Jim Lambers
CS143 Compilers
Summer Quarter 2000-01
Introduction to Bison

`bison` is a tool that generates an LR parser from the description of a grammar, and implements the parser in C. This tool is very similar to the well-known parser generator `yacc`.

1 Using Bison

To create a parser using `bison`, you create an input file in which you specify the grammar for the language you wish to parse, along with actions to be taken during parsing. Often, you will use `bison` with a lexical analyzer created using `lex` or `flex`. In this case, global variables and functions are used for communication between the parser and the lexical analyzer.

2 The `bison` input file

Your input file to `bison` is organized as follows:

```
%{
Declarations
%}
Definitions
%%
Productions
%%
Additional code
```

The Declarations section includes C++ declarations that are needed to compile any code that you add to this file. The Definitions section is used for a number of purposes, including, but not limited to, the following:

- defining token values that will be used by a lexical analyzer generated by `flex`, if applicable
- listing precedence and associativity rules for operators

- defining the type of the global attribute variable `yylval`
- specifying how `yylval` is to be interpreted when used with each terminal or non-terminal symbol

The Productions section is where the actual grammar is described to `bison`. Each production is associated with zero or more *actions*, similar to actions in `flex`. The code in each action is executed whenever a reduction using the given production takes place during parsing.

After the second `%%` delimiter, you may add any additional C++ code that you may need.

3 Grammar Symbols and Attributes

In your input file, you define both terminal and non-terminal symbols. Non-terminals are named like typical identifiers: using a sequence of letters and/or digits, starting with a letter. A terminal symbol may be specified using the `%token` directive in the Definitions section. The format is

`%token` $name_1$ $name_2$ ... $name_n$

This directive tells `bison` to define the given names as numeric constants. These values are to be returned by the lexical analyzer to inform the parser which terminal symbol has just been recognized in the input.

Both terminals and non-terminals can have attributes. For this reason, `bison` allows you to define a type for storing attribute values. The name `YYSTYPE` is defined to be the type of object in which attributes are stored. You may either `#define YYSTYPE` to be the type of your choice, or you may use the `%union` directive in the Definitions section to define the attribute type as a union:

`%union` {
    field-declarations
}

The type `YYSTYPE` is then defined to be this union. If you use the `%union` directive, then you must associate each grammar symbol with a field from this union, if you intend to use the symbol's attribute during parsing. To create such an association, you may use the `%type` directive for non-terminals,

or specify the field along with the token name when it is defined using the `%token` directive.

Specifically, you define tokens as follows:

`%token` $< field\_name > token\_name < field\_name > token\_name \cdots$

and the type of non-terminals as follows:

`%type`$< field\_name > symbol\_name\ symbol\_name \cdots$

The global variable `yylval` is declared to be of type `YYSTYPE`. This variable is used by the lexical analyzer to communicate attribute values for terminals back to the parser. The parser also stores objects of this type on the stack during parsing, so that the attribute of a non-terminal on the left side of a production may be determined using the attributes of the symbols on the right side during a reduction.

4 Specifying Productions

Productions and their accompanying actions are described to `bison` using the following format:

$$
\begin{array}{lcl}
left\_side & : & right\_side \ \{ \ action \ \} \\
 & | & right\_side \ \{ \ action \ \} \\
 & \vdots & \\
 & ; & \\
 & & \\
left\_side & : & right\_side \ \{ \ action \ \} \\
 & \vdots & \\
 & ; & \\
 & & \\
 & \vdots &
\end{array}
$$

The right side is a sequence of grammar symbols, separated by white space. Alternatives for the right side must be separated by the vertical bar

|, and the list of alternatives for a given left side must be terminated by a semicolon.

Non-terminals must be named like typical identifiers (sequences of letters and digits), while terminal symbols can either be token values defined using the `%token` directive, or single characters enclosed in single quotes. The non-terminal on the left side of the first production is assumed to be the start symbol of the grammar.

The format above shows actions occurring after the right side of a production. Actions may be added within right sides, but this causes the actions to be treated as grammar symbols, and forces the parser to commit to parsing a particular production early, much like a predictive parser. This can cause conflicts to be introduced into the parsing table. Another restriction is that the attribute of the left side of a production cannot be set within an action inserted in the middle of the right side.

5 Using Attributes within Actions

To access a grammar symbol's attribute within the code for an action, simply use `$`n, where n is the position of the symbol. The position of the non-terminal on the left side is denoted by another `$`. The position of a symbol on the right side is indicated by a number, with 1 denoting the first symbol on the right side, 2 denoting the second symbol, and so on.

If you have defined `YYSTYPE` as a predefined type, then an object denoted by `$`n or `$$` is of that type. Otherwise, if you have used the `%union` directive to define YYSTYPE, then the type of each grammar symbol is determined by the field within the union that you have previously assigned to the symbol in the Definitions section.

A notation similar to the use of the `$` can be used to obtain information about the location of symbols in the input. `bison` maintains a stack of objects of type `YYLTYPE`, which is a structure containing four members: `first_line`, `first_column`, `last_line`, and `last_column`. To obtain the location of a grammar symbol on the right side, you simply use the notation `@`n, where n once again denotes the position of the symbol within the right side. This notation denotes an object of type `YYLTYPE`. The location of a terminal symbol is furnished by the lexical analyzer via the global variable `yylloc`. During a reduction $A \rightarrow \alpha$, the location of the non-terminal A is derived from the location of all symbols in the handle α that is being reduced to A.

6 Precedence and Associativity

To resolve ambiguities without having to modify the grammar, `bison` provides various directives for specifying precedence and associativity among operators. The `%left`, `%right` and `%nonassoc` directives indicate that their arguments, which are terminal symbols, are operators with left, right or no associativity, respectively. These directives must be used with the Definitions section of the input file.

The order in which these directives are used indicate precedence. The first use of these directives is used for the operators of lowest precedence. For example, the convention in arithmetic is that addition and subtraction have lowest precedence, followed by multiplication and division, with exponentiation having the highest precedence. All of these operations are left associative, except for exponentiation, which is right associative. In bison, these rules would be specified as follows:

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

Note that all terminals listed with a single directive have equal precedence. Terminals may be specified using quoted characters or token names.

Another way to use precedence to resolve conflicts is by using the `%prec` directive. This directive is used at the end of the right side of a production, with a terminal symbol a as its only argument. The effect of this directive is to assign the right side the same precedence as a. As a result, given a choice to reduce using this right side or shift some symbol b onto the stack, the shift will be chosen if a has lower precedence than b, and the reduce will be chosen if a has higher precedence than b.

The terminal a can be a token that is never even returned by the lexical analyzer, in which case it is a terminal that denotes the empty string ϵ. Adding such a terminal can be helpful in disambiguating various constructs. Note that such "imaginary" terminal symbols do not need to be defined using the `%token` directive; they can be introduced using the `%left`, `%right`, or `%nonassoc` directives, which specify their precedence relative to actual tokens.

7 Bison Global Variables and Functions

The following global variables and/or functions are also of interest:

- `yyparse` is a function that serves as the driver for the parser. It reads a string from standard input and attempts to parse it. It returns 0 if the string is accepted, and 1 if a syntax error is detected.
- `yylex` serves as the lexical analyzer used by `yyparse` to recognize tokens in the input string. It returns an `int` denoting a particular token, which will be one of the constants defined using the `%token` directive or a `char` value denoting a single-character token.
- `yyerror` is a function called by `yyparse` if a syntax error is detected. You must supply the implementation of this function.
- `yydebug` is used to enable debugging. If it is set to a nonzero value and the preprocessor macro `YYDEBUG` is also defined to have a nonzero value, then `bison` will output debugging information as it parses. You must `#include <stdio.h>` when debugging since debug output is written to `stderr`.

8 Invoking `bison`

Once your input file is prepared, you may then run `bison` on your input file, using the command

`bison` [*-options*] *input_filename*

By default, `bison` will produce a file with the same name as your input file, except a `.tab.c` suffix, instead of the input file's suffix, which by convention is `.y`. `bison` will inform you of any syntax errors in your input file, as well as any conflicts arising in the parsing table generated from your grammar.

The following options may be useful to you:

- `-d` tells `bison` to generate a header file (with a `.tab.h` extension) containing the definitions of the token values, as well as the definition of `YYSTYPE` and a few other declarations. You must use this option if you define `yylex` in a separate source file, which is the case if you use `bison` in conjunction with a lexical analyzer generator.
- `-t` causes `YYDEBUG` to be defined, thus enabling debugging facilities.

- `-v` causes `bison` to generate an additional output file, with a `.output` extension, containing verbose descriptions of the parsing states and actions. This is especially useful in debugging conflicts.
- `-y` specifies that output files are to be named using the `yacc` convention, using `y` as the base name. Thus the output file will be named `y.tab.c`, the header file `y.tab.h`, and the verbose output file `y.output`.

9 A Sample `bison` File

A `bison` file that recognizes the grammar from the simple arithmetic expression language presented in the first lecture is available in the course directory, in the directory

```
/usr/class/cs143/sample/bison
```

This directory contains the `bison` input file `compiler.y`, an accompanying `flex` input file `compiler.l` (similar to the `flex` sample posted earlier), a driver program `main.cc`, a header file `compiler.h` with common declarations, and a `Makefile`.

To compile the parser defined in these files, execute the `make` command. Observe the commands executed to create the final executable, which is named `compiler`.

This program, when executed, will read one expression from standard input. If it is a valid expression, the program will evaluate it and display the result on standard output. Otherwise, it will indicate that there is a syntax error and then exit.

10 Bison Resources

The manual page on `bison` provides detailed information about the invocation of bison, including descriptions of all command-line options. You can also learn more about `bison` on the GNU web site at

```
http://www.gnu.org/software/bison/
```