MP 6 – A Lexer for PicoML

CS 421 – Summer 2013

Revision 1.0

Assigned June 27, 2013
Due July 5, 2013, 11:59 PM
Extension 48 hours (20% penalty)

1 Change Log

1.0 Initial Release.

2 Overview

To complete this MP, make sure you are familiar with the lectures on DFAs and NFAs, regular expressions, and lexing. After completing this MP, you should understand how to implement a practical lexer using a lexer generator. The language we are making a lexer for is called PicoML, a simple 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 values of a user-defined variant 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 but consist of valid tokens.

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`

We will give an overview here. *ocamllex*'s lexer specification is 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*. Generally, 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 data { ... data ... }
```

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 – whatever comes after the whitespace that was just matched. By writing `mylexer lexbuf`, we recursively call our lexing rule on the remainder of the input and return 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 with the `as` keyword, whatever string matched the regex is bound to the name `data` and available inside the action code for this rule (as is `lexbuf`). Note that you can also name just *parts* of the regex. The return value from this action should somehow use the value of `data`.

You can also define multiple lexing functions (also called *entry points*) – see the lecture slides or online documentation for more details. In the action of one rule, you can call a different lexing function. Think of the whole lexer as being a giant state machine that changes lexing behaviors based on the state it is in. This is convenient for defining different behavior when lexing inside comments, strings, etc.

5 Provided Code

mp6parse.cmi is a special binary object file containing the definition of our tokens. This file must be present for the lexer specification to compile. Please don't modify it.

mp6common.cmo contains additional functions and datatypes.

mp6-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. The footer section defines the `get_all_tokens` function that drives the lexer, and should not be changed. Rename this file to ***mp6.mll***, modify it, and hand it in.

6 Compiling, Testing & Handing In

6.1 Compiling & Testing

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

```
ocamllex mp6.mll
```

This creates a file called `mp6.ml` (note that it takes in a `.mll` file and produces a `.ml` file). Then you can run tests on your lexer in OCaml using the `token` function that is already included in `mp6.mll`. To see all the tokens producible from a string, use `get_all_tokens` .

```
# #load "mp6common.cmo";;
# #use "mp6.ml";;
...
# get_all_tokens "some string to test";;
- : Mp6parse.token list = [ some list of tokens ]
```

7 Problems

1. (7 pts) Define all the keywords and operator symbols of our PicoML language. Each of these tokens is represented by a constructor in our disjoint datatype.

Token	Constructor
`;;`	DBLSEMI
`+`	PLUS
`-`	MINUS
`*`	TIMES
`/`	DIV
`+.`	DPLUS
`-.`	DMINUS
`*.`	DTIMES
`/.`	DDIV
`^`	CARAT
`**`	EXP
`<`	LT
`>`	GT
`<=`	LEQ
`>=`	GEQ
`=`	EQUALS
`&&`	AND
`||`	OR
`|`	PIPE
`->`	ARROW
`::`	DCOLON
`let`	LET
`rec`	REC
`;`	SEMI
`in`	IN
`if`	IF
`then`	THEN
`else`	ELSE
`fun`	FUN
`[`	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 `DBLSEMI` token and not two `SEMI` tokens (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 ;; + ** , ; ( - )";;
- : Mp6parse.token list =
[LET; EQUALS; IN; DBLSEMI; PLUS; EXP; COMMA; SEMI; LPAREN; MINUS; RPAREN]
```

2. (8 pts) Implement integers and floats using regular expressions. There is a token constructer `INT` that takes an integer as an argument, and a token `FLOAT` that takes a float as an argument. Do not worry about negative integers, but make sure that integers have at least one digit. Floats must have a decimal point and at least one digit before the decimal point. They do not require any digits after the decimal point. You may use `int_of_string` and `float_of_string` to convert strings to integers and floats respectively.

```
# get_all_tokens "42 100.5 0";;
- : Mp6parse.token list = [INT 42; FLOAT 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 ()";;
- : Mp6parse.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 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 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`
`int`	`Int`

```
# get_all_tokens "this is where if";;
- : Mp6parse.token list = [IDENT "this"; IDENT "is"; IDENT "where"; IF]
```

5. (20 pts) Implement comments. Line comments in PicoML begin with two slashes, "`//`", and continue to the end of the line (that is, until the character '\n'). 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 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.

Lastly, a "`**)`" is a *super* close comment. It closes all block comments opened previously, provided there is at least one unclosed open comment preceding it. If there is no preceding unmatched open comment, then a Failure exception must 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";;
- : Mp6parse.token list = [IDENT "this"; IDENT "test"]
# get_all_tokens "this // is a test";;
- : Mp6parse.token list = [IDENT "this"]
# get_all_tokens "this // is a\n test";;
- : Mp6parse.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 (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. To be precise, the printable characters are those with ASCII codes between 32 (space) and 126 ($\sim$), except for `"` and `\`. You can refer to a character by its character code in a regular expression using the $\backslash ddd$ format described below.

   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`

   Each such two-character sequence must be converted into the corresponding single character. For example, the two-character string `"\t"` (`\` followed by `t`) must become the single character `'\t'` in your string token.

   Additionally, you must handle the following escaped sequence:

   $\backslash ddd$

   where ddd represents an integer value between 0 and 255 (numbers greater than 255 are malformed tokens and should not be lexed). The above escaped sequence is used to escape specific ASCII integer values. Your job is to map the integer to its single character value. For example, the escaped character $\backslash 100$ is the character 'd'. You can use the function `char_of_int` to turn a character code into an OCaml character.

   You will probably find it easiest to create a new entrypoint, possibly taking an argument, to handle the parsing of strings. You may also find the functions `int_of_string` and `String.make` useful. `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.

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\"";;
- : Mp6parse.token list = [STRING "some string"]
# get_all_tokens "\" she said, \\\"hello\\\"\"";;
- : Mp6parse.token list = [STRING " she said, \"hello\""]
# get_all_tokens "\" \\100 \\001 \"";;
- : Mp6parse.token list = [STRING " d \001 "]
# get_all_tokens "\"next line \\n starts here; indentation \\t starts here
 next string\" \"starts here\"";;
- : Mp6parse.token list =
[STRING "next line \n starts here; indentation \t starts here next string";
 STRING "starts here"]
```

8 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 `mp6-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, defined in `mp6common.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
   exception SuperCloseComm 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 bookkeeping. In the event that there are multiple unclosed comments, return the position of the most recently opened one.
- `*)` is encountered outside of a comment, raise `CloseComm pos` where `pos` is the position of the first character of the errant `*)`.
- `**)` is encountered outside of a comment, raise `SuperCloseComm 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:
Mp6common.OpenComm {Mp6common.line_num = 3; Mp6common.char_num = 2}.
# get_all_tokens "let\n123\n\"hi\\n\"*)\n";;
Exception:
Mp6common.CloseComm {Mp6common.line_num = 3; Mp6common.char_num = 7}.
# get_all_tokens "aaaa\nbbbb**)\nc\n";;
Exception:
Mp6common.SuperCloseComm {Mp6common.line_num = 2; Mp6common.char_num = 5}.
```