
MP 6 – A Lexer for MicroML

CS 421 – Spring 2007

Revision 1.0

Assigned March 14, 2007

Due March 30, 2007 23:59pm

Extension 48 hours (20% penalty)

1 Change Log

1.0 Initial Release.

2 Overview

In this assignment, you will develop a lexer for a MicroML type language using *ocamllex*. In a later MP, you will build a parser using *ocamlyacc*.

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 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.

MicroML is a simplified version of the Standard ML programming language. A MicroML program is just one expression consisting of a space-separated sequence of expressions.

3 Overview of Lexical Analysis (Lexing)

Recall from lecture that the process of transforming program code (i.e. as ASCII 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 list 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 that language. 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. The lexer will ONLY return the next token, and NOT the list of all tokens.

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` which does the parsing. 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 list of tokens it finds.

Here is a quick example:

```
rule mylexer = parse
| [' ' '\t' '\n'] {mylexer lexbuf}
| ['x' 'y' 'z']+ as 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 regex that is named. 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`, but should also use the result of a recursive call back to `mylexer` (otherwise the rest of the input would be discarded). For this MP, most actions will not have a recursive call in them because the lexer will only return the next token.

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 as a 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

micromlparse.cmi is a special binary object file containing the definition of our tokens. This file must be present for the lexer specification to compile.

micromllex-skeleton.mll is the skeleton for the lexer specification. `mml_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. First rename the file to `micromllex.mll` as before and work on it.

The `lertest` function takes the input string and returns a function. Each time the function is called it returns the next token in the string.

Other than the different types of tokens defined for lexer, a special `DUMMY` token has been defined. When the `lertest` function has finished tokenizing the string, all the subsequent calls will return the `DUMMY` token.

6 Problems

- (5 points). 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
=	EQUALS
(LPAREN
)	RPAREN
,	COMMA
[LBRAC
]	RBRAC
::	DBLCOLON
#	STRCAT
=>	MAPSTO
and	AND
andalso	ANDALSO
end	END
fn	FN
fun	FUN
if	IF
else	ELSE
in	IN
let	LET
rec	REC
local	LOCAL
op	OP
orelse	ORELSE
val	VAL

Each token should have its own rule in the lexer specification. The lexer tries to match the longest string possible each time. Hence, “=>” is matched with MAPSTO, instead of an EQUALS token and a symbolic identifier “ > ” (Symbolic identifiers are introduced in problem 6).

Remember also that the regular expression rules are tried on the input from top to bottom. Hence, if “fun” is defined as a reserved word (FUN) ahead of alphanumeric identifiers (introduced in problem 6), the lexer will recognise it as FUN and not VAR "fun".

```
# let f = lextest " fun = ) / not fun = ( " ;;
val f : unit -> Micromlparse.token = <fun>
# f ();;
- : Micromlparse.token = FUN
# f ();;
- : Micromlparse.token = EQUALS
# f ();;
- : Micromlparse.token = RPAREN
```

- (5 points). Implement integers using regular expressions. There is a token constructor INT which takes the integer as an argument. Do not worry about negative integers, and make sure that integers have at least one digit.

```
# let f = lextest "42 100 0" ;;
val f : unit -> Micromlparse.token = <fun>
```

```
# f ();;
- : Micromlparse.token = INT 42
# f ();;
- : Micromlparse.token = INT 100
# f ();;
- : Micromlparse.token = INT 0
```

3. (10 points). Implement floats using regular expressions. There is a token constructor `FLOAT` which takes the float as an argument. Do not worry about negative numbers. The floats may have zero or more leading 0's.

```
# let f = lextest "42.01 .1 0.02";;
val f : unit -> Micromlparse.token = <fun>
# f ();;
- : Micromlparse.token = FLOAT 42.01
# f ();;
- : Micromlparse.token = FLOAT 0.1
# f ();;
- : Micromlparse.token = FLOAT 0.02
```

4. (5 points). Implement booleans and the unit expression. The relevant constructors are `UNIT` and `BOOL`.

```
# let f = lextest "true false ()";;
val f : unit -> Micromlparse.token = <fun>
# f ();;
- : Micromlparse.token = BOOL true
# f ();;
- : Micromlparse.token = BOOL false
# f ();;
- : Micromlparse.token = UNIT
```

5. (15 points). Implement strings. These are not as simple as integers because of string escapes. We will implement the C# style strings.

We treat the strings as it is, without considering the escape characters. A string begins with an `@` character; everything that is surrounded by the two double quotes following the `@` character is literally part of the string. A string does not consider escape characters (i.e. takes the string verbatim), except for `' "`, which is escaped by again a `' "`.

Remember, OCaml uses escape characters to describe special characters such as tab and newline. Thus `"\\ "` is a string of length one, containing the character `' \ '`. Note here, that when we input string in OCaml we have to use the `"\\"` character to escape the input. Hence, a string of length one containing a single backslash character would be input as `"@\"\\ \"` and the returned token will be `STRING "\\ "`.

The syntax for a single double quote would be `@" " "` which would be given by the OCaml string `"@\"\" \"` and the returned token will be `STRING "\" "`.

Use the `STRING` constructor to represent the string. It takes a single argument (the contents of the string).

```

# let f = lextest "@\\Directory\\File\" ;
val f : unit -> Micromlparse.token = <fun>
# f ();;
- : Micromlparse.token = [STRING "\\Directory\\File"]

# let f = lextest "@\"She said \\\"Hello Mike\\\" when they met.\" ;
val f : unit -> Micromlparse.token = <fun>
# f ();;
- : Micromlparse.token = [STRING "She said \\\"Hello Mike\\\" when they met."]

```

We don't care about the quotation marks which surround the string in the input. 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 (I know of at least 2). In any case, don't include the delimiting quotation marks in the token that is returned.

There is a helper function that you have to write. It is called `unescape`. It takes care of the string escapes. It will check for two adjacent `'` characters and return one of them. In case there is only one `'` character in the string, an error will be returned. The error code will be `"singleton '\'' character"` (you can use `failwith "singleton '\'' character"` for this). You might also use the two helper functions named `strpop` and `chr2str`.

6. (10 points). Implement identifiers. An identifier is either alphanumeric: any sequence of letters, digits, primes (`'`) and underscores (`_`) starting with a letter (both uppercase and lowercase), or symbolic: any non-empty sequence of the following symbols,

`+ - / * < >= <= >`

Use the `VAR` constructor, which takes a string argument (the name of the variable).

Variable Tokens	Not Variable Tokens
asdf1234	1234asdf
A_'bb1	_123
Fun	fun
+++	

```

# let f = lextest "this is fun" ;
val f : unit -> Micromlparse.token = <fun>
# f ();;
- : Micromlparse.token = VAR "this"
# f ();;
- : Micromlparse.token = VAR "is"
# f ();;
- : Micromlparse.token = FUN

# let f = lextest "+-/* A__'d a_1_2_3";
val f : unit -> Micromlparse.token = <fun>
# f ();;
- : Micromlparse.token = VAR "+-/*"
# f ();;
- : Micromlparse.token = VAR "A__'d"
# f ();;
- : Micromlparse.token = VAR "a_1_2_3"

```

7. (20 points). Implement comments. Comments in MicroML begin with “[*” and end at the first “*]” that follows. In other words, comments may not be nested.

There are two ways to do this: either make a regular expression to match an entire comment, or define a second lexing rule (entrypoint) that gets entered when a “[*” is seen, and which returns to our token rule after it sees a “*]”. There is no token for comments, they are simply ignored (just like whitespace).

Another type of comment is the single-line comment. This starts with a“(’ ” symbol. The remainder of that line is commented.

```
# let f = lextest "foo = (* blah blah blah blah *) bar" ;;
val f : unit -> Micromlparse.token = <fun>
# f ();;
- : Micromlparse.token = VAR "foo"
# f ();;
- : Micromlparse.token = EQUALS
# f ();;
- : Micromlparse.token = VAR "bar"

# let f = lextest "foo = bar ( ' blah blah blah blah " ;;
val f : unit -> Micromlparse.token = <fun>
# f ();;
- : Micromlparse.token = VAR "foo"
# f ();;
- : Micromlparse.token = EQUALS
# f ();;
- : Micromlparse.token = VAR "bar"
```

We will assume the separation of two types of comments. Whenever OCaml starts to match for a“(’ ” the entire output in that line is ignored. So, if that line contained a “[* ” or “*] ” symbol, it will be ignored. This is why the lextest fails to match a token the second time it is called in the following example.

```
# let f = lextest " ( ' [* a
  b *] c
  [* d
    e ( ' f *] g" ;;
val f : unit -> Micromlparse.token = <fun>
# f ();;
- : Micromlparse.token = VAR "b"
# f ();;
Exception: Failure "unmatched ' *]'".
```

More example of comments.

```
# let f = lextest "a [* b
c ( ` e *] f
( ' g [* h
i " ;;
val f : unit -> Micromlparse.token = <fun>
# f ();;
- : Micromlparse.token = VAR "a"
# f ();;
- : Micromlparse.token = VAR "f"
# f ();;
```

```
| - : Micromlparse.token = VAR "i"  
| # f ();;  
| - : Micromlparse.token = DUMMY
```

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

7 Compiling, Testing & Handing In

7.1 Compiling & Testing

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

```
| ocamllex micromllex.mll
```

This creates a file called `micromllex.ml` (note the slight difference in names). Then you can run tests on your lexer in OCaml using the `lextest` function that is already included in `micromllex.mll`.

```
| # #use "micromllex.ml";;  
| ...  
| # let f = lextest "some string to test";;  
| val f : unit -> Micromlparse.token = <fun>
```