IEEE P1003.2 Draft 11.2 – September 1991

Copyright © 1991 by the
Institute of Electrical and Electronics Engineers, Inc.
345 East 47th Street
New York, NY 10017, USA
All rights reserved as an unpublished work.

This is an unapproved and unpublished IEEE Standards Draft, subject to change. The publication, distribution, or copying of this draft, as well as all derivative works based on this draft, is expressly prohibited except as set forth below.

Permission is hereby granted for IEEE Standards Committee participants to reproduce this document for purposes of IEEE standardization activities only, and subject to the restrictions contained herein.

Permission is hereby also granted for member bodies and technical committees of ISO and IEC to reproduce this document for purposes of developing a national position, subject to the restrictions contained herein.

Permission is hereby also granted to the preceding entities to make limited copies of this document in an electronic form only for the stated activities.

The following restrictions apply to reproducing or transmitting the document in any form: 1) all copies or portions thereof must identify the document's IEEE project number and draft number, and must be accompanied by this entire notice in a prominent location; 2) no portion of this document may be redistributed in any modified or abridged form without the prior approval of the IEEE Standards Department.

Other entities seeking permission to reproduce this document, or any portion thereof, for standardization or other activities, must contact the IEEE Standards Department for the appropriate license.

Use of information contained in this unapproved draft is at your own risk.

IEEE Standards Department
Copyright and Permissions
445 Hoes Lane, P.O. Box 1331
Piscataway, NJ 08855-1331, USA
+1 (908) 562-3800
+1 (908) 562-1571 [FAX]

Table A-1 – `lex` Table Size Declarations

Declaration	Description	Minimum Value
`%p` *n*	Number of positions	2500
`%n` *n*	Number of states	500
`%a` *n*	Number of transitions	2000
`%e` *n*	Number of parse tree nodes	1000
`%k` *n*	Number of packed character classes	1000
`%o` *n*	Size of the output array	3000

A.2.7.2 `lex` Rules

The rules in `lex` source files are a table in which the left column contains regular expressions and the right column contains actions (C program fragments) to be executed when the expressions are recognized.

ERE action
ERE action
...

The extended regular expression (*ERE*) portion of a rule shall be separated from *action* by one or more `<blank>`s. A regular expression containing `<blank>`s shall be recognized under the following conditions: the entire expression appears within double-quotes; or, the `<blank>`s appear within double-quotes or square brackets; or, each `<blank>` is preceded by a backslash character.

A.2.7.3 `lex` User Subroutines

Anything in the user subroutines section shall be copied to `lex.yy.c` following *yylex* ().

A.2.7.4 `lex` Regular Expressions

The `lex` utility shall support the set of extended regular expressions (see 2.8.4), with the following additions and exceptions to the syntax:

`"..."` Any string enclosed in double-quotes shall represent the characters within the double-quotes as themselves, except that backslash escapes (which appear in Table A-2) shall be recognized. Any backslash-escape sequence shall be terminated by the closing quote. For example, `"\01""1"` represents a single string: the octal value 1 followed by the character 1.

<state >r
<state1 ,state2 ,...>r
The regular expression *r* shall be matched only when the program

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

is in one of the start conditions indicated by *state* , *state1* , etc.; see A.2.7.5. (As an exception to the typographical conventions of the rest of this standard, in this case <*state* > does not represent a metavariable, but the literal angle-bracket characters surrounding a symbol.) The start condition shall be recognized as such only at the beginning of a regular expression.

r/*x* The regular expression *r* shall be matched only if it is followed by an occurrence of regular expression *x*. The token returned in *yytext* shall only match *r*. If the trailing portion of *r* matches the beginning of *x*, the result is unspecified. The *r* expression cannot include further trailing context or the `$` (match-end-of-line) operator; *x* cannot include the `^` (match-beginning-of-line) operator, nor trailing context, nor the `$` operator. That is, only one occurrence of trailing context is allowed in a `lex` regular expression, and the `^` operator only can be used at the beginning of such an expression.

{*name* } When *name* is one of the substitution symbols from the *Definitions* section (see A.2.7.1), the string, including the enclosing braces, shall be replaced by the *substitute* value. The *substitute* value shall be treated in the extended regular expression as if it were enclosed in parentheses. No substitution shall occur if {*name* } occurs within a bracket expression or within double-quotes.

Within an ERE , a backslash character shall be considered to begin an escape sequence as specified in Table 2-15 (see 2.12). In addition, the escape sequences in Table A-2 shall be recognized.

A literal `<newline>` character cannot occur within an ERE ; the escape sequence `\n` can be used to represent a `<newline>`. A `<newline>` shall not be matched by a period operator.

The order of precedence given to extended regular expressions for `lex` differs from that specified in Table 2-13. The order of precedence for `lex` shall be as shown in Table A-3, from high to low.

NOTE: The escaped characters entry is not meant to imply that these are operators, but they are included in the table to show their relationships to the true operators. The start condition, trailing context, and anchoring notations have been omitted from the table because of the placement restrictions described in this subclause; they can only appear at the beginning or ending of an ERE.

The ERE anchoring operators (`^` and `$`) do not appear in Table A-3. With `lex` regular expressions, these operators are restricted in their use: the `^` operator can only be used at the beginning of an entire regular expression, and the `$` operator only at the end. The operators apply to the entire regular expression. Thus, for example, the pattern `(^abc)|(def$)` is undefined; it can instead be written as two separate rules, one with the regular expression `^abc` and one with `def$`, which share a common action via the special `|` action (see below). If the pattern were written `^abc|def$`, it would match either of `abc` or `def` on a line by itself. Note also that `$` is a form of trailing context (it is equivalent to `/\n`) and as such

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

Table A-2 – `lex` Escape Sequences

Escape Sequence	Description	Meaning
digits	<backslash> followed by the longest sequence of one, two, or three octal-digit characters (01234567). If all of the digits are 0, (i.e., representation of the NUL character), the behavior is undefined.	The character whose encoding is represented by the one-, two-, or three-digit octal integer. If the size of a byte on the system is greater than nine bits, the valid escape sequence used to represent a byte is implementation-defined. Multibyte characters require multiple, concatenated escape sequences of this type, including the leading \ for each byte.
\x*digits*	<backslash> followed by the longest sequence of hexadecimal-digit characters (01234567abcdef ABCDEF). If all of the digits are 0, (i.e., representation of the NUL character), the behavior is undefined.	The character whose encoding is represented by the hexadecimal integer.
c	<backslash> followed by any character not described in this table or in Table 2-15	The character *c*, unchanged.

Table A-3 – `lex` ERE Precedence

	collation-related bracket symbols	[= =] [: :] [. .]
	escaped characters	\<*special character* >
	bracket expression	[ ]
	quoting	"..."
	grouping	( )
	definition	{*name* }
	single-character RE duplication	* + ?
	concatenation	
	interval expression	{*m* , *n*}
	alternation	|

cannot be used with regular expressions containing another instance of the operator (see the preceding discussion of trailing context).

The additional regular expressions trailing-context operator / can be used as an ordinary character if presented within double-quotes, "/"; preceded by a backslash, \/; or within a bracket expression, [/]. The start-condition < and > operators shall be special only in a start condition at the beginning of a regular

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

expression; elsewhere in the regular expression they shall be treated as ordinary characters.

A.2.7.5 `lex` Actions

The action to be taken when an *ERE* is matched can be a C program fragment or the special actions described below; the program fragment can contain one or more C statements, and can also include special actions. The empty C statement `;` shall be a valid action; any string in the `lex.yy.c` input that matches the pattern portion of such a rule is effectively ignored or skipped. However, the absence of an action shall not be valid, and the action `lex` takes in such a condition is undefined.

The specification for an action, including C statements and/or special actions, can extend across several lines if enclosed in braces:

ERE <*blank(s)*> `{` *program statement*
program statement `}`

The default action when a string in the input to a `lex.yy.c` program is not matched by any expression shall be to copy the string to the output. Because the default behavior of a program generated by `lex` is to read the input and copy it to the output, a minimal `lex` source program that has just `%%` shall generate a C program that simply copies the input to the output unchanged.

Four special actions shall be available: ‘`|`’’,‘`ECHO;`’’,‘`REJECT;`’’,and ‘`BEGIN`’’:

`|` The action `|` means that the action for the next rule is the action for this rule. Unlike the other three actions, `|` cannot be enclosed in braces or be semicolon-terminated; it shall be specified alone, with no other actions.

`ECHO;` Write the contents of the string *yytext* on the output.

`REJECT;` Usually only a single expression is matched by a given string in the input. `REJECT` means ‘‘continue to the next expression that matches the current input,’’ and causes whatever rule was the second choice after the current rule to be executed for the same input. Thus, multiple rules can be matched and executed for one input string or overlapping input strings. For example, given the regular expressions `xyz` and `xy` and the input `xyz`, usually only the regular expression `xyz` would match. The next attempted match would start after `z`. If the last action in the `xyz` rule is `REJECT`, both this rule and the `xy` rule would be executed. The `REJECT` action may be implemented in such a fashion that flow of control does not continue after it, as if it were equivalent to a `goto` to another part of *yylex* (). The use of `REJECT` may result in somewhat larger and slower scanners.

`BEGIN` The

`BEGIN` *newstate* ;

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

action switches the state (start condition) to *newstate* . If the string *newstate* has not been declared previously as a start condition in the *Definitions* section, the results are unspecified. The initial state is indicated by the digit `0` or the token `INITIAL`.

The functions or macros described below are accessible to user code included in the `lex` input. It is unspecified whether they appear in the C code output of `lex`, or are accessible only through the `-l l` operand to `c89` (the `lex` library).

`int yylex(void)` Performs lexical analysis on the input; this is the primary function generated by the `lex` utility. The function shall return zero when the end of input is reached; otherwise it shall return nonzero values (tokens) determined by the actions that are selected.

`int yymore(void)` When called, indicates that when the next input string is recognized, it is to be appended to the current value of *yytext* rather than replacing it; the value in *yyleng* shall be adjusted accordingly.

`int yyless(int n)` Retains *n* initial characters in *yytext* , NUL -terminated, and treats the remaining characters as if they had not been read; the value in *yyleng* shall be adjusted accordingly.

`int input(void)` Returns the next character from the input, or zero on end of file. It shall obtain input from the stream pointer *yyin* , although possibly via an intermediate buffer. Thus, once scanning has begun, the effect of altering the value of *yyin* is undefined. The character read is removed from the input stream of the scanner without any processing by the scanner.

`int unput(int c)` Returns the character *c* to the input; *yytext* and *yyleng* are undefined until the next expression is matched. The result of *unput* ting more characters than have been input is unspecified.

The following functions appear only in the `lex` library accessible through the `-l l` operand; they can therefore be redefined by a portable application:

`int yywrap(void)` Called by *yylex* () at end of file; the default *yywrap* () always shall return 1. If the application requires *yylex* () to continue processing with another source of input, then the application can include a function *yywrap* (), which associates another file with the external variable `FILE *`*yyin* and shall return a value of zero.

`int main(int argc, char *argv[])`
Calls *yylex* () to perform lexical analysis, then exits. The user code can contain *main* () to perform application-specific operations, calling *yylex* () as

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

771 applicable.

772 Except for *input* (), *unput* (), and *main* (), all external and static names generated
773 by `lex` shall begin with the prefix `yy` or `YY`.

774 **A.2.8 Exit Status**

775 The `lex` utility shall exit with one of the following values:

776 0 Successful completion.

777 >0 An error occurred.

778 **A.2.9 Consequences of Errors**

779 Default.

780 **A.2.10 Rationale.** *(This subclause is not a part of P1003.2)*

781 **Examples, Usage**

782 The following is an example of a `lex` program that implements a rudimentary
783 scanner for a Pascal-like syntax:

```
784 %{
785 /* need this for the call to atof() below */
786 #include <math.h>
787 /* need this for printf(), fopen(), and stdin below */
788 #include <stdio.h>
789 %}
790 DIGIT    [0-9]
791 ID       [a-z][a-z0-9]*
792 %%
793 {DIGIT}+    {
794         printf("An integer: %s (%d)\n", yytext,
795                 atoi(yytext));
796         }
797 {DIGIT}+"."{DIGIT}*        {
798         printf("A float: %s (%g)\n", yytext,
799                 atof(yytext));
800         }
801 if|then|begin|end|procedure|function        {
802         printf("A keyword: %s\n", yytext);
803         }
804 {ID}    printf("An identifier: %s\n", yytext);
805 "+"|"-"|"*"|"/"         printf("An operator: %s\n", yytext);
806 "{"[^}\n]*"}"           /* eat up one-line comments */
```

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

```
[ \t\n]+                /* eat up white space */
.       printf("Unrecognized character: %s\n", yytext);
%%
int main(int argc, char *argv[])
{
        ++argv, --argc;  /* skip over program name */
        if (argc > 0)
                yyin = fopen(argv[0], "r");
        else
                yyin = stdin;
        yylex();
}
```

The following examples have been included to clarify the differences between `lex` regular expressions and regular expressions appearing elsewhere in this document. For regular expressions of the form *r*/*x*, the string matching *r* is always returned; confusion may arise when the beginning of *x* matches the trailing portion of *r*. For example, given the regular expression `a*b/cc` and the input `aaabcc`, *yytext* would contain the string `aaab` on this match. But given the regular expression `x*/xy` and the input `xxxy`, the token `xxx`, not `xx`, is returned by some implementations because `xxx` matches `x*`.

In the rule `ab*/bc`, the `b*` at the end of *r* will extend *r*'s match into the beginning of the trailing context, so the result is unspecified. If this rule were `ab/bc`, however, the rule matches the text `ab` when it is followed by the text `bc`. In this latter case, the matching of *r* cannot extend into the beginning of *x*, so the result is specified.

Unlike the general ERE rules, embedded anchoring is not allowed by most historical `lex` implementations. An example of embedded anchoring would be for patterns such as `(^| )foo( |$)` to match `foo` when it exists as a complete word. This functionality can be obtained using existing `lex` features:

```
^foo/[ \n]      |
" foo"/[ \n]    /* found foo as a separate word */
```

The precedence of regular expressions in `lex` does not match that of extended regular expressions in Section 2 because of historical practice. In System V `lex` and its predecessors, a regular expression of the form `ab{3}` matches `ababab`; an ERE, such as used by `egrep`, would match `abbb`. Changing this precedence for uniformity with `egrep` would have been desirable, but too many applications would break in nonobvious ways.

Conforming applications are warned that in the *Rules* section, an *ERE* without an action is not acceptable, but need not be detected as erroneous by `lex`. This may result in compilation or run-time errors.

The purpose of *input* () is to take characters off the input stream and discard them as far as the lexical analysis is concerned. A common use is to discard the body of a comment once the beginning of a comment is recognized.

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

History of Decisions Made

Even though the −c option and references to the C language are retained in this description, lex may be generalized to other languages, as was done at one time for EFL , Extended FORTRAN Language. Since the lex input specification is essentially language independent, versions of this utility could be written to produce Ada, Modula-2, or Pascal code, and there are known historical implementations that do so.

The current description of lex bypasses the issue of dealing with internationalized regular expressions in the lex source code or generated lexical analyzer. If it follows the model used by awk, (the source code is assumed to be presented in the POSIX Locale, but input and output are in the locale specified by the environment variables), then the tables in the lexical analyzer produced by lex would interpret regular expressions specified in the lex source in terms of the environment variables specified when lex was executed. The desired effect would be to have the lexical analyzer interpret the regular expressions given in the lex source according to the environment specified when the lexical analyzer is executed, but this is not possible with the current lex technology.

Major international vendors believe that only limited internationalization is required for the POSIX. 2 lex. The theoretically desirable goal of runtime-selectable locales is not feasible in the near future. Furthermore, the very nature of the lexical analyzers produced by lex must be closely tied to the lexical requirements of the input language being described, which will frequently be locale-specific anyway. (For example, writing an analyzer that is used for French text will not automatically be useful for processing other languages.) The text in the Environment Variable subclause allows locale-specific regular expression handling, but mandates only something similar to that provided in historical implementations.

The description of octal- and hexadecimal-digit escape sequences agrees with the C Standard {7} use of escape sequences. See the rationale for ed for a discussion of bytes larger than nine bits being represented by octal values. Hexadecimal values can represent larger bytes and multibyte characters directly, using as many digits as required.

There is no detailed output format specification. The observed behavior of lex under four different historical implementations was that none of these implementations consistently reported the line numbers for error and warning messages. Furthermore, there was a desire that lex be allowed to output additional diagnostic messages. Leaving message formats unspecified sidesteps these formatting questions and also avoids problems with internationalization.

Although the %x specifier for exclusive start conditions is not existing practice, it is believed to be a minor change to historical implementations, and greatly enhances the usability of lex programs since it permits an application to obtain the expected functionality with fewer statements.

The %array and %pointer declarations were added as a compromise between historical systems. The System V-based lex has copied the matched text to a

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

yytext array. The `flex` program, supported in BSD and GNU systems, uses a pointer. In the latter case, significant performance improvements are available for some scanners. Most existing programs should require no change in porting from one system to another because the string being referenced is null-terminated in both cases. (The method used by `flex` in its case is to null-terminate the token in-place by remembering the character that used to come right after the token and replacing it before continuing on to the next scan.) Multifile programs with external references to *yytext* outside the scanner source file should continue to operate on their existing systems, but would require one of the new declarations to be considered strictly portable.

The description of regular expressions avoids unnecessary duplication of regular expression details. Specifically, the `|` operator and {*m,n* } interval expression are not listed in A.2.7.4 because their meanings within a `lex` regular expression are the same as that for extended regular expressions.

The reason for the undefined condition associated with text beginning with a `<blank>` or within `%{` and `%}` delimiter lines appearing in the *Rules* section is historical practice. Both BSD and System V `lex` copy the indented (or enclosed) input in the *Rules* section (except at the beginning) to unreachable areas of the *yylex* () function (the code is written directly after a `break` statement). In some cases, the System V `lex` generates an error message or a syntax error, depending on the form of indented input.

The intention in breaking the list of functions into those that may appear in `lex.yy.c` versus those that only appear in `libl.a` is that only those functions in `libl.a` can be reliably redefined by a portable application.

The descriptions of Standard Output and Standard Error are somewhat complicated because historical `lex` implementations chose to issue diagnostic messages to standard output (unless `−t` was given). POSIX. 2 allows this behavior, but leaves an opening for the more expected behavior of using standard error for diagnostics. Also, the System V behavior of writing the statistics when any table sizes are given is allowed, while BSD-derived systems can avoid it. The programmer can always precisely obtain the desired results by using either the `−t` or `−n` options.

The Operands subclause does not mention the use of – as a synonym for standard input; not all historical implementations support such usage for any of the *file* operands.

The description of the *Translation Table* was deleted from earlier drafts because of its relatively low usage in historical applications.

The change to the definition of the *input* () function that allows buffering of input presents the opportunity for major performance gains in some applications.

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

A.3 yacc — Yet another compiler compiler

A.3.1 Synopsis

yacc [–dltv][–b *file_prefix*][–p *sym_prefix*] *grammar*

A.3.2 Description

The yacc utility shall read a description of a context-free grammar in *file* and write C source code, conforming to the C Standard {7}, to a code file, and optionally header information into a header file, in the current directory. The C code shall define a function and related routines and macros for an automaton that executes a parsing algorithm meeting the requirements in A.3.7.8.

The form and meaning of the grammar is described in A.3.7.

The C source code and header file shall be produced in a form suitable as input for the C compiler (see c89 in A.1).

A.3.3 Options

The yacc utility shall conform to the utility argument syntax guidelines described in 2.10.2.

The following options shall be supported by the implementation:

–b *file_prefix*
: Use *file_prefix* instead of y as the prefix for all output filenames. The code file y.tab.c, the header file y.tab.h (created when –d is specified), and the description file y.output (created when –v is specified), shall be changed to *file_prefix* .tab.c, *file_prefix* .tab.h, and *file_prefix* .output, respectively.

–d
: Write the header file; by default only the code file is written.

–l
: Produce a code file that does not contain any #line constructs. If this option is not present, it is unspecified whether the code file or header file contains #line directives.

–p *sym_prefix*
: Use *sym_prefix* instead of yy as the prefix for all external names produced by yacc. The names affected shall include the functions *yyparse* (), *yylex* (), and *yyerror* (), and the variables *yylval* , *yychar* , and *yydebug* . (In the remainder of this clause, the six symbols cited are referenced using their default names only as a notational convenience.) Local names may also be affected by the –p option; however, the –p option shall not affect yacc-generated #define symbols.

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

968 `-t` Modify conditional compilation directives to permit compilation of
969 debugging code in the code file. Runtime debugging statements
970 shall be always contained in the code file, but by default condi-
971 tional compilation directives prevent their compilation.

972 `-v` Write a file containing a description of the parser and a report of
973 conflicts generated by ambiguities in the grammar.

974 ### A.3.4 Operands

975 The following operand is required:

976 *grammar* A pathname of a file containing instructions, hereafter called
977 *grammar* , for which a parser is to be created. The format for the
978 grammar is described in A.3.7.

979 ### A.3.5 External Influences

980 #### A.3.5.1 Standard Input

981 None.

982 #### A.3.5.2 Input Files

983 The file *grammar* shall be a text file formatted as specified in A.3.7.

984 #### A.3.5.3 Environment Variables

985 The following environment variables shall affect the execution of `yacc`:

986 **LANG** This variable shall determine the locale to use for the
987 locale categories when both **LC_ALL** and the correspond-
988 ing environment variable (beginning with **LC_**) do not
989 specify a locale. See 2.6.

990 **LC_ALL** This variable shall determine the locale to be used to over-
991 ride any values for locale categories specified by the set-
992 tings of **LANG** or any environment variables beginning
993 with **LC_**.

994 **LC_CTYPE** This variable shall determine the locale for the interpreta-
995 tion of sequences of bytes of text data as characters (e.g.,
996 single- versus multibyte characters in arguments and
997 input files).

998 **LC_MESSAGES** This variable shall determine the language in which mes-
999 sages should be written.

1000 The **LANG** and **LC_*** variables shall affect the execution of the `yacc` utility as
1001 stated. The *main* () function defined in A.3.7.6 shall call

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

```
setlocale(LC_ALL, "")
```

and thus, the program generated by `yacc` shall also be affected by the the contents of these variables at runtime.

A.3.5.4 Asynchronous Events

Default.

A.3.6 External Effects

A.3.6.1 Standard Output

None.

A.3.6.2 Standard Error

If shift/reduce or reduce/reduce conflicts are detected in *grammar* , `yacc` writes a report of those conflicts to the standard error in an unspecified format.

Standard error is also used for diagnostic messages.

A.3.6.3 Output Files

The code file, the header file, and the description file shall be text files. All are described in the following subclauses.

A.3.6.3.1 Code file

This file shall contain the C source code for the *yyparse* () routine. It shall contain code for the various semantic actions with macro substitution performed on them as described in A.3.7. It shall also contain a copy of the `#define` statements in the header file. If a `%union` declaration is used, the declaration for YYSTYPE shall be also included in this file.

The contents of the Program Section (see A.3.7.1.4) of the input file shall then be included.

A.3.6.3.2 Header file

The header file shall contain `#define` statements that associate the token numbers with the token names. This allows source files other than the code file to access the token codes. If a `%union` declaration is used, the declaration for YYSTYPE and an `extern YYSTYPE yylval` declaration shall be also included in this file.

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

A.3.6.3.3 Description file

The description file shall be a text file containing a description of the state machine corresponding to the parser, using an unspecified format. Limits for internal tables (see A.3.7.9) also shall be reported, in an implementation-defined manner.

A.3.7 Extended Description

The `yacc` command accepts a language that is used to define a grammar for a target language to be parsed by the tables and code generated by `yacc`. The language accepted by `yacc` as a grammar for the target language is described below using the `yacc` input language itself.

The input *grammar* includes rules describing the input structure of the target language, and code to be invoked when these rules are recognized to provide the associated semantic action. The code to be executed shall appear as bodies of text that are intended to be C language code. The C language inclusions are presumed to form a correct function when processed by `yacc` into its output files. The code included in this way shall be executed during the recognition of the target language.

Given a grammar, the `yacc` utility generates the files described in A.3.6.3. The code file can be compiled and linked using `c89`. If the declaration and programs sections of the grammar file did not include definitions of *main* (), *yylex* (), and *yyerror* (), the compiled output requires linking with externally supplied version of those functions. Default versions of *main* () and *yyerror* () are supplied in the `yacc` library and can be linked in by using the `-l y` operand to `c89`. The `yacc` library interfaces need not support interfaces with other than the default `yy` symbol prefix. The application provides the lexical analyzer function, *yylex* (); the `lex` utility (see A.2) is specifically designed to generate such a routine.

A.3.7.1 Input Language

Every specification file shall consist of three sections: *declarations* , *grammar rules* , and *programs* , separated by double percent-signs (%%). The declarations and programs sections can be empty. If the latter is empty, the preceding %% mark separating it from the rules section can be omitted.

The input is free form text following the structure of the grammar defined below.

A.3.7.1.1 Lexical Structure of the Grammar

The characters `<blank>`s, `<newline>`s, and `<form-feed>`s shall be ignored, except that they shall not appear in names or multicharacter reserved symbols. Comments shall be enclosed in `/* ... */`, and can appear wherever a name is valid.

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

Names are of arbitrary length, made up of letters, periods (.), underscores (_), and noninitial digits. Upper- and lowercase letters are distinct. Portable applications shall not use names beginning in `yy` or `YY` since the `yacc` parser uses such names. Many of the names appear in the final output of `yacc`, and thus they should be chosen to conform with any additional rules created by the C compiler to be used. In particular they will appear in `#define` statements.

A literal shall consist of a single character enclosed in single-quotes (′). All of the escape sequences supported for character constants by the C Standard {7}(3.1.3.4) shall be supported by `yacc`.

The relationship with the lexical analyzer is discussed in detail below.

The NUL character shall not be used in grammar rules or literals.

A.3.7.1.2 Declarations Section

The declarations section is used to define the symbols used to define the target language and their relationship with each other. In particular, much of the additional information required to resolve ambiguities in the context-free grammar for the target language is provided here.

Usually `yacc` assigns the relationship between the symbolic names it generates and their underlying numeric value. The declarations section makes it possible to control the assignment of these values.

It is also possible to keep semantic information associated with the tokens currently on the parse stack in a user-defined C language `union`, if the members of the union are associated with the various names in the grammar. The declarations section provides for this as well.

The first group of declarators below all take a list of names as arguments. That list can optionally be preceded by the name of a C union member (called a *tag* below) appearing within ‘‘<’’and ‘‘>’’. (As an exception to the typographical conventions of the rest of this standard, in this case *<tag>* does not represent a metavariable, but the literal angle bracket characters surrounding a symbol.) The use of *tag* specifies that the tokens named on this line are to be of the same C type as the union member referenced by *tag*. This is discussed in more detail below.

For lists used to define tokens, the first appearance of a given token can be followed by a positive integer (as a string of decimal digits). If this is done, the underlying value assigned to it for lexical purposes shall be taken to be that number.

`%token` [*<tag>*] *name* [*number*] [*name* [*number*]]...
: Declares *name* (s) to be a token. If *tag* is present, the C type for all tokens on this line shall be declared to be the type referenced by *tag*. If a positive integer, *number* , follows a *name* , that value shall be assigned to the token.

`%left` [*<tag>*] *name* [*number*] [*name* [*number*]]...
`%right` [*<tag>*] *name* [*number*] [*name* [*number*]]...
: Declares *name* to be a token, and assigns precedence to it. One or

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

more lines, each beginning with one of these symbols can appear in this section. All tokens on the same line have the same precedence level and associativity; the lines are in order of increasing precedence or binding strength. `%left` denotes that the operators on that line are left associative, and `%right` similarly denotes right associative operators. If *tag* is present, it shall declare a C type for *name* (s) as described for `%token`.

`%nonassoc` **[**<*tag*>**]** *name* **[***number* **]** **[***name* **[***number* **]]**...
Declares *name* to be a token, and indicates that this cannot be used associatively. If the parser encounters associative use of this token it shall report an error. If *tag* is present, it shall declare a C type for *name* (s) as described for `%token`.

`%type` <*tag*> *name* ...
Declares that union member *name* (s) are nonterminals, and thus it is required to have a *tag* field at its beginning. Because it deals with nonterminals only, assigning a token number or using a literal is also prohibited. If this construct is present, `yacc` shall perform type checking; if this construct is not present, the parse stack shall hold only the `int` type.

Every name used in *grammar* undefined by a `%token`, `%left`, `%right`, or `%nonassoc` declaration is assumed to represent a nonterminal symbol. The `yacc` utility shall report an error for any nonterminal symbol that does not appear on the left side of at least one grammar rule.

Once the type, precedence, or token number of a name is specified, it shall not be changed. If the first declaration of a token does not assign a token number, `yacc` shall assign a token number. Once this assignment is made, the token number shall not be changed by explicit assignment.

The following declarators do not follow the previous pattern.

`%start` *name*
Declares the nonterminal *name* to be the *start symbol* , which represents the largest, most general structure described by the grammar rules. By default, it is the left-hand side of the first grammar rule; this default can be overridden with this declaration.

`%union { ` *body of union (in C)* `}`
Declares the `yacc` value stack to be a union of the various types of values desired. By default, the values returned by actions (see below) and the lexical analyzer shall be integers. The `yacc` utility keeps track of types, and shall insert corresponding union member names in order to perform strict type checking of the resulting parser.

Alternatively, given that at least one <*tag*> construct is used, the union can be declared in a header file (which shall be included in the declarations section by using an `#include` construct within

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

%{ and %}), and a `typedef` used to define the symbol YYSTYPE to represent this union. The effect of `%union` is to provide the declaration of YYSTYPE directly from the input.

`%{ ... %}` C language declarations and definitions can appear in the declarations section, enclosed by these marks. These statements shall be copied into the code file, and have global scope within it so that they can be used in the rules and program sections.

The declarations section shall be terminated by the token `%%`.

A.3.7.1.3 Grammar Rules

The rules section defines the context-free grammar to be accepted by the function `yacc` generates, and associates with those rules C language actions and additional precedence information. The grammar is described below, and a formal definition follows.

The rules section is comprised of one or more grammar rules. A grammar rule has the form:

```
A : BODY ;
```

The symbol `A` represents a nonterminal name, and `BODY` represents a sequence of zero or more *name* s, *literal* s, and *semantic action* s that can then be followed by optional *precedence rule* s. Only the names and literals participate in the formation of the grammar; the semantic actions and precedence rules are used in other ways. The colon and the semicolon are `yacc` punctuation. If there are several successive grammar rules with the same left-hand side, the vertical bar | can be used to avoid rewriting the left-hand side; in this case the semicolon appears only after the last rule. The BODY part can be empty (or empty of names and literals) to indicate that the nonterminal symbol matches the empty string.

The `yacc` utility assigns a unique number to each rule. Rules using the vertical bar notation are distinct rules. The number assigned to the rule appears in the description file.

The elements comprising a BODY are:

name
literal These form the rules of the grammar: *name* is either a *token* or a *nonterminal* ; *literal* stands for itself (less the lexically required quotation marks).

semantic action
With each grammar rule, the user can associate actions to be performed each time the rule is recognized in the input process. [Note that the word ‘‘action’’ can also refer to the actions of the parser (shift, reduce, etc.).]

These actions can return values and can obtain the values returned by previous actions. These values shall be kept in objects of type YYSTYPE (see `%union`). The result value of the action shall be kept on the parse stack with the left-hand side of

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

the rule, to be accessed by other reductions as part of their right-hand side. By using the <*tag*> information provided in the declarations section, the code generated by `yacc` can be strictly type checked and contain arbitrary information. In addition, the lexical analyzer can provide the same kinds of values for tokens, if desired.

An action is an arbitrary C statement, and as such can do input or output, call subprograms, and alter external variables. An action is one or more C statements enclosed in curly braces `{` and `}`.

Certain pseudo-variables can be used in the action. These are macros for access to data structures known interally to `yacc`.

`$$` The value of the action can be set by assigning it to `$$`. If type checking is enabled and the type of the value to be assigned cannot be determined, a diagnostic message may be generated.

`$`*number*
This refers to the value returned by the component specified by the token *number* in the right side of a rule, reading from left to right; *number* can be zero or negative. If it is, it refers to the data associated with the name on the parser's stack preceding the leftmost symbol of the current rule. (That is, `$0` refers to the name immediately preceding the leftmost name in the current rule, to be found on the parser's stack, and `$-1` refers to the symbol to *its* left.) If *number* refers to an element past the current point in the rule, or beyond the bottom of the stack, the result is undefined. If type checking is enabled and the type of the value to be assigned cannot be determined, a diagnostic message may be generated.

`$`<*tag*>*number*
These correspond exactly to the corresponding symbols without the *tag* inclusion, but allow for strict type checking (and preclude unwanted type conversions). The effect is that the macro is expanded to use *tag* to select an element from the YYSTYPE union (using *dataname.tag*). This is particularly useful if *number* is not positive.

`$`<*tag*>`$`
This imposes on the reference the type of the union member referenced by *tag* . This construction is applicable when a reference to a left context value occurs in the grammar, and provides `yacc` with a means for selecting a type.

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

Actions can occur in the middle of a rule as well as at the end; an action can access values returned by actions to its left, and in turn the value it returns can be accessed by actions to its right. An action appearing in the middle of a rule shall be equivalent to replacing the action with a new nonterminal symbol and adding an empty rule with that nonterminal symbol on the left-hand side. The semantic action associated with the new rule shall be equivalent to the original action. The use of actions within rules might introduce conflicts that would not otherwise exist.

By default, the value of a rule shall be the value of the first element in it. If the first element does not have a type (particularly in the case of a literal) and type checking is turned on by `%type` an error message shall result.

precedence The keyword `%prec` can be used to change the precedence level associated with a particular grammar rule. Examples of this are in cases where a unary and binary operator have the same symbolic representation, but need to be given different precedences, or where the handling of an ambiguous if-else construction is necessary. The reserved symbol `%prec` can appear immediately after the body of the grammar rule and can be followed by a token name or a literal. It shall cause the precedence of the grammar rule to become that of the following token name or literal. The action for the rule as a whole can follow `%prec`.

If a program section follows, the grammar rules shall be terminated by %%.

A.3.7.1.4 Programs Section

The *programs* section can include the definition of the lexical analyzer *yylex* (), and any other functions, for example those used in the actions specified in the grammar rules. This is C language code, and shall be included in the code file after the tables and code generated by `yacc`. It is unspecified whether the programs section precedes or follows the semantic actions in the output file; therefore, if the application contains any macro definitions and declarations intended to apply to the code in the semantic actions, it shall place them within `%{ ... %}` in the declarations section.

A.3.7.1.5 Input Grammar

The following input to `yacc` yields a parser for the input to `yacc`. This is to be taken as the formal specification of the grammar of `yacc`, notwithstanding conflicts that may appear elsewhere.

The lexical structure is defined less precisely; the previous section on A.3.7.1.1 defines most terms. The correspondence between the previous terms and the tokens below is as follows.

`IDENTIFIER` This corresponds to the concept of *name* , given previously. It also includes literals as defined previously.

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

C_IDENTIFIER	This is a name, and additionally it is known to be followed by a colon. A literal cannot yield this token.
NUMBER	A string of digits (a nonnegative decimal integer).
TYPE LEFT MARK etc.	These correspond directly to %type, %left, %%, etc.
{ ... }	This indicates C language source code, with the possible inclusion of $ macros as discussed previously.

```
/*      Grammar for the input to yacc */

/*      Basic entries */

/*      The following are recognized by the lexical analyzer */

%token  IDENTIFIER      /* includes identifiers and literals */
%token  C_IDENTIFIER    /* identifier (but not literal)
                           followed by a : */
%token  NUMBER         /* [0-9][0-9]* */

/*      Reserved words : %type=>TYPE %left=>LEFT, etc. */

%token  LEFT RIGHT NONASSOC TOKEN PREC TYPE START UNION

%token  MARK            /* the %% mark */
%token  LCURL           /* the %{ mark */
%token  RCURL           /* the }% mark */

/*      8-bit character literals stand for themselves; */
/*      tokens have to be defined for multibyte characters */

%start  spec

%%

spec  : defs MARK rules tail
      ;

tail  : MARK
      {
        /* In this action, set up the rest of the file */
      }
      | /* empty; the second MARK is optional */
      ;

defs  : /* empty */
      | defs def
      ;

def   : START IDENTIFIER
      | UNION
      {
        /* Copy union definition to output */
```

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

```
1322        }
1323        | LCURL
1324        {
1325          /* Copy C code to output file */
1326        }
1327          RCURL
1328        | rword tag nlist
1329        ;
1330  rword : TOKEN
1331        | LEFT
1332        | RIGHT
1333        | NONASSOC
1334        | TYPE
1335        ;
1336  tag   : /* empty: union tag id optional */
1337        | '<' IDENTIFIER '>'
1338        ;
1339  nlist : nmno
1340        | nlist nmno
1341        ;
1342  nmno  : IDENTIFIER             /* Note: literal invalid with % type */
1343        | IDENTIFIER NUMBER       /* Note: invalid with % type */
1344        ;
1345  /*      rule section */
1346  rules : C_IDENTIFIER rbody prec
1347        | rules  rule
1348        ;
1349  rule  : C_IDENTIFIER rbody prec
1350        | '|' rbody prec
1351        ;
1352  rbody : /* empty */
1353        | rbody IDENTIFIER
1354        | rbody act
1355        ;
1356  act   : '{'
1357          {
1358                 /* Copy action, translate $$, etc. */
1359          }
1360          '}'
1361        ;
1362  prec  : /* empty */
1363        | PREC IDENTIFIER
1364        | PREC IDENTIFIER act
1365        | prec ';'
1366        ;
```

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.