Go to the previous, next section.

The Concepts of Bison

This chapter introduces many of the basic concepts without which the details of Bison will not make sense. If you do not already know how to use Bison or Yacc, we suggest you start by reading this chapter carefully.

Languages and Context-Free Grammars

In order for Bison to parse a language, it must be described by a *context-free grammar*. This means that you specify one or more *syntactic groupings* and give rules for constructing them from their parts. For example, in the C language, one kind of grouping is called an `expression'. One rule for making an expression might be, "An expression can be made of a minus sign and another expression". Another would be, "An expression can be an integer". As you can see, rules are often recursive, but there must be at least one rule which leads out of the recursion.

The most common formal system for presenting such rules for humans to read is *Backus-Naur Form* or "BNF", which was developed in order to specify the language Algol 60. Any grammar expressed in BNF is a context-free grammar. The input to Bison is essentially machine-readable BNF.

Not all context-free languages can be handled by Bison, only those that are LALR(1). In brief, this means that it must be possible to tell how to parse any portion of an input string with just a single token of look-ahead. Strictly speaking, that is a description of an LR(1) grammar, and LALR(1) involves additional restrictions that are hard to explain simply; but it is rare in actual practice to find an LR(1) grammar that fails to be LALR(1). See section Mysterious Reduce/Reduce Conflicts, for more information on this.

In the formal grammatical rules for a language, each kind of syntactic unit or grouping is named by a *symbol*. Those which are built by grouping smaller constructs according to grammatical rules are called *nonterminal symbols*; those which can't be subdivided are called *terminal symbols* or *token types*. We call a piece of input corresponding to a single terminal symbol a *token*, and a piece corresponding to a single nonterminal symbol a *grouping*.

We can use the C language as an example of what symbols, terminal and nonterminal, mean. The tokens of C are identifiers, constants (numeric and string), and the various keywords, arithmetic operators and punctuation marks. So the terminal symbols of a grammar for C include `identifier', `number', `string', plus one symbol for each keyword, operator or punctuation mark: `if', `return', `const', `static', `int', `char', `plus-sign', `open-brace', `close-brace', `comma' and many more. (These tokens can be subdivided into characters, but that is a matter of lexicography, not grammar.)

Here is a simple C function subdivided into tokens:

```
int             /* keyword `int' */
square (x)      /* identifier, open-paren, */
                /* identifier, close-paren */
     int x;     /* keyword `int', identifier, semicolon */
{               /* open-brace */
  return x * x; /* keyword `return', identifier, */
                /* asterisk, identifier, semicolon */
}               /* close-brace */
```

The syntactic groupings of C include the expression, the statement, the declaration, and the function definition. These are represented in the grammar of C by nonterminal symbols `expression', `statement', `declaration' and `function definition'. The full grammar uses dozens of additional language constructs, each with its own nonterminal symbol, in order to express the meanings of these four. The example above is a function definition; it contains one declaration, and one statement. In the statement, each `x' is an expression and so is `x * x'.

Each nonterminal symbol must have grammatical rules showing how it is made out of simpler constructs. For example, one kind of C statement is the `return` statement; this would be described with a grammar rule which reads informally as follows:

> A `statement' can be made of a `return' keyword, an `expression' and a `semicolon'.

There would be many other rules for `statement', one for each kind of statement in C.

One nonterminal symbol must be distinguished as the special one which defines a complete utterance in the language. It is called the *start symbol*. In a compiler, this means a complete input program. In the C language, the nonterminal symbol `sequence of definitions and declarations' plays this role.

For example, `` `1 + 2' `` is a valid C expression--a valid part of a C program--but it is not valid as an *entire* C program. In the context-free grammar of C, this follows from the fact that `expression' is not the start symbol.

The Bison parser reads a sequence of tokens as its input, and groups the tokens using the grammar rules. If the input is valid, the end result is that the entire token sequence reduces to a single grouping whose symbol is the grammar's start symbol. If we use a grammar for C, the entire input must be a `sequence of definitions and declarations'. If not, the parser reports a syntax error.

From Formal Rules to Bison Input

A formal grammar is a mathematical construct. To define the language for Bison, you must write a file expressing the grammar in Bison syntax: a *Bison grammar* file. See section Bison Grammar Files.

A nonterminal symbol in the formal grammar is represented in Bison input as an identifier, like an identifier in C. By convention, it should be in lower case, such as `expr`, `stmt` or `declaration`.

The Bison representation for a terminal symbol is also called a *token type*. Token types as well can be represented as C-like identifiers. By convention, these identifiers should be upper case to distinguish them from nonterminals: for example, `INTEGER`, `IDENTIFIER`, `IF` or `RETURN`. A terminal symbol that stands for a particular keyword in the language should be named after that keyword converted to upper case. The terminal symbol `error` is reserved for error recovery. See section Symbols, Terminal and Nonterminal.

A terminal symbol can also be represented as a character literal, just like a C character constant. You should do this whenever a token is just a single character (parenthesis, plus-sign, etc.): use that same character in a literal as the terminal symbol for that token.

The grammar rules also have an expression in Bison syntax. For example, here is the Bison rule for a C `return` statement. The semicolon in quotes is a literal character token, representing part of the C syntax for the statement; the naked semicolon, and the colon, are Bison punctuation used in every rule.

```
stmt:   RETURN expr ';'
        ;
```

See section Syntax of Grammar Rules.

Semantic Values

A formal grammar selects tokens only by their classifications: for example, if a rule mentions the terminal symbol `integer constant', it means that *any* integer constant is grammatically valid in that position. The precise value of the constant is irrelevant to how to parse the input: if `` `x+4' `` is grammatical then `` `x+1' `` or `` `x+3989' `` is equally grammatical.

But the precise value is very important for what the input means once it is parsed. A compiler is useless if it fails to distinguish between 4, 1 and 3989 as constants in the program! Therefore, each token in a Bison grammar has both a token type and a *semantic value*. See section Defining Language Semantics, for details.

The token type is a terminal symbol defined in the grammar, such as `INTEGER`, `IDENTIFIER` or `','`. It tells everything you need to know to decide where the token may validly appear and how to group it with other tokens. The grammar rules know nothing about tokens except their types.

The semantic value has all the rest of the information about the meaning of the token, such as the value of an integer, or the name of an identifier. (A token such as `','` which is just punctuation doesn't need to have any semantic value.)

For example, an input token might be classified as token type `INTEGER` and have the semantic value 4. Another input token might have the same token type `INTEGER` but value 3989. When a grammar rule says that `INTEGER` is allowed, either of these tokens is acceptable because each is an `INTEGER`. When the parser accepts the token, it keeps track of the token's semantic value.

Each grouping can also have a semantic value as well as its nonterminal symbol. For example, in a calculator, an expression typically has a semantic value that is a number. In a compiler for a programming language, an expression typically has a semantic value that is a tree structure describing the meaning of the expression.

Semantic Actions

In order to be useful, a program must do more than parse input; it must also produce some output based on the input. In a Bison grammar, a grammar rule can have an *action* made up of C statements. Each time the parser recognizes a match for that rule, the action is executed. See section Actions. Most of the time, the purpose of an action is to compute the semantic value of the whole construct from the semantic values of its parts. For example, suppose we have a rule which says an expression can be the sum of two expressions. When the parser recognizes such a sum, each of the subexpressions has a semantic value which describes how it was built up. The action for this rule should create a similar sort of value for the newly recognized larger expression.

For example, here is a rule that says an expression can be the sum of two subexpressions:

```
expr: expr '+' expr   { $$ = $1 + $3; }
        ;
```

The action says how to produce the semantic value of the sum expression from the values of the two subexpressions.

Bison Output: the Parser File

When you run Bison, you give it a Bison grammar file as input. The output is a C source file that parses the language described by the grammar. This file is called a *Bison parser*. Keep in mind that the Bison utility and the Bison parser are two distinct programs: the Bison utility is a program whose output is the Bison parser that becomes part of your program.

The job of the Bison parser is to group tokens into groupings according to the grammar rules--for example, to build identifiers and operators into expressions. As it does this, it runs the actions for the grammar rules it uses.

The tokens come from a function called the *lexical analyzer* that you must supply in some fashion (such as by writing it in C). The Bison parser calls the lexical analyzer each time it wants a new token. It doesn't know what is "inside" the tokens (though their semantic values may reflect this). Typically the lexical analyzer makes the tokens by parsing characters of text, but Bison does not depend on this. See section The Lexical Analyzer Function `yylex`.

The Bison parser file is C code which defines a function named `yyparse` which implements that grammar. This function does not make a complete C program: you must supply some additional functions. One is the lexical analyzer. Another is an error-reporting function which the parser calls to report an error. In addition, a complete C program must start with a function called `main`; you have to provide this, and arrange for it to call `yyparse` or the parser will never run. See section Parser C-Language Interface.

Aside from the token type names and the symbols in the actions you write, all variable and function names used in the Bison parser file begin with `yy' or `YY'. This includes interface functions such as the lexical analyzer function `yylex`, the error reporting function `yyerror` and the parser function `yyparse` itself. This also includes numerous identifiers used for internal purposes. Therefore, you should avoid using C identifiers starting with `yy' or `YY' in the Bison grammar file except for the ones defined in this manual.

Stages in Using Bison

The actual language-design process using Bison, from grammar specification to a working compiler or interpreter, has these parts:

1. Formally specify the grammar in a form recognized by Bison (see section Bison Grammar Files). For each grammatical rule in the language, describe the action that is to be taken when an instance of that rule is recognized. The action is described by a sequence of C statements.

2. Write a lexical analyzer to process input and pass tokens to the parser. The lexical analyzer may be written by hand in C (see section The Lexical Analyzer Function `yylex`). It could also be produced using Lex, but the use of Lex is not discussed in this manual.

3. Write a controlling function that calls the Bison-produced parser.

4. Write error-reporting routines.

To turn this source code as written into a runnable program, you must follow these steps:

1. Run Bison on the grammar to produce the parser.

2. Compile the code output by Bison, as well as any other source files.

3. Link the object files to produce the finished product.

The Overall Layout of a Bison Grammar

The input file for the Bison utility is a *Bison grammar file*. The general form of a Bison grammar file is as follows:

```
%{
C declarations
%}

Bison declarations

%%
Grammar rules
%%
Additional C code
```

The `%%', `%{' and `%}' are punctuation that appears in every Bison grammar file to separate the sections.

The C declarations may define types and variables used in the actions. You can also use preprocessor commands to define macros used there, and use `#include` to include header files that do any of these things.

The Bison declarations declare the names of the terminal and nonterminal symbols, and may also describe operator precedence and the data types of semantic values of various symbols.

The grammar rules define how to construct each nonterminal symbol from its parts.

The additional C code can contain any C code you want to use. Often the definition of the lexical analyzer `yylex` goes here, plus subroutines called by the actions in the grammar rules. In a simple program, all the rest

of the program can go here.

Go to the previous, next section.