first rule says that any whitespace character (either a space, tab, or newline) should be ignored. `lexbuf` is a special object that represents "the rest of the input" - the stuff after the whitespace that was just matched. By saying `mylexer lexbuf`, we are recursively calling our lexing rule on the remainder of the input and returning its result. Since we do nothing with the whitespace that was matched, it is effectively ignored.

The second rule shows a *named* regex. By naming the regex like this, whatever string matched the regex is bound to the name `thingy` and available inside the action code for this rule (as is `lexbuf` as before). Note that you can also name just *parts* of the regex. The return value from this action should somehow use the value of `thingy`.

You can also define multiple lexing functions - see the online documentation for more details (they are referred to as "entrypoints"). Then from the action of one rule, you can call a different lexing function. Think of the lexer on the whole as being a big state machine, where you can change lexing behaviors based on the state you are in (and transition to a different state after seeing a certain token). This is convenient for defining different behavior when lexing inside comments, strings, etc.

5 Provided Code

common.cmo contains the definition of the type for our tokens, a type, and exceptions to support the completion of Problem 2.

mp4.mll is the skeleton for the lexer specification. `token` is the name of the lexing rule that is already partially defined. You may find it useful to add your own helper functions to the header section. The footer section defines the `get_all_tokens` function that drives the lexer, and should not be changed. Modify ***mp4.mll***, and enter it in PrairieLearn as usual.

6 Problems

1. (6 pts) Define all the keywords and operator symbols of the OCaml type system. Each of these tokens is represented by a constructor in our disjoint datatype.

Token	Constructor
~	NEG
+	PLUS
-	MINUS
*	TIMES
/	DIV
+.	DPLUS
-.	DMINUS
*.	DTIMES
/.	DDIV
^	CARAT
<	LT
>	GT
<=	LEQ
>=	GEQ
=	EQUALS
<>	NEQ
|	PIPE
->	ARROW
;	SEMI
;;	DSEMI
::	DCOLON
@	AT
[]	NIL
let	LET
rec	REC
and	AND
end	END
in	IN
if	IF
then	THEN
else	ELSE
fun	FUN
mod	MOD
raise	RAISE
try	TRY
with	WITH
not	NOT
&&	LOGICALAND
||	LOGICALOR
[	LBRAC
]	RBRAC
(	LPAREN
)	RPAREN
,	COMMA
_	UNDERSCORE
true	TRUE
false	FALSE
()	UNIT

Each token should have its own rule in the lexer specification. Be sure that, for instance, "<>" is lexed as the NEQ token and not the two tokens LT and GT. (Remember that the regular expression rules are tried by the "longest

match" rule first, and then by the input from top to bottom).

```
# get_all_tokens "let = in + , ; ( - )";;
- : Common.token list =
[LET; EQUALS; IN; PLUS; COMMA; SEMI; LPAREN; MINUS; RPAREN]
```

2. (10 pts)

   a. Implement (decimal) integers using regular expressions. There is a token constructor `INT` that takes an integer as an argument. Do not worry about negative integers, but make sure that integers have at least one digit. You may use `int_of_string : string -> int` to convert strings to integers.

   ```
   # get_all_tokens "42 0 7";;
   - : Common.token list = [INT 42; INT 0; INT 7]
   ```

   b. Implement a representation of binary numbers using regular expressions. In particular, binary numbers are represented with the prefix `0b`, followed by at least one symbol, where each symbol can be one of two unique symbols: 0 to 1. There is a token constructor `INT` that takes a (decimal) integer as an argument. You may use `int_of_string : string -> int` to convert strings to integers.

   ```
   # get_all_tokens "0b110100101";;
   - : Common.token list = [INT 421]
   ```

   c. Implement a representation of hexadecimal numbers using regular expressions. In particular, hexadecimal numbers are represented with the prefix `0x`, followed by at least one symbol, where each symbol can be one of 16 unique symbols: 0 to 9 and `a` to `f`. There is a token constructor `INT` that takes a (decimal) integer as an argument.You may use `int_of_string : string -> int` to convert strings to integers.

   ```
   # get_all_tokens "0x8844ffaa11";;
   - : Common.token list = [INT 585273158161]
   ```

   d. Implement floats using regular expressions. There is a token `FLOAT` that takes a float as an argument. Floats must have a decimal point and at least one digit before the decimal point (and optionally may have digits after). You may use `float_of_string : string -> float` to convert strings to floats.

   ```
   # get_all_tokens "3.14 100.5 1.414";;
   - : Common.token list = [FLOAT 3.14; FLOAT 100.5; FLOAT 1.414]
   ```

   e. Implement a representation of floats which support scientific notation. In particular, the float representation described in Question 2d is augmented with an optional *exponent part*. The exponent part is defined as the character `e` followed by one or more digits. You may use `float_of_string : string -> float` to convert strings to floats.

   ```
   # get_all_tokens "2.7e10";;
   - : Common.token list = [FLOAT 27000000000.]
   ```

3. (3 pts) Implement booleans and the unit expression. The relevant constructors are `UNIT` and `BOOL`.

   ```
   # get_all_tokens "true false () true";;
   - : Mp8common.token list = [TRUE; FALSE; UNIT; TRUE]
   ```

4. (6 pts) Implement identifiers. An identifier is any sequence of letter and number characters, along with `_` (underscore) or `'` (single quote, or prime), that begins with a lowercase letter. Remember that if it is possible to match a token by two different rules, the longest match will win over the shorter match, and then if the string lengths are the same, the first rule will win. This applies to identifiers and certain alphabetic keywords. Use the `IDENT` constructor, which takes a string argument (the name of the identifier).

Identifier Tokens	Not Identifier Tokens
asdf1234	1234asdf
abcd_	_123
a'	then
xABC_d	'abc
a'b_c'd'''e	A_Nice_Name

```
# get_all_tokens "this is where if";;
- : Common.token list = [IDENT "this"; IDENT "is"; IDENT "where"; IF]
```

5. (8 pts) Implement comments. Line comments in PicoML are made up of two slashes, "`//`". Block comments in PicoML begin with "`(*`" and end with "`*)`", and can be nested. An exception should be raised if the end of file is reached while processing a block comment; this can be done by associating the following action with this condition:

   ```
   raise (Failure "unmatched open comment")
   ```

   Any time you raise `Failure` for such an error, you must use the text `"unmatched open comment"` verbatim in order to get points. Furthermore, an unmatched close comment ("`*)`") should also cause a `Failure "unmatched closed comment"` exception to be raised.

   The easiest way to handle block comments will be to create a new entry point, like we saw in lecture, since we will need to keep track of the depth. For line comments, a new entry point is not needed – instead, you should just be able to craft a regular expression that will consume the rest of the line. There is no token for comments, since we just discard what is in them. A block comment may contain the character sequence "`//`" and a line comment may contain either or both of "`(*`" and "`*)`".

   ```
   # get_all_tokens "this (* is a *) test";;
   - : Mp8common.token list = [IDENT "this"; IDENT "test"]
   # get_all_tokens "this // is a test";;
   - : Mp8common.token list = [IDENT "this"]
   # get_all_tokens "this // is a\n test";;
   - : Mp8common.token list = [IDENT "this"; IDENT "test"]
   # get_all_tokens "this (* is (* a test *)";;
   Exception: Failure "unmatched open comment".
   ```

6. (25 pts) Implement strings. A string begins with a double quote (`"`), followed by a (possibly empty) sequence of printable characters and escaped sequences, followed by a closing double quote (`"`).

   A printable character is one that would appear on an old fashioned `qwerty` keyboard on a mechanical typewriter, including the space. Here, we must exclude `"` and \ because they are given special meaning, as described below. The printable characters include the 52 uppercase and lowercase alphabetics, the ten digits, space, and 30 of the 32 special characters, excluding the `"` and \.

   Note that a string cannot contain an unescaped quote (`"`) because it is used to end the string. However, it can contain the two character sequence representing an escaped quote (\`"`). More generally, we use \ to begin an escaped sequence.

   Specifically, you must recognize the following two-character sequences that represent escaped characters:

\\
\'
\"
\t
\n
\r
\b
space

Each such two-character sequence must be converted into the corresponding single character. For example, the two-character string `" \ t"` (where the first character is the ASCII code for \, 92) must become the single character `' \ t '` (ASCII character number 9).

Additionally, you must handle the following escaped sequence:

ddd

The above escaped sequence is used to escape specific ASCII integer values. *ddd* represents an integer value between 0 and 255. Your job is to map the integer to its single character value. For example, the escaped character \100 is the character 'd'.

Lastly, to allow splitting long string literals across lines, the sequence \ followed by a newline, followed by any number of spaces and tabs at the beginning of the next line, is ignored inside string literals.

The escape character \ cannot legally precede any other character in a string.

You will probably find it easiest to create a new entry point, possibly taking an argument, to handle the parsing of strings.

You will also find the following function useful:

```
String.make : int -> char -> string
```

where `String.make n c` creates the string consisting of n copies of c. In particular, `String.make 1 c` converts c from a character to a string.

You may also use `char_of_int : int -> char` and `int_of_string : string -> int`.

Note that if you test your solution in OCaml, you will have to include extra backslashes to represent strings and escaped characters.

For example:

```
# get_all_tokens "\"some string\"";;
- : Common.token list = [STRING "some string"]
# get_all_tokens "\" she said, \\\"hello\\\"\"";;
- : Common.token list = [STRING " she said, \"hello\""]
# get_all_tokens "\" \\100 \\001 \"";;
- : Common.token list = [STRING " d \001 "]
# get_all_tokens
 "\"a line \\n starts here; indent \\t starts here next string\" \"starts here\"";;
- : Common.token list =
[STRING "a line \n starts here; indent \t starts here next string";
 STRING "starts here"]
```

7 Compiling, Testing & Handing In

7.1 Compiling & Testing

To compile your lexer specification to OCaml code, use the command

```
ocamllex mp4.mll
```

This creates a file called `mp4.ml` (note the slight difference in names). Then you can run tests on your lexer in OCaml using the `token` function that is already included in `mp4.mll`. To see all the tokens producible from a string, use `get_all_tokens`.

```
# #load "common.cmo";;
# #use "mp4.ml";;
...
# get_all_tokens "some string to test";;
- : Common.token list =
[IDENT "some"; IDENT "string"; IDENT "to"; IDENT "test"]
```