MP 7 – A Lexer for MicroML

CS 421 – su 2011

Revision 1.0

Assigned July 16, 2011
Due July 24, 2011, at 23:59pm
Extension 48 hours (20% penalty)

1 Change Log

1.0 Initial Release.

2 Overview

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

The language for which we are making a parser is called MicroML, which is a subset of SML. It includes functions, lists, integers, strings, let expressions, declarations, 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/manual026.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 well as `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

mp7common.ml contains the definition of our tokens and some additional exceptions and types.

mp7-skeleton.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, and the footer section defines the **get_all_tokens** function that drives the lexer, and should not be changed. Rename this file to ***mp7.mll***, modify it, and hand it in.

6 Problems

1. (7 pts) Define all the keywords and operator symbols of our MicroML language. 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
::	DCOLON
@	AT
nil	NIL
let	LET
local	LOCAL
val	VAL
rec	REC
and	AND
end	END
in	IN
if	IF
then	THEN
else	ELSE
fun	FUN
fn	FN
op	OP
mod	MOD
raise	RAISE
handle	HANDLE
with	WITH
not	NOT
andalso	ANDALSO
orelse	ORELSE
[	LBRAC
]	RBRAC
(	LPAREN
)	RPAREN
,	COMMA
_	UNDERSCORE

Each token should have its own rule in the lexer specification. Be sure that, for instance, "<>" is lexed as the `NEQ` token and not 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 + , ; ( - )";;
- : Mp7common.token list =
[LET; EQUALS; IN; PLUS; COMMA; SEMI; LPAREN; MINUS; RPAREN]
```

2. (8 pts) Implement integers and reals using regular expressions. There is a token constructor `INT` that takes an integer as an argument, and a token `REAL` that takes a real as an argument. Do not worry about negative integers, but make sure that integers have at least one digit. Reals must have a decimal point and at least one digit before the decimal point, and one after. You may use `int_of_string : string -> int` and `real_of_string : string -> real` to convert strings to integers and reals respectively.

```
# get_all_tokens "42 100.5 0";;
- : Mp7common.token list = [INT 42; REAL 100.5; INT 0]
```

3. (6 pts)

   Implement booleans and the unit expression. The relevant constructors are `UNIT` and `BOOL`.

```
# get_all_tokens "true false ()";;
- : Mp7common.token list = [BOOL true; BOOL false; UNIT]
```

4. (10 pts)

   Implement identifiers. An identifier is any sequence of letter and number characters, along with _ (underscore) or ' (single quote, or prime), that begins with either a lowercase or an uppercase 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
ABC_d	'abc

```
# get_all_tokens "this is where if";;
- : Mp7common.token list = [IDENT "this"; IDENT "is"; IDENT "where"; IF]
```

5. (20 pts)

   Implement comments. Line comments in MicroML are made up of two slashes, "`//`". Block comments in MicroML 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 comment")
```

   Any time you raise `Failure`, you must use the text `"unmatched comment"` verbatim in order to get points. Furthermore, an unmatched close comment ("`*)`") should also cause a `Failure "unmatched 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";;
- : Mp7common.token list = [IDENT "this"; IDENT "test"]
# get_all_tokens "this // is a test";;
- : Mp7common.token list = [IDENT "this"]
# get_all_tokens "this // is a\n test";;
- : Mp7common.token list = [IDENT "this"; IDENT "test"]
# get_all_tokens "this (* is (* a test *)";;
Exception: Failure "unmatched comment".
```

6. (25 pts)

   Implement strings. A string begins with a double quote(`"`), followed by a sequence of characters, followed by a closing double quote(`"`).

   Note that a string cannot contain an unescaped quote (`"`). However, it can contain the two character sequence representing an escaped quote (\").

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

   \\
   \"
   $\backslash t$
   $\backslash n$
   $\backslash r$

   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 two escaped sequences:

   $\backslash f_1 ... f_n \backslash$
   $\backslash ddd$

   The first is used to help with formatting. $f_1 ... f_n$ represents any non-zero number of whitespace formatting that includes tabs ($\backslash t$), spaces (" "), and new lines. Make sure to consider whether the user explicitly put $\backslash n$ in the string or if they used the return key. When passing the string to the parser, the whitespace formatting f_i's and the backslashes should be ignored.

   The second additional 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 $\backslash 100$ is the character 'd'.

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

   You will also find the following library functions useful:

```
String.make : int -> char -> string
String.sub : string -> int -> int -> 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. For substring manipulation, use `String.sub`. `String.sub s i n c` creates a string using the n characters after position i in the string s.

   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\"";;
- : Mp7common.token list = [STRING "some string"]
# get_all_tokens "\" she said, \\\"hello\\\"\"";;
- : Mp7common.token list = [STRING " she said, \"hello\""]
# get_all_tokens
  "\"a line \\n starts here; indent \\t starts here next string\" \"starts here\"";;
- : Mp7common.token list =
[STRING "a line \n starts here; indent \t starts here next string";
 STRING "starts here"]
# get_all_tokens "\"Hello, \\ \n \\ world! \"";;
- : Mp7common.token list = [STRING "Hello,  world! "]
# get_all_tokens "\" \\100 \\001 \"";;
- : Mp7common.token list = [STRING " d \001 "]
```

7 Extra Credit

7. (8 pts) Keep track of line and column numbers, and, upon failure of lexing due to errant or unmatched comments, raise exceptions that indicate not only the error encountered, but also the column and line number of the error.

   To keep track of character and line numbers, you will need help from some side-effecting functions. The current line and character count are stored as `int ref`s. Don't worry about the exact nature of a `ref`, as we have not covered it in class, but just know that it is the type of mutable objects in OCaml.

   The following references and functions are defined at the top of `mp7-skeleton.mll`:

   ```
   let line_count = ref 1
   let char_count = ref 1

   let cinc n = char_count := !char_count + n
   let linc n = line_count := (char_count := 1; !line_count + n)
   ```

   `line_count` and `char_count` are references to the current line and character counts, respectively. You will need to update these as your program progresses.

   The `!` operator extracts the value of a references. So, to access the current line count, use `!line_count`, and to access the current character count, use `!char_count`. To increment `line_count`, call `linc n` where `n` is the number of lines to increment the count by. Note that when you increment the `line_count` via this function, the character count is automatically reset to 1. To increment `char_count`, call `cinc n` where `n` is the number of characters to increment the count by. You will need to call `cinc` as part of the action for most of your rules, and you will need to call `linc` when you encounter a newline. The real challenge comes in raising exceptions upon encountering unmatched or errant comment markers. The following datatype, hidden in `mp7common.cmo`, defines the exceptions that you will need to raise in place of the `raise (Failure "unmatched comment")`s that you had before:

   ```
   type position = {line_num : int; char_num : int}

   exception OpenComm of position
   exception CloseComm of position
   ```

   If...

   - ... `eof` is reached before `*)` inside a comment, raise `OpenComm pos` where `pos` is the position of the first character of the unmatched `(*`. This will require some extra book-keeping.

- ... `*)` is encountered outside of a comment, raise `CloseComm pos` where `pos` is the position of the first character of the errant `*)`.

Here is some sample output:

```
# get_all_tokens "aaaa\nbbbb\nc(*\n";;
Exception: Mp7common.OpenComm {line_num = 3; char_num = 2}.
# get_all_tokens "let\n123\n\"hi\\n\"*)\n";;
Exception: Mp7common.CloseComm {line_num = 3; char_num = 7}.
```

8 Compiling, Testing & Handing In

8.1 Compiling & Testing

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

```
ocamllex mp7.mll
```

This creates a file called `mp7.ml` (note the slight difference in names). Then you can run tests on your lexer in OCaml using the `token` function that defined by the main rule in `mp7.mll`. To see all the tokens producible from a string, use `get_all_tokens` .

```
# #load "mp7common.cmo"
# #use "mp7.ml";;
...
# get_all_tokens "some string to test";;
- : Mp7common.token list = [ some list of tokens ]
```