A lexical analyzer generator for Standard ML. Version 1.6.0, October 1994

Andrew W. Appel[1]
James S. Mattson
David R. Tarditi[2]

[1]Department of Computer Science, Princeton University
[2]School of Computer Science, Carnegie Mellon University

New in this version:

- REJECT is much less costly than before.
- Lexical analyzers with more than 255 states can now compile in your lifetime.

Contents

1 General Description

Computer programs often need to divide their input into words and distinguish between different kinds of words. Compilers, for example, need to distinguish between integers, reserved words, and identifiers. Applications programs often need to be able to recognize components of typed commands from users.

The problem of segmenting input into words and recognizing classes of words is known as lexical analysis. Small cases of this problem, such as reading text strings separated by spaces, can be solved by using hand-written programs. Larger cases of this problem, such as tokenizing an input stream for a compiler, can also be solved using hand-written programs.

A hand-written program for a large lexical analysis problem, however, suffers from two major problems. First, the program requires a fair amount of programmer time to create. Second, the description of classes of words is not explicit in the program. It must be inferred from the program code. This makes it difficult to verify if the program recognizes the correct words for each class. It also makes future maintenance of the program difficult.

Lex, a programming tool for the Unix system, is a successful solution to the general problem of lexical analysis. It uses regular expressions to describe classes of words. A program fragment is associated with each class of words. This information is given to Lex as a specification (a Lex program). Lex produces a program for a function that can be used to perform lexical analysis.

The function operates as follows. It finds the longest word starting from the current position in the input stream that is in one of the word classes. It executes the program fragment associated with the class, and sets the current position in the input stream to be the character after the word. The program fragment has the actual text of the word available to it, and may be any piece of code. For many applications it returns some kind of value.

Lex allows the programmer to make the language description explicit, and to concentrate on what to do with the recognized words, not how to recognize the words. It saves programmer time and increases program maintainability.

Unfortunately, Lex is targeted only C. It also places artificial limits on the size of strings that can be recognized.

ML-Lex is a variant of Lex for the ML programming language. ML-Lex has a syntax similar to Lex, and produces an ML program instead of a C program. ML-Lex produces a program that runs very efficiently. Typically the program will be as fast or even faster than a hand-coded lexer implemented in Standard ML.

The program typically uses only a small amount of space. ML-Lex thus allows ML programmers the same benefits that Lex allows C programmers. It also does not place artificial limits on the size of recognized strings.

2 ML-Lex specifications

An ML-Lex specification has the general format:

user declarations %% ML-Lex definitions %% rules

Each section is separated from the others by a %% delimiter.

The rules are used to define the lexical analysis function. Each rule has two parts—a regular expression and an action. The regular expression defines the word class that a rule matches. The action is a program fragment to be executed when a rule matches the input. The actions are used to compute values, and must all return values of the same type.

The user can define values available to all rule actions in the user declarations section. The user must define two values in this section—a type lexresult and a function eof. Lexresult defines the type of values returned by the rule actions. The function "eof" is called by the lexer when the end of the input stream is reached. It will typically return a value signalling eof or raise an exception. It is called with the same argument as lex (see `%arg`, below), and must return a value of type lexresult.

In the definitions section, the user can define named regular expressions, a set of start states, and specify which of the various bells and whistles of ML-Lex are desired.

The start states allow the user to control when certain rules are matched. Rules may be defined to match only when the lexer is in specific start states. The user may change the lexer's start state in a rule action. This allows the user to specify special handling of lexical objects.

This feature is typically used to handle quoted strings with escapes to denote special characters. The rules to recognize the inside contents of a string are defined for only one start state. This start state is entered when the beginning of a string is recognized, and exited when the end of the string is recognized.

3 Regular expressions

Regular expressions are a simple language for denoting classes of strings. A regular expression is defined inductively over an alphabet with a set of basic operations. The alphabet for ML-Lex is the Ascii character set (character codes 0–127; or if `%full` is used, 0–255).

The syntax and semantics of regular expressions will be described in order of decreasing precedence (from the most tightly binding operators to the most weakly binding):

- An individual character stands for itself, except for the reserved characters `? * + | ( ) ^ $ / ; . = < > [ { " \`

A backslash followed by one of the reserved characters stands for that character.

- A set of characters enclosed in square brackets [ ] stands for any one of those characters. Inside the brackets, only the symbols `\ - ^` are reserved. An initial up-arrow `^` stands for the complement of the characters listed, e.g. `[^abc]` stands any character except a, b, or c. The hyphen - denotes a range of characters, e.g. `[a-z]` stands for any lower-case alphabetic character, and `[0-9a-fA-F]` stands for any hexadecimal digit. To include `^` literally in a bracketed set, put it anywhere but first; to include `-` literally in a set, put it first or last.

. The dot `.` character stands for any character except newline, i.e. the same as `[^\n]`

- The following special escape sequences are available, inside or outside of square-brackets:

`\b`	backspace
`\n`	newline
`\t`	tab
`\h`	stands for all characters with codes > 127, when 7-bit characters are used.
`\ddd`	where `ddd` is a 3 digit decimal escape.

" A sequence of characters will stand for itself (reserved characters will be taken literally) if it is enclosed in double quotes `" "`.

{} A named regular expression (defined in the "definitions" section) may be referred to by enclosing its name in braces `{ }`.

() Any regular expression may be enclosed in parentheses `( )` for syntactic (but, as usual, not semantic) effect.

* The postfix operator `*` stands for Kleene closure: zero or more repetitions of the preceding expression.

+ The postfix operator `+` stands for one or more repetitions of the preceding expression.

? The postfix operator `?` stands for zero or one occurrence of the preceding expression.

- A postfix repetition range $\{n_1, n_2\}$ where n_1 and n_2 are small integers stands for any number of repetitions between n_1 and n_2 of the preceding expression. The notation $\{n_1\}$ stands for exactly n_1 repetitions.

- Concatenation of expressions denotes concatenation of strings. The expression e_1e_2 stands for any string that results from the concatenation of one string that matches e_1 with another string that matches e_2.

- | The infix operator | stands for alternation. The expression e_1 | e_2 stands for anything that either e_1 or e_2 stands for.

/ The infix operator / denotes lookahead. Lookahead is not implemented and cannot be used, because there is a bug in the algorithm for generating lexers with lookahead. If it could be used, the expression e_1/e_2 would match any string that e_1 stands for, but only when that string is followed by a string that matches e_2.

- When the up-arrow ^ occurs at the beginning of an expression, that expression will only match strings that occur at the beginning of a line (right after a newline character).

$ The dollar sign of C Lex $ is not implemented, since it is an abbreviation for lookahead involving the newline character (that is, it is an abbreviation for `/\n`).

Here are some examples of regular expressions, and descriptions of the set of strings they denote:

`0 | 1 | 2 | 3`	A single digit between 0 and 3
`[0123]`	A single digit between 0 and 3
`0123`	The string "0123"
`0*`	All strings of 0 or more 0's
`00*`	All strings of 1 or more 0's
`0+`	All strings of 1 or more 0's
`[0-9]{3}`	Any three-digit decimal number.
`\\[ntb]`	A newline, tab, or backspace.
`(00)*`	Any string with an even number of 0's.

4 ML-Lex syntax summary

4.1 User declarations

Anything up to the first `%%` is in the user declarations section. The user should note that no symbolic identifier containing `%%` can be used in this section.

4.2 ML-Lex definitions

Start states can be defined with

`%s` identifier list ;

An identifier list consists of one or more identifiers.

An identifier consists of one or more letters, digits, underscores, or primes, and must begin with a letter.

Named expressions can be defined with

identifier = regular expression ;

Regular expressions are defined below.

The following % commands are also available:

`%reject` create REJECT() function

`%count` count newlines using yylineno

`%posarg` pass initial-position argument to makeLexer

`%full` create lexer for the full 8-bit character set, with characters in the range 0–255 permitted as input.

`%structure {identifier}` name the structure in the output program identifier instead of Mlex

`%header` use code following it to create header for lexer structure

`%arg` extra (curried) formal parameter argument to be passed to the lex functions, and to be passed to the eof function in place of ()

These functions are discussed in section 5.

4.3 Rules

Each rule has the format:

`<`*start state list*`>` *regular expression* `=> (` *code* `);`

All parentheses in *code* must be balanced, including those used in strings and comments.

The *start state list* is optional. It consists of a list of identifiers separated by commas, and is delimited by triangle brackets `< >`. Each identifier must be a start state defined in the `%s` section above.

The regular expression is only recognized when the lexer is in one of the start states in the start state list. If no start state list is given, the expression is recognized in all start states.

The lexer begins in a pre-defined start state called `INITIAL`.

The lexer resolves conflicts among rules by choosing the rule with the longest match, and in the case two rules match the same string, choosing the rule listed first in the specification.

The rules should match all possible input. If some input occurs that does not match any rule, the lexer created by ML-Lex will raise an exception LexError. Note that this differs from C Lex, which prints any unmatched input on the standard output.

5 Values available inside the code associated with a rule.

ML-Lex places the value of the string matched by a regular expression in `yytext`, a string variable.

The user may recursively call the lexing function with `lex()`. (If `%arg` is used, the lexing function may be re-invoked with the same argument by using continue().) This is convenient for ignoring white space or comments silently:

```
[\ \t\n]+        => ( lex());
```

To switch start states, the user may call `YYBEGIN` with the name of a start state.

The following values will be available only if the corresponding `%` command is in the ML-Lex definitions sections:

Value	**% command**	**description**
`REJECT`	`%reject`	`REJECT()` causes the current rule to be "rejected." The lexer behaves as if the current rule had not matched; another rule that matches this string, or that matches the longest possible prefix of this string, is used instead.
`yypos`		The position of the first character of `yytext`, relative to the beginning of the file.
`yylineno`	`%count`	Current line number

These values should be used only if necessary. Adding `REJECT` to a lexer will slow it down by 20%; adding `yylineno` will slow it down by another 20%, or more. (It is much more efficient to recognize
`n` and have an action that increments the line-number variable.) The use of the lookahead operator `/` will also slow down the entire lexer. The character-position, `yypos`, is not costly to maintain, however.

Bug. The position of the first character in the file is reported as 2 (unless the `%posarg` feature is used). To preserve compatibility, this bug has not been fixed.

6 Running ML-Lex

From the Unix shell, run `sml-lex myfile.lex` The output file will be myfile.lex.sml. The extension `.lex` is not required but is recommended.

Within an interactive system [not the preferred method]: Use `lexgen.sml`; this will create a structure LexGen. The function LexGen.lexGen creates a program for a lexer from an input specification. It takes a string argument – the name of the file containing the input specification. The output file name is determined by appending "`.sml`" to the input file name.

7 Using the program produced by ML-Lex

When the output file is loaded, it will create a structure Mlex that contains the function `makeLexer` which takes a function from $int \rightarrow string$ and returns a lexing function:

```
val makeLexer : (int->string) -> yyarg -> lexresult
```

where `yyarg` is the type given in the `%yyarg` directive, or `unit` if there is no `%yyarg` directive.

For example,

```
val lexer = Mlex.makeLexer (inputc (open_in "f"))
```

creates a lexer that operates on the file whose name is f.

When the `%posarg` directive is used, the type of `makeLexer` is

```
val makeLexer : ((int->string)*int) -> yyarg -> lexresult
```

where the extra `int` argument is one less than the `yypos` of the first character in the input. The value k would be used, for example, when creating a lexer to start in the middle of a file, when k characters have already been read. At the beginning of the file, $k = 0$ should be used.

The $int \rightarrow string$ function should read a string of characters from the input stream. It should return a null string to indicate that the end of the stream has been reached. The integer is the number of characters that the lexer wishes to read; the function may return any non-zero number of characters. For example,

```
val lexer =
  let val input_line = fn f =>
        let fun loop result =
           let val c = input (f,1)
       val result = c :: result
           in if String.size c = 0 orelse c = "\n" then
    String.implode (rev result)
```

```
        else loop result
    end
         in loop nil
         end
    in Mlex.makeLexer (fn n => input_line std_in)
    end
```

is appropriate for interactive streams where prompting, etc. occurs; the lexer won't care that `input_line` might return a string of more than or less than n characters.

The lexer tries to read a large number of characters from the input function at once, and it is desirable that the input function return as many as possible. Reading many characters at once makes the lexer more efficient. Fewer input calls and buffering operations are needed, and input is more efficient in large block reads. For interactive streams this is less of a concern, as the limiting factor is the speed at which the user can type.

To obtain a value, invoke the lexer by passing it a unit:

```
val nextToken = lexer()
```

If one wanted to restart the lexer, one would just discard `lexer` and create a new lexer on the same stream with another call to `makeLexer`. This is the best way to discard any characters buffered internally by the lexer.

All code in the user declarations section is placed inside a structure UserDeclarations. To access this structure, use the path name `Mlex.UserDeclarations`.

If any input cannot be matched, the program will raise the exception `Mlex.LexError`. An internal error (i.e. bug) will cause the exception `Internal.LexerError` to be raised.

If `%structure` is used, remember that the structure name will no longer be Mlex, but the one specified in the command.

8 Sample

Here is a sample lexer for a calculator program:

```
datatype lexresult= DIV | EOF | EOS | ID of string | LPAREN |
                    NUM of int | PLUS | PRINT | RPAREN | SUB | TIMES

val linenum = ref 1
val error = fn x => output(std_out,x ^ "\n")
val eof = fn () => EOF
%%
%structure CalcLex
alpha=[A-Za-z];
digit=[0-9];
```

```
ws = [\ \t];
%%
\n       => (inc linenum; lex());
{ws}+    => (lex());
"/"      => (DIV);
";"      => (EOS);
"("      => (LPAREN);
{digit}+ => (NUM (revfold (fn(a,r)=>ord(a)-ord("0")+10*r) (explode yytext) 0));
")"      => (RPAREN);
"+"      => (PLUS);
{alpha}+ => (if yytext="print" then PRINT else ID yytext);
"-"      => (SUB);
"*"      => (TIMES);
.        => (error ("calc: ignoring bad character "^yytext); lex());
```

Here is the parser for the calculator:

```
(* Sample interactive calculator to demonstrate use of lexer

   The original grammar was

       stmt_list -> stmt_list stmt
       stmt -> print exp ;  | exp ;
       exp -> exp + t | exp - t | t
       t -> t * f | t/f | f
       f -> (exp) | id | num

  The function parse takes a stream and parses it for the calculator
  program.

  If a syntax error occurs, parse prints an error message and calls
  itself on the stream.  On this system that has the effect of ignoring
  all input to the end of a line.
*)

structure Calc =
 struct
   open CalcLex
   open UserDeclarations
   exception Error
   fun parse strm =
    let
      val say = fn s => output(std_out,s)
      val input_line = fn f =>
          let fun loop result =
             let val c = input (f,1)
         val result = c :: result
```

```
        in if String.size c = 0 orelse c = "\n" then
 String.implode (rev result)
    else loop result
end
     in loop nil
     end
 val lexer = makeLexer (fn n => input_line strm)
 val nexttok = ref (lexer())
 val advance = fn () => (nexttok := lexer(); !nexttok)
 val error = fn () => (say ("calc: syntax error on line" ^
                      (makestring(!linenum)) ^ "\n"); raise Error)
 val lookup = fn i =>
   if i = "ONE" then 1
   else if i = "TWO" then 2
   else  (say ("calc: unknown identifier '" ^ i ^ "'\n"); raise Error)
fun STMT_LIST () =
    case !nexttok of
       EOF => ()
     | _ => (STMT(); STMT_LIST())

and STMT() =
    (case !nexttok
      of EOS  => ()
       | PRINT => (advance(); say ((makestring (E():int)) ^ "\n"); ())
       | _ => (E(); ());
    case !nexttok
      of EOS => (advance())
       | _ => error())
and E () = E' (T())
and E' (i : int ) =
    case !nexttok of
       PLUS => (advance (); E'(i+T()))
     | SUB => (advance (); E'(i-T()))
     | RPAREN => i
     | EOF => i
     | EOS => i
     | _ => error()
and T () =  T'(F())
and T' i =
   case !nexttok of
       PLUS => i
     | SUB => i
     | TIMES => (advance(); T'(i*F()))
     | DIV => (advance (); T'(i div F()))
     | EOF => i
     | EOS => i
     | RPAREN => i
```

```
        | _ => error()
   and F () =
      case !nexttok of
          ID i => (advance(); lookup i)
        | LPAREN =>
            let val v = (advance(); E())
            in if !nexttok = RPAREN then (advance (); v) else error()
            end
        | NUM i => (advance(); i)
        | _ => error()
  in STMT_LIST () handle Error => parse strm
  end
end
```