THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 57, No. 6, July-August 1978
Printed in U. S. A.

UNIX Time-Sharing System:

Language Development Tools

By S. C. JOHNSON and M. E. LESK
(Manuscript received December 27, 1977)

The development of new programs on the UNIX system is facilitated by tools for language design and implementation. These are frequently program generators, compiling into C, which provide advanced algorithms in a convenient form, while not restraining the user to a preconceived set of jobs. Two of the most important such tools are Yacc, a generator of LALR(1) parsers, and Lex, a generator of regular expression recognizers using deterministic finite automata. They have been used in a wide variety of applications, including compilers, desk calculators, typesetting languages, and pattern processors.*

I. INTRODUCTION

On the UNIX system, an effort has been made to package language development aids for general use, so that all users can share the newest tools. As a result, these tools have been used to design pleasant, structured applications languages, as well as in their more traditional roles in compiler construction. The packaging is crucial, since if the underlying algorithms are not well packaged, the tools will not be used; applications programmers will rarely spend weeks learning theory in order to use a tool.

Traditionally, algorithms have been packaged as system commands (such as **sort**), subroutines (such as **sin**), or as part of the supported features of a compiler or higher level language environment

* UNIX is a trademark of Bell Laboratories.

(such as the heap allocation in Algol 68). Another way of packaging, which is particularly appropriate in the UNIX operating system, is as a *program generator*. Program generators take a specification of a task and write a program which performs that task. The programming language in which this program is generated (called the *host language*) may be high or low level, although most of ours are high level. Unlike compilers, which typically implement an entire general-purpose source language, program generators can restrict themselves to doing one job, but doing it well.

Program generators have been used for some time in business data processing, typically to implement sorting and report generation applications. Usually, the specifications used in these applications describe the entire job to be done, and the fact that a program is generated is important, but really only one feature of the implementation of the applications package. In contrast, our program generators might better be termed module generators; the intent is to provide a single module that does an important part of the total job. The host language, augmented perhaps by other generators, can provide the other features needed in the application. This approach gains many of the advantages of modularity, as well as the advantages which the advanced algorithms provide. In particular:

(*i*) Each generator handles only one job, and thus is easier to write and to keep up to date.

(*ii*) The user can select exactly the tools needed; one is not forced to accept many unwanted features in order to get the one desired.

(*iii*) The user can also select what manuals have to be read; not only is an unused tool not paid for, but it also need not be learned.

(*iv*) Portability can be enhanced, since only the host language compiler must know the object machine code.

(*v*) Since the interfaces between the tools are well-defined and the output of the tools is in human-readable form, it is easy to make independent changes to the tools and to determine the source of difficulty when a combination of tools fails to work.

Obviously, this all depends on the specific tools fitting together well, so that several can be used in a job. On the UNIX system, this is achieved in a variety of ways. One is the use of *filters*, programs that read one input stream and write one output stream. Filters are easy to fit together; they are simply connected end to end. On the UNIX system, the command line syntax makes it easy to specify a

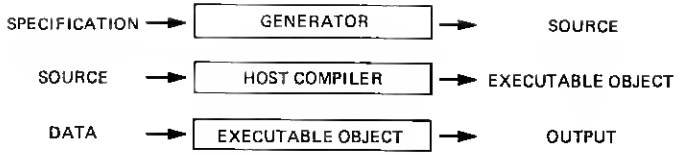

Fig. 1—Program generator.

sequence of commands, each of which uses as its input the output of the preceding command. An example appears in typesetting:

```
refer source—files | tbl | eqn | troff ...
```

where **refer** processes the references, **tbl** the tables, **eqn** the equations, and finally **troff** the text.[1] Each of the first three programs is really a program generator writing code in the host language **troff**, which in turn produces "object code" in the form of typesetter device commands.

This paper focuses on Yacc and Lex. A detailed description of the underlying theory of both programs can be found in Aho and Ullman's book,[2] while the appropriate users' manual can be consulted for further examples and details.[3,4]

Since program generators have output which is in turn input to a compiler and the compiler output is a program which in turn may have both input and output, some terminology is essential. To clarify the discussion, throughout this paper the term *specification* will be used to refer to the input of Yacc or Lex. The output program generated then becomes the *source*, which is compiled by the host language compiler. The resulting *executable object* program may then read *data* and produce *output*. To use a generator:

(*i*) The user writes a specification for the generator, containing grammar rules (for Yacc) or regular expressions (for Lex).
(*ii*) The specification is fed through the generator to produce a source code file.
(*iii*) The source code is processed by the compiler to produce an executable file.
(*iii*) The user's real data is processed by the executable file to produce the real output.

This can be diagrammed as shown in Fig. 1. Both Yacc and Lex accept both C and Ratfor[5] as host languages, although C is far more widely used.

The remainder of this paper gives more detail on the two main program generators, Yacc in Section II and Lex in Section III.

Section IV describes an example of the combined use of both generators to do a simple job, reading a date (month, day, year) and producing the day of the week on which it falls. Finally, Section V contains more general comments about program generators and host languages.

II. THE YACC PARSER GENERATOR

Yacc is a tool which turns a wide class of context-free grammars (also known as Backus-Naur Form, or BNF, descriptions) into parsers that accept the language specified by the grammar. A simple example of such a description might look like

```
date :    month day year ;

month:    "Jan" | "Feb" | "Mar" |
          "Apr" | "May" | "Jun" |
          "Jul" | "Aug" | "Sep" |
          "Oct" | "Nov" | "Dec" ;

day:      number ;

year:     "," number |     ;

number:   DIGIT |
          number DIGIT ;
```

In words, this says that a date is a month, day, and year, in that order; in the Yacc style of writing BNF, colons and semicolons are syntactic connectives that aren't really included in the actual description. The vertical bar stands for "or," so a month is **Jan** or **Feb**, and so on. Quoted strings can stand for literal appearances of the quoted characters. A day is just a number (discussed below). A year is either a comma followed by a number, or it can in fact be missing entirely. Thus, this example would allow as a date either *Jul 4, 1776*, or *Jul 4*.

The two rules for **number** say that a number is either a single digit, or a number followed by a digit. Thus, in this formulation, the number *123* is made up of the number *12*, followed by the digit *3*; the number *12* is made up of the number *1* followed by the digit *2*; and the number *1* is made up simply of the digit *1*.

Using Yacc, an action can be associated with each of the BNF

rules, to be performed upon recognizing that rule. The actions can be any arbitrary program fragments. In general, some value or meaning is associated with the components of the rule and part of the job of the action for a rule is to compute the value or meaning to be associated with the left side of the current rule. Thus, a mechanism has been provided for these program fragments to obtain the values of the components of the rule and return a value. Using the **number** example above, suppose a value has been associated with each possible DIGIT; the value of *1* is 1, etc. The rules describing the structure of numbers can be followed by associated program fragments which compute the meaning or value of the numbers. Assuming that numbers are decimal, then the value of a number which is a single digit is just the value of the digit, while the value of a number which is a number followed by a digit is 10 times the value of the number, plus the value of the digit. In order to specify the values of numbers, we can write:

```
number      : DIGIT
                        { $$ = $1; }
            | number DIGIT
                        { $$ = 10 * $1 + $2; }
            ;
```

Notice that the values of the components of the right-hand sides of the rule are described by the *pseudo-variables* **$1**, **$2**, etc. which refer to the first, second, etc. elements of the right side of the rule. A value is returned for the rule by assigning to the pseudo-variable **$$**. After writing the above actions, the other rules which use **number** will be able to access the value of the number.

Recall that the values for the digits were assumed known. In practice, BNF is rarely used to describe the complete structure of the input. Usually a previous stage, the *lexical analyzer*, is responsible for actually reading the input characters and assembling them into *tokens*, the basic input units for the BNF specification. Lex, described in the next section, is used to help build lexical analyzers; among the issues usually dealt with in the Lex specification are the assembly of alphabetic characters into names, the recognition of classes of characters (such as DIGITS), and the treatment of blanks, newlines, comments, and other similar issues. In particular, the lexical analyzer will be able to associate values to the tokens which it represents, and these values will be accessible in the BNF specification.

The programs generated by Yacc, called *parsers*, read the input

data and associate the rules and actions of the BNF to this input, or report an error if there is no correct association. If the above BNF example is given to Yacc, together with an appropriate lexical analyzer, it will produce a program that will read dates and only dates, report error if something is read that does not fit the BNF description of a date, and associate the correct actions, values, or meanings to the structures encountered during input.

Thus, parsing is like listening to prose; programmers say, "I've never parsed a thing!" but, in fact, every Fortran READ statement does parsing. Fortran FORMAT statements are simply parser specifications. BNF is very powerful, however, and, what is important in practice, many BNF specifications can be turned automatically into fast parsers with good error detection properties.

Yacc provides a number of facilities that go beyond BNF in the strict sense. For example, there is a mechanism which permits the user some control over the behavior of the parser when an error is encountered. Theoretically, one may be justified in terminating the processing when the data are discovered to be in error, but, in practice, this is unduly hostile, since it leads to the detection of only one error per run. To use such a parser to accept input interactively is totally unacceptable; one often wants to prompt the naive user if input is in error, and encourage correct input. The Yacc facilities for error recovery are used by including additional rules, in addition to those which specify the correct input. These rules may use the special token *error*. When an error is detected, the parser will attempt to recover by behaving as if it had just seen the special *error* token immediately before the token which triggered the error. It looks for the "nearest" rule (in a precise sense) for which the *error* token is legal, and resumes processing at this rule. In general, it is also necessary to skip over a part of the input in order to resume processing at an appropriate place; this can also be specified in the error rule. This mechanism, while somewhat unintuitive and not completely general, has proved to be powerful and inexpensive to implement. As an example, consider a language in which every statement ends with a semicolon. A reasonable error recovery rule might be

```
statement : error ';'
```

which, when added to the specification file, would cause the parser to advance the input to the next semicolon when an error was encountered, and then perform any action associated with this rule. One of the trickiest areas of error recovery is the semantic recovery:

how to repair partially built symbol table entries and expression trees that may be left after an error, for example. This problem is difficult and depends strongly on the particular application.

Yacc provides another very useful facility for specifying arithmetic expressions. In most programming languages, there may be a number of arithmetic operators, such as +, −, /, etc. These typically have an ordering, or *precedence*, associated with them. As an example, the expression

```
a + b * c
```

is typically taken to mean

```
a + ( b * c )
```

because the multiplication operator (*) is of higher precedence or binding power than the addition operator (+). In pure BNF, specification of precedence levels is somewhat indirect and requires a technical trick which, while easy to learn, is nevertheless unintuitive. Yacc provides the ability to write simple rules that specify the parsing of arithmetic expressions except for precedence, and then supply the precedence information about the operators separately. In addition, the left or right associativity can be specified. For example, the sequence

```
%left '+' '-'
%left '*' '/'
```

indicates that addition and subtraction are of lower precedence than multiplication and division, and that all are left associative operators. This facility has been very successful; it is not only easier for the nonspecialist to use, but actually produces faster, smaller parsers.[6,7]

Yacc provides a case history of the packaging of a piece of theory in a useful and effective way. For one thing, while BNF is very powerful it does not do everything. It is important to permit escapes from BNF, to permit real applications that can take advantage of the power of BNF, while having some relief from its restrictions. Allowing a general lexical analyzer and general C programs as actions serves this purpose in Yacc. This in turn is made possible by the packaging of the theory as a program generator; the Yacc system does not have to make available to the user all facilities for lexical analysis and actions, but can restrict itself to building fast parsers, and let these other issues be taken care of by other modules.

It is also possible to enclose the Yacc-generated parser in a larger

program. Yacc translates the user's specification into a program named **yyparse**. This program behaves like a finite automaton that recognizes the user's grammar; it is represented by a set of tables and an interpreter to process them. If the user does not supply an explicit main program, **yyparse** is invoked and it reads and parses the input sequence delivered by the lexical analyzer. If the user wishes, however, a main program can be supplied to perform any desired actions before or after calling the parser, and the parser may be invoked repeatedly. The function value returned by **yyparse** indicates whether or not a legal sentence in the specified language was recognized.

It is also possible for the user to introduce his own code at a lower level, since the Yacc parser depends on a routine **yylex** for its input and lexical analysis. This subroutine may be written with Lex (see the next section) or directly by the user. In either case, each time it is called it must return a lexical token name to the Yacc parser. It can also assign a value to the current token by assigning to the variable **yylval**. Such values are used in the same way as values assigned to the **$$** variables in parsing actions.

Thus the user's code may be placed (*i*) above the parser, in the main program; (*ii*) in the parser, as action statements on rules; and/or (*iii*) below the parser, in the lexical analyzer. All of these are in the same core load, so they may communicate through external variables as desired. This gives even the fussiest programmers enough rope to hang themselves. Note, however, that despite the presence of user code even within the parser, both the finite automaton tables and the interpreter are entirely under the control of, and generated by, Yacc, so that changes in the automaton representation need not affect the user.

In addition to generality, good packaging demands that tools be easy to use, inexpensive, and produce high quality output. Over the years, Yacc has developed increased speed and greater power, with little negative effect on the user community. The time required for Yacc to process most specifications is faster than the time required to compile the resulting C programs. The parsers are also comparable in space and time with those that may be produced by hand, but are typically very much easier to write and modify.

To summarize, Yacc provides a tool for turning a wide class of BNF descriptions into efficient parsers. It provides facilities for error recovery, specification of operator precedence, and a general action facility. It is packaged as a program generator, and requires a lexical analyzer to be supplied. The next section will discuss a

complementary tool, Lex, which builds lexical analyzers suitable for Yacc, and is also useful for many other functions.

III. THE LEX LEXICAL ANALYZER GENERATOR

Lex is similar in spirit to Yacc, and there are many similarities in its input format as well. Like Yacc, Lex input consists of rules and associated actions. Like Yacc, when a rule is recognized, the action is performed. The major differences arise from the typical input data and the model used to process them. Yacc is prepared to recognize BNF rules on input which is made up of tokens. These tokens may represent several input characters, such as names or numbers, and there may be characters in the input text that are never seen by the BNF description (such as blanks). Programs generated by Lex, on the other hand, are designed to read the input characters directly. The model implemented by Lex is more powerful than Yacc at dealing with local information — context, character classes, and repetition — but is almost totally lacking in more global structuring facilities, such as recursion. The basic model is that of the theory of regular expressions, which also underlies the UNIX text editor **ed** and a number of other UNIX programs that process text. The class of rules is chosen so that Lex can generate a program that is a deterministic finite state automaton; this means that the resulting analyzer is quite fast, even for large sets of regular expressions. The program fragments written by the user are executed in the order in which the corresponding regular expressions are matched in the input stream.

The lexical analysis programs written with Lex accept ambiguous specifications and choose the longest match possible at each input point. If necessary, substantial look-ahead is performed on the input, but the input stream will be backed up to the end of the final string matched, making this look-ahead invisible to the user.

For a trivial example, consider the specification for a program to delete from the input text all appearances of the word *theoretical*.

```
%%
theoretical   ;
```

This specification contains a %% delimiter to mark the beginning of the rules, and one rule. This rule contains a regular expression which matches precisely the string of characters "theoretical." No action is specified, so when these characters are seen, they are ignored. All characters which are not matched by some rule are

copied to the output, so all the rest of the text is copied. To also change *theory* to *practice*, just add another rule:

```
%%
theoretical          ;
theory               printf( "practice" );
```

The finite automaton generated for this source will scan for both rules at once, and, when a match is found, execute the desired rule action.

Lex-generated programs can handle data that may require substantial lookahead. For example, suppose there were a third rule, matching **the**, and the input data was the text string **theoretician**. The automaton generated by Lex would have to read the initial string **theoretici** before realizing that the input will not match **theoretical**. It then backs up the input, matching **the**, and leaving the input poised to read **oretician**. Such backup is more costly than the processing of simpler specifications.

As with Yacc, Lex actions may be general program fragments. Since the input is believed to be text, a character array (called **yytext**) can be used to hold the string which was matched by the rule. Actions can obtain the actual characters matched by accessing this array.

The structure of Lex output is similar to that of Yacc. A function named **yylex** is produced, which contains tables and an interpreter representing a deterministic finite automaton. By default, **yylex** is invoked from the main program, and it reads characters from the standard input. The user may provide his own main program, however. Alternatively, when Yacc is used, it automatically generates calls to **yylex** to obtain input tokens. In this case, each Lex rule which recognizes a token should have as an action

return (*token-number*)

to signal the kind of token recognized to the parser. It may also assign a value to **yylval** if desired.

The user can also change the Lex input routines, so long as it is remembered that Lex expects to be able to look ahead on and then back up the input stream. Thus, as with Yacc, user code may be above, within, and below the Lex automaton. It is even easy to have a lexical analyzer in which some tokens are recognized by the automaton and some by user-written code. This may be necessary when some input structure is not easily specified by even the large class of regular expressions supported by Lex.

The definitions of regular expressions are very similar to those in qed[8] and the UNIX text editor ed.[9] A regular expression specifies a set of strings to be matched. It contains text characters (which match the corresponding characters in the strings being compared) and operator characters (which specify repetitions, choices, and other features). The letters of the alphabet and the digits are always text characters; thus the regular expression

```
integer
```

matches the string *integer* wherever it appears and the expression

```
a57D
```

looks for the string *a57D*. It is also possible to use the standard C language escapes to refer to certain special characters, such as \n for newline and \t for tab. The operators may be used to:

(*i*) Specify a repetition of 0 or more, or 1 or more repetitions of a regular expression: * and +.
(*ii*) Specify that an expression is optional: ?.
(*iii*) Allow a choice of two or more patterns: |.
(*iv*) Match the beginning or the end of a line of text: ^ and $.
(*v*) Match any non-newline character: . (dot).
(*vi*) Group sub-expressions: (and).
(*vii*) Allow escaping and quoting special characters: \ and " .
(*viii*) Define classes of characters: [and].
(*ix*) Access defined patterns: { and }.
(*x*) Specify additional right context: /.

Some simple examples are

```
[0-9]
```

which recognizes the individual digits from 0 through 9,

```
[0-9]+
```

which recognizes strings of one or more digits, and

```
-?[0-9]+
```

which recognizes strings of digits optionally preceded by a minus sign. A more complicated pattern is

```
[A-Za-z][A-Za-z0-9]*
```

which matches all alphanumeric strings with a leading alphabetic

character. This is a typical expression for recognizing identifiers in computer programming languages.

Lex programs go beyond the pure theory of regular expressions in their ability to recognize patterns. As one example, Lex rules can recognize a small amount of surrounding context. The two simplest operators for this are ^ and $. If the first character of an expression is ^, the expression will only be matched at the beginning of a line (after a newline character, or at the beginning of the input stream). If the very last character is $, the expression will only be matched at the end of a line (when immediately followed by a newline). The latter operator is a special case of the / operator, which indicates trailing context. The expression

```
ab/cd
```

matches the string *ab*, but only if followed by *cd*. Left context is handled in Lex by *start conditions*. In effect, start conditions can be used to selectively enable or disable sets of rules, depending on what has come before.

Another feature of Lex is the ability to handle ambiguous specifications. When more than one expression can match the current input, Lex chooses as follows:

(*i*) The longest match is preferred.
(*ii*) Among rules which matched the same number of characters, the rule given first is preferred.

Thus, suppose the rules

```
integer          keyword action ...;
[a-z]+           identifier action ...;
```

to be given in that order. If the input is *integers*, it is taken as an identifier, because `[a-z]+` matches 8 characters while `integer` matches only 7. If the input is *integer*, both rules match 7 characters, and the keyword rule is selected because it was given first. Anything shorter (e.g., *int*) will not match the expression `integer` and so the identifier interpretation is used.

Note that a Lex program normally partitions the input stream, rather than search for all possible matches of each expression. This means that each character is accounted for once and only once. Sometimes the user would like to override this choice. The action REJECT means "go do the next alternative." It causes to be executed whatever rule was next choice after the current rule. The position of the input pointer is adjusted accordingly. In general, REJECT is

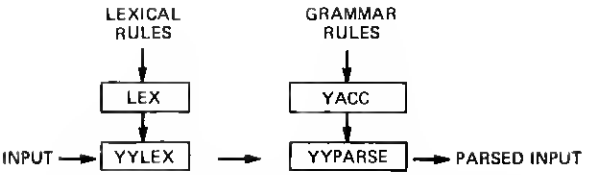

Fig. 2—Yacc and Lex cooperating.

useful whenever the purpose of a Lex program is not to partition the input stream but to detect all examples of some items in the input, and the instances of these items may overlap or include each other.

IV. COOPERATION OF YACC AND LEX: AN EXAMPLE

This section gives an example of the cooperation of Yacc and Lex to do a simple program which, nevertheless, would be difficult to write directly in many high-level languages. Before the specific example, however, let us summarize the various mechanisms available for making Lex- and Yacc-generated programs cooperate.

Since Yacc generates parsers and Lex can be used to make lexical analyzers, it is often desirable to use them together to make the first stage of a language analyzer. In such an application, two specifications are needed: a set of lexical rules to define the input data tokens and a set of grammar rules to define how these tokens may appear in the language. The input data text is read, divided up into tokens by the lexical analyzer, and then passed to the parser and organized into the larger structures of the input language. In principle, this could be done with pipes, but usually the code produced by Lex and Yacc are compiled together to produce one program for execution. Conventionally, the Yacc program is named **yyparse** and it calls a program named **yylex** to obtain tokens; therefore, this is the name used by Lex for its output source program. The overall appearance is shown in Fig. 2.

To make this cooperation work, it is necessary for Yacc and Lex to agree on the numeric codes used to differentiate token types. These codes can be specified by the user, but ordinarily the user allows Yacc to choose these numbers, and Lex obtains the values by including a header file, written by Yacc, which contains the definitions. It is also necessary to provide a mechanism by which Yacc can obtain the values of tokens returned from Lex. These values are passed through the external variable **yylval**.

Yacc and Lex were designed to work together, and are frequently

used together. The programs using this technology include the portable C compiler and the C language preprocessor.

As a simple example, we shall specify a complete program which will allow the input of dates, such as

July 4, 1776

and it will output the days of the week on which they fall. The program will also permit dates to be input as three numbers, separated by slashes:

7 / 4 / 1776

and in European format:

4 July 1776

Moreover, the month names can be given by their common abbreviations (with an optional '.' following) or spelled in full, but nothing in between.

Conceptually, there are three parts of the program. The Yacc specification describes a list of dates, one per line, in terms of the two tokens DIGIT and MONTH, and various punctuation symbols such as comma and newline. The Lex specification recognizes MONTHs and DIGITs, deletes blanks, and passes other characters through to Yacc. Finally, the Yacc actions call a set of routines which actually carry out the day of the week computation. We will discuss each of these in turn.

```
%token DIGIT MONTH
%%
input :     /* empty file is legal */
      |     input date '\n'
      |     input error '\n'
                  { yyerrok;  /* ignore line if error */ }
      ;
date  :     MONTH day ',' year
                  { date( $1, $2, $4 ); }
      |     day MONTH year
                  { date( $2, $1, $4 ); }
      |     number '/' number '/' number
                  { date( $1, $3, $5 ); }
      ;
day   :     number
      ;
```

```
year    :       number
        ;
number  :       DIGIT
        |       number DIGIT
                        {  $$ = 10 * $1 + $2; }
        ;
```

The Yacc specification file is quite simple. The first line declares the two names DIGIT and MONTH as tokens, whose meaning is to be supplied by the lexical analyzer. The %% mark separates the declarations from the rules. The input is described as either empty or some input followed by a date and a newline. Another rule specifies error recovery action in case a line is entered with an illegally formed date; the parser is to skip to the end of the line and then behave as if the error had never been seen.

Dates are legal in the three forms discussed above. In each case, the effect is to call the routine **date**, which does the work required to actually figure out the day of the week. The syntactic categories **day** and **year** are simply numbers; the routine **date** checks them to ensure that they are in the proper range. Finally, numbers are either DIGITs or a number followed by a DIGIT. In the latter case, an action is supplied to return the decimal value of the number. In the case of the first rule, the action

```
{  $$ = $1;  }
```

is the implied default, and need not be specified.

Note that the Yacc specification assumes that the lexical analyzer returns values 0 through 9 for the DIGITs, and a month number from 1 to 12 for the MONTHs.

We turn now to the Lex specification.

```
%{
# include "y.tab.h"
extern int yylval;
# define MON(x) {yylval= x; return(MONTH);}
%}
%%
Jan("."|uary)?          MON(1);
Feb("."|ruary)?         MON(2);
Mar("."|ch)?            MON(3);
Apr("."|il)?            MON(4);
May                     MON(5);
```

```
Jun("."|e)?          MON(6);
Jul("."|y)?          MON(7);
Aug("."|ust)?        MON(8);
Sep("."|"t"|"t."|tember)?  MON(9);
Oct("."|ober)?       MON(10);
Nov("."|ember)?      MON(11);
Dec("."|ember)?      MON(12);
[0-9]                {    yylval = yytext[0] - '0';
                          return( DIGIT ); }
[ ]                  {  ;  /* delete blanks */ }
"\n"                 |
.                    {  return( yytext[0] );  /* return
                                single characters */ }
```

The Lex specification includes the file y.tab.h which is produced by Yacc; this defines the token names DIGIT and NUMBER, so they can be used by the Lex program. The variable yylval is defined, which is used to communicate the values of the tokens to Yacc. Finally, to make it easier to return the values of MONTHs the macro MON is defined which assigns its argument to yylval and returns MONTH.

The next portion of the Lex specification is concerned with the month names. Typically, the full month name is legal, as well as the three-letter abbreviation, with or without a following period. The action when a month name is recognized is to set yylval to the number of the month, and return the token indication MONTH; this tells Yacc that a MONTH has been seen. Similarly, the digits 0 through 9 are recognized as a character class, their value stored into yylval, and the indication DIGIT returned. The remaining rules serve to delete blanks, and to pass all other characters, including newline, to Yacc for further processing.

Finally, for completeness, we present the subroutine date which actually carries out the computation. A good fraction of the logic is concerned with leap years, in particular the rather baroque rule that a year is a leap year if it is exactly divisible by 4, and not exactly divisible by 100 unless it is also divisible by 400. Notice also that the month and day are checked to ensure that they are in range.

```
/* here are the routines that really do the work */
# include <stdio.h>
int noleap [] {
```

```
        0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31,
        };
int leap[] {
        0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31,
        };
char * dayname[] {
        "Sunday",
        "Monday",
        "Tuesday",
        "Wednesday",
        "Thursday",
        "Friday",
        "Saturday",
        };
date( month, day, year ){ /* this routine does the real work */
        int *daysin;
        daysin = isleap( year ) ? leap : noleap;
        /* check the month */
        if( month < 1 || month > 12 ){
                printf( "month out of range\n" );
                return;
                }
        /* check the day of the month */
        if( day < 1 || day > daysin[month] ){
                printf( "day of month out of range\n" );
                return;
                }
        /* now, take the day of the month,
        add the days of previous months */
        while( month > 1 ) day += daysin[ -- month ];
        /* now, make day (mod 7) offset from Jan 1, 0000 */
        if( year > 0 ){
                --year;  /* make corrections for previous years */
                day += year;  /* since 365 = 1 (mod 7) */
                /* leap year correction */
                day += year/4 - year/100 + year/400;
                }
        /* Jan 1, 0000 was a Sunday, so no correction needed */
        printf( "    %s\n", dayname[day%7] );
        }
isleap( year ){
        if( year % 4 != 0 ) return( 0 );  /* not a leap year */
```

```
    if( year % 100 != 0 ) return( 1 );  /* is a leap year */
    if( year % 400 != 0 ) return( 0 );  /* not a leap year */
    return( 1 );  /* a leap year */
    }
```

Some of the Lex specification (such as the optional period after month names) might have been done in Yacc. Notice also that some of the things done in Yacc (such as the recognition of numbers) might have been done in Lex. Moreover, additional checking (such as ensuring that days of the month have only one or two digits) might have been placed into Yacc. In general, there is considerable flexibility in dividing the work between Yacc, Lex, and the action programs.

As an exercise, the reader might consider how this program might be written in his favorite programming language. Notice that the Lex program takes care of looking ahead on the input stream, and remembering characters that may delimit tokens but not be part of them. The Yacc program arranges to specify alternative forms and is clearly easy to expand. In fact, this example uses none of the precedence and little of the powerful recursive features of Yacc. Finally, languages such as Snobol in which one might reasonably do the same things as Yacc and Lex do, for this example, would be very unpleasant to write the **date** function in. Practical applications of both Yacc and Lex frequently run to hundreds of rules in the specifications.

V. CONCLUSIONS

Yacc and Lex are quite specialized tools by comparison with some "compiler-writing" systems. To us this is an advantage; it is a deliberate effort at modular design. Rather than grouping tools into enormous packages, enforcing virtually an entire way of life onto a user, we prefer a set of individually small and adaptable tools, each doing one job well. As a result, our tools are used for a wider variety of jobs than most; we have jocularly defined a successful tool as one that was used to do something undreamed of by its author (both Yacc and Lex are successful by this definition).

More seriously, a successful tool must be used. A form of Darwinism is practiced on our UNIX system; programs which are not used are removed from the system. Utilities thus compete for the available jobs and users. Lex, for example, seems to have found an ecological niche in the input phase of programs which accept

complex input languages. Originally it had been thought that it would also be employed for jobs now handled by editor scripts, but most users seem to be sticking with the various editors. Some particularly complicated rearrangements (those which involve memory), however, are done with Lex. Data validation and statistics gathering is still an open area; the editors are unsuitable, and Lex competes with C programs and a new language called **awk**,[10] with no tool having clear dominance. Yacc has a secure role as the major tool now used for the first pass of compilers. It is also used for complex input to many application programs, including Lex, the desk calculator **bc**, and the typesetting language **eqn**. Yacc is also used, with or without Lex, for some kinds of syntactic data validation.

Packaging is very important. Ideally, these tools would be available in several forms. Among the possible modes of access to an algorithm might be a subroutine library, a program generator, a command, or a full compiler. Of these, the program generator is very attractive. It does not restrict the user as much as a command or full compiler, since it is mixed with the user's own code in the host language. On the other hand, since the generator has a reasonable overview of the user's job, it can be more powerful than a subroutine library. Few operating systems today make it possible to have an algorithm available in all forms without additional work, and the program generator is a suitable compromise. The previous compiler-compiler system on UNIX was a more restrictive and inclusive system, TMG,[11] and it is now almost unused. All the users seem to prefer the greater flexibility of the program generators.

The usability and portability of the generators, however, depend on the host language(s). The host language has a difficult task: it must be a suitable target for both user and program generator. It also must be reasonably portable; otherwise, the generator output is not portable. Efficiency is important; the generator must write sufficiently good code that the users do not abandon it to write their own code directly. The host language must also not constrain the generator unduly; for example, mechanically generated **gotos** are not as dangerous as hand-written ones and should not be forbidden. As a result, the best host languages are relatively low level. Another way of seeing this is to observe that if the host language has many complex compiling algorithms in it already, there may not be much scope left for the generator. On the other hand if the language is too low level (after all, the generators typically would have little trouble writing in assembler), the users cannot use it.

What is needed is a semantically low-level but syntactically convenient (and portable) language; C seems to work well.

Giving machine-generated code to a compiler designed for human use sometimes creates problems, however. Compiler writers may limit the number of initializers that can be written to several hundred, for example, since "clearly" no reasonable user would write more than that number of array elements by hand. Unfortunately, Yacc and Lex can generate arrays of thousands of elements to describe their finite automata. Also, if the compiler lacks a facility for adjusting diagnostic line numbers, the error messages will not reflect the extra step in code generation, and the line numbers in them may be unrelated to the user's original input file. (Remember that, although the generated code is presumably error-free, there is also user-written code intermixed).

Other difficulties arise from built-in error checking and type handling. Typically, the generator output is quite reliable, as it often is based on a tight mathematical model or construction. Thus, often one may have nearly perfect confidence that array bounds do not overflow, that defaults in switches are never taken, etc. Nevertheless, compilers which provide such checks often have no way of selectively, or generally, overriding them. In the worst case, this checking can dominate the inner loop of the algorithm embodied in the generated module, removing a great deal of the attractiveness of the generated program.

We should not exaggerate the problems with host languages: in general, C has proven very suitable for the job. The concept of splitting the work between a generator and a host language is very profitable for both sides; it relieves pressure on the host language to provide many complex features, and it relieves pressure on the program generators to turn into complete general-purpose languages. It encourages modularity; and beneath all the buzzwords of "top-down design" and "structured programming," modularity is really what good programming is all about.

REFERENCES

1. B. W. Kernighan, M. E. Lesk, and J. F. Ossanna, "UNIX Time-Sharing System: Document Preparation," B.S.T.J., this issue, pp. 2115-2135.
2. A. V. Aho and J. D. Ullman, *Principles of Compiler Design*, Reading, Mass.: Addison-Wesley, 1977.
3. S. C. Johnson, "Yacc — Yet Another Compiler-Compiler," Comp. Sci. Tech. Rep. No. 32, Bell Laboratories (July 1975).
4. M. E. Lesk, "Lex — A Lexical Analyzer Generator," Comp. Sci. Tech. Rep. No. 39, Bell Laboratories (October 1975).

5. B. W. Kernighan, "Ratfor — A Preprocessor for a Rational Fortran," Software — Practice and Experience, *5* (1975), pp. 395-406.
6. A. V. Aho, S. C. Johnson, and J. D. Ullman, "Deterministic Parsing of Ambiguous Grammars," Commun. Assn. Comp. Mach., *18* (August 1975), pp. 441-452.
7. A. V. Aho and S. C. Johnson, "LR Parsing," Comp. Surveys, *6* (June 1974), pp. 99-124.
8. B. W. Kernighan, D. M. Ritchie, and K. Thompson, "QED Text Editor," Comp. Sci. Tech. Rep. No. 5, Bell Laboratories (May 1972).
9. K. Thompson and D. M. Ritchie, *UNIX Programmer's Manual,* Bell Laboratories, May 1975, section ed(1).
10. A. V. Aho, B. W. Kernighan, and P. J. Weinberger, unpublished work (1977).
11. R. M. McClure, "TMG—a Syntax Directed Compiler," Proc. 20th ACM National Conf. (1965), pp. 262-274.