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All rights reserved as an unpublished work.

This is an unapproved and unpublished IEEE Standards Draft, subject to change.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]

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.

A.3.7.2 Conflicts

The parser produced for an input grammar may contain states in which conflicts occur. The conflicts occur because the grammar is not LALR(1). An ambiguous grammar always contains at least one LALR(1) conflict. The `yacc` utility shall resolve all conflicts, using either default rules or user-specified precedence rules.

Conflicts are either ''shift/reduce conflicts'' or ''reduce/reduce conflicts.'' A shift/reduce conflict is where, for a given state and lookahead symbol, both a shift action and a reduce action are possible. A reduce/reduce conflict is where, for a given state and lookahead symbol, reductions by two different rules are possible.

The rules below describe how to specify what actions to take when a conflict occurs. Not all shift/reduce conflicts can be successfully resolved this way because the conflict may be due to something other than ambiguity, so incautious use of these facilities can cause the language accepted by the parser to be much different than was intended. The description file shall contain sufficient information to understand the cause of the conflict. Where ambiguity is the reason either the default or explicit rules should be adequate to produce a working parser.

The declared precedences and associativities (see A.3.7.1.2) are used to resolve parsing conflicts as follows:

(1) A precedence and associativity is associated with each grammar rule; it is the precedence and associativity of the last token or literal in the body of the rule. If the `%prec` keyword is used, it overrides this default. Some grammar rules might not have both precedence and associativity.

(2) If there is a shift/reduce conflict, and both the grammar rule and the input symbol have precedence and associativity associated with them, then the conflict is resolved in favor of the action (shift or reduce) associated with the higher precedence. If the precedences are the same, then the associativity is used; left associative implies reduce, right associative implies shift, and nonassociative implies an error in the string being parsed.

(3) When there is a shift/reduce conflict that cannot be resolved by rule (2), the shift is done. Conflicts resolved this way are counted in the diagnostic output described in A.3.7.3.

(4) When there is a reduce/reduce conflict, a reduction is done by the grammar rule that occurs earlier in the input sequence. Conflicts resolved this way are counted in the diagnostic output described in A.3.7.3.

Conflicts resolved by precedence or associativity shall not be counted in the shift/reduce and reduce/reduce conflicts reported by `yacc` on either standard error or in the description file.

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

A.3.7.3 Error Handling

The token `error` shall be reserved for error handling. The name `error` can be used in grammar rules. It indicates places where the parser can recover from a syntax error. The default value of `error` shall be 256. Its value can be changed using a `%token` declaration. The lexical analyzer should not return the value of `error`.

The parser shall detect a syntax error when it is in a state where the action associated with the lookahead symbol is `error`. A semantic action can cause the parser to initiate error handling by executing the macro YYERROR. When YYERROR is executed, the semantic action shall pass control back to the parser. YYERROR cannot be used outside of semantic actions.

When the parser detects a syntax error, it normally calls `yyerror` with the character string `"syntax error"` as its argument. The call shall not be made if the parser is still recovering from a previous error when the error is detected. The parser is considered to be recovering from a previous error until the parser has shifted over at least three normal input symbols since the last error was detected or a semantic action has executed the macro `yyerrok`. The parser shall not call `yyerror` when YYERROR is executed.

The macro function YYRECOVERING () shall return 1 if a syntax error has been detected and the parser has not yet fully recovered from it. Otherwise, zero shall be returned.

When a syntax error is detected by the parser, the parser shall check if a previous syntax error has been detected. If a previous error was detected, and if no normal input symbols have been shifted since the preceding error was detected, the parser checks if the lookahead symbol is an endmarker (see A.3.7.4). If it is, the parser shall return with a nonzero value. Otherwise, the lookahead symbol shall be discarded and normal parsing shall resume.

When YYERROR is executed or when the parser detects a syntax error and no previous error has been detected, or at least one normal input symbol has been shifted since the previous error was detected, the parser shall pop back one state at a time until the parse stack is empty or the current state allows a shift over `error`. If the parser empties the parse stack, it shall return with a nonzero value. Otherwise, it shall shift over `error` and then resume normal parsing. If the parser reads a lookahead symbol before the error was detected, that symbol shall still be the lookahead symbol when parsing is resumed.

The macro `yyerrok` in a semantic action shall cause the parser to act as if it has fully recovered from any previous errors. The macro `yyclearin` shall cause the parser to discard the current lookahead token. If the current lookahead token has not yet been read, `yyclearin` shall have no effect.

The macro YYACCEPT shall cause the parser to return with the value zero. The macro YYABORT shall cause the parser to return with a nonzero value.

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

A.3.7.4 Interface to the Lexical Analyzer

The *yylex* () function is an integer-valued function that returns a *token number* representing the kind of token read. If there is a value associated with the token returned by *yylex* () (see the discussion of *tag* above), it shall be assigned to the external variable *yylval* .

If the parser and *yylex* () do not agree on these token numbers, reliable communication between them cannot occur. For (one character) literals, the token is simply the numeric value of the character in the current character set. The numbers for other tokens can either be chosen by `yacc`, or chosen by the user. In either case, the `#define` construct of C is used to allow *yylex* () to return these numbers symbolically. The `#define` statements are put into the code file, and the header file if that file is requested. The set of characters permitted by `yacc` in an identifier is larger than that permitted by C. Token names found to contain such characters shall not be included in the `#define` declarations.

If the token numbers are chosen by `yacc`, the tokens other than literals shall be assigned numbers greater than 256, although no order is implied. A token can be explicitly assigned a number by following its first appearance in the declarations section with a number. Names and literals not defined this way retain their default definition. All assigned token numbers shall be unique and distinct from the token numbers used for literals. If duplicate token numbers cause conflicts in parser generation, `yacc` shall report an error; otherwise, it is unspecified whether the token assignment is accepted or an error is reported.

The end of the input is marked by a special token called the *endmarker* , which has a token number that is zero or negative. (These values are invalid for any other token.) All lexical analyzers shall return zero or negative as a token number upon reaching the end of their input. If the tokens up to, but excluding, the endmarker form a structure that matches the start symbol, the parser shall accept the input. If the endmarker is seen in any other context, it shall be considered an error.

A.3.7.5 Completing the Program

In addition to *yyparse* () and *yylex* (), the functions *yyerror* () and *main* () are required to make a complete program. The application can supply *main* () and *yyerror* (), or those routines can be obtained from the `yacc` library.

A.3.7.6 `yacc` Library

The following functions appear only in the `yacc` library accessible through the `-l y` operand to `c89`; they can therefore be redefined by a portable application:

`int main(void)`

This function shall call *yyparse* () and exit with an unspecified value. Other actions within this function are unspecified.

`int yyerror(const char *s)`

This function shall write the NUL-terminated argument to

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

standard error, followed by a <newline>.

The order of the `-l y` and `-l l` operands given to `c89` is significant; the application shall either provide its own *main* () function or ensure that `-l y` precedes `-l l`.

A.3.7.7 Debugging the Parser

The parser generated by `yacc` shall have diagnostic facilities in it that can be optionally enabled at either compile time or at run time (if enabled at compile time). The compilation of the runtime debugging code is under the control of YYDEBUG , a preprocessor symbol. If YYDEBUG has a nonzero value, the debugging code shall be included. If its value is zero, the code shall not be included.

In parsers where the debugging code has been included, the external `int yydebug` can be used to turn debugging on (with a nonzero value) and off (zero value) at run time. The initial value of *yydebug* shall be zero.

When `-t` is specified, the code file shall be built such that, if YYDEBUG is not already defined at compilation time (using the `c89 -D YYDEBUG` option, for example), YYDEBUG shall be set explicitly to 1. When `-t` is not specified, the code file shall be built such that, if YYDEBUG is not already defined, it shall be set explicitly to zero.

The format of the debugging output is unspecified but includes at least enough information to determine the shift and reduce actions, and the input symbols. It also provides information about error recovery.

A.3.7.8 Algorithms

The parser constructed by `yacc` implements an LALR (1) parsing algorithm as documented in the literature. It is unspecified whether the parser is table-driven or direct-coded.

A parser generated by `yacc` shall never request an input symbol from *yylex* () while in a state where the only actions other than the error action are reductions by a single rule.

The literature of parsing theory defines these concepts.

A.3.7.9 Limits

The `yacc` utility may have several internal tables. The minimum maximums for these tables are shown in Table A-4. The exact meaning of these values is implementation defined. The implementation shall define the relationship between these values and between them and any error messages that the implementation may generate should it run out of space for any internal structure. An implementation may combine groups of these resources into a single pool as long as the total available to the user does not fall below the sum of the sizes specified by this subclause.

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

Table A-4 – yacc Internal Limits

Limit	Minimum Maximum	Description
{NTERMS }	126	Number of tokens.
{NNONTERM }	200	Number of nonterminals.
{NPROD }	300	Number of rules.
{NSTATES }	600	Number of states.
{MEMSIZE }	5200	Length of rules. The total length, in names (tokens and nonterminals), of all the rules of the grammar. The left-hand side is counted for each rule, even if it is not explicitly repeated, as specified in A.3.7.1.3.
{ACTSIZE }	4000	Number of actions. ''Actions'' here (and in the description file) refer to parser actions (shift, reduce, etc.) not to semantic actions defined in A.3.7.1.3.

A.3.8 Exit Status

The yacc utility shall exit with one of the following values:

0 Successful completion.

>0 An error occurred.

A.3.9 Consequences of Errors

If any errors are encountered, the run is aborted and yacc exits with a nonzero status. Partial code files and header files files may be produced. The summary information in the description file shall always be produced if the –v flag is present.

A.3.10 Rationale. *(This subclause is not a part of P1003.2)*

The references in the Bibliography may be helpful in constructing the parser generator. The Pennello-DeRemer {B26} paper (along with the works it references) describe a technique to generate parsers that conform to this standard. Work in this area continues to be done, so implementors should consult current literature before doing any new implementations. The original paper by Knuth {B27} is the theoretical basis for this kind of parser, but the tables it generates are impractically large for reasonable grammars, and should not be used. The ''equivalent to'' wording is intentional to assure that the best tables that are LALR(1) can be generated.

There has been confusion between the class of grammars, the algorithms needed to generate parsers, and the algorithms needed to parse the languages. They are all reasonably orthogonal. In particular, a parser generator that accepts the full range of LR(1) grammars need not generate a table any more complex than one that accepts SLR(1) (a relatively weak class of LR grammars) for a grammar that

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

happens to be SLR(1). Such an implementation need not recognize the case, either; table compression can yield the SLR(1) table (or one even smaller than that) without recognizing that the grammar is SLR(1). The speed of a LR(1) parser for any class is dependent more upon the table representation and compression (or the code generation if a direct parser is generated) than upon the class of grammar that the table generator handles.

The speed of the parser generator is somewhat dependent upon the class of grammar it handles. However, the original Knuth {B27} algorithms for constructing LR parsers was judged by its author to be impractically slow at that time. Although full LR is more complex than LALR(1), as computer speeds and algorithms improve, the difference (in terms of acceptable wall-clock execution time) is becoming less significant.

Potential authors are cautioned that the Penello-DeRemer paper previously cited identifies a bug (an oversimplification of the computation of LALR(1) lookahead sets) in some of the LALR(1) algorithm statements that preceded it to publication. They should take the time to seek out that paper, as well as current relevant work, particularly Aho's {B22}.

Examples, Usage

Access to the `yacc` library is obtained with library search operands to `c89`. To use the `yacc` library *main* (),

```
c89 y.tab.c -l y
```

Both the `lex` library and the `yacc` library contain *main* (). To access the `yacc` *main* (),

```
c89 y.tab.c lex.yy.c -l y -l l
```

This ensures that the `yacc` library is searched first, so that its *main* () is used.

The historical `yacc` libraries have contained two simple functions that are normally coded by the application programmer. These library functions are similar to the following code:

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

```
#include <locale.h>
int main(void)
{
        extern int yyparse();

        setlocale(LC_ALL, "");

        /* If the following parser is one created by lex, the
           application must be careful to ensure that LC_CTYPE
           and LC_COLLATE are set to the POSIX Locale.  */
        (void) yyparse();
        return (0);
}

#include <stdio.h>

int yyerror(const char *msg)
{
        (void) fprintf(stderr, "%s\n", msg);
        return (0);
}
```

Historical implementations experience name conflicts on the names `yacc.tmp`, `yacc.acts`, `yacc.debug`, `y.tab.c`, `y.tab.h`, and `y.output` if more than one copy of `yacc` is running in a single directory at one time. The `-b` option was added to overcome this problem. The related problem of allowing multiple `yacc` parsers to be placed in the same file was addressed by adding a `-p` option to override the previously hardcoded `yy` variable prefix. (The `-p` option name was selected from a historical implementation.) Implementations will also have to be cognizant of 2.11.6.3, which requires that any temporary files used by `yacc` also be named to avoid collisions.

The description of the `-p` option specifies the minimal set of function and variable names that cause conflict when multiple parsers are linked together. YYSTYPE does not need to be changed. Instead, the programmer can use `-b` to give the header files for different parsers different names, and then the file with the *yylex* () for a given parser can include the header for that parser. Names such as *yyclearerr* don't need to be changed because they are used only in the actions; they do not have linkage. It is possible that an implementation will have other names, either internal ones for implementing things such as *yyclearerr* , or providing non-standard features, that it wants to change with `-p`.

The `-b` option was added to provide a portable method for permitting `yacc` to work on multiple separate parsers in the same directory. If a directory contains more than one `yacc` grammar, and both grammars are constructed at the same time (by, say, a parallel `make` program), conflict results. While the solution is not historical practice, it corrects a known deficiency in historical implementations. Corresponding changes were made to all sections that referenced the filenames `y.tab.c` (now ''the code file''), `y.tab.h` (now ''the header file''), and `y.output` (now ''the description file'').

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

The grammar for `yacc` input is based on System V documentation. The textual description shows there that the `;` is required at the end of the rule. The grammar and the implementation do not require this. (The use of `C_IDENTIFIER` causes a reduce to occur in the right place.) 1

Also, in that implementation, the constructs such as `%token` can be terminated by a semicolon, but this is not permitted by the grammar. The keywords such as `%token` can also appear in uppercase, which is again not discussed. In most places where `%` is used, `\` can be substituted, and there are alternate spellings for some of the symbols (e.g. `%LEFT` can be `%<` or even `\<`). 1

Multibyte characters should be recognized by the lexical analyzer and returned as tokens. They should not be returned as multibyte character literals. The token `error` that is used for error recovery is normally assigned the value 256 in the historical implementation. Thus, the token value 256, which used in many multibyte character sets, is not available for use as the value of a user-defined token. 2

Historically, <*tag*> can contain any characters except `>`, including white space, in the implementation. However, since the *tag* must reference a Standard C union member, in practice conforming implementations need only support the set of characters for Standard C identifiers in this context.

Some historical implementations are known to accept actions that are terminated by a period. Historical implementations often allow `$` in names. A conforming implementation need support neither of these behaviors.

Unary operators that are the same token as a binary operator in general need their precedence adjusted. This is handled by the `%prec` advisory symbol associated with the particular grammar rule defining that unary operator. See A. Applications are not required to use this operator for unary operators, but the grammars that do not require it are rare.

Deciding when to use `%prec` illustrates the difficulty in specifying the behavior of `yacc`. There may be situations in which the *grammar* is not strictly speaking in error, and yet `yacc` cannot interpret it unambiguously. The resolution of ambiguities in the grammar can in many instances be resolved by providing additional information, such as using `%type` or `%union` declarations. It is often easier and it usually yields a smaller parser to take this alternative when it is appropriate.

The size and execution time of a program produced without the runtime debugging code is usually smaller and slightly faster in historical implementations.

There is a fair amount of material in this that appears tutorial in nature; some of it has been moved to the Rationale in Draft 9 to simplify the specification. It is hard to avoid because of the need to define terms at least informally. The alternative is to bring in one of the parser generator texts and use its terminology directly, but since there is some variation in that terminology, it was felt that informal definitions of the terms so that someone who understood the concepts would be sure to understand the terms would make the standard stand alone from any specific text.

Statistics messages from several historical implementations include the following types of information:

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

1680 *n* /512 terminals, *n* /300 nonterminals
1681 *n* /600 grammar rules, *n* /1500 states
1682 *n* shift/reduce, *n* reduce/reduce conflicts reported
1683 *n* /350 working sets used
1684 memory: states,etc. *n* /15000, parser *n* /15000
1685 *n* /600 distinct lookahead sets
1686 *n* extra closures
1687 *n* shift entries, *n* exceptions
1688 *n* goto entries
1689 *n* entries saved by goto default
1690 Optimizer space used: input *n* /15000, output *n* /15000
1691 *n* table entries, *n* zero
1692 maximum spread: *n*, maximum offset: *n*

1693 The report of internal tables in the description file is left implementation defined 2
1694 because all aspects of these limits are also implementation defined. Some imple- 2
1695 mentations may use dynamic allocation techniques and have no specific limit 2
1696 values to report. 2

1697 **History of Decisions Made**

1698 The format of the `y.output` file is not given because specification of the format
1699 was not seen to enhance application portability. The listing is primarily intended
1700 to help human users understand and debug the parser; use of `y.output` by a
1701 portable application script is far-fetched. Furthermore, implementations have not
1702 produced consistent output and no clear winner was apparent. The format
1703 selected by the implementation should be human-readable, in addition to the
1704 requirement that it be a text file.

1705 Standard error reports are not specifically described because they are seldom of
1706 use to portable applications and there was no reason to restrict implementations.

1707 Some implementations recognize `={` as equivalent to `{`, because it appears in his-
1708 torical documentation. This construction was recognized and documented as
1709 obsolete as long ago as 1978, in the original paper *Yacc: Yet Another Compiler-*
1710 *Compiler* by Stephen C. Johnson. POSIX. 2 chose to leave it as obsolete and omit
1711 it.

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

Annex B
(norm ative)

C Language Bindings Option

This annex describes the C language bindings to the language-independent services described in Section 7.

The interfaces described in this annex may be provided by the conforming system; however, any system claiming conformance to the Language-Independent System Services C Language Bindings Option shall provide all of the interfaces described here.

B.0.1 C Language Bindings Option Rationale. *(This subclause is not a part of P1003.2)*

In this version of POSIX. 2, the language-independent descriptions in Section 7 have not been developed. The language-independent syntax is being created in parallel by the POSIX. 1 working group. Therefore, the C language bindings described in this annex are actually the full functional specifications. It is the intention of the POSIX. 2 working group to rectify this situation in a revision to this standard, by moving the majority of the functional specifications back into Section 7, leaving Annex B with only brief descriptions of the C bindings to those services.

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