PROLOG

This manual page is part of the POSIX Programmer's Manual. The Linux implementation of this interface may differ (consult the corresponding Linux manual page for details of Linux behavior), or the interface may not be implemented on Linux.

NAME

yacc — yet another compiler compiler (**DEVELOPMENT**)

SYNOPSIS

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

DESCRIPTION

The *yacc* utility shall read a description of a context-free grammar in *grammar* and write C source code, conforming to the ISO C standard, to a code file, and optionally header information into a header file, in the current directory. The generated source code shall not depend on any undefined, unspecified, or implementation-defined behavior, except in cases where it is copied directly from the supplied grammar, or in cases that are documented by the implementation. The C code shall define a function and related routines and macros for an automaton that executes a parsing algorithm meeting the requirements in *Algorithms*.

The form and meaning of the grammar are described in the EXTENDED DESCRIPTION section.

The C source code and header file shall be produced in a form suitable as input for the C compiler (see *c99*).

OPTIONS

The *yacc* utility shall conform to the Base Definitions volume of POSIX.1-2017, *Section 12.2*, *Utility Syntax Guidelines*, except for Guideline 9.

The following options shall be supported:

−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. See the OUTPUT FILES section.

−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. This should only be used after the grammar and the associated actions are fully debugged.

−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 section, 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 **#define** symbols generated by *yacc*.

−t Modify conditional compilation directives to permit compilation of debugging code in the code file. Runtime debugging statements shall always be contained in the code file, but by default conditional compilation directives prevent their compilation.

−v Write a file containing a description of the parser and a report of conflicts generated by ambiguities in the grammar.

OPERANDS

The following operand is required:

grammar A pathname of a file containing instructions, hereafter called *grammar*, for which a parser is to be created. The format for the grammar is described in the EXTENDED DESCRIPTION section.

STDIN

Not used.

INPUT FILES

The file *grammar* shall be a text file formatted as specified in the EXTENDED DESCRIPTION section.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *yacc*:

LANG Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of POSIX.1-2017, *Section 8.2*, *Internationalization Variables* for the precedence of internationalization variables used to determine the values of locale categories.)

LC_ALL If set to a non-empty string value, override the values of all the other internationalization variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

NLSPATH Determine the location of message catalogs for the processing of *LC_MESSAGES*.

The *LANG* and *LC_** variables affect the execution of the *yacc* utility as stated. The *main*() function defined in *Yacc Library* shall call:

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

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

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

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

Standard error shall also be used for diagnostic messages.

OUTPUT FILES

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

Code File

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

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 also be included in this file.

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 *Limits*) shall also be reported, in an implementation-defined manner. (Some implementations may use dynamic allocation techniques and have no specific limit values to report.)

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 tar get 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. These bodies of text shall not contain C-language trigraphs. 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 the OUTPUT FILES section. The code file can be compiled and linked using *c99*. 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 versions 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 *c99*. The*yacc* library interf aces need not support interfaces with other than the default **yy** symbol prefix. The application provides the lexical analyzer function, *yylex*(); the *lex* utility is specifically designed to generate such a routine.

Input Language

The application shall ensure that every specification file consists of three sections in order: *declarations*, *grammar rules*, and *pr ograms*, separated by double <percent-sign> characters (**"%%"**). 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.

Lexical Structure of the Grammar

The <blank>, <newline>, and <form-feed> character shall be ignored, except that the application shall ensure that they do not appear in names or multi-character reserved symbols. Comments shall be enclosed in **"/* ... */"**, and can appear wherever a name is valid.

Names are of arbitrary length, made up of letters, periods (**'.'**), underscores (**'_'**), and non-initial digits. Uppercase and lowercase letters are distinct. Conforming 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 appear in **#define** statements.

A literal shall consist of a single character enclosed in single-quote characters. All of the escape sequences supported for character constants by the ISO C standard shall be supported by *yacc*.

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

The application shall ensure that the NUL character is not used in grammar rules or literals.

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 volume of POSIX.1-2017, 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 shall 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.

The following declares *name* to be a token:

```
%token [<tag>] name [number] [name [number]]...
```

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.

The following declares *name* to be a token, and assigns precedence to it:

```
%left [<tag>] name [number] [name [number]]...
%right [<tag>] name [number] [name [number]]...
```

One or 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 associati ve, and **%right** similarly denotes right associative operators. If *tag* is present, it shall declare a C type for *name*s as described for **%token**.

The following declares *name* to be a token, and indicates that this cannot be used associatively:

```
%nonassoc [<tag>] name [number] [name [number]]...
```

If the parser encounters associative use of this token it reports an error. If *tag* is present, it shall declare a C type for *name*s as described for **%token**.

The following declares that union member *name*s are non-terminals, and thus it is required to have a *tag* field at its beginning:

```
%type <tag> name...
```

Because it deals with non-terminals 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* not defined by a **%token**, **%left**, **%right**, or **%nonassoc** declaration is assumed to represent a non-terminal symbol. The *yacc* utility shall report an error for any non-terminal 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.

The following declares the non-terminal *name* to be the *start symbol*, which represents the largest, most general structure described by the grammar rules:

```
%start name
```

By default, it is the left-hand side of the first grammar rule; this default can be overridden with this declaration.

The following declares the *yacc* value stack to be a union of the various types of values desired.

```
%union { body of union (in C) }
```

The body of the union shall not contain unbalanced curly brace preprocessing tokens.

By default, the values returned by actions (see below) and the lexical analyzer shall be of type **int**. The *yacc* utility keeps track of types, and it 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 a **#include** construct within **%{** 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 *yacc* input.

C-language declarations and definitions can appear in the declarations section, enclosed by the following 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 statements shall not contain **"%}"** outside a comment, string literal, or multi-character constant.

The application shall ensure that the declarations section is terminated by the token **%%**.

Grammar Rules in yacc

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 non-terminal 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-line> (**'|'**) 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 non-terminal 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 *non-terminal*; *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, and so on.)

These actions can return values and can obtain the values returned by previous actions. These values are 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 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 **'}'**. The statements shall not contain unbalanced curly brace preprocessing tokens.

Certain pseudo-variables can be used in the action. These are macros for access to data structures known internally 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 *number* is zero or negative, 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.

Actions can occur anywhere in a rule (not just 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 non-terminal symbol and adding an empty rule with that non-terminal 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 application shall ensure that the grammar rules are terminated by **%%**.

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. 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.

Input Grammar

The following input to *yacc* yields a parser for the input to *yacc*. This formal syntax takes precedence over the preceding text syntax description.

The lexical structure is defined less precisely; *Lexical Structure of the Grammar* 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.

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 non-negative decimal integer).

TYPE, **LEFT**, **MARK**, **LCURL**, **RCURL**
These correspond directly to **%type**, **%left**, **%%**, **%{**, and **%}**.

{ . . . } 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, and so on */

%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 multi-byte 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. */
      }
      |    LCURL
      {
        /* Copy C code to output file. */
      }
        RCURL
      |    rword tag nlist
      ;
rword : TOKEN
      | LEFT
      | RIGHT
      | NONASSOC
      | TYPE
      ;
tag   : /* Empty: union tag ID optional. */
      | '<' IDENTIFIER '>'
      ;
nlist : nmno
      | nlist nmno
      ;
nmno  : IDENTIFIER         /* Note: literal invalid with % type. */
      | IDENTIFIER NUMBER  /* Note: invalid with % type. */
      ;

/* Rule section */

rules : C_IDENTIFIER rbody prec
      | rules  rule
      ;
rule  : C_IDENTIFIER rbody prec
      | '|' rbody prec
      ;
rbody : /* empty */
      | rbody IDENTIFIER
      | rbody act
      ;
act   : '{'
        {
          /* Copy action, translate $$, and so on. */
        }
        '}'
      ;
prec  : /* Empty */
      | PREC IDENTIFIER
      | PREC IDENTIFIER act
      | prec ';'
      ;
```

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 from that which 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 *Declarations Section*) 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 non-associative 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 *Error Handling*.
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 *Error Handling*.

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.

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 passes 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*yyerr or*() 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 *Interface to the*

Lexical Analyzer). If it is, the parser shall return with a non-zero 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 non-zero 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 non-zero value.

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 (single-byte 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 token numbers assigned by *yacc* shall be unique and distinct from the token numbers used for literals and user-assigned tokens. 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.

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.

Yacc Library

The following functions shall appear only in the *yacc* library accessible through the **−l y** operand to *c99*; they can therefore be redefined by a conforming 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 standard error, followed by a <newline>.

The order of the **−l y** and **−l l** operands given to *c99* is significant; the application shall either provide its own *main*() function or ensure that **−l y** precedes **−l l**.

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 runtime (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 non-zero 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 non-zero value) and off (zero value) at runtime. 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 *c99* **−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.

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.

Limits

The *yacc* utility may have several internal tables. The minimum maximums for these tables are shown in the following table. 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 section.

Table: Internal Limits in *yacc*

Limit	Minimum Maximum	Description
{NTERMS}	126	Number of tokens.
{NNONTERM}	200	Number of non-terminals.
{NPROD}	300	Number of rules.
{NSTATES}	600	Number of states.
{MEMSIZE}	5 200	Length of rules. The total length, in names (tokens and non-terminals), 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*Gr ammar Rules in yacc*.
{ACTSIZE}	4 000	Number of actions. ‘‘Actions’’ here (and in the description file) refer to parser actions (shift, reduce, and so on) not to semantic actions defined in*Gr ammar Rules in yacc*.

EXIT STATUS

The following exit values shall be returned:

0 Successful completion.

>0 An error occurred.

CONSEQUENCES OF ERRORS

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

The following sections are informative.

APPLICATION USAGE

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 hard-coded **yy** variable prefix.

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* do not need to be changed because they are used only in the actions; they do not have linkage. It is possible that an implementation has other names, either internal ones for implementing things such as *yyclearerr*, or providing non-standard features that it wants to change with **−p**.

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*Gr ammar Rules in yacc*.) Applications are not required to use this operator for unary operators, but the grammars that do not require it are rare.

EXAMPLES

Access to the *yacc* library is obtained with library search operands to *c99*. To use the *yacc* library *main*():

```
c99 y.tab.c -l y
```

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

```
c99 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 functions are similar to the following code:

```
#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);
}
```

RATIONALE

The references in **Referenced Documents** may be helpful in constructing the parser generator. The referenced DeRemer and Pennello article (along with the works it references) describes a technique to generate parsers that conform to this volume of POSIX.1-2017. Work in this area continues to be done, so implementors should consult current literature before doing any new implementations. The original Knuth article 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 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 an 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 article algorithms for constructing LR parsers were 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 referenced DeRemer and Pennello article previously cited identifies a bug (an over-simplification 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.

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, for example, 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’’).

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.)

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, <backslash> can be substituted, and there are alternate spellings for some of the symbols (for example, **%LEFT** can be **"%<"** or even "\<").

Historically, <*tag*> can contain any characters except **’>’**, including white space, in the implementation. However, since the *tag* must reference an ISO C standard union member, in practice conforming implementations need to support only the set of characters for ISO C standard 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 does not need to support either of these behaviors.

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.

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

n/512 terminals, *n*/300 non-terminals
n/600 grammar rules, *n*/1 500 states
n shift/reduce, *n* reduce/reduce conflicts reported
n/350 working sets used
Memory: states, etc. *n*/15 000, parser *n*/15 000
n/600 distinct lookahead sets
n extra closures
n shift entries, *n* exceptions
n goto entries
n entries saved by goto default
Optimizer space used: input *n*/15 000, output *n*/15 000
n table entries, *n* zero
Maximum spread: *n*, Maximum offset: *n*

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

The format of the **y.output** file is not given because specification of the format was not seen to enhance applications portability. The listing is primarily intended to help human users understand and debug the parser; use of **y.output** by a conforming application script would be unusual. Furthermore, implementations have not produced consistent output and no popular format was apparent. The format selected by the implementation should be human-readable, in addition to the requirement that it be a text file.

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

Some implementations recognize **"={"** as equivalent to **'{'** because it appears in historical documentation. This construction was recognized and documented as obsolete as long ago as 1978, in the referenced *Yacc: Yet Another Compiler-Compiler*. This volume of POSIX.1-2017 chose to leave it as obsolete and omit it.

Multi-byte characters should be recognized by the lexical analyzer and returned as tokens. They should not be returned as multi-byte 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 is used in many multi-byte character sets, is not available for use as the value of a user-defined token.

FUTURE DIRECTIONS

None.

SEE ALSO

c99, *lex*

The Base Definitions volume of POSIX.1-2017,*Chapter 8*, *En vironment Variables*, *Section 12.2*, *Utility Syntax Guidelines*

COPYRIGHT

Any typographical or formatting errors that appear in this page are most likely to have been introduced during the conversion of the source files to man page format. To report such errors, see https://www.kernel.org/doc/man-pages/reporting_bugs.html .