-- --

IEEE P1003.2 Draft 11.2 – September 1991

A.2 `lex` — Generate programs for lexical tasks

A.2.1 Synopsis

`lex [−t] [ −n | −v ]` [*file* . . .]

Obsolescent Version :

`lex −c [−t] [ −n | −v ]` [*file* . . .]

A.2.2 Description

The `lex` utility shall generate C programs to be used in lexical processing of character input, and that can be used as an interface to `yacc` (see A.3). The C programs shall be generated from `lex` source code and conform to the C Standard {7}. Usually, the `lex` utility writes the program it generates to the file `lex.yy.c`; the state of this file is unspecified if `lex` exits with a nonzero exit status. See A.2.7 for a complete description of the `lex` input language.

A.2.3 Options

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

The following options shall be supported by the implementation:

`−c` (Obsolescent.) Indicate C-language action (default option).

`−n` Suppress the summary of statistics usually written with the `−v` option. If no table sizes are specified in the `lex` source code and the `−v` option is not specified, then `−n` is implied.

`−t` Write the resulting program to standard output instead of `lex.yy.c`.

`−v` Write a summary of `lex` statistics to the standard output. (See the discussion of `lex` table sizes in A.2.7.1.) If the `−t` option is specified and `−n` is not specified, this report shall be written to standard error. If table sizes are specified in the `lex` source code, and if the `−n` option is not specified, the `−v` option may be enabled.

A.2.4 Operands

The following operand shall be supported by the implementation:

file A pathname of an input file. If more than one such *file* is specified, all files shall be concatenated to produce a single `lex` program. If no *file* operands are specified, or if a *file* operand is −, the standard input shall be used.

A.2.5 External Influences

A.2.5.1 Standard Input

The standard input shall be used if no *file* operands are specified, or if a *file* operand is −. See Input Files.

A.2.5.2 Input Files

The input files shall be text files containing `lex` source code, as described in A.2.7.

A.2.5.3 Environment Variables

The following environment variables shall affect the execution of `lex`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_COLLATE This variable shall determine the locale for the behavior of ranges, equivalence classes, and multicharacter collating elements within regular expressions. If this variable is not set to the POSIX Locale, the results are unspecified.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments and input files) and the behavior of character classes within extended regular expressions. If this variable is not set to the POSIX Locale, the results are unspecified.

LC_MESSAGES This variable shall determine the language in which messages should be written.

A.2.5.4 Asynchronous Events

Default.

A.2.6 External Effects

A.2.6.1 Standard Output

If the `-t` option is specified, the text file of C source code output of `lex` shall be written to standard output.

If the `-t` option is not specified:

(1) Implementation-defined informational, error, and warning messages concerning the contents of `lex` source code input shall be written to either the standard output or standard error.

(2) If the `-v` option is specified and the `-n` option is not specified, `lex` statistics shall also be written to either the standard output or standard error, in an implementation-defined format. These statistics may also be generated if table sizes are specified with a `%` operator in the *Definitions* section (see A.2.7), as long as the `-n` option is not specified.

A.2.6.2 Standard Error

If the `-t` option is specified, implementation-defined informational, error, and warning messages concerning the contents of `lex` source code input shall be written to the standard error.

If the `-t` option is not specified:

(1) Implementation-defined informational, error, and warning messages concerning the contents of `lex` source code input shall be written to either the standard output or standard error.

(2) If the `-v` option is specified and the `-n` option is not specified, `lex` statistics shall also be written to either the standard output or standard error, in an implementation-defined format. These statistics may also be generated if table sizes are specified with a `%` operator in the *Definitions* section (see A.2.7), as long as the `-n` option is not specified.

A.2.6.3 Output Files

A text file containing C source code shall be written to `lex.yy.c`, or to the standard output if the `-t` option is present.

A.2.7 Extended Description

Each input file contains `lex` source code, which is a table of regular expressions with corresponding actions in the form of C program fragments.

When `lex.yy.c` is compiled and linked with the `lex` library (using the `-l l` operand with `c89`), the resulting program reads character input from the standard input and partitions it into strings that match the given expressions.

When an expression is matched, these actions shall occur:

— The input string that was matched is left in *yytext* as a null-terminated string; *yytext* is either an external character array or a pointer to a character string. As explained in A.2.7.1, the type can be explicitly selected using the `%array` or `%pointer` declarations, but the default is implementation defined.

— The external *int yyleng* is set to the length of the matching string.

— The expression's corresponding program fragment, or action, is executed.

During pattern matching, `lex` shall search the set of patterns for the single long- 1
est possible match. Among rules that match the same number of characters, the 1
rule given first shall be chosen.

The general format of `lex` source is:

Definitions
%%
Rules
%%
User Subroutines

The first %% is required to mark the beginning of the rules (regular expressions and actions); the second %% is required only if user subroutines follow.

Any line in the *Definitions* section beginning with a `<blank>` shall be assumed to be a C program fragment and shall be copied to the external definition area of the `lex.yy.c` file. Similarly, anything in the *Definitions* section included between delimiter lines containing only `%{` and `%}` shall also be copied unchanged to the external definition area of the `lex.yy.c` file.

Any such input (beginning with a `<blank>` or within `%{` and `%}` delimiter lines) appearing at the beginning of the *Rules* section before any rules are specified shall be written to `lex.yy.c` after the declarations of variables for the *yylex*() function and before the first line of code in *yylex*(). Thus, user variables local to *yylex*() can be declared here, as well as application code to execute upon entry to *yylex*().

The action taken by `lex` when encountering any input beginning with a `<blank>` or within `%{` and `%}` delimiter lines appearing in the *Rules* section but coming after one or more rules is undefined. The presence of such input may result in an erroneous definition of the *yylex*() function.

A.2.7.1 `lex` Definitions

Definitions appear before the first `%%` delimiter. Any line in this section not contained between `%{` and `%}` lines and not beginning with a `<blank>` shall be assumed to define a `lex` substitution string. The format of these lines is:

name substitute

If a *name* does not meet the requirements for identifiers in the C Standard {7}, the result is undefined. The string *substitute* shall replace the string {*name* } when it is used in a rule. The *name* string shall be recognized in this context only when the braces are provided and when it does not appear within a bracket expression or within double-quotes.

In the *Definitions* section, any line beginning with a `%` (percent-sign) character and followed by an alphanumeric word beginning with either `s` or `S` shall define a set of start conditions. Any line beginning with a `%` followed by a word beginning with either `x` or `X` shall define a set of exclusive start conditions. When the generated scanner is in a `%s` state, patterns with no state specified shall be also active; in a `%x` state, such patterns shall not be active. The rest of the line, after the first word, shall be considered to be one or more `<blank>`-separated names of start conditions. Start condition names shall be constructed in the same way as definition names. Start conditions can be used to restrict the matching of regular expressions to one or more states as described in the section A.2.7.4.

Implementations shall accept either of the following two mutually exclusive declarations in the *Definitions* section:

`%array` Declare the type of *yytext* to be a null-terminated character array.

`%pointer` Declare the type of *yytext* to be a pointer to a null-terminated character string.

The default type of *yytext* is implementation defined. If an application refers to *yytext* outside of the scanner source file (i.e., via an `extern`), the application shall include the appropriate `%array` or `%pointer` declaration in the scanner source file.

Implementations shall accept declarations in the *Definitions* section for setting certain internal table sizes. The declarations are shown in Table A-1. In the table, *n* represents a positive decimal integer, preceded by one or more `<blank>`s. The exact meaning of these table size numbers is implementation defined. The implementation shall document how these numbers affect the `lex` utility and how they are related to any output that may be generated by the implementation should space limitations be encountered during the execution of `lex`. It shall be possible to determine from this output which of the table size values needs to be modified to permit `lex` to successfully generate tables for the input language. The values in the column Minimum Value represent the lowest values conforming implementations shall provide.

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 1
yylex(). 1

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

<*state*>*r* 1
<*state1*, *state2*, ...>*r* 1
The regular expression *r* shall be matched only when the program 1

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 `$` (match-end-of-line) operator; *x* cannot include the ˆ (match-beginning-of-line) operator, nor trailing context, nor the `$` operator. That is, only one occurrence of trailing context is allowed in a `lex` regular expression, and the ˆ 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 (ˆ and `$`) do not appear in Table A-3. With `lex` regular expressions, these operators are restricted in their use: the ˆ operator can only be used at the beginning of an entire regular expression, and the `$` operator only at the end. The operators apply to the entire regular expression. Thus, for example, the pattern `(ˆabc)|(def$)` is undefined; it can instead be written as two separate rules, one with the regular expression `ˆabc` and one with `def$`, which share a common action via the special `|` action (see below). If the pattern were written `ˆabc|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

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.	1
\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.	1
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* >	1
bracket expression	[ ]	1
quoting	"..."	1
grouping	( )	1
definition	{*name* }	1
single-character RE duplication	* + ?	1
concatenation		1
interval expression	{*m*,*n*}	2
alternation	|	2

cannot be used with regular expressions containing another instance of the opera- 2
tor (see the preceding discussion of trailing context). 2

The additional regular expressions trailing-context operator / can be used as an 1
ordinary character if presented within double-quotes, "/"; preceded by a 1
backslash, \/; or within a bracket expression, [/]. The start-condition < and > 1
operators shall be special only in a start condition at the beginning of a regular 1

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* ;

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

applicable.

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

A.2.8 Exit Status

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

0 Successful completion.

>0 An error occurred.

A.2.9 Consequences of Errors

Default.

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

Examples, Usage

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

```
%{
/* need this for the call to atof() below */
#include <math.h>
/* need this for printf(), fopen(), and stdin below */
#include <stdio.h>
%}

DIGIT    [0-9]
ID       [a-z][a-z0-9]*

%%

{DIGIT}+    {
        printf("An integer: %s (%d)\n", yytext,
                atoi(yytext));
        }

{DIGIT}+"."{DIGIT}*        {
        printf("A float: %s (%g)\n", yytext,
                atof(yytext));
        }

if|then|begin|end|procedure|function        {
        printf("A keyword: %s\n", yytext);
        }

{ID}    printf("An identifier: %s\n", yytext);

"+"|"-"|"*"|"/"          printf("An operator: %s\n", yytext);

"{"[^}\n]*"}"            /* eat up one-line comments */
```

```
[ \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 histori- 2
cal `lex` implementations. An example of embedded anchoring would be for pat- 2
terns such as `(^| )foo( |$)` to match `foo` when it exists as a complete word. 2
This functionality can be obtained using existing `lex` features: 2

```
      ^foo/[ \n]     |                                            2
      " foo"/[ \n]   /* found foo as a separate word */           2
```

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.

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 1
C Standard {7} use of escape sequences. See the rationale for `ed` for a discussion 1
of bytes larger than nine bits being represented by octal values. Hexadecimal 1
values can represent larger bytes and multibyte characters directly, using as 1
many digits as required. 1

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 *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.

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 2
produced by `yacc`. The names affected shall include the func- 2
tions *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 2
`−p` option; however, the `−p` option shall not affect `yacc`-generated 2
`#define` symbols. 2