MP 3 – A Lexer for MiniJava

CS 421 – Spring 2011

Revision 1.1

Assigned Tuesday, February 1, 2011
Due Monday, February 7, at 23:59
Extension 48 hours (penalty 20% of total points possible)
Total points 50 (+5 points extra credit)

1 Change Log

1.1 Clarify the definition of Problem 3.

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 we are making a parser for is called MiniJava, which is basically a subset of Java.

3 Collaboration

Collaboration is allowed on this assignment.

4 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 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 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 == if if else`" which are not valid programs.

5 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.

5.1 *ocamllex* specification

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 }
...
```

where `rule` and `parse` are required words, but `myrule` can be any OCaml identifier (as long as it starts with a lower-case letter). By the way, even though ocamllex uses the word "parse," it is a misnomer; ocamllex generates *lexers*, not parsers.

When this specification is compiled, it creates a 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. The token type is described in the problem description below, and will also be included in file `mp3common.mli`.

The syntax and meaning of regular expressions are given in the ocamllex manual, and are described in Lecture 6.

An "action" is any expression of type token, giving the token to be returned when the corresponding regular expression is matched. For example, if the action is `{ LSHIFT }`, the lexer will return the `LSHIFT` token. Within an action, you may refer to the variable `lexbuf`, which represents the part of the input remaining after the token is removed. Thus, if you simply want to ignore the token — as you might want to do, for example, with comments — the action would be `myrule lexbuf` — meaning, scan the *remaining* input and return whatever token it returns.

To bind part of an expression to a variable which you can use in the action, use the `as` keyword.

For example:

```
  | '+' (_* as s) '+'              { s }
```

recognizes strings surrounded by '+', and returns the characters inside.

Here is a quick example:

```
rule tokenize = parse
    | [' ' '\t' '\n']  { tokenize lexbuf }
    | "<<"             { LSHIFT }
    | (['x' 'y' 'z']+ as s)   { IDENTIFIER s }
    | _                { failwith ("Illegal character") }
```

The function `tokenize` matches the following: (1) a single whitespace character; it ignores it and returns the token obtained from the remainder of the input (which may also begin with whitespace that will be ignored similarly); or (2) a left-shift operator; it returns the `LSHIFT` token; or (3) a sequence of one or more of the characters x, y, and z; it returns the `IDENTIFIER` token constructed from the matched sequence. In all other cases, it throws an exception.

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. The functions you define in this way can also have additional arguments, as in :

```
rule aux n = parse
   | regex1   { ... aux (n+1) lexbuf ... }
   ...
```

Technically, the lexing function `tokenize` created by the example above has type `lexbuf -> token`. How do you turn the input string into a `lexbuf`? There is a function for that called `Lexing.from_string: string -> lexbuf`. But you don't have to worry about that, because we'll provide the following functions (in `mp3lex-skeleton.mll`):

```
   (* lextest: string -> token *)
   let lextest s = tokenize (Lexing.from_string s)

   (* get_all_tokens s: string -> token list *)
   let get_all_tokens s =
      let buf = Lexing.from_string (s^"\n")
      in let rec aux () = match tokenize buf with
                            EOF -> []
                          | t   -> t:: aux ()
   in aux ()
```

`lextest s` returns the first token from `s`, while `get_all_tokens s` returns the list of all tokens in `s`. (The definition of `get_all_tokens` — or, more specifically, of `aux` — must be confusing, because we just keep calling `aux ()` repeatedly and it is not clear why we are not in a loop. The call `tokenize buf` not only finds the next token in `buf`, it also has a side effect on `buf`, removing the matched characters. Thus, in effect, each time this call is made, `buf` gets shorter, until it only contains the `EOF` character.)

6 Provided Code

mp3common.mli is a special file containing the definition of tokens, posted separately on the MP3 page. Please examine but do not modify it.

mp3common.cmi is a binary version of the same file. It must be in the folder where you execute your code.

mp3lex-skeleton.mll is the skeleton for the lexer specification (please rename it to `mp3lex.mll`). `token` is the name of the lexing rule that is already partially defined. There are also a few user-defined functions in the header and footer section that you may find useful. This is the file you will modify and hand in.

7 Problems

1. (5 pts) Define all the keywords and operator symbols of our MiniJava language. Each of these tokens is represented by a constructor in our disjoint datatype (defined in `mp3common.mli`).

Token	Constructor
boolean	BOOLEAN
break	BREAK
case	CASE
class	CLASS
continue	CONTINUE
else	ELSE
extends	EXTENDS
float	FLOAT
default	DEFAULT
int	INT
new	NEW
if	IF
public	PUBLIC
switch	SWITCH
return	RETURN
static	STATIC
while	WHILE
this	THIS
null	NULL_LITERAL
(	LPAREN
)	RPAREN
{	LBRACE
}	RBRACE
[	LBRACK
]	RBRACK
;	SEMICOLON
,	COMMA
.	DOT
=	EQ
<	LT
!	NOT
:	COLON
&&	ANDAND
||	OROR
+	PLUS
-	MINUS
*	MULT
/	DIV
&	AND
|	OR

Each token should have its own rule in the lexer specification. Be sure that, for instance, "`&&`" is lexed as the `ANDAND` token and not two `AND` tokens (remember that the regular expression rules are tried by the "longest match" rule first, and then by the input from top to bottom).

There are token categories for integers, booleans, strings, and floats, specifically: `INTEGER_LITERAL of int`, `BOOLEAN_LITERAL of boolean`, `STRING_LITERAL of string`, and `FLOAT_LITERAL of float`. The following problems will handle these tokens. There is no token for comments — because comments don't produce a token — but you will need to recognize and discard them (as we did with whitespace in the example above).

2. (6 pts) Recognize integer constants. An integer is a string of 1 or more decimal digits. You may use `int_of_string: string -> int` to convert strings to integers, and the constructor `INTEGER_LITERAL` to convert integers to tokens. (Note that negative integer constants are not individual tokens, but instead consist of two tokens, `MINUS` and integer literal.)

3. (8 pts) Implement floats. A numeric sub-string **(with at least one digit)** represents a float (as opposed to an integer) if the following hold:

   - It contains exactly one decimal point (and at least one digit to the left or to the right of that decimal point)
   - The character `e`, optionally followed by a + or −, then followed by an integer can be appended.

   There is a token constructer `FLOAT_LITERAL` that takes a float as an argument. (In real Java, the tokens we are describing are *double* literals, but in our version of Java we only have the `float` type, so we are omitting the "`f`" that is used in Java to make a float literal.)

   You may use `float_of_string: string -> float` to convert strings to floats.

4. (3 pts)

   Implement booleans. The patterns to be matched are "false" and "true". The relevant constructor is `BOOLEAN_LITERAL`.

5. (6 pts)

   Recognize identifiers. An identifier is a letter followed by any number of alphanumeric characters (letters and decimal digits). Use the `IDENTIFIER` constructor, which takes a string argument (the name of the identifier).

6. (8 pts)

   Implement strings. A string begins with a double quote (`"`), followed by a sequence of characters, followed by a closing double quote (`"`). Double quotes, line breaks and backslashes may not occur within string literals.

   You may find it most convenient to use ocamllex's `[^ ...]` syntax to create a character set which contains all characters except those specified in the `...`.

   Hint: You probably want to use `StringCharacter`, which already does much of the work for you.

```
# get_all_tokens  "\"some string\"";;                                  ]
- : Mp3common.token list = [Mp3common.STRING_LITERAL "some string"]
```

   We don't want the quotation marks which surround the string in the input as part of the string. One way you can ignore these by binding to a name only the part of the regular expression (with the `as` keyword – see the *ocamllex* documentation). Or maybe you can find another way to do it. In any case, don't include the delimiting quotation marks in the token that is returned.

7. (3 pts)

   Implement line comments. Line comments in MiniJava are made up of two slashes, "`//`" and include all characters up to the next newline.

8. (4 pts)

   Implement traditional C-style comments. C-style comments begin with "`/*`" and end with "`*/`", and contain any characters, including newlines. Unlike OCaml's comments, C-tyle comments **cannot** be nested.

   You must handle C-style comments by creating an additional lexing function after the original lexing function, `tokenize`. You may want to refer to lecture notes for an example. You should raise an exception if `eof` is reached before the end of a comment; e.g., `failwith "unterminated comment"`.

9. (7 pts)

   Implement nested comments (ala OCaml) using `(*` and `*)`. Again, you will need to create an additional lexing function. You should raise an exception if `eof` is reached before the end of a comment; e.g., `failwith "unterminated comment"`.

10. (5 pts extra credit)

    You may also implement JavaDoc comments for extra credit. A JavaDoc comment begins with `/**` and ends with `*/`, and contains any characters, including newlines. Unlike ordinary comments, JavaDoc comments should not be discarded. Use the `JAVADOC` constructor, which takes a string argument.

```
# get_all_tokens "/** some description  */";;
- : Mp3common.token list = [Mp3common.JAVADOC " some description  "]
```

    When you capture its contents, you don't need to worry about tags, formattings and so on. If a line starts with any number of whitespaces followed by `*`, however, you should discard the whitespaces and `*`. Except this rule, you can simply copy all characters between `/**` and `*/`, including newlines.

    Here is an example:

```
# get_all_tokens "/**\n1\n 2\n*3\n *4\n * 5\n */";;
- : Mp3common.token list = [Mp3common.JAVADOC "\n1\n 2\n3\n4\n 5\n "]
```

    That's it! See, it wasn't that bad, was it?

8 Compiling, Testing & Handing In

To compile your lexer specification (renamed to `mp3lex.mll`) to OCaml code, use the command:

```
make
```

Now you can run `./grader`.

But, you also now have a file called `student.ml` (note that this file ends in .ml, not .mll). Then you can run tests on your lexer in OCaml using the `tokenize` function that is already included in `mp3lex.mll`. To see all the tokens producible from a string, use `get_all_tokens` .

```
# #use "student.ml";;
...
# get_all_tokens "some string to test";;
- : Picomlparse.token list = [ some list of tokens ]
```

To test the solution, use `solution.cmo` instead:

```
# #load "solution.cmo";;
# open Solution;;
...
# get_all_tokens "some string to test";;
- : Picomlparse.token list = [ some list of tokens ]
```