Chapter 6. A Reference for Bison Specifications

In this chapter we describe the syntax of bison programs, along with the various options and support functions available. POSIX yacc is almost an exact subset of bison, so we note which parts of bison are extensions beyond what POSIX requires.

After the section on the structure of a bison grammar, the sections in this chapter are in alphabetical order by feature.

Structure of a Bison Grammar

A bison grammar consists of three sections: the definition section, the rules section, and the user subroutines section.

```
... definition section ...
%%
... rules section ...
%%
... user subroutines section ...
```

The sections are separated by lines consisting of two percent signs. The first two sections are required, although a section may be empty. The third section and the preceding `%%` line may be omitted.

Symbols

A bison grammar is constructed from symbols, which are the “words” of the grammar. Symbols are strings of letters, digits, periods, and underscores that do not start with a digit. The symbol `error` is reserved for error recovery; otherwise, bison attaches no fixed meaning to any symbol. (Since bison defines a C preprocessor symbol for each token, you also need to be sure token names don’t collide with C reserved words or bison’s own symbols such as `yyparse`, or strange errors will ensue.)

Symbols produced by the lexer are called *terminal symbols* or *tokens*. Those that are defined on the left-hand side of rules are called *nonterminal symbols* or *nonterminals*. Tokens may also be literal quoted characters. (See Literal Tokens.) A widely followed convention makes token

names all uppercase and nonterminals lowercase. We follow that convention throughout the book.

Definition Section

The definition section can include a literal block, which is C code copied verbatim to the beginning of the generated C file, usually containing declaration and `#include` lines in `%{ %}` or `%code` blocks. There may be `%union`, `%start`, `%token`, `%type`, `%left`, `%right`, and `%nonassoc` declarations. (See %union Declaration, %start Declaration, Tokens, %type Declaration, and Precedence and Associativity Declarations.) The definition section can also contain comments in the usual C format, surrounded by `/*` and `*/`. All of these are optional, so in a very simple parser the definition section may be completely empty.

Rules Section

The rules section contains grammar rules and actions containing C code. See below and Rules for details.

User Subroutines Section

Bison copies the contents of the user subroutines section verbatim to the C file. This section typically includes routines called from the actions.

In a large program, it is usually more convenient to put the supporting code in a separate source file to minimize the amount of material recompiled when you change the bison file.

Actions

An *action* is C code executed when bison matches a rule in the grammar. The action must be a C compound statement, for example:

```
date: month '/' day '/' year { printf("date found"); } ;
```

The action can refer to the values associated with the symbols in the rule by using a dollar sign followed by a number, with the first symbol after the colon being 1, for example:

```
date: month '/' day '/' year { printf("date %d-%d-%d found", $1, $3, $5
      ;
```

The name `$$` refers to the value for the left-hand side (LHS) symbol, the one to the left of the colon. Symbol values can have different C types. See Tokens, %type Declaration, and %union Declaration for details.

For rules with no action, bison uses a default of the following:

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

If you have a rule with no RHS symbols and the LHS symbol has a declared type, you *must* write an action to set its value.

Embedded Actions

Even though bison's parsing technique allows actions only at the end of a rule, bison can simulate actions embedded within a rule. If you write an action within a rule, bison invents a rule with an empty right-hand side and a made-up name on the left, makes the embedded action into the action for that rule, and replaces the action in the original rule with the made-up name. For example, these are equivalent:

```
thing:          A { printf("seen an A"); } B ;

thing:          A fakename B ;
fakename:       /* empty */ { printf("seen an A"); } ;
```

Although this feature is quite useful, it can have some surprising consequences. The embedded action turns into a symbol in the rule, so its value (whatever it assigns to `$$`) is available to an end-of-rule action like any other symbol:

```
thing:          A { $$ = 17; } B C
                { printf("%d", $2); }
          ;
```

This example prints "17". Either action can refer to the value of `A` as `$1`, and the end-of-rule action can refer to the value of the embedded action as `$2` and can refer to the values of `B` and `C` as `$3` and `$4`.

Embedded actions can cause shift/reduce or reduce/reduce conflicts in otherwise acceptable grammars. For example, this grammar causes no problem:

```
%%
thing:      abcd | abcz ;
```

```
abcd: 'A' 'B' 'C' 'D' ;
abcz: 'A' 'B' 'C' 'Z' ;
```

But if you add an embedded action, it has a shift/reduce conflict:

```
%%
thing:      abcd | abcz ;

abcd: 'A' 'B' { somefunc(); } 'C' 'D' ;
abcz: 'A' 'B' 'C' 'Z' ;
```

In the first case, the parser doesn't need to decide whether it's parsing an `abcd` or an `abcz` until it's seen all four tokens, when it can tell which it's found. In the second case, it needs to decide after it parses the `B`, but at that point it hasn't seen enough of the input to decide which rule it is parsing. If the embedded action came after the `C`, there would be no problem, since bison could use its one-token lookahead to see whether a `D` or a `Z` is next.

Symbol Types for Embedded Actions

Since embedded actions aren't associated with grammar symbols, there is no way to declare the type of the value returned by an embedded action. If you are using `%union` and typed symbol values, you have to put the value in angle brackets when referring to the action's value, for example, `$<` *type* `>$` when you set it in the embedded action and `$<` *type* `>3` (using the appropriate number) when you refer to it in the action at the end of the rule. See Symbol Values. If you have a simple parser that uses all `int` values, as in the previous example, you don't need to give a type.

Ambiguity and Conflicts

Bison may report conflicts when it translates a grammar into a parser. In some cases, the grammar is truly ambiguous; that is, there are two possible parses for a single input string, and bison cannot handle that. In others, the grammar is unambiguous, but the standard parsing technique that bison uses is not powerful enough to parse the grammar. The problem in an unambiguous grammar with conflicts is that the parser would need to look more than one token ahead to decide which of two possible parses to use. Usually you can rewrite the grammar so that one token of lookahead is enough, but bison also offers a more powerful technique, GLR, that provides lookahead of unlimited length.

See Precedence and Associativity Declarations and Chapter 7 for more details and suggestions on how to fix these problems, and see GLR Parsing for details on GLR parsing.

Types of Conflicts

There are two kinds of conflicts that can occur when bison tries to create a parser: shift/reduce and reduce/reduce.

Shift/Reduce Conflicts

A *shift/reduce* conflict occurs when there are two possible parses for an input string and one of the parses completes a rule (the reduce option) and one doesn't (the shift option). For example, this grammar has one shift/reduce conflict:

```
%%
e:      'X'
    |   e '+' e
    ;
```

For the input string `X+X+X` there are two possible parses, `(X+X)+X` or `X+(X+X)`. Taking the reduce option makes the parser use the first parse, and taking the shift option makes the parser take the second. Bison chooses the shift unless the user puts in operator precedence declarations. See Precedence and Associativity Declarations for more information.

Reduce/Reduce Conflicts

A *reduce/reduce* conflict occurs when the same token could complete two different rules. For example:

```
%%
prog: proga | progb ;

proga:      'X' ;
progb:      'X' ;
```

An `X` could either be a `proga` or be a `progb`. Most reduce/reduce conflicts are less obvious than this, but they usually indicate mistakes in the grammar. See Chapter 8 for details on handling conflicts.

%expect

Occasionally you may have a grammar that has a few conflicts, you are confident that bison will resolve them the way you want, and it's too much hassle to rewrite the grammar to get rid of them. (If/then/else-style constructs are the most common place this happens.) The declaration `%expect N` tells bison that your parser should have N shift/reduce conflicts, so bison reports a compile-time error if the number is different.

You can also use `%expect-rr N` to tell it how many reduce/reduce conflicts to expect; unless you are using a GLR parser, do not use this feature since reduce/reduce conflicts almost always indicate an error in the grammar.

GLR Parsers

Sometimes a grammar is truly too hard for bison's normal LALR parsing algorithm to handle. In that case, you can tell bison to create a Generalized LR (GLR) parser instead, by including a `%glr-parser` declaration. When a GLR parser encounters a conflict, it conceptually splits and continues both possible parses in parallel, with each one consuming each token. If there are several conflicts, it can create a tree of partial parses, splitting each time there is a conflict. At the end of the parse, either there will be only one surviving parse, the rest having been abandoned because they didn't match the rest of the input, or, if the grammar is truly ambiguous, there may be several parses, and it is up to you to decide what to do with them.

See GLR Parsing for examples and more details.

Bugs in Bison Programs

Bison itself is pretty robust, but there are a few common programming mistakes that can cause serious failures in your bison parser.

Infinite Recursion

A common error in bison grammars is to create a recursive rule with no way to terminate the recursion. Bison will diagnose this grammar with the somewhat mysterious "start symbol xlist does not derive any sentence."

```
%%
xlist:    xlist 'X' ;
```

Interchanging Precedence

People occasionally try to use `%prec` to swap the precedence of two tokens:

```
%token  NUMBER
%left PLUS
%left MUL
%%
expr  :    expr PLUS expr %prec MUL
      |    expr MUL expr %prec PLUS
      |    NUMBER
      ;
```

This example seems to give `PLUS` higher precedence than `MUL`, but in fact it makes them the same. The precedence mechanism resolves shift/reduce conflicts by comparing the precedence of the token to be shifted to the precedence of the rule. In this case, there are several conflicts. A typical conflict arises when the parser has seen "expr PLUS expr" and the next token is a `MUL`. In the absence of a `%prec`, the rule would have the precedence of `PLUS`, which is lower than that of `MUL`, and bison takes the shift. But with `%prec`, both the rule and the token have the precedence of `MUL`, so it reduces because `MUL` is left associative.

One possibility would be to introduce pseudotokens, for example, `XPLUS` and `XMUL`, with their own precedence levels to use with `%prec`. A far better solution is to rewrite the grammar to say what it means, in this case exchanging the `%left` lines (see Precedence and Associativity Declarations).

Embedded Actions

When you write an action in the middle of a rule rather than at the end, bison has to invent an anonymous rule to trigger the embedded action. Occasionally the anonymous rule causes unexpected shift/reduce conflicts. See Actions for more details.

C++ Parsers

Bison can produce a C++ parsers. If your bison file is called `comp.yxx`, it will create the C++ source `comp.tab.cxx` and header `comp.tab.hxx` files. (You can change the output filenames with the `-o` flag, perhaps as `-o comp.c++`, which will also rename the header to `comp.h++`.) It also creates the header files `stack.hh`, `location.hh`, and `position.hh`, which define three classes used in the parser. The contents of these three files are always the same unless you use `-p` or `%name-prefix` to change the namespace of the parser from "yy" to something else.

Bison defines a class called `yy::parser` (unless you change its name) with a main `parse` routine and some minor routines for error reporting and debugging. See C++ Parsers for more details.

%code Blocks

Bison has always accepted C code in the declaration section surrounded by `%{ ... %}` brackets. Sometimes the code has to go in a particular place in the generated program, before or after particular parts of the standard parser skeleton code. The `%code` directive allows more specific placement of the code.

```
%code [place] {
  ... code here ...
}
```

The optional *`place`*, which the bison manual calls the *qualifier*, says where in the generated program to put the code. Current places in C language programs are `top`, `provides`, and `requires`. They put the code at the top of the file, before the definitions of `YYSTYPE` and `YYLTYPE`, and after those definitions. This feature is considered experimental, so the places are likely to change; therefore, check the current bison manual to see what the current options are. The use of `%code` without a place will definitely remain and is intended to replace `%{ %}`.

End Marker

Each bison grammar includes a pseudotoken called the *end marker*, which marks the end of input. In bison listings, the end marker is usually indicated as `$end`.

The lexer must return a zero token to indicate the end of input.

Error Token and Error Recovery

Bison parsers will always detect syntax errors as early as possible, that is, as soon as they read a token for which there is no potential parse. When bison detects a syntax error, that is, when it receives an input token that it cannot parse, it attempts to recover from the error using this procedure:

1. It calls `yyerror("syntax error")`. This usually reports the error to the user.

2. It discards any partially parsed rules until it returns to a state in which it could shift the special `error` symbol.
3. It resumes parsing, starting by shifting an `error`.
4. If another error occurs before three tokens have been shifted successfully, bison does not report another error but returns to step 2.

See Chapter 8 for more details on error recovery. Also, see yyerror(), YYRECOVERING(), yyclearin, and yyerrok for details on features that help control error recovery.

%destructor

When bison is trying to recover from a parse error, it discards symbols and their values from the parse stack. If the value is a pointer to dynamically allocated memory, or otherwise needs special treatment when it's discarded, the `%destructor` directive lets you get control when particular symbols, or symbols with values of particular types, are deleted. It also will handle the value of the start symbol after a successful parse. See Chapter 8 for more details.

Inherited Attributes ($0)

Bison symbol values can act as *inherited attributes* or *synthesized attributes*. (What bison calls *values* are usually referred to in compiler literature as *attributes*.) The usual synthesized attributes start as token values, which are the leaves of the parse tree. Information conceptually moves up the parse tree each time a rule is reduced, and each action synthesizes the value of its resulting symbol (`$$`) from the values of the symbols on the right-hand side of the rule.

Sometimes you need to pass information the other way, from the root of the parse tree toward the leaves. Consider this example:

```
declaration:     class type namelist ;

class:      GLOBAL      { $$ = 1; }
      |     LOCAL       { $$ = 2; }
      ;

type:       REAL        { $$ = 1; }
      |     INTEGER     { $$ = 2; }
      ;

namelist:   NAME        { mksymbol($0, $-1, $1); }
      |     namelist NAME { mksymbol($0, $-1, $2); }
      ;
```

It would be useful to have the class and type available in the actions for `namelist`, both for error checking and for entering into the symbol table. Bison makes this possible by allowing access to symbols on its internal stack to the left of the current rule, via `$0`, `$-1`, etc. In the example, the `$0` in the call to `mksymbol()` refers to the value of the `type`, which is stacked just before the symbol(s) for the `namelist` production, and it will have the value 1 or 2 depending on whether the type was `REAL` or `INTEGER`. `$-1` refers to the class, which will have the value 1 or 2 if the class was `GLOBAL` or `LOCAL`.

Although inherited attributes can be useful, they can also be a source of hard-to-find bugs. An action that uses inherited attributes has to take into account every place in the grammar where its rule is used. In this example, if you changed the grammar to use a `namelist` somewhere else, you'd have to make sure that, in the new place where the `namelist` occurs, appropriate symbols precede it so that `$0` and `$-1` will get the right values:

```
declaration:      STRING namelist ; /* won't work! */
```

Inherited attributes can occasionally be very useful, particularly for syntactically complex constructs such as C language variable declarations. But it's usually safer and easier to use a global variable for the value that would have been fetched from a synthesized attribute. Or it's often nearly as easy to use synthesized attributes. In the previous example, the `namelist` rules could create a linked list of references to the names to be declared and return a pointer to that list as its value. The action for declaration could take the class, type, and `namelist` values and at that point assign the class and type to the names in the `namelist`.

Symbol Types for Inherited Attributes

When you use the value of an inherited attribute, the usual value declaration techniques (e.g., `%type`) don't work. Since the symbol corresponding to the value doesn't appear in the rule, bison cannot tell what the correct type is. You have to supply type names in the action code using an explicit type. In the previous example, if the types of `class` and `type` were `cval` and `tval`, the last two lines would actually read like this:

```
namelist:    NAME        { mksymbol($<tval>0, $<cval>-1, $1); }
 |     namelist NAME     { mksymbol($<tval>0, $<cval>-1, $2); }
```

See Symbol Values for additional information.

%initial-action

If you need to initialize something when your parser starts up, you can use `%initial-action { some-code }` to tell bison to copy `some-code` near the beginning of `yyparse`. The place where the code is copied comes after the standard initialization code, so you cannot usefully put variable declarations in the code. (They'll be accepted, but they won't be accessible in your actions.) If you need to define your own parse-time variables, you have to either use static globals or pass them as arguments via `%parse-param`.

Lexical Feedback

Parsers can sometimes feed information back to the lexer to handle otherwise difficult situations. For example, consider an input syntax like this:

```
message (any characters)
```

where in this particular context the parentheses are acting as string quotes. You can't just decide to parse a string any time you see an open parenthesis, because open parentheses might be used differently elsewhere in the grammar.

A straightforward way to handle this situation is to feed context information from the parser back to the lexer, for example, set a flag in the parser when a context-dependent construct is expected:

```
/* parser */
%{
int parenstring = 0;
}%
. . .
%%
statement: MESSAGE { parenstring = 1; } '(' STRING ')' ;
```

and look for it in the lexer:

```
%{
extern int parenstring;
%}
%s PSTRING
%%
. . .
"message"  return MESSAGE;
```

```
"("   {    if(parenstring)
                  BEGIN PSTRING;
             return '(';
          }
<PSTRING>[^)]*   {
                 yylval.svalue = strdup(yytext);   /* pass string to parse
                 BEGIN INITIAL;
                 return STRING;
                 }
```

This code is not bulletproof, because if there is some other rule that starts with `MESSAGE`, bison might have to use a lookahead token, in which case the in-line action wouldn't be executed until after the open parenthesis had been scanned. In most real cases that isn't a problem because the syntax tends to be simple. If the parser does error recovery, the error code needs to reset `parenstring`.

In this example, you could also handle the special case in the lexer by setting `parenstring` in the lexer, for example:

```
"message("  { parenstring = 1; return MESSAGE; }
```

This could cause problems, however, if the token `MESSAGE` is used elsewhere in the grammar and is not always followed by a parenthesized string. You usually have the choice of doing lexical feedback entirely in the lexer or doing it partly in the parser, with the best solution depending on how complex the grammar is. If the grammar is simple and tokens do not appear in multiple contexts, you can do all of your lexical hackery in the lexer, while if the grammar is more complex, it is easier to identify the special situations in the parser.

This approach can be taken to extremes—I wrote a complete Fortran 77 parser in yacc, bison's predecessor (but, not lex, since tokenizing Fortran is just too strange), and the parser needed to feed a dozen special context states back to the lexer. It was messy, but it was far easier than writing the whole parser and lexer in C.

Literal Block

The literal block in the definition section is bracketed by the lines `%{` and `%}`.

```
%{
... C code and declarations ...
%}
```

The contents of the literal block are copied verbatim to the generated C source file near the beginning, before the beginning of `yyparse()`. The literal block usually contains declarations of variables and functions used by code in the rules section, as well as `#include` lines for any necessary header files.

Bison also provides an experimental `%code POS { ... }` where `POS` is a keyword to suggest where in the generated parser the code should go. See %code Blocks for current details.

Literal Tokens

Bison treats a character in single quotes as a token. In this example,

```
expr: '(' expr ')' ;
```

the open and close parentheses are literal tokens. The token number of a literal token is the numeric value in the local character set, usually ASCII, and is the same as the C value of the quoted character.

The lexer usually generates these tokens from the corresponding single characters in the input, but as with any other token, the correspondence between the input characters and the generated tokens is entirely up to the lexer. A common technique is to have the lexer treat all otherwise unrecognized characters as literal tokens. For example, in a flex scanner:

```
.      return yytext[0];
```

this covers all of the single-character operators in a language and lets bison catch all unrecognized characters in the input with parse errors.

Bison also allows you to declare strings as aliases for tokens, for example:

```
%token NE "!="
%%
...
exp: exp "!=" exp ;
```

This defines the token `NE` and lets you use `NE` and `!=` interchangeably in the parser. The lexer must still return the internal token value for `NE` when the token is read, not a string.

Locations

To aid in error reporting, bison offers locations, a feature to track the source line and column range for every symbol in the parser. Locations are enabled explicitly with `%locations` or implicitly by referring to a location in the action code. The lexer has to track the current line and column and set the location range for each token in `yylloc` before returning the token. (Flex lexers will automatically track the line number, but tracking the column number is up to you.) A default rule in the parser invoked every time a rule is reduced sets the location range of the LHS symbol from the beginning line and column of the first RHS symbol to the ending line and column of the last RHS symbol.

Within action code, the location for each symbol is referred to as `@$` for the LHS and as `@1`, and so forth, for the RHS symbols. Each location is actually a structure, with references to fields like `@3.first_column`.

Locations are overkill for the error-reporting needs of many, perhaps most, parsers. Parse errors still just report the single token where the parse failed, so the only time a location range is visible to the user is if your action code reports it. The most likely place to use them is in an integrated development environment that could use them to highlight the source code responsible for semantic errors. For some examples, see Chapter 8.

%parse-param

Normally, you call `yyparse()` with no arguments. If the parser needs to import information from the surrounding program, it can use global variables, or you can add parameters to its definition:

```
%parse-param {char *modulename}
%parse-param {int intensity}
```

This allows you to call `yyparse("mymodule", 42)` and refer to `modulename` and `intensity` in action code in the parser. Note that there are no semicolons and no trailing punctuation, since the parameters are just placed between the parentheses in the definition of `yyparse`.

In normal parsers, there's little reason to use parse parameters rather than global variables, but if you generate a pure parser that can be called multiple times, either recursively or in multiple threads, parameters are the easiest way to provide the parameters to each instance of the parser.

Portability of Bison Parsers

There are two levels at which you can port a parser: the original bison grammar or the generated C source file.

Porting Bison Grammars

For the most part, you can write bison parsers and expect anyone's version of bison to handle them, since the core features bison have changed very little in the past 15 years. If you know that your parser requires features added in recent years, you can declare the minimum version of bison needed to compile it:

```
%require "2.4"
```

Porting Generated C Parsers

Bison parsers are in general very portable among reasonably modern C or C++ implementations, C89 or later, or ANSI/ISO C++.

Libraries

The only routines in the bison library are usually `main()` and `yyerror()`. Any nontrivial parser has its own version of those two routines, so the library usually isn't necessary.

Character Codes

Moving a generated parser between machines that use different character codes can be tricky. In particular, you must avoid literal tokens like `'0'` since the parser uses the character code as an index into internal tables; so, a parser generated on an ASCII machine where the code for `'0'` is 48 will fail on an EBCDIC machine where the code is 240.

Bison assigns its own numeric values to symbolic tokens, so a parser that uses only symbolic tokens should port successfully.

Precedence and Associativity Declarations

Normally, all bison grammars have to be unambiguous. That is, there is only one possible way to parse any legal input using the rules in the

grammar.

Sometimes, an ambiguous grammar is easier to use. Ambiguous grammars cause *conflicts*, situations where there are two possible parses and hence two different ways that bison can process a token. When bison processes an ambiguous grammar, it uses default rules to decide which way to parse an ambiguous sequence. Often these rules do not produce the desired result, so bison includes operator declarations that let you change the way it handles shift/reduce conflicts that result from ambiguous grammars. (See also Ambiguity and Conflicts.)

Most programming languages have complicated rules that control the interpretation of arithmetic expressions. For example, the C expression:

```
a = b = c + d / e / f
```

is treated as follows:

```
a = (b = (c + ((d / e) / f)))
```

The rules for determining what operands group with which operators are known as *precedence* and *associativity*.

Precedence

Precedence assigns each operator a precedence "level." Operators at higher levels bind more tightly; for example, if `*` has higher precedence than `+`, `A+B*C` is treated as `A+(B*C)`, while `D*E+F` is `(D*E)+F`.

Associativity

Associativity controls how the grammar groups expressions using the same operator or different operators with the same precedence. They can group from the left, from the right, or not at all. If `-` were left associative, the expression `A-B-C` would mean `(A-B)-C`, while if it were right associative, it would mean `A-(B-C)`.

Some operators such as Fortran `.GE.` are not associative either way; that is, `A .GE. B .GE. C` is not a valid expression.

Precedence Declarations

Precedence declarations appear in the definition section. The possible declarations are `%left`, `%right`, and `%nonassoc`. The `%left` and

`%right` declarations make an operator left or right associative, respectively. You declare nonassociative operators with `%nonassoc` .

Operators are declared in increasing order of precedence. All operators declared on the same line are at the same precedence level. For example, a Fortran grammar might include the following:

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

The lowest precedence operators here are `+` and `-` , the middle predecence operators are `*` and `/` , and the highest is `POW` , which represents the `**` power operator.

Using Precedence and Associativity to Resolve Conflicts

Every token in a grammar can have a precedence and an associativity assigned by a precedence declaration. Every rule can also have a precedence and an associativity, which is taken from a `%prec` clause in the rule or, failing that, the rightmost token in the rule that has a precedence assigned.

Whenever there is a shift/reduce conflict, bison compares the precedence of the token that might be shifted to that of the rule that might be reduced. It shifts if the token's precedence is higher or reduces if the rule's precedence is higher. If both have the same precedence, bison checks the associativity. If they are left associative, it reduces; if they are right associative, it shifts; and if they are nonassociative, bison generates an error.

Typical Uses of Precedence

Although you can in theory use precedence to resolve any kind of shift/reduce conflict, you should use precedence only for a few well-understood situations and rewrite the grammar otherwise. Precedence declarations were designed to handle expression grammars, with large numbers of rules like this:

```
expr OP expr
```

Expression grammars are almost always written using precedence.

The only other common use is if/then/else, where you can resolve the "dangling else" problem more easily with precedence than by rewriting the grammar.

See Chapter 7 for details. Also see Bugs in Bison Programs for a common pitfall using `%prec` .

Recursive Rules

To parse a list of items of indefinite length, you write a *recursive* rule, one that is defined in terms of itself. For example, this parses a possibly empty list of numbers:

```
numberlist: /* empty */
      | numberlist NUMBER
      ;
```

The details of the recursive rule vary depending on the exact syntax to be parsed. The next example parses a nonempty list of expressions separated by commas, with the symbol `expr` being defined elsewhere in the grammar:

```
exprlist:     expr
      |       exprlist ',' expr
      ;
```

It's also possible to have mutually recursive rules that refer to each other:

```
exp:       term
      |    term '+' term
      ;
term:     '(' exp ')'
      |   VARIABLE
      ;
```

Any recursive rule and each rule in a group of mutually recursive rules must have at least one nonrecursive alternative (one that does not refer to itself). Otherwise, there is no way to terminate the string that it matches, which is an error.

Left and Right Recursion

When you write a recursive rule, you can put the recursive reference at the left end or the right end of the right-hand side of the rule, for

example:

```
exprlist:    exprlist ',' expr ;    /* left recursion */

exprlist:    expr ',' exprlist ;    /* right recursion */
```

In most cases, you can write the grammar either way. Bison handles left recursion much more efficiently than right recursion. This is because its internal stack keeps track of all symbols seen so far for all partially parsed rules. If you use the right-recursive version of `exprlist` and have an expression with 10 expressions in it, by the time the 10th expression is read, there will be 20 entries on the stack: an `expr` and a comma for each of the 10 expressions. When the list ends, all of the nested `exprlist` s will be reduced, starting from right to left. On the other hand, if you use the left-recursive version, the `exprlist` rule is reduced after each `expr` , so the list will never have more than three entries on the internal stack.

A 10-element expression list poses no problems in a parser, but grammars often parse lists that are hundreds or thousands of items long, particularly when a program is defined as a list of statements:

```
%start program
%%
program:   statementlist ;

statementlist :   statement
            | statementlist ';' statement
            ;
statement: . . .
```

In this case, a 5,000-statement program is parsed as a 10,000-element list of statements and semicolons, and a right-recursive list of 10,000 elements is too large for most bison parsers.

Right-recursive grammars can be useful for a list of items that you know will be short and that you want to make into a linked list of values:

```
thinglist:   THING { $$ = $1; }
      |      THING thinglist { $1->next = $2; $$ = $1; }
      ;
```

With a left-recursive grammar, either you end up with the list linked in reverse order with a reversal step at the end or you need extra code to

search for the end of the list at each stage in order to add the next thing to the end.

You can control the size of the parser stack by defining `YYINITDEPTH`, which is the initial stack size that is normally 200, and `YYMAXDEPTH`, which is the maximum stack size that is normally 10,000. For example:

```
%{
#define YYMAXDEPTH 50000
%}
```

Each stack entry is the size of a semantic value (the largest size in the `%union` entries) plus two bytes for the token number plus, if you are using locations, 16 bytes for the location. On a workstation with a gigabyte of virtual memory, a stack of 100,000 entries would be a manageable 2 or 3 megabytes, but on a smaller embedded system, you'd probably want to rewrite your grammar to limit the stack size.

Rules

A bison grammar consists of a set of *rules*. Each rule starts with a nonterminal symbol and a colon and is followed by a possibly empty list of symbols, literal tokens, and actions. Rules by convention end with a semicolon, although the semicolon is technically optional. For example,

```
date: month '/' day '/' year ;
```

says that a date is a month, a slash, a day, another slash, and a year. (The symbols month, day, and year must be defined elsewhere in the grammar.) The initial symbol and colon are called the *left-hand side (LHS)* of the rule, and the rest of the rule is the *right-hand side (RHS)*. The right-hand side may be empty.

If several consecutive rules in a grammar have the same LHS, the second and subsequent rules may start with a vertical bar rather than the name and the colon. These two fragments are equivalent:

```
declaration:     EXTERNAL name ;
declaration:     ARRAY name '(' size ')' ;

declaration:     EXTERNAL name
      |    ARRAY name '(' size ')' ;
```

The form with the vertical bar is better style. The semicolon must be omitted before a vertical bar. Multiple rules with the same LHS need not occur together. See the SQL grammar in Chapter 4 where there are multiple rules defining the term `sql` throughout the grammar.

An *action* is a C compound statement that is executed whenever the parser reaches the point in the grammar where the action occurs:

```
date: month '/' day '/' year
                    { printf("Date recognized.\n"); }
      ;
```

The C code in actions may have some special constructs starting with `$` or `@` that are specially treated by bison. (See Actions and Locations for details.) Actions that occur anywhere except at the end of a rule are treated specially. (See Embedded Actions for details.)

A rule may have an explicit precedence at the end:

```
expr: expr '*' expr
    | expr '-' expr
    | '-' expr %prec UMINUS ;
```

The precedence is used only to resolve otherwise ambiguous parses. See Precedence and Associativity Declarations for details. In a GLR parser, a rule can also have a `%dprec` precedence to resolve ambiguous parses.

Special Characters

Since bison deals with symbolic tokens rather than literal text, its input character set is considerably simpler than lex's. Here is a list of the special characters that it uses:

%
: A line with two percent signs separates the parts of a bison grammar (see Structure of a Bison Grammar). All of the declarations in the definition section start with `%`, including `%{ %}`, `%start`, `%token`, `%type`, `%left`, `%right`, `%nonassoc`, and `%union`. See Literal Block, %start Declaration, %type Declaration, Precedence and Associativity Declarations, and %union Declaration.

$
: In actions, a dollar sign introduces a value reference, for example, `$3` for the value of the third symbol in the rule's right-hand side.

See Symbol Values.

@

In actions, an @ sign introduces a location reference, such as `@2` for the location of the second symbol in the RHS.

'

Literal tokens are enclosed in single quotes, for example, `'Z'`. See Literal Tokens.

"

Bison lets you declare quoted strings as parser aliases for tokens. See Literal Tokens.

< >

In a value reference in an action, you can override the value's default type by enclosing the type name in angle brackets, for example, `$<xtype>3`. See Declaring Symbol Types.

{ }

The C code in actions is enclosed in curly braces. (See Actions.) C code in the literal block declarations section is enclosed in `%{` and `%}`. See Literal Block.

;

Each rule in the rules section should end with a semicolon, except those that are immediately followed by a rule that starts with a vertical bar. The semicolons are optional, but they are always a good idea. See Rules.

|

When two consecutive rules have the same left-hand side, the second rule may replace the symbol and colon with a vertical bar. See Rules.

:

In each rule, a colon follows the symbol on the rule's left-hand side. See Rules.

_

Symbols may include underscores along with letters, digits, and periods.

.

Symbols may include periods along with letters, digits, and underscores. This can cause trouble because C identifiers cannot include

periods. In particular, do not use tokens whose names contain periods, since the token names are all `#define`'d as C preprocessor symbols.

%start Declaration

Normally, the start rule, the one that the parser starts trying to parse, is the one named in the first rule. If you want to start with some other rule, in the declaration section you can write the following:

```
%start somename
```

to start with rule *somename*.

In most cases, the clearest way to present the grammar is top-down, with the start rule first, so no `%start` is needed.

Symbol Values

Every symbol in a bison parser, both tokens and nonterminals, can have a value associated with it. If the token were `NUMBER`, the value might be the particular number; if it were `STRING`, the value might be a pointer to a copy of the string; and if it were `SYMBOL`, the value might be a pointer to an entry in the symbol table that describes the symbol. Each of these kinds of value corresponds to a different C type: `int` or `double` for the number, `char *` for the string, and a pointer to a structure for the symbol. Bison makes it easy to assign types to symbols so that it automatically uses the correct type for each symbol.

Declaring Symbol Types

Internally, bison declares each value as a C union that includes all of the types. You list all of the types in `%union` declarations. Bison turns them into a `typedef` for a union type called `YYSTYPE`. Then for each symbol whose value is set or used in action code, you have to declare its type. Use `%type` for nonterminals. Use `%token`, `%left`, `%right`, or `%nonassoc` for tokens to give the name of the union field corresponding to its type.

Then, whenever you refer to a value using `$$`, `$1`, etc., bison automatically uses the appropriate field of the union.

Bison doesn't understand any C, so any symbol typing mistakes you make, such as using a type name that isn't in the union or using a field in a way

that C doesn't allow, will cause errors in the generated C program.

Explicit Symbol Types

Bison allows you to declare an explicit type for a symbol value reference by putting the type name in angle brackets between the dollar sign and the symbol number or between the two dollar signs, for example, `$<xxx>3` or `$<zzz>$`.

The feature is rarely used, since in nearly all cases it is easier and more readable to declare the symbols. The most plausible uses are when referring to inherited attributes and when setting and referring to the value returned by an embedded action. See Inherited Attributes ($0) and Actions for details.

Tokens

Tokens or *terminal symbols* are symbols that the lexer passes to the parser. Whenever a bison parser needs another token, it calls `yylex()`, which returns the next token from the input. At the end of input, `yylex()` returns zero.

Tokens may be either symbols defined by `%token` or individual characters in single quotes. (See Literal Tokens.) All symbols used as tokens must be defined explicitly in the definition section, for example:

```
%token UP DOWN LEFT RIGHT
```

Tokens can also be declared by `%left`, `%right`, or `%nonassoc` declarations, each of which has exactly the same syntax options as `%token` has. See Precedence and Associativity Declarations.

Token Numbers

Within the lexer and parser, tokens are identified by small integers. The token number of a literal token is the numeric value in the local character set, usually ASCII, and is the same as the C value of the quoted character.

Symbolic tokens usually have values assigned by bison, which gives them numbers higher than any possible character's code, so they will not conflict with any literal tokens. You can assign token numbers yourself by following the token name by its number in `%token`:

```
%token UP 50 DOWN 60 LEFT 17 RIGHT 25
```

It is an error to assign two tokens the same number. In most cases it is easier and more reliable to let bison choose its own token numbers.

The lexer needs to know the token numbers in order to return the appropriate values to the parser. For literal tokens, it uses the corresponding C character constant. For symbolic tokens, you can tell bison with the `-d` command-line flag to create a C header file with definitions of all the token numbers. If you `# include` that header file in your lexer, you can use the symbols, for example, `UP`, `DOWN`, `LEFT`, and `RIGHT`, in its C code. The header file is normally called `xxx.tab.h` if your source file was `xxx.y`, or you can rename it with the `%defines` declaration or the `--defines=` *filename* command-line option.

```
%defines "xxxsyms.h"
```

Token Values

Each symbol in a bison parser can have an associated value. (See Symbol Values.) Since tokens can have values, you need to set the values as the lexer returns tokens to the parser. The token value is always stored in the variable `yylval`. In the simplest parsers, `yylval` is a plain `int` variable, and you might set it like this in a flex scanner:

```
[0-9]+      { yylval = atoi(yytext); return NUMBER; }
```

In most cases, though, different symbols have different value types. See %union Declaration, Symbol Values, and below.

In the parser you must declare the value types of all tokens that have values. Put the name of the appropriate union tag in angle brackets in the `%token` or precedence declaration. You might define your values types like this:

```
%union {
      enum optype opval;
      double dval;
}

%nonassoc <opval> RELOP
%token <dval> REAL

%union { char *sval; }
. . .
%token <sval> STRING
```

In this case `RELOP` might be a relational operator such as `==` or `>`, and the token value says which operator it is.

You set the appropriate field of `yylval` when you return the token. In this case, you'd do something like this in lex:

```
%{
#include "parser.tab.h"
%}
. . .
[0-9]+\.[0-9]*   { yylval.dval = atof(yytext); return REAL; }
\"[^"]*\"  { yylval.sval = strdup(yytext); return STRING; }
"=="       { yyval.opval = OPEQUAL; return RELOP; }
```

The value for `REAL` is a `double`, so it goes into `yylval.dval`, while the value for `STRING` is a `char *`, so it goes into `yylval.sval`.

%type Declaration

You declare the types of nonterminals using `%type`. Each declaration has the following form:

```
%type <type> name, name, . . .
```

Each `type` name must have been defined by a `%union`. (See %union Declaration.) Each `name` is the name of a nonterminal symbol. See Declaring Symbol Types for details and an example.

Use `%type` to declare nonterminals. To declare tokens, you can also use `%token`, `%left`, `%right`, or `%nonassoc`. See Tokens and Precedence and Associativity Declarations for details.

%union Declaration

The `%union` declaration identifies all of the possible C types that a symbol value can have. (See Symbol Values.) The declaration takes this form:

```
%union {
  ... field declarations...
}
```

The field declarations are copied verbatim into a `C union` declaration of the type `YYSTYPE` in the output file. Bison does not check to see whether the contents of the `%union` are valid C. If you have more than one

`%union` declaration, their contents are concatenated to create the C or C++ union declaration.

In the absence of a `%union` declaration, bison defines `YYSTYPE` to be `int`, so all of the symbol values are integers.

You associate the types declared in `%union` with particular symbols using the `%type` declaration.

Bison puts the generated C union declaration both in the generated C file and in the optional generated header file (called *name*`.tab.h` unless you tell it otherwise), so you can use `YYSTYPE` in other source files by including the generated header file. Conversely, you can put your own declaration of `YYSTYPE` in an include file that you reference with `#include` in the definition section. In this case, there must be at least one `%type` or other declaration that specifies a symbol type to warn bison that you are using explicit symbol types.

Variant and Multiple Grammars

You may want to have parsers for two partially or entirely different grammars in the same program. For example, an interactive debugging interpreter might have one parser for the programming language and another for debugger commands. A one-pass C compiler might need one parser for the preprocessor syntax and another for the C language itself.

There are two ways to handle two grammars in one program: combine them into a single parser or put two complete parsers into the program.

Combined Parsers

If you have several similar grammars, you can combine them into one by adding a new start rule that depends on the first token read. For example:

```
%token CSTART PPSTART
%%
combined:  CSTART cgrammar
      |    PPSTART ppgrammar
      ;

cgrammar:  . . .

ppgrammar: . . .
```

In this case, if the first token is `CSTART`, it parses the grammar whose start rule is `cgrammar`, while if the first token is `PPSTART`, it parses the grammar whose start rule is `ppgrammar`.

You also need to put code in the lexer that returns the appropriate special token the first time that the parser asks the lexer for a token:

```
%%
%{
     extern first_tok;

     if(first_tok) {
          int holdtok = first_tok;

          first_tok = 0;
          return holdtok;
     }
%}
. . . <the rest of the lexer>
```

In this case, you set `first_tok` to the appropriate token before calling `yyparse()`.

One advantage of this approach is that the program is smaller than it would be with multiple parsers, since there is only one copy of the parsing code. Another is that if you are parsing related grammars, for example, C preprocessor expressions and C itself, you should be able to share some parts of the grammar. The disadvantages are that you cannot call one parser while the other is active unless you create a pure parser and that you have to use different symbols in the two grammars except where they deliberately share rules, and the possibilities for hard-to-find errors are rife if you accidentally use the same symbol in the two grammars.

In practice, this approach is useful when you want to parse slightly different versions of a single language, for example, a full language that is compiled and an interactive subset that you interpret in a debugger. If one language is actually a subset of the other, a better approach would be to use a single parser for both, check in the action code in the rules excluded from the subset for which version is being parsed, and if it's the subset, report an error to the user.

Multiple Parsers

The other approach is to include two complete parsers in a single program. Every bison parser normally has the same entry point,

`yyparse()`, and calls the same lexer, `yylex()`, which uses the same token value variable `yylval`. Also, the parse tables and parser stack are in global variables with names like `yyact` and `yyv`. If you just translate two grammars and compile and link the two resulting files, you get a long list of multiply defined symbols. The trick is to change the names that bison uses for its functions and variables.

Using %name-prefix or the -p Flag

You can use a declaration in the bison source code to change the prefix used on the names in the parser generated by bison.

```
%name-prefix "pdq"
```

This produces a parser with the entry point `pdqparse()`, which calls the lexer `pdqlex()`, and so forth.

Specifically, the names affected are `yyparse()`, `yylex()`, `yyerror()`, `yylval`, `yychar`, and `yydebug`. (The variable `yychar` holds the most recently read token, which is sometimes useful when printing error messages.) The other variables used in the parser may be renamed or may be made `static` or `auto`; in any event, they are guaranteed not to collide. There is also a `-p` flag to specify the prefix on the command line rather than in the source file, and there is a a `-b` flag to specify the prefix of the generated C file; for example,

```
bison -d -p pdq -b pref mygram.y
```

would produce `pref.tab.c` and `pref.tab.h` with a parser whose entry point is `pdqparse`.

You have to provide properly named versions of `yyerror()` and `yylex()`.

Lexers for Multiple Parsers

If you use a flex lexer with your multiple parsers, you need to make adjustments to the lexer to correspond to the changes to the parser. (See Multiple Lexers in One Program.) You will usually want to use a combined lexer with a combined parser and use multiple lexers with multiple parsers.

Pure Parsers

A slightly different problem is that of recursive parsing, calling `yyparse()` a second time while the original call to `yyparse()` is still active. This can be an issue when you have combined parsers. If you have a combined C language and C preprocessor parser, you can call `yyparse()` in C language mode once to parse the whole program, and you can call it recursively whenever you see a `#if` to parse a preprocessor expression.

Pure parsers are also useful in threaded programs, where each thread might be parsing input from a separate source. See Pure Scanners and Parsers in Chapter 9 for the details on pure parsers.

y.output Files

Bison can create a log file, traditionally named *`y.output`* or now more often *`name`*`.output`, that shows all of the states in the parser and the transitions from one state to another. Use the *`--report=all`* flag to generate a log file.

Here's part of the log for the bison grammar in Chapter 1:

```
state 3

   10 term: NUMBER .

    $default  reduce using rule 10 (term)

state 4

   11 term: ABS . term

    NUMBER  shift, and go to state 3
    ABS     shift, and go to state 4

    term  go to state 9


state 5

    2 calclist: calclist calc . EOL

    EOL  shift, and go to state 10
```

The dot in each state shows how far the parser has gotten parsing a rule when it gets to that state. When the parser is in state 4, for example, if the parser sees a `NUMBER` token, it shifts the `NUMBER` onto the stack and

switches to state 3. If it sees an `ABS`, it shifts and switches back to state 4, and any other token is an error. If a subsequent reduction returns to this state with a `term` on the top of the stack, it switches to state 9. In state 3, it always reduces rule 10. (Rules are numbered in the order they appear in the input file.) After the reduction, the `NUMBER` is replaced on the parse stack by a `term`, and the parser pops back to state 4, at which point the `term` makes it go to state 9.

When there are conflicts, the states with conflicts show the conflicting shift and reduce actions.

```
State 19 conflicts: 3 shift/reduce

state 19

    5 exp: exp . ADD exp
    5    | exp ADD exp .
    6    | exp . SUB factor
    7    | exp . ABS factor

    ADD  shift, and go to state 12
    SUB  shift, and go to state 13
    ABS  shift, and go to state 14

    ADD       [reduce using rule 5 (exp)]
    SUB       [reduce using rule 5 (exp)]
    ABS       [reduce using rule 5 (exp)]
    $default  reduce using rule 5 (exp)
```

In this case, there is a shift/reduce conflict when bison sees a plus sign. You could fix it either by rewriting the grammar or by adding an operator declaration for the plus sign. See Precedence and Associativity Declarations.

Bison Library

Bison inherits a library of helpful routines from its predecessor yacc. You can include the library by giving the `-ly` flag at the end of the `cc` command line. The library contains `main()` and `yyerror()`.

main()

The library has a minimal main program that is sometimes useful for quickie programs and for testing. It's so simple we can reproduce it here:

```
main(ac, av)
{
      yyparse();
      return 0;
}
```

As with any library routine, you can provide your own `main()`. In nearly any useful application you will want to provide a `main()` that accepts command-line arguments and flags, opens files, and checks for errors.

yyerror()

Bison also provides a simple error-reporting routine. It's also simple enough to list in its entirety:

```
yyerror(char *errmsg)
{
      fprintf(stderr, "%s\n", errmsg);
}
```

This sometimes suffices, but a better error routine that reports at least the line number and the most recent token (available in `yytext` if your lexer is written with lex) will make your parser much more usable.

YYABORT

The special statement

```
YYABORT;
```

in an action makes the parser routine `yyparse()` return immediately with a nonzero value, indicating failure.

It can be useful when an action routine detects an error so severe that there is no point in continuing the parse.

Since the parser may have a one-token lookahead, the rule action containing the `YYABORT` may not be reduced until the parser has read another token.

YYACCEPT

The special statement

```
YYACCEPT;
```

in an action makes the parser routine `yyparse()` return immediately with a value 0, indicating success.

It can be useful in a situation where the lexer cannot tell when the input data ends but the parser can.

Since the parser may have a one-token lookahead, the rule action containing the `YYACCEPT` may not be reduced until the parser has read another token.

YYBACKUP

The macro `YYBACKUP` lets you unshift the current token and replace it with something else. The syntax is as follows:

```
sym:    TOKEN  { YYBACKUP(newtok, newval); }
```

It discards the symbol `sym` that would have been substituted by the reduction and pretends that the lexer just read the token `newtok` with the value `newval`. If there is a look-ahead token or the rule has more than one symbol on the right side, the rule fails with a call to `yyerror()`.

It is extremely difficult to use `YYBACKUP()` correctly, so you're best off not using it. (It's documented here in case you come across an existing grammar that does use it.)

yyclearin

The macro `yyclearin` in an action discards a lookahead token if one has been read. It is most often useful in error recovery in an interactive parser to put the parser into a known state after an error:

```
stmtlist: stmt | stmtlist stmt ;

stmt: error  { reset_input(); yyclearin; } ;
```

After an error, this calls the user routine `reset_input()`, which presumably puts the input into a known state, and then uses `yyclearin` to prepare to start reading tokens anew.

See YYRECOVERING() and below for more information.

yydebug and YYDEBUG

Bison can optionally compile in trace code that reports everything that the parser does. These reports are extremely verbose but are often the only way to figure out what a recalcitrant parser is doing.

YYDEBUG

Since the trace code is large and slow, it is not automatically compiled into the object program. To include the trace code, either use the `-t` flag on the bison command line or else define the C preprocessor symbol `YYDEBUG` to be nonzero either on the C compiler command line or by including something like this in the definition section:

```
%{
#define YYDEBUG 1
%}
```

yydebug

The integer variable `yydebug` in the running parser controls whether the parser actually produces debug output. If it is nonzero, the parser produces debugging reports, while if it is zero, it doesn't. You can set `yydebug` nonzero in any way you want, for instance, in response to a flag on the program's command line or by patching it at runtime with a debugger.

yyerrok

After bison detects a syntax error, it normally refrains from reporting another error until it has shifted three consecutive tokens without another error. This somewhat alleviates the problem of multiple error messages resulting from a single mistake as the parser gets resynchronized.

If you know when the parser is back in sync, you can return to the normal state in which all errors are reported. The macro `yyerrok` tells the parser to return to the normal state.

For example, assume you have a command interpreter in which all commands are on separate lines. No matter how badly the user botches a command, you know the next line is a new command.

```
cmdlist: cmd | cmdlist cmd ;

cmd:  error '\n' { yyerrok; } ;
```

The rule with `error` skips input after an error up to a newline, and `yyerrok` tells the parser that error recovery is complete.

See also YYRECOVERING() and yyclearin.

YYERROR

Sometimes your action code can detect context-sensitive syntax errors that the parser itself cannot. If your code detects a syntax error, you can call the macro `YYERROR` to produce exactly the same effect as if the parser had read a token forbidden by the grammar. As soon as you invoke `YYERROR`, the parser goes into error recovery mode looking for a state where it can shift an `error` token. See Error Token and Error Recovery for details. If you want to produce an error message, you have to call `yyerror` yourself.

yyerror()

Whenever a bison parser detects a syntax error, it calls `yyerror()` to report the error to the user, passing it a single argument: a string describing the error. (Unless you have calls to `yyerror()` in your own code, usually the only error you ever get is "syntax error.") The default version of `yyerror` in the bison library merely prints its argument on the standard output. Here is a slightly more informative version:

```
yyerror(const char *msg)
{
      printf("%d: %s at '%s'\n", yylineno, msg, yytext);
}
```

We assume `yylineno` is the current line number. (See Line Numbers and yylineno.) `yytext` is the flex token buffer that contains the current token.

Since bison doggedly tries to recover from errors and parse its entire input, no matter how badly garbled, you may want to have `yyerror()` count the number of times it's called and exit after 10 errors, on the theory that the parser is probably hopelessly confused by the errors that have already been reported.

You can and probably should call `yyerror()` yourself when your action routines detect other sorts of errors.

yyparse()

The entry point to a bison-generated parser is `yyparse()`. When your program calls `yyparse()`, the parser attempts to parse an input stream. The parser returns a value of zero if the parse succeeds and nonzero if not. The parser normally takes no arguments, but see %parse-param for more information.

Every time you call `yyparse()`, the parser starts parsing anew, forgetting whatever state it might have been in the last time it returned. This is quite unlike the scanner `yylex()` generated by lex, which picks up where it left off each time you call it.

See also YYACCEPT and YYABORT.

YYRECOVERING()

After bison detects a syntax error, it normally enters a recovery mode in which it refrains from reporting another error until it has shifted three consecutive tokens without another error. This somewhat alleviates the problem of multiple error messages resulting from a single mistake as the parser gets resynchronized.

The macro `YYRECOVERING()` returns nonzero if the parser is currently in the error recovery mode and zero if it is not. It is sometimes convenient to test `YYRECOVERING()` to decide whether to report errors discovered in an action routine.

See also yyclearin and yyerrok.