A Lexical Analyzer Generator in ML

James S. Mattson Jr.

Department of Computer Science
Junior Independent Work
Advisor: Andrew W. Appel

INTRODUCTION

Lex [1] and Yacc [2] are standard Unix[1] tools designed primarily to generate routines which will impart a structure to the input for a computer program. Lex produces code for a *lexical analyzer*, a routine which will scan the input stream character by character and partition it into strings matching the regular expressions given in the user's Lex specifications. In general, these strings, called *tokens*, are then passed on to the function generated by Yacc, the *parser*, which organizes the tokens according to the grammar rules given in the user's Yacc specifications. When the input matches a grammar rule, the user's action for that rule is performed. In other words, the parser analyzes the syntax of the tokens in the input stream, and so perhaps Yacc is better understood as a *syntax analyzer generator*. (Actually, Lex is powerful enough to generate stand-alone code which can perform simple manipulations on the input stream without a parser, but such applications are not important to this discussion.) The language in which Lex and Yacc actions are written is called the host language. Currently, Lex only supports two host languages: C and Ratfor. To make matters worse, Yacc only supports C as a host language.

This paper is a description of *lex.ml*, a Lexical Analyzer Generator very much like Lex which supports ML [3,4] as a host language. ML is not a procedural programming language like C or any other member of the Algol family; ML is a functional programming language with a close

---

[1] Unix is a trademark of AT&T Bell Laboratories

resemblance to the lambda-calculus. The primary motivation for writing Lex in ML is to further the development of a new tool which will aid compiler development. This tool, a *semantics analyzer generator* will produce code which can derive the meaning of a program from the output of the parser. One technique for describing the meaning of program fragments is denotational semantics [5]. Since the meta-language of denotational semantics is the lambda-calculus, a "semantics analyzer generator" which employs this technique will no doubt be much easier to write in a functional programming language like ML than it would be to write in C. Such a tool would not be very useful, however, if the underlying support of Lex and Yacc were not available for the same language. Thus, *lex.ml* is the first step in such a project.

For the remainder of this paper, a familiarity with Lex and ML will be assumed. Readers who desire more background information are urged to consult the references.

LEX SPECIFICATIONS

Though some attempts have been made to improve the format of the Lex source, those who have used Lex should be able to use *lex.ml* without any real difficulty. The general format of the *lex.ml* source file is:

```
{user subroutines}
%%
{definitions}
%%
{rules}
%%
```

The last %% is optional, but the other two are required. User subroutines have been moved to the beginning of the file to reflect the fact that functions in ML must be defined before they are used.

As with Lex, user subroutines are simply copied character by character into the output file. Global variables and anything else that might have appeared between %{ and %} in the definitions section of standard Lex should also be included here.

The format of the definitions section has been modified slightly so that *lex.ml* may ignore whitespace. Thus, the format for a rule is now:

```
name = regular-expression
```

Similarly, the format for rules has been modified:

```
regular-expression => (ML-action)
```

Note that this format still allows for such constructs as:

```
a       |
b       |
c       => (ECHO());
```

Care should be taken that all actions return the same type of result, since they will all be gathered as matches of a case statement. (For instance, ECHO() returns a unit; thus, if ECHO() is used, all actions must return a unit.)

REGULAR EXPRESSIONS

Most standard Lex operators are supported. Double quotes may be used to indicate that the string between quotes is to be taken

literally; e.g. "++". Backslash may be used to indicate that the next character is to be taken literally; e.g. \+\+. The exceptions to this are \n = newline; \t = tab; and \b = backspace. As usual, \ must be entered as \\. The backslash operator remains in effect even within quoted strings. Escaping into octal is not supported. Brackets may be used to indicate a character class. Unlike standard Lex, whitespace is ignored even within a character class, so space and tab must be escaped using \. Only three characters are special within a character class: \ - and ^. They work as in standard Lex. The period matches any character but newline. The question mark indicates an optional expression; e.g. ab?c. An asterisk may be used to indicate "any number of repetitions," and a plus may be used to indicate "one or more repetitions"; e.g. a* or a+. A vertical bar indicates alternation; e.g. ab|cd. Parentheses may be used for grouping complex expressions; e.g. (ab|cd+)?(ef)*. The implied operator is, of course, catenation. A carat at the beginning of an expression indicates that the expression is to be matched only when it occurs at the beginning of a line, and a dollar sign at the end of an expression indicates that the expression is to be matched only when it occurs at the end of a line. Trailing context may be indicated with a slash; e.g ab/cd. Start conditions may be specified by placing the names of the start states within angle brackets prior to the expression; e.g. <AAA,BBB>abc. Repetitions may be specified using braces; e.g. a{12} or a{1,5}. Any name within braces is expanded as per the definition of that name within the definitions section; e.g. {digit}.

START STATES

If certain rules are to be matched only when the lexer is in a specific start state, the names of these states must be specified on a line in the definitions section of the format:

%s name1 name2 ... namen

Rules which are not preceded by a list of start states in angle brackets are matched in all states. The initial start state of the lexer is called "INITIAL." This state should not be specified in the %s line of the definitions section.

LEX FUNCTIONS AND VARIABLES

To avoid conflicts with standard ML functions, input(), unput(), and output() have been renamed to yyinput(), yyunput(), and yyoutput(). The user may override the standard versions of these functions by including new definitions for them in the user subroutines section. Also available are yyless():unit, yymore():unit, and yywrap():bool, which perform the usual functions. (The supplied version of yywrap() returns false.) BEGIN(), ECHO(), and REJECT() are supplied as functions rather than macros, but they work just as in Lex. Finally, yytext:string is the text matched, and yyleng has been discarded, because it is not generally needed in ML.

USAGE

To load the Lex program, first enter the ml interpreter and then enter:

```
use ["lex.ml"]
```

Once loaded, Lex is invoked with a call such as:

```
LexGen("inputfile","outputfile")
```

All of the necessary ML source code for the lexical analyzer will be generated in the outputfile; that file may be loaded without loading *lex.ml*.

IMPLEMENTATION

The regular expressions specified in the rules section are first augmented with an end-marker and collected into a list of (start state list * expression parse tree) pairs. From this list, a deterministic finite automaton is generated directly without an intermediate non-deterministic finite automaton [6]. The important (non-epsilon) leaves of the parse tree are numbered, and an array indexed by leaf number is generated which indicates which leaves may follow each leaf. The states of the DFA correspond to lists of leaves. For example, the state "INITIAL" consists of all leaves which may correspond to the first character of each regular expression which is recognized from that start

state. Other start states are defined similarly. For each state, then, every possible input character is considered in turn. All leaves which may follow any leaf of a given state marked with a given character are collected to form a new state. The transitions from state to state are stored in an array indexed by state of a list of (character * state) pairs.

The DFA is simulated in the lexer by a function which performs the appropriate state transitions based upon its input until there are no transitions available to it. Each time a new state is entered, the leaves of that state which correspond to end-markers are added to a growing list of accepting leaf numbers along with the number of characters matched. When no transitions are available, control passes to a function which performs the user actions associated with the most recent accepting leaf. If REJECT() is called, the actions associated with the previous accepting leaf are performed. If the list of accepting leaves is nil, ECHO() is called to copy the input to the output.

REFERENCES

1. Lesk, M.E. and Schmidt, E., "Lex--A Lexical Analyzer Generator," Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, Murray Hill, NJ (1975).

2. Johnson, S.C., "Yacc: Yet Another Compiler Compiler," Comp. Sci. Tech. Rep. No. 32, Bell Laboratories, Murray Hill, NJ (1975).

3. Miller, Robin, "The Standard ML Core Language," University of Edinburgh (1985).

4. Harper, Robert, "Introduction to Standard ML" (Preliminary Draft), University of Edinburgh (1986).

5. Appel, Andrew W., "Compile-time Evaluation and Code Generation for Semantics-directed Compilers," Carnegie-Mellon University (1985).

6. Aho, Alfred, Sethi, Ravi, and Ullman, Jeffrey D., *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA (1986).