**<http://www.cs.ecu.edu/karl/5220/spr16/Notes/Attribute/index.html>**

## 10.1. Bison

Bison is a parser generator that creates LALR(1) parsers.

(LALR(1) uses the same LR(0) state machine as SLR(1), but it is able to remove some parsing conflicts that the SLR(1) algorithm creates unnecessarily.)

Bison is an update of an earlier parser generator called Yacc, which is described in the Dragon Book.

**10.2. Bison File Format**

A Bison file has a similar form to a Flex file.

**%{**

***What is written here is copied***

***verbatim to the beginning of***

***file y.tab.c, the C file that***

***Bison generates.***

**%}**

***Here, you define tokens, precedence***

***of tokens and types of attributes.***

**%%**

***Here, you write the grammar.***

**%%**

***What is written here is copied***

***verbatim to the end of file y.tab.c.***

**Defining tokens and precedence**

Bison recognizes character constants, such as '=', as tokens. But they must be a *single character*. You need to tell Bison about other tokens.

Line

**%token TOK\_ARROW**

tells Bison that TOK\_ARROW is a token.

**Writing the grammar**

Write the grammar between the two lines that have only %% on them.

Use a colon instead of → in a production. After the first production in a group, a | introduces another right-hand side with the same left-hand side as before. For example,

**expression : expression '+' expression**

**| expression '\*' expression**

**| TOK\_NUMBER**

**;**

describes three productions

|  |  |  |
| --- | --- | --- |
| expression | → | expression **+** expression |
| expression | → | expression **\*** expression |
| expression | → | TOK\_NUMBER |

Note the semicolon at the end of the group of productions. ***It is required.***

Programming languages can have large grammars, with a lot of nonterminals. It is a good idea to use descriptive names for nonterminals rather than single-character names.

**ε-productions**

To write an ε-production, write nothing at all after : or |. But you probably want to write a comment (ignored by Bison). For example,

**StatementList : /\* empty \*/**

**| …**

creates an erasing production for StatementList.

**10.3. Running Bison**

Command

**bison -y -v -d parser.y**

runs Bison on file parser.y. Option -y tells it to do things the same way that Yacc does, so that what you read in the Dragon Book will be correct.

When run as it is above, Bison writes three files.

1. File **y.tab.c** contains the parsing tables and a definition of function yyparse, with heading
2. **void yyparse(void)**

Calling yyparse will run the parser. It assumes that the lexer is called yylex and that yylex puts token attributes into variable yylval, of type YYSTYPE.

1. Option -d asks Bison to write a file **y.tab.h** that contains definitions of tokens.

For example, if you have a token called TOK\_ARROW, then Bison chooses a number for that token and defines TOK\_ARROW in y.tab.h.

You will want to make your lexer include y.tab.h so that it gets the correct numbers for tokens. You no longer need token.h.

1. Option -v asks Bison to write file **y.output**. It contains the information about the LR(0) finite state machine that Bison created, and shows where conflicts are, if there are any.

**Debug mode for yyparse**

You can ask yyparse to show you what it is doing as follows.

1. In the section %{ … %} at the beginning of the parser file, write
2. **#define YYDEBUG 1**

to get debugging code to be compiled into yyparse.

1. Set global variable yydebug to 1 to turn on debugging. You will need to include line
2. **extern int yydebug;**

to set yydebug in a module other than the parser module.

Bison will write a trace to the standard output. It might take a bit of work to figure out exactly what it is saying, but remember, from the point of view of a bottom-up parser, productions are used backwards, and that is how Bison shows them. Ignore the $ signs.

**10.4. Precedence and Associativity**

**Precedence and associativity of tokens**

Use %left, %right and %nonassoc instead of %token to declare precedence and associativity of tokens. Lines

**%left TOK\_FROG**

**%right TOK\_TOAD**

**%nonassoc TOK\_SALAMANDER**

tell Bison that

* TOK\_FROG is left-associative,
* TOK\_TOAD is right-associative and
* TOK\_SALAMENDER has no associativity, meaning its precedence should not be allowed to be compared to itself.

Precedence is from low to high, in the order written. So the above lines indicate that TOK\_SALAMANDER has highest precedence, then TOK\_TOAD, and TOK\_FROG has the lowest precedence.

You can declare a few tokens to have the same precedence and associativity. For example,

**%left '+' '-'**

defines '+' and '-' to have the same precedence, and left-associativity.

**Precedence and associativity of productions**

By default, the precedence of a production is the same as the precedence of the first token on its right-hand side, if there is one.

If you want to choose your own precedence, follow the production by

**%prec *t***

where *t* is a token. For example,

**expression : expression '-' expression**

**%prec '+'**

defines a production whose precedence and associativity are the same as for token '+'.

**10.5. Actions**

After a production, you can write an action that is performed when the parser does a reduce by that production. Write it in C, surrounded by braces. For example,

**expression : expression '+' expression**

**{printf("Reducing by production E -> E + E\n");**

**}**

**Getting token attributes in actions**

Type YYSTYPE is a union type. Suppose that it is defined by

**typedef union**

**{**

**int ival;**

**const char\* strval;**

**} YYSTYPE;**

You will want to tell which field of the union is used for each token. Lines

**%token <ival> TOK\_NUMBER**

**%token <strval> TOK\_IDENTIFIER**

tell Bison that TOK\_NUMBER uses field ival and TOK\_IDENTIFIER uses field strval.

In an action, refer to a token by its position on the right-hand side of the production, numbering from 1. The attribute of the first thing is $1, of the second thing is $2, etc.

For example, in

**TOK\_IDENTIFIER expression : TOK\_LET**

**TOK\_IN expression '='**

**TOK\_END expression**

**{const char\* id = $2;**

**printf("I see a let of %s\n", id);**

**}**

the action shows the attribute of the TOK\_IDENTIFIER token.

Notice that you don't explicitly write **.strval**. Bison adds that automatically.

## 10.6 Embedded Actions

There are times when you want to perform an action before you get to the end of the right-hand side of a production.

You can do that by creating a nonterminal that has only one production, an erasing production, and attaching the action to the end of that production.

For example, you can set variables that indicate context, or you can perform initializations.

**expr : TOK\_FOO initForFoo**

**expr TOK\_ENDFOO**

**{**

**printf("Foo %d\n", $4);**

**insideFoo = 0;**

**}**

**…**

**;**

**initForFoo : /\* empty \*.**

**{**

**insideFoo = 1;**

**}**

**;**

That kind of thing is sufficiently common that Bison has a convenient way to do it. Just write an action in place of a token or nonterminal. So you can write

**expr : TOK\_FOO**

**{**

**insideFoo = 1;**

**}**

**expr TOK\_ENDFOO**

**{**

**printf("Foo %d\n", $4);**

**insideFoo = 0;**

**}**

**…**

**;**

Bison replaces the extra action by a new nonterminal and creates an erasing production for it. Consequently, the extra action counts as one of the things on the right-hand side of the production.

So TOK\_ENDFOO is still $4. $2 refers to the second thing, which is the automatically create nonterminal.

**10.7. Panic-Mode Error Recovery**

Bison supports an error recovery mode called *panic-mode* error recovery.

There is a special token, **error**, that is used in recovery from a syntax error. Use it in a production to give the parser an error recovery point.

For example, your grammar might say

**definition : TOK\_DEF idlist '=' expr TOK\_END**

**| error TOK\_END**

**;**

In the event of an error, the parser starts a panic.

1. When a syntax error occurs, yyparse calls yyerror("syntax error").
2. Next, the parser begins popping its stack until it encounters a state that contains an LR(0) item of the form *A* → **⋅** **error** α.
3. If α is an empty string,
   1. yyparse reduces by production *A* → **error**, and performs the action associated with that production.
   2. yyparse throws away tokens in the input until it finds one on which normal parsing can resume.
   3. yyparse resumes parsing in provisional mode (see item 5).
4. If α is not empty, then yyparse begins throwing away tokens until it finds a sequence of tokens that can match α.

Normally, α is a sequence of tokens. After it has shifted α, yyparse reduces by production *A* → **error** α and resumes parsing in provisional mode (see item 5).

1. After error recovery, yyparse normally stays in a provisional error mode until it has successfully shifted three tokens.

If another syntax error occurs before three successful shifts, yyparse does not call yyerror(). Instead, it just starts another panic. That avoids lots of syntax error reports as the parser recovers.

Within an action you can use **yyerrok;** to exit provisional error mode and resume normal mode. For example:

**definition : error TOK\_END**

**{yyerrok;**

**}**

But only do that if you are sure that the parser has fully recovered.