Go to the previous, next section.

Bison Grammar Files

Bison takes as input a context-free grammar specification and produces a C-language function that recognizes correct instances of the grammar.

The Bison grammar input file conventionally has a name ending in `.y'.

Outline of a Bison Grammar

A Bison grammar file has four main sections, shown here with the appropriate delimiters:

```
%{
C declarations
%}

Bison declarations

%%
Grammar rules
%%

Additional C code
```

Comments enclosed in `/* ... */' may appear in any of the sections.

The C Declarations Section

The *C declarations* section contains macro definitions and declarations of functions and variables that are used in the actions in the grammar rules. These are copied to the beginning of the parser file so that they precede the definition of `yyparse`. You can use `#include' to get the declarations from a header file. If you don't need any C declarations, you may omit the `%{' and `%}' delimiters that bracket this section.

The Bison Declarations Section

The *Bison declarations* section contains declarations that define terminal and nonterminal symbols, specify precedence, and so on. In some simple grammars you may not need any declarations. See section Bison Declarations.

The Grammar Rules Section

The *grammar rules* section contains one or more Bison grammar rules, and nothing else. See section Syntax of Grammar Rules.

There must always be at least one grammar rule, and the first `%%' (which precedes the grammar rules) may never be omitted even if it is the first thing in the file.

The Additional C Code Section

The *additional C code* section is copied verbatim to the end of the parser file, just as the *C declarations* section is copied to the beginning. This is the most convenient place to put anything that you want to have in the parser file but which need not come before the definition of `yyparse`. For example, the definitions of `yylex` and `yyerror` often go here. See section Parser C-Language Interface.

If the last section is empty, you may omit the `%%' that separates it from the grammar rules.

The Bison parser itself contains many static variables whose names start with `` `yy' `` and many macros whose names start with `` `YY' ``. It is a good idea to avoid using any such names (except those documented in this manual) in the additional C code section of the grammar file.

Symbols, Terminal and Nonterminal

Symbols in Bison grammars represent the grammatical classifications of the language.

A *terminal symbol* (also known as a *token type*) represents a class of syntactically equivalent tokens. You use the symbol in grammar rules to mean that a token in that class is allowed. The symbol is represented in the Bison parser by a numeric code, and the `yylex` function returns a token type code to indicate what kind of token has been read. You don't need to know what the code value is; you can use the symbol to stand for it.

A *nonterminal symbol* stands for a class of syntactically equivalent groupings. The symbol name is used in writing grammar rules. By convention, it should be all lower case.

Symbol names can contain letters, digits (not at the beginning), underscores and periods. Periods make sense only in nonterminals.

There are two ways of writing terminal symbols in the grammar:

- A *named token type* is written with an identifier, like an identifier in C. By convention, it should be all upper case. Each such name must be defined with a Bison declaration such as `%token`. See section Token Type Names.
- A *character token type* (or *literal token*) is written in the grammar using the same syntax used in C for character constants; for example, `'+'` is a character token type. A character token type doesn't need to be declared unless you need to specify its semantic value data type (see section Data Types of Semantic Values), associativity, or precedence (see section Operator Precedence).

  By convention, a character token type is used only to represent a token that consists of that particular character. Thus, the token type `'+'` is used to represent the character `` `+' `` as a token. Nothing enforces this convention, but if you depart from it, your program will confuse other readers.

  All the usual escape sequences used in character literals in C can be used in Bison as well, but you must not use the null character as a character literal because its ASCII code, zero, is the code `yylex` returns for end-of-input (see section Calling Convention for `yylex`).

How you choose to write a terminal symbol has no effect on its grammatical meaning. That depends only on where it appears in rules and on when the parser function returns that symbol.

The value returned by `yylex` is always one of the terminal symbols (or 0 for end-of-input). Whichever way you write the token type in the grammar rules, you write it the same way in the definition of `yylex`. The numeric code for a character token type is simply the ASCII code for the character, so `yylex` can use the identical character constant to generate the requisite code. Each named token type becomes a C macro in the parser file, so `yylex` can use the name to stand for the code. (This is why periods don't make sense in terminal symbols.) See section Calling Convention for `yylex`.

If `yylex` is defined in a separate file, you need to arrange for the token-type macro definitions to be available there. Use the `` `-d' `` option when you run Bison, so that it will write these macro definitions into a separate header file `` `*name*.tab.h' `` which you can include in the other source files that need it. See section Invoking Bison.

The symbol `error` is a terminal symbol reserved for error recovery (see section Error Recovery); you shouldn't use it for any other purpose. In particular, `yylex` should never return this value.

Syntax of Grammar Rules

A Bison grammar rule has the following general form:

```
result: components...
        ;
```

where *result* is the nonterminal symbol that this rule describes and *components* are various terminal and nonterminal symbols that are put together by this rule (see section Symbols, Terminal and Nonterminal).

For example,

```
exp:      exp '+' exp
        ;
```

says that two groupings of type `exp`, with a `` `+' `` token in between, can be combined into a larger grouping of type `exp`.

Whitespace in rules is significant only to separate symbols. You can add extra whitespace as you wish.

Scattered among the components can be *actions* that determine the semantics of the rule. An action looks like this:

```
{C statements}
```

Usually there is only one action and it follows the components. See section Actions.

Multiple rules for the same *result* can be written separately or can be joined with the vertical-bar character `` `|' `` as follows:

```
result:    rule1-components...
        | rule2-components...
        ...
        ;
```

They are still considered distinct rules even when joined in this way.

If *components* in a rule is empty, it means that *result* can match the empty string. For example, here is how to define a comma-separated sequence of zero or more `exp` groupings:

```
expseq:   /* empty */
        | expseq1
        ;

expseq1:  exp
        | expseq1 ',' exp
        ;
```

It is customary to write a comment `` `/* empty */' `` in each rule with no components.

Recursive Rules

A rule is called *recursive* when its *result* nonterminal appears also on its right hand side. Nearly all Bison grammars need to use recursion, because that is the only way to define a sequence of any number of somethings. Consider this recursive definition of a comma-separated sequence of one or more expressions:

```
expseq1:  exp
        | expseq1 ',' exp
        ;
```

Since the recursive use of `expseq1` is the leftmost symbol in the right hand side, we call this *left recursion*. By contrast, here the same construct is defined using *right recursion*:

```
expseq1:  exp
        | exp ',' expseq1
        ;
```

Any kind of sequence can be defined using either left recursion or right recursion, but you should always use left recursion, because it can parse a sequence of any number of elements with bounded stack space. Right recursion uses up space on the Bison stack in proportion to the number of elements in the sequence, because all the elements must be shifted onto the stack before the rule can be applied even once. See section The Bison Parser Algorithm, for further explanation of this.

Indirect or *mutual* recursion occurs when the result of the rule does not appear directly on its right hand side, but does appear in rules for other nonterminals which do appear on its right hand side.

For example:

```
expr:     primary
        | primary '+' primary
        ;

primary:  constant
        | '(' expr ')'
        ;
```

defines two mutually-recursive nonterminals, since each refers to the other.

Defining Language Semantics

The grammar rules for a language determine only the syntax. The semantics are determined by the semantic values associated with various tokens and groupings, and by the actions taken when various groupings are recognized.

For example, the calculator calculates properly because the value associated with each expression is the proper number; it adds properly because the action for the grouping `` `x + y' `` is to add the numbers associated with *x* and *y*.

Data Types of Semantic Values

In a simple program it may be sufficient to use the same data type for the semantic values of all language constructs. This was true in the RPN and infix calculator examples (see section Reverse Polish Notation Calculator).

Bison's default is to use type `int` for all semantic values. To specify some other type, define `YYSTYPE` as a macro, like this:

```
#define YYSTYPE double
```

This macro definition must go in the C declarations section of the grammar file (see section Outline of a Bison Grammar).

More Than One Value Type

In most programs, you will need different data types for different kinds of tokens and groupings. For example, a numeric constant may need type `int` or `long`, while a string constant needs type `char *`, and an identifier might need a pointer to an entry in the symbol table.

To use more than one data type for semantic values in one parser, Bison requires you to do two things:

- Specify the entire collection of possible data types, with the `%union` Bison declaration (see section The Collection of Value Types).

- Choose one of those types for each symbol (terminal or nonterminal) for which semantic values are used. This is done for tokens with the `%token` Bison declaration (see section Token Type Names) and for groupings with the `%type` Bison declaration (see section Nonterminal Symbols).

Actions

An action accompanies a syntactic rule and contains C code to be executed each time an instance of that rule is recognized. The task of most actions is to compute a semantic value for the grouping built by the rule from the semantic values associated with tokens or smaller groupings.

An action consists of C statements surrounded by braces, much like a compound statement in C. It can be placed at any position in the rule; it is executed at that position. Most rules have just one action at the end of the rule, following all the components. Actions in the middle of a rule are tricky and used only for special purposes (see section Actions in Mid-Rule).

The C code in an action can refer to the semantic values of the components matched by the rule with the construct `$n`, which stands for the value of the *n*th component. The semantic value for the grouping being constructed is `$$`. (Bison translates both of these constructs into array element references when it copies the actions into the parser file.)

Here is a typical example:

```
exp:    ...
        | exp '+' exp
            { $$ = $1 + $3; }
```

This rule constructs an `exp` from two smaller `exp` groupings connected by a plus-sign token. In the action, `$1` and `$3` refer to the semantic values of the two component `exp` groupings, which are the first and third symbols on the right hand side of the rule. The sum is stored into `$$` so that it becomes the semantic value of the addition-expression just recognized by the rule. If there were a useful semantic value associated with the `` `+' `` token, it could be referred to as `$2`.

If you don't specify an action for a rule, Bison supplies a default: `$$ = $1`. Thus, the value of the first symbol in the rule becomes the value of the whole rule. Of course, the default rule is valid only if the two data types match. There is no meaningful default action for an empty rule; every empty rule must have an explicit action unless the rule's value does not matter.

`$n` with *n* zero or negative is allowed for reference to tokens and groupings on the stack *before* those that match the current rule. This is a very risky practice, and to use it reliably you must be certain of the context in which the rule is applied. Here is a case in which you can use this reliably:

```
foo:      expr bar '+' expr  { ... }
        | expr bar '-' expr  { ... }
        ;

bar:      /* empty */
        { previous_expr = $0; }
        ;
```

As long as `bar` is used only in the fashion shown here, `$0` always refers to the `expr` which precedes `bar` in the definition of `foo`.

Data Types of Values in Actions

If you have chosen a single data type for semantic values, the `$$` and `$n` constructs always have that data type.

If you have used `%union` to specify a variety of data types, then you must declare a choice among these types for each terminal or nonterminal symbol that can have a semantic value. Then each time you use `$$` or `$n`, its data type is determined by which symbol it refers to in the rule. In this example,

```
exp:    ...
        | exp '+' exp
            { $$ = $1 + $3; }
```

`$1` and `$3` refer to instances of `exp`, so they all have the data type declared for the nonterminal symbol `exp`. If `$2` were used, it would have the data type declared for the terminal symbol `'+'`, whatever that might be.

Alternatively, you can specify the data type when you refer to the value, by inserting `` `<type>' `` after the `` `$' `` at the beginning of the reference. For example, if you have defined types as shown here:

```
%union {
  int itype;
  double dtype;
}
```

then you can write `$<itype>1` to refer to the first subunit of the rule as an integer, or `$<dtype>1` to refer to it as a double.

Actions in Mid-Rule

Occasionally it is useful to put an action in the middle of a rule. These actions are written just like usual end-of-rule actions, but they are executed before the parser even recognizes the following components.

A mid-rule action may refer to the components preceding it using `$n`, but it may not refer to subsequent components because it is run before they are parsed.

The mid-rule action itself counts as one of the components of the rule. This makes a difference when there is another action later in the same rule (and usually there is another at the end): you have to count the actions along with the symbols when working out which number *n* to use in `$n`.

The mid-rule action can also have a semantic value. The action can set its value with an assignment to `$$`, and actions later in the rule can refer to the value using `$n`. Since there is no symbol to name the action, there is no way to declare a data type for the value in advance, so you must use the `` `$<...>' `` construct to specify a data type each time you refer to this value.

There is no way to set the value of the entire rule with a mid-rule action, because assignments to `$$` do not have that effect. The only way to set the value for the entire rule is with an ordinary action at the end of the rule.

Here is an example from a hypothetical compiler, handling a `let` statement that looks like `` `let (variable) statement' `` and serves to create a variable named *variable* temporarily for the duration of *statement*. To parse this construct, we must put *variable* into the symbol table while *statement* is parsed, then remove it afterward. Here is how it is done:

```
stmt:   LET '(' var ')'
                { $<context>$ = push_context ();
                  declare_variable ($3); }
        stmt    { $$ = $6;
                  pop_context ($<context>5); }
```

As soon as `` `let (variable)' `` has been recognized, the first action is run. It saves a copy of the current semantic context (the list of accessible variables) as its semantic value, using alternative `context` in the data-type union. Then it calls `declare_variable` to add the new variable to that list. Once the first action is finished, the embedded statement `stmt` can be parsed. Note that the mid-rule action is component number 5, so the `` `stmt' `` is component number 6.

After the embedded statement is parsed, its semantic value becomes the value of the entire `let`-statement. Then the semantic value from the earlier action is used to restore the prior list of variables. This removes the temporary `let`-variable from the list so that it won't appear to exist while the rest of the program is parsed.

Taking action before a rule is completely recognized often leads to conflicts since the parser must commit to a parse in order to execute the action. For example, the following two rules, without mid-rule actions, can coexist in a working parser because the parser can shift the open-brace token and look at what follows before deciding whether there is a declaration or not:

```
compound: '{' declarations statements '}'
        | '{' statements '}'
        ;
```

But when we add a mid-rule action as follows, the rules become nonfunctional:

```
compound: { prepare_for_local_variables (); }
          '{' declarations statements '}'
        | '{' statements '}'
        ;
```

Now the parser is forced to decide whether to run the mid-rule action when it has read no farther than the open-brace. In other words, it must commit to using one rule or the other, without sufficient information to do it correctly. (The open-brace token is what is called the *look-ahead* token at this time, since the parser is still deciding what to do about it. See section Look-Ahead Tokens.)

You might think that you could correct the problem by putting identical actions into the two rules, like this:

```
compound: { prepare_for_local_variables (); }
          '{' declarations statements '}'
        | { prepare_for_local_variables (); }
          '{' statements '}'
        ;
```

But this does not help, because Bison does not realize that the two actions are identical. (Bison never tries to understand the C code in an action.)

If the grammar is such that a declaration can be distinguished from a statement by the first token (which is true in C), then one solution which does work is to put the action after the open-brace, like this:

```
compound: '{' { prepare_for_local_variables (); }
          declarations statements '}'
        | '{' statements '}'
        ;
```

Now the first token of the following declaration or statement, which would in any case tell Bison which rule to use, can still do so.

Another solution is to bury the action inside a nonterminal symbol which serves as a subroutine:

```
subroutine: /* empty */
          { prepare_for_local_variables (); }
        ;


compound: subroutine
          '{' declarations statements '}'
        | subroutine
          '{' statements '}'
        ;
```

Now Bison can execute the action in the rule for `subroutine` without deciding which rule for `compound` it will eventually use. Note that the action is now at the end of its rule. Any mid-rule action can be converted to an end-of-rule action in this way, and this is what Bison actually does to implement mid-rule actions.

Bison Declarations

The *Bison declarations* section of a Bison grammar defines the symbols used in formulating the grammar and the data types of semantic values. See section Symbols, Terminal and Nonterminal.

All token type names (but not single-character literal tokens such as `'+'` and `'*'`) must be declared. Nonterminal symbols must be declared if you need to specify which data type to use for the semantic value (see section More Than One Value Type).

The first rule in the file also specifies the start symbol, by default. If you want some other symbol to be the start symbol, you must declare it explicitly (see section Languages and Context-Free Grammars).

Token Type Names

The basic way to declare a token type name (terminal symbol) is as follows:

```
%token name
```

Bison will convert this into a `#define` directive in the parser, so that the function `yylex` (if it is in this file) can use the name *name* to stand for this token type's code.

Alternatively, you can use `%left`, `%right`, or `%nonassoc` instead of `%token`, if you wish to specify precedence. See section Operator Precedence.

You can explicitly specify the numeric code for a token type by appending an integer value in the field immediately following the token name:

```
%token NUM 300
```

It is generally best, however, to let Bison choose the numeric codes for all token types. Bison will automatically select codes that don't conflict with each other or with ASCII characters.

In the event that the stack type is a union, you must augment the `%token` or other token declaration to include the data type alternative delimited by angle-brackets (see section More Than One Value Type).

For example:

```
%union {              /* define stack type */
  double val;
  symrec *tptr;
}
%token <val> NUM      /* define token NUM and its type */
```

Operator Precedence

Use the `%left`, `%right` or `%nonassoc` declaration to declare a token and specify its precedence and associativity, all at once. These are called *precedence declarations*. See section Operator Precedence, for general information on operator precedence.

The syntax of a precedence declaration is the same as that of `%token`: either

```
%left symbols...
```

or

```
%left <type> symbols...
```

And indeed any of these declarations serves the purposes of `%token`. But in addition, they specify the associativity and relative precedence for all the *symbols*:

- The associativity of an operator *op* determines how repeated uses of the operator nest: whether ``x op y op z'` is parsed by grouping *x* with *y* first or by grouping *y* with *z* first. `%left` specifies left-associativity (grouping *x* with *y* first) and `%right` specifies right-associativity (grouping *y* with *z* first). `%nonassoc` specifies no associativity, which means that `` `x op y op z' `` is considered a syntax error.

- The precedence of an operator determines how it nests with other operators. All the tokens declared in a single precedence declaration have equal precedence and nest together according to their associativity. When two tokens declared in different precedence declarations associate, the one declared later has the higher precedence and is grouped first.

The Collection of Value Types

The `%union` declaration specifies the entire collection of possible data types for semantic values. The keyword `%union` is followed by a pair of braces containing the same thing that goes inside a `union` in C.

For example:

```
%union {
  double val;
  symrec *tptr;
}
```

This says that the two alternative types are `double` and `symrec *`. They are given names `val` and `tptr`; these names are used in the `%token` and `%type` declarations to pick one of the types for a terminal or nonterminal symbol (see section Nonterminal Symbols).

Note that, unlike making a `union` declaration in C, you do not write a semicolon after the closing brace.

Nonterminal Symbols

When you use `%union` to specify multiple value types, you must declare the value type of each nonterminal symbol for which values are used. This is done with a `%type` declaration, like this:

```
%type <type> nonterminal...
```

Here *nonterminal* is the name of a nonterminal symbol, and *type* is the name given in the `%union` to the alternative that you want (see section The Collection of Value Types). You can give any number of nonterminal symbols in the same `%type` declaration, if they have the same value type. Use spaces to separate the symbol names.

Suppressing Conflict Warnings ☆ intresting! Implement if you have time!

Bison normally warns if there are any conflicts in the grammar (see section Shift/Reduce Conflicts), but most real grammars have harmless shift/reduce conflicts which are resolved in a predictable way and would be difficult to eliminate. It is desirable to suppress the warning about these conflicts unless the number of conflicts changes. You can do this with the `%expect` declaration.

The declaration looks like this:

```
%expect n
```

Here *n* is a decimal integer. The declaration says there should be no warning if there are *n* shift/reduce conflicts and no reduce/reduce conflicts. The usual warning is given if there are either more or fewer conflicts, or if there are any reduce/reduce conflicts.

In general, using `%expect` involves these steps:

- Compile your grammar without `%expect`. Use the `` `-v' `` option to get a verbose list of where the conflicts occur. Bison will also print the number of conflicts.

- Check each of the conflicts to make sure that Bison's default resolution is what you really want. If not, rewrite the grammar and go back to the beginning.
- Add an `%expect` declaration, copying the number *n* from the number which Bison printed.

Now Bison will stop annoying you about the conflicts you have checked, but it will warn you again if changes in the grammar result in additional conflicts.

The Start-Symbol

Bison assumes by default that the start symbol for the grammar is the first nonterminal specified in the grammar specification section. The programmer may override this restriction with the `%start` declaration as follows:

```
%start symbol
```

A Pure (Reentrant) Parser

A *reentrant* program is one which does not alter in the course of execution; in other words, it consists entirely of *pure* (read-only) code. Reentrancy is important whenever asynchronous execution is possible; for example, a nonreentrant program may not be safe to call from a signal handler. In systems with multiple threads of control, a nonreentrant program must be called only within interlocks.

The Bison parser is not normally a reentrant program, because it uses statically allocated variables for communication with `yylex`. These variables include `yylval` and `yylloc`.

The Bison declaration `%pure_parser` says that you want the parser to be reentrant. It looks like this:

```
%pure_parser
```

The effect is that the two communication variables become local variables in `yyparse`, and a different calling convention is used for the lexical analyzer function `yylex`. See section Calling for Pure Parsers, for the details of this. The variable `yynerrs` also becomes local in `yyparse` (see section The Error Reporting Function `yyerror`). The convention for calling `yyparse` itself is unchanged.

Bison Declaration Summary

Here is a summary of all Bison declarations:

`%union`
: Declare the collection of data types that semantic values may have (see section The Collection of Value Types).

`%token`
: Declare a terminal symbol (token type name) with no precedence or associativity specified (see section Token Type Names).

`%right`
: Declare a terminal symbol (token type name) that is right-associative (see section Operator Precedence).

`%left`
: Declare a terminal symbol (token type name) that is left-associative (see section Operator Precedence).

`%nonassoc`
: Declare a terminal symbol (token type name) that is nonassociative (using it in a way that would be associative is a syntax error) (see section Operator Precedence).

`%type`
: Declare the type of semantic values for a nonterminal symbol (see section Nonterminal Symbols).

`%start`
: Specify the grammar's start symbol (see section The Start-Symbol).

`%expect`
: Declare the expected number of shift-reduce conflicts (see section Suppressing Conflict Warnings).

`%pure_parser`
: Request a pure (reentrant) parser program (see section A Pure (Reentrant) Parser).

Multiple Parsers in the Same Program

Most programs that use Bison parse only one language and therefore contain only one Bison parser. But what if you want to parse more than one language with the same program? Then you need to avoid a name conflict between different definitions of `yyparse`, `yylval`, and so on.

The easy way to do this is to use the option `` `-p *prefix*' `` (see section Invoking Bison). This renames the interface functions and variables of the Bison parser to start with *prefix* instead of `` `yy' ``. You can use this to give each parser distinct names that do not conflict.

The precise list of symbols renamed is `yyparse`, `yylex`, `yyerror`, `yylval`, `yychar` and `yydebug`. For example, if you use `` `-p c' ``, the names become `cparse`, `clex`, and so on.

All the other variables and macros associated with Bison are not renamed. These others are not global; there is no conflict if the same name is used in different parsers. For example, `YYSTYPE` is not renamed, but defining this in different ways in different parsers causes no trouble (see section Data Types of Semantic Values).

The `` `-p' `` option works by adding macro definitions to the beginning of the parser source file, defining `yyparse` as `*prefix*parse`, and so on. This effectively substitutes one name for the other in the entire parser file.

Go to the previous, next section.