Chapter 9. Advanced Flex and Bison

Bison was originally a version of yacc, the original Unix parser generator that generated LALR parsers in C. In recent years it's grown a lot of new features. We discuss some of the most useful ones here.

Pure Scanners and Parsers

A flex scanner and bison parser built in the usual way is not reentrant and can parse only one input stream at a time. That's because both the scanner and the parser use static data structures to keep track of what they're doing and to communicate with each other and with the calling program. Both flex and bison can create "pure" reentrant code, which replaces the static data structures with one passed as an argument to each routine in the scanner and parser. This both allows recursive calls to the scanner and parser, which is occasionally useful, and allows scanners and parsers to be used in multithreaded programs where there may be several parses going on at once in different threads.

As a demonstration, we'll take the calculator from Chapter 3 and modify it to use a pure scanner and parser. Rather than having the parser execute each line of code immediately, it'll return the AST to the caller. As is usual in reentrant programs, the calling routine allocates a structure with space for the per-instance data and passes it along in each call to the scanner and parser.

Unfortunately, as of the time this book went to press (mid-2009), the code for flex pure scanners and yacc pure scanners is a mess. Bison's calling sequence for a pure `yylex()` is different from flex's, and the way they handle per-instance data is different. It's possible to paper over the problems in the existing code and persuade pure scanners and parsers to work together, which is what we will do in this chapter, but before doing so, check the latest flex and bison documentation. Most of the incompatibilities could be fixed by relatively simple changes to the code skeletons used to create scanners and parsers, and with any luck someone will have done it so you don't have to do it.

Pure Scanners in Flex

A single scanning job may involve many calls to `yylex()` because it returns tokens to the calling program. Since the scanner's state has to be saved between calls, you have to manage the per-scanner data yourself. Flex provides routines that create and destroy a scanner's context, as well as routines to access scanner values that used to be in static variables like `yyin` and `yytext` to allow routines outside `yylex()` to get and set them.

```
yyscan_t scaninfo;          a pointer to the per-instance scanner data

int yylex_init(&scaninfo);      create a scanner
int yylex_init_extra(userstuff, &scaninfo); or create a scanner with a pointer to user data

yyset_in(stdin, scaninfo); set the input file and other parameters

while( ... ) {
   tok = yylex(scaninfo);   call until done
}

yylex_destroy(scaninfo);    free the scanner data
```

The `yyscan_t` structure contains all of the per-scanner state such as the input and output files and pointers to remember where in the buffered input to resume scanning. It also includes a stack of pointers to `YY_BUFFER_STATE` structures to track the active input buffer. (As we saw in Chapter 2, the built-in buffer stack isn't too useful since you usually need to remember extra per-buffer information.)

The `userstuff` argument to `yylex_init_extra` allows you to provide your own per-instance data to the scanner, such as the address of the symbol table for it to use. It is a value of type `YY_EXTRA_TYPE`, by default defined to be `void *` but easily overridden with `%option extra-type`. The per-instance data is invariably stored in a structure, so the `userstuff` is a pointer to that structure. As we'll see in a moment, in one line you can retrieve it within the scanner and put it in a pointer variable with a reasonable name and type.

Within the scanner, flex defines macros for `yytext`, `yyleng`, and a few other fields that refer to the instance data. The values of `yylineno` and `yycolumn`, which is a variable not present in nonreentrant scanners, are stored in the current buffer structure, making it easier to track line and column information in multiple input files, while the rest are in the `yyscan_t` structure. Flex maintains `yylineno` as it does in nonreentrant scanners, but the only thing it does automatically to `yycolumn` is set it to zero when it sees a `\n` character, so you still have to track the column yourself using the techniques in Chapter 8.

Example 9-1 shows the word count program from Chapter 2, modified to use a pure scanner.

Example 9-1. Pure version of word count program

```
/* pure version of word count program */
%option noyywrap nodefault reentrant
%{
struct pwc {    our per-scanner data
    int chars;
    int words;
    int lines;
```

```
};
%}
%option extra-type="struct pwc *"

%%
%{                              this code goes at the top of yylex
  struct pwc *pp = yyextra;     yyextra is a flex-defined macro
%}

[a-zA-Z]+       { pp->words++; pp->chars += strlen(yytext); }
\n              { pp->chars++; pp->lines++; }
.               { pp->chars++; }

%%
```

The three variables to count characters, words, and lines are now placed in a structure, with a copy allocated for each instance of the scanner and with `%option extra-type` making the `userstuff` in flex's scanner a pointer to that structure.

The first thing in the rules section is a line of code that flex will place at the top of `yylex`, after its own variable definitions but before any executable code. This line lets us declare a pointer to our own instance data and initialize it to `yyextra`, which is a macro provided by flex that refers to the extra data field in the scanner's current per-instance data. In the rules themselves, what were references to static variables are now references to our instance data.

```
main(argc, argv)
int argc;
char **argv;
{
  struct pwc mypwc = { 0, 0, 0 }; /* my instance data */
  yyscan_t scanner;               /* flex instance data */

  if(yylex_init_extra(&mypwc, &scanner)) {
    perror("init alloc failed");
    return 1;
  }

  if(argc > 1) {
    FILE *f;

    if(!(f = fopen(argv[1], "r"))) {
      perror(argv[1]);
      return (1);
    }
    yyset_in(f, scanner);
  } else
    yyset_in(stdin, scanner);

  yylex(scanner);
  printf("%8d%8d%8d\n", mypwc.lines, mypwc.words, mypwc.chars);

  if(argc > 1)
```

```
    fclose(yyget_in(scanner));

  yylex_destroy( scanner );
}
```

The main routine declares and initializes `mypwc`, our own instance data, and declares `scanner`, which will be the flex instance data. The call to `yylex_init_extra` takes a pointer to `scanner`, so it can fill it in with a pointer to the newly allocated instance, and the call returns 0 for success or 1 for failure (the `malloc` for the instance data failed).

If there's a file argument, we open the file and use `yyset_in` to store it in the `yyin`-ish field in the scanner data. Then we call `yylex`, passing it the flex instance data; print out the results that the scanner stored in our own instance data; and then free and deallocate the scanner.

This was more work than a regular nonpure scanner, but the changes were for the most part mechanical: move static data into a structure, change references to the static data to references to the structure, and add the code to create and destroy the scanner instance.

Pure Parsers in Bison

Pure parsers are a little easier to create than pure scanners, because an entire bison parse happens in a single call to `yyparse`. Hence, the parser can create its per-instance data at the start of the parse, do the parsing work, and then free it without needing explicit programmer help. The parser does typically need some application instance data, passed as an argument to `yyparse`. The static variables used to communicate with the scanner— `yylval` and, if the parser uses locations, `yyloc` —become instance variables that have to be passed to `yylex`, probably along with the application instance data.

Bison will create a pure parser if it sees the `%define api.pure` (formerly `%pure_parser`) declaration. This declaration makes the parser reentrant. To get a pure parser started, you pass in a pointer to the application per-instance data. The contents of the `%parse-param` declaration are placed between the parentheses in the definition of `yyparse()`, so you can declare as many arguments as you want, although one pointer to the per-instance data is usually all you need:

```
%define api.pure
%parse-param { struct pureparse *pp }
```

Pure parsers also change the calling sequence to `yylex()`, passing as arguments pointers to the current copies of `yylval` and, if locations are in use, `yylloc`.

```
/* generated calls within the parser */
token = yylex(YYSTYPE *yylvalp);                    without locations
```

```
token = yylex(YYSTYPE *yylvalp, YYLTYPE *yylocp); with them
```

If you want to pass application data, you can declare it with `%lex-param{ }` or by `#define YYLEX_PARAM`. An ill-advised overoptimization strips the argument to `%lex-param` to the last token in the braces, so define `YYLEX_PARAM` instead. (The documentation shows that the implementer assumed you'd put a declaration there, as in `parse-param`, but since the argument for a flex scanner has to be the scanner's `yyscan_t`, I always fetch it from in a field in the application per-instance data.)

```
%code {
#define YYLEX_PARAM pp->scaninfo
%}
%%
   /* generated calls within the parser */
   token = yylex(YYSTYPE *yylvalp, pp->scaninfo);   without locations
   token = yylex(YYSTYPE *yylvalp, YYLTYPE *yylocp, pp->scaninfo); with them
```

When the generated parser encounters a syntax error, it calls `yyerror()`, passing a pointer to the current location, if the parser uses locations, and the parser parameter in addition to the usual error message string: [20]

```
yyerror(struct pureparse *pp, "Syntax error");       without locations
yyerror(YYLTYPE &yyllocp,struct pureparse *pp, "Syntax error"); with them
```

If you're using a handwritten scanner, these are all the hooks you need for a pure parser. When you call internal routines from the scanner and parser, you'll want to pass along the instance data, as we'll see later in this chapter in the reentrant calculator. But since we're using a flex scanner, first we have to deal with flex and bison's incompatible calling sequence.

Using Pure Scanners and Parsers Together

If you compare the calling sequence for `yylex` in pure scanners and pure parsers, you'll note that they're incompatible. Flex wants the first argument to be the scanner instance data, but bison makes the first argument a pointer to `yylval`. While it is possible to use some undocumented C preprocessor symbols to fudge this, the maintainer of flex took pity on programmers and added the `bison-bridge` option to make its pure calling sequence compatible with bison's. If you use `%option bison-bridge`, the declaration of yylex becomes the following:

```
int yylex(YYSTYPE* lvalp, yyscan_t scaninfo);
```

If you use `%option bison-bridge bison-locations`, the declaration is as follows:

```
int yylex (YYSTYPE* lvalp, YYLTYPE* llocp, yyscan_t scaninfo);
```

Flex defines the macros `yylval` and (optionally) `yylloc`, which are copies of the arguments, but they are both pointers to the bison value union and location structure, so `yylval.field` and `yylloc.first_line` have to become `yylval->field` and `yylloc->first_line`. This is also a bug, but it's documented in the flex and bison manuals, so it is unlikely to change.

MULTIPLE INSTANCES OF MULTIPLE SCANNERS AND PARSERS

The discussion of pure parsing in this chapter covers the way you can have several copies of a scanner or parser, all that use the same set of rules. It's also possible to have several instances of different scanners or parsers in a single program. The sections Multiple Lexers in One Program and Variant and Multiple Grammars describe how to put several different scanners parsers into a single program, renaming their symbols with a prefix other than "yy". You can use the same features to rename pure scanners and parsers and then call them as needed. This level of complexity will be a challenge to debug, but the features are there if you need them.

A Reentrant Calculator

To make the calculator from Chapter 3 reentrant, most of the changes are mechanical, putting static data into per-instance structures. Rather than executing the parsed ASTs within the parser, this version parses one expression or function definition at a time, and it returns the ASTs to the caller, where they can be run immediately or saved for later. The scanner, on the other hand, is managed as a single session, used for all the calls to `yyparse`, so that there's no problem of losing buffered input each time the parser restarts. This means that the program creates the scanner context when it starts, and then it passes the same context to the parser each time. Example 9-2 shows the modified header file for the calculator.

Example 9-2. Reentrant calc header file purecalc.h

```
/*
 * Declarations for a calculator, pure version
 */

/* per-parse data */
struct pcdata {
  yyscan_t scaninfo;            /* scanner context */
  struct symbol *symtab;        /* symbols for this parse */
  struct ast *ast;              /* most recently parsed AST */
};
```

The new structure `pcdata` contains the application context for the parser. It points to the symbol table, allowing different parses to have dif-

ferent namespaces; the scanner context that the parser passes to `yylex`; and a place for the parser to return the AST that it parsed. (Remember that the value of `yyparse` is 1 or 0 to report whether the parse succeeded, so it can't directly return the AST.)

The changes in the rest of the header add an initial context argument to all of the functions, both the ones specific to the calculator and `yyerror`.

```
/* symbol table */
struct symbol {         /* a variable name */
  char *name;
  double value;
  struct ast *func;     /* AST for the function */
  struct symlist *syms; /* list of dummy args */
};

/* simple symtab of fixed size */
#define NHASH 9997
struct symbol symtab[NHASH];

struct symbol *lookup(struct pcdata *, char*);

/* list of symbols, for an argument list */
struct symlist {
  struct symbol *sym;
  struct symlist *next;
};

struct symlist *newsymlist(struct pcdata *, struct symbol *sym, struct symlist *n
void symlistfree(struct pcdata *, struct symlist *sl);

/* node types
 *  + - * / |
 *  0-7 comparison ops, bit coded 04 equal, 02 less, 01 greater
 *  M unary minus
 *  L statement list
 *  I IF statement
 *  W WHILE statement
 *  N symbol ref
 *  = assignment
 *  S list of symbols
 *  F built in function call
 *  C user function call
 */

enum bifs {                     /* built-in functions */
  B_sqrt = 1,
  B_exp,
  B_log,
  B_print
};

/* nodes in the abstract syntax tree */
/* all have common initial nodetype */
```

... all nodes unchanged from the original version ...

```
/* build an AST */
struct ast *newast(struct pcdata *, int nodetype, struct ast *l, struct ast *r);
struct ast *newcmp(struct pcdata *, int cmptype, struct ast *l, struct ast *r);
struct ast *newfunc(struct pcdata *, int functype, struct ast *l);
struct ast *newcall(struct pcdata *, struct symbol *s, struct ast *l);
struct ast *newref(struct pcdata *, struct symbol *s);
struct ast *newasgn(struct pcdata *, struct symbol *s, struct ast *v);
struct ast *newnum(struct pcdata *, double d);
struct ast *newflow(struct pcdata *, int nodetype, struct ast *cond, struct ast *
            struct ast *tr);

/* define a function */
void dodef(struct pcdata *, struct symbol *name, struct symlist *syms, struct ast

/* evaluate an AST */
double eval(struct pcdata *, struct ast *);

/* delete and free an AST */
void treefree(struct pcdata *, struct ast *);

/* interface to the scanner */
void yyerror(struct pcdata *pp, char *s, ...);
```

The scanner has the `reentrant` and `bison-bridge` options to make a reentrant bison-compatible scanner. For the first time we also tell flex to create a header file analogous to the one that bison creates. The file contains declarations of the various routines used to get and set variables in a scanner context, as well as the definition of `yyscan_t` that the parser will need.

DON'T INCLUDE THE SCANNER HEADER IN YOUR SCANNER!

If you create a scanner header file such as `purecalc.lex.h` in this example, be sure *not* to include the header directly or indirectly into the scanner itself. For some reason, the header has `#undef`s for several internal scanner macros, which will cause attempts to compile the scanner to fail with cryptic error messages about undefined variables. (Guess how I found this out.)

Either protect the `#include` statements with `#ifndef`/`#define` lines if you include the scanner header file in a common header file or do what this example does and include it directly only in the files that need it.

Example 9-3. Reentrant calculator scanner purecalc.l

```
/* recognize tokens for the calculator */
/* pure scanner and parser version */
/* $Header: /usr/home/johnl/flnb/RCS/ch09.tr,v 1.4 2009/05/19 18:28:27 johnl Exp
%option noyywrap nodefault yylineno reentrant bison-bridge

%option header-file="purecalc.lex.h"
%option extra-type="struct pcdata*"
```

```
%{
#include "purecalc.tab.h"
#include "purecalc.h"
%}

/* float exponent */
EXP     ([Ee][-+]?[0-9]+)

%%
%{
  struct pcdata *pp = yyextra;
%}
 /* single character ops */
"+" |
"-" |
"*" |
"/" |
"=" |
"|" |
"," |
";" |
"(" |
")"     { return yytext[0]; }

 /* comparison ops */
">"     { yylval->fn = 1; return CMP; }
"<"     { yylval->fn = 2; return CMP; }
"<>"    { yylval->fn = 3; return CMP; }
"=="    { yylval->fn = 4; return CMP; }
">="    { yylval->fn = 5; return CMP; }
"<="    { yylval->fn = 6; return CMP; }

 /* keywords */

"if"    { return IF; }
"then"  { return THEN; }
"else"  { return ELSE; }
"while" { return WHILE; }
"do"    { return DO; }
"let"   { return LET;}

 /* built-in functions */
"sqrt"  { yylval->fn = B_sqrt; return FUNC; }
"exp"   { yylval->fn = B_exp; return FUNC; }
"log"   { yylval->fn = B_log; return FUNC; }
"print" { yylval->fn = B_print; return FUNC; }

 /* names */
[a-zA-Z][a-zA-Z0-9]*  { yylval->s = lookup(pp, yytext); return NAME; }

[0-9]+"."[0-9]*{EXP}? |
"."?[0-9]+{EXP}? { yylval->d = atof(yytext); return NUMBER; }

"//".*
[ \t]   /* ignore whitespace */
\\n    printf("c> "); /* ignore line continuation */
"\n"    { return EOL; }
```

```
.       { yyerror(pp, "Mystery character %c\n", *yytext); }

<<EOF>>  { exit(0); }
%%
```

Example 9-3 shows the scanner with modifications to make it reentrant. A line of code at the top of the rules puts the pointer to the application instance data accessed via macro `yyextra` into variable `pp`, which is of the right type in case we need to access fields in it. The references to `yylval` are adjusted to use it as a pointer, and the call to `yyerror` passes the instance data.

At the end of the lexer is an `<<EOF>>` rule that just exits. This is not a particularly elegant way to end the program, but for our purposes it will do. Possible alternative approaches to ending the program are discussed later.

Example 9-4 shows the parser, modified to be reentrant. It has two kinds of changes. Some of them are the mechanical changes to handle explicit state; the rest change the parser to handle one statement at a time.

Example 9-4. Reentrant calculator parser purecalc.y

```
/* calculator with AST */
%define api.pure
%parse-param { struct pcdata *pp }

%{
#  include <stdio.h>
#  include <stdlib.h>
%}

%union {
  struct ast *a;
  double d;
  struct symbol *s;             /* which symbol */
  struct symlist *sl;
  int fn;                       /* which function */
}

%{
#  include "purecalc.lex.h"
#  include "purecalc.h"
#define YYLEX_PARAM pp->scaninfo
%}

/* declare tokens */
%token <d> NUMBER
%token <s> NAME
%token <fn> FUNC
%token EOL

%token IF THEN ELSE WHILE DO LET
```

```
%nonassoc <fn> CMP
%right '='
%left '+' '-'
%left '*' '/'
%nonassoc '|' UMINUS

%type <a> exp stmt list explist
%type <sl> symlist

%start calc
%%
```

The parser file defines `api.pure` to generate a reentrant parser and uses `parse-param` to declare that the parser now takes an argument, which is the pointer to the application state. A few lines further down, a code block includes `purecalc.lex.h`, which is the header generated by flex, and defines `YYLEX_PARAM` to pass the scanner state, which is stored in the instance state, to the scanner.

```
calc: /* nothing */ EOL { pp->ast = NULL; YYACCEPT; }
   | stmt EOL { pp->ast = $1; YYACCEPT; }
   | LET NAME '(' symlist ')' '=' list EOL {
                        dodef(pp, $2, $4, $7);
                        printf("%d: Defined %s\n", yyget_lineno(pp->scaninfo),
                               $2->name);
                        pp->ast = NULL; YYACCEPT; }
 ;
```

The top-level rule is now `calc`, which handles an empty line, a statement that is parsed into an AST, or a function definition that is stored in the local symbol table. Normally a bison parser reads a token stream up to the end-of file-token. This parser uses `YYACCEPT` to end the parse. When the parser ends, it leaves the scanner's state unchanged, so the next time the parser starts up, using the same scanner state, it resumes reading where the previous parse left off. An alternate approach would be to have the lexer return an end-of-file token when the user types a newline, which would also work; in a situation like this, there's no strong reason to prefer one approach or the other.

The rest of the parser is the same as the nonreentrant version, except that every call to an external routine now passes the pointer to the instance state. As we will see, many of the routines don't do anything with the state variable, but it's easier to pass it to all of them than to try to remember which ones need it and which ones don't.

```
stmt: IF exp THEN list           { $$ = newflow(pp, 'I', $2, $4, NULL); }
   | IF exp THEN list ELSE list  { $$ = newflow(pp, 'I', $2, $4, $6); }
   | WHILE exp DO list           { $$ = newflow(pp, 'W', $2, $4, NULL); }
   | exp
;
```

```
list: /* nothing */ { $$ = NULL; }
   | stmt ';' list { if ($3 == NULL)
                       $$ = $1;
                     else
                       $$ = newast(pp, 'L', $1, $3);
                   }
   ;

exp: exp CMP exp          { $$ = newcmp(pp, $2, $1, $3); }
   | exp '+' exp          { $$ = newast(pp, '+', $1,$3); }
   | exp '-' exp          { $$ = newast(pp, '-', $1,$3);}
   | exp '*' exp          { $$ = newast(pp, '*', $1,$3); }
   | exp '/' exp          { $$ = newast(pp, '/', $1,$3); }
   | '|' exp              { $$ = newast(pp, '|', $2, NULL); }
   | '(' exp ')'          { $$ = $2; }
   | '-' exp %prec UMINUS { $$ = newast(pp, 'M', $2, NULL); }
   | NUMBER               { $$ = newnum(pp, $1); }
   | FUNC '(' explist ')' { $$ = newfunc(pp, $1, $3); }
   | NAME                 { $$ = newref(pp, $1); }
   | NAME '=' exp         { $$ = newasgn(pp, $1, $3); }
   | NAME '(' explist ')' { $$ = newcall(pp, $1, $3); }
;

explist: exp
 | exp ',' explist  { $$ = newast(pp, 'L', $1, $3); }
;
symlist: NAME       { $$ = newsymlist(pp, $1, NULL); }
 | NAME ',' symlist { $$ = newsymlist(pp, $1, $3); }
;
%%
```

Example 9-5 shows the helper functions, adjusted for a reentrant scanner and parser. There is now a symbol table per instance state, so the routines that do symbol table lookups need to get the symbol table pointer from the state structure.

Example 9-5. Helper functions purecalcfuncs.c

```
/*
 * helper functions for purecalc
 */
#  include <stdio.h>
#  include <stdlib.h>
#  include <stdarg.h>
#  include <string.h>
#  include <math.h>

#  include "purecalc.tab.h"
#  include "purecalc.lex.h"
#  include "purecalc.h"

/* symbol table */
/* hash a symbol */
static unsigned
symhash(char *sym)
{
```

```
  unsigned int hash = 0;
  unsigned c;

  while(c = *sym++) hash = hash*9 ^ c;

  return hash;
}

struct symbol *
lookup(struct pcdata *pp, char* sym)
{
  struct symbol *sp = &(pp->symtab)[symhash(sym)%NHASH];
  int scount = NHASH;           /* how many have we looked at */

  while(--scount >= 0) {
    if(sp->name && !strcmp(sp->name, sym)) { return sp; }

    if(!sp->name) {             /* new entry */
      sp->name = strdup(sym);
      sp->value = 0;
      sp->func = NULL;
      sp->syms = NULL;
      return sp;
    }

    if(++sp >= pp->symtab+NHASH) sp = pp->symtab; /* try the next entry */
  }
  yyerror(pp, "symbol table overflow\n");
  abort(); /* tried them all, table is full */

}

struct ast *
newast(struct pcdata *pp, int nodetype, struct ast *l, struct ast *r)
{
  struct ast *a = malloc(sizeof(struct ast));

  if(!a) {
    yyerror(pp, "out of space");
    exit(0);
  }
  a->nodetype = nodetype;
  a->l = l;
  a->r = r;
  return a;
}

struct ast *
newnum(struct pcdata *pp, double d)
{
  struct numval *a = malloc(sizeof(struct numval));

  if(!a) {
    yyerror(pp, "out of space");
    exit(0);
  }
  a->nodetype = 'K';
```

```
  a->number = d;
  return (struct ast *)a;
}

struct ast *
newcmp(struct pcdata *pp, int cmptype, struct ast *l, struct ast *r)
{
  struct ast *a = malloc(sizeof(struct ast));

  if(!a) {
    yyerror(pp, "out of space");
    exit(0);
  }
  a->nodetype = '0' + cmptype;
  a->l = l;
  a->r = r;
  return a;
}

struct ast *
newfunc(struct pcdata *pp, int functype, struct ast *l)
{
  struct fncall *a = malloc(sizeof(struct fncall));

  if(!a) {
    yyerror(pp, "out of space");
    exit(0);
  }
  a->nodetype = 'F';
  a->l = l;
  a->functype = functype;
  return (struct ast *)a;
}

struct ast *
newcall(struct pcdata *pp, struct symbol *s, struct ast *l)
{
  struct ufncall *a = malloc(sizeof(struct ufncall));

  if(!a) {
    yyerror(pp, "out of space");
    exit(0);
  }
  a->nodetype = 'C';
  a->l = l;
  a->s = s;
  return (struct ast *)a;
}

struct ast *
newref(struct pcdata *pp, struct symbol *s)
{
  struct symref *a = malloc(sizeof(struct symref));

  if(!a) {
    yyerror(pp, "out of space");
    exit(0);
```

```
  }
  a->nodetype = 'N';
  a->s = s;
  return (struct ast *)a;
}

struct ast *
newasgn(struct pcdata *pp, struct symbol *s, struct ast *v)
{
  struct symasgn *a = malloc(sizeof(struct symasgn));

  if(!a) {
    yyerror(pp, "out of space");
    exit(0);
  }
  a->nodetype = '=';
  a->s = s;
  a->v = v;
  return (struct ast *)a;
}

struct ast *
newflow(struct pcdata *pp, int nodetype, struct ast *cond, struct ast *tl, struct
{
  struct flow *a = malloc(sizeof(struct flow));

  if(!a) {
    yyerror(pp, "out of space");
    exit(0);
  }
  a->nodetype = nodetype;
  a->cond = cond;
  a->tl = tl;
  a->el = el;
  return (struct ast *)a;
}

struct symlist *
newsymlist(struct pcdata *pp, struct symbol *sym, struct symlist *next)
{
  struct symlist *sl = malloc(sizeof(struct symlist));

  if(!sl) {
    yyerror(pp, "out of space");
    exit(0);
  }
  sl->sym = sym;
  sl->next = next;
  return sl;
}

void
symlistfree(struct pcdata *pp, struct symlist *sl)
{
  struct symlist *nsl;

  while(sl) {
```

```
    nsl = sl->next;
    free(sl);
    sl = nsl;
  }
}

/* define a function */
void
dodef(struct pcdata *pp, struct symbol *name, struct symlist *syms, struct ast *f
{
  if(name->syms) symlistfree(pp, name->syms);
  if(name->func) treefree(pp, name->func);
  name->syms = syms;
  name->func = func;
}

static double callbuiltin(struct pcdata *pp, struct fncall *);
static double calluser(struct pcdata *pp, struct ufncall *);

double
eval(struct pcdata *pp, struct ast *a)
{
  double v;

  if(!a) {
    yyerror(pp, "internal error, null eval");
    return 0.0;
  }

  switch(a->nodetype) {
    /* constant */
  case 'K': v = ((struct numval *)a)->number; break;

    /* name reference */
  case 'N': v = ((struct symref *)a)->s->value; break;

    /* assignment */
  case '=': v = ((struct symasgn *)a)->s->value =
      eval(pp, ((struct symasgn *)a)->v); break;

    /* expressions */
  case '+': v = eval(pp, a->l) + eval(pp, a->r); break;
  case '-': v = eval(pp, a->l) - eval(pp, a->r); break;
  case '*': v = eval(pp, a->l) * eval(pp, a->r); break;
  case '/': v = eval(pp, a->l) / eval(pp, a->r); break;
  case '|': v = fabs(eval(pp, a->l)); break;
  case 'M': v = -eval(pp, a->l); break;

    /* comparisons */
  case '1': v = (eval(pp, a->l) > eval(pp, a->r))? 1 : 0; break;
  case '2': v = (eval(pp, a->l) < eval(pp, a->r))? 1 : 0; break;
  case '3': v = (eval(pp, a->l) != eval(pp, a->r))? 1 : 0; break;
  case '4': v = (eval(pp, a->l) == eval(pp, a->r))? 1 : 0; break;
  case '5': v = (eval(pp, a->l) >= eval(pp, a->r))? 1 : 0; break;
  case '6': v = (eval(pp, a->l) <= eval(pp, a->r))? 1 : 0; break;

  /* control flow */
```

```
  /* null if/else/do expressions allowed in the grammar, so check for them */
  case 'I':
    if( eval(pp,  ((struct flow *)a)->cond) != 0) {
      if( ((struct flow *)a)->tl) {
        v = eval(pp,  ((struct flow *)a)->tl);
      } else
        v = 0.0;               /* a default value */
    } else {
      if( ((struct flow *)a)->el) {
        v = eval(pp, ((struct flow *)a)->el);
      } else
        v = 0.0;               /* a default value */
    }
    break;

  case 'W':
    v = 0.0;           /* a default value */

    if( ((struct flow *)a)->tl) {
      while( eval(pp, ((struct flow *)a)->cond) != 0)
        v = eval(pp, ((struct flow *)a)->tl);
    }
    break;                     /* last value is value */

  case 'L': eval(pp, a->l); v = eval(pp, a->r); break;

  case 'F': v = callbuiltin(pp, (struct fncall *)a); break;

  case 'C': v = calluser(pp, (struct ufncall *)a); break;

  default: printf("internal error: bad node %c\n", a->nodetype);
  }
  return v;
}

static double
callbuiltin(struct pcdata *pp, struct fncall *f)
{
  enum bifs functype = f->functype;
  double v = eval(pp, f->l);

 switch(functype) {
 case B_sqrt:
   return sqrt(v);
 case B_exp:
   return exp(v);
 case B_log:
   return log(v);
 case B_print:
   printf("= %4.4g\n", v);
   return v;
 default:
   yyerror(pp, "Unknown built-in function %d", functype);
   return 0.0;
 }
}
```

```
static double
calluser(struct pcdata *pp, struct ufncall *f)
{
  struct symbol *fn = f->s;     /* function name */
  struct symlist *sl;           /* dummy arguments */
  struct ast *args = f->l;      /* actual arguments */
  double *oldval, *newval;      /* saved arg values */
  double v;
  int nargs;
  int i;

  if(!fn->func) {
    yyerror(pp, "call to undefined function", fn->name);
    return 0;
  }

  /* count the arguments */
  sl = fn->syms;
  for(nargs = 0; sl; sl = sl->next)
    nargs++;

  /* prepare to save them */
  oldval = (double *)malloc(nargs * sizeof(double));
  newval = (double *)malloc(nargs * sizeof(double));
  if(!oldval || !newval) {
    yyerror(pp, "Out of space in %s", fn->name); return 0.0;
  }

  /* evaluate the arguments */
  for(i = 0; i < nargs; i++) {
    if(!args) {
      yyerror(pp, "too few args in call to %s", fn->name);
      free(oldval); free(newval);
      return 0;
    }

    if(args->nodetype == 'L') { /* if this is a list node */
      newval[i] = eval(pp, args->l);
      args = args->r;
    } else {                    /* if it's the end of the list */
      newval[i] = eval(pp, args);
      args = NULL;
    }
  }

  /* save old values of dummies, assign new ones */
  sl = fn->syms;
  for(i = 0; i < nargs; i++) {
    struct symbol *s = sl->sym;

    oldval[i] = s->value;
    s->value = newval[i];
    sl = sl->next;
  }

  free(newval);
```

```
  /* evaluate the function */
  v = eval(pp, fn->func);

  /* put the dummies back */
  sl = fn->syms;
  for(i = 0; i < nargs; i++) {
    struct symbol *s = sl->sym;

    s->value = oldval[i];
    sl = sl->next;
  }

  free(oldval);
  return v;
}

void
treefree(struct pcdata *pp, struct ast *a)
{
  switch(a->nodetype) {

    /* two subtrees */
  case '+':
  case '-':
  case '*':
  case '/':
  case '1':  case '2':  case '3':  case '4':  case '5':  case '6':
  case 'L':
    treefree(pp, a->r);

    /* one subtree */
  case '|':
  case 'M': case 'C': case 'F':
    treefree(pp, a->l);

    /* no subtree */
  case 'K': case 'N':
    break;

  case '=':
    free( ((struct symasgn *)a)->v);
    break;

  case 'I': case 'W':
    free( ((struct flow *)a)->cond);
    if( ((struct flow *)a)->tl) free( ((struct flow *)a)->tl);
    if( ((struct flow *)a)->el) free( ((struct flow *)a)->el);
    break;

  default: printf("internal error: free bad node %c\n", a->nodetype);
  }

  free(a); /* always free the node itself */
}
```

The `yyerror` function now gets the current line number that was in the static `yylineno` from the scanner state using `yyget_lineno`.

```
void
yyerror(struct pcdata *pp, char *s, ...)
{
  va_list ap;
  va_start(ap, s);

  fprintf(stderr, "%d: error: ", yyget_lineno(pp->scaninfo));
  vfprintf(stderr, s, ap);
  fprintf(stderr, "\n");
}
```

The main function needs to create instance data and link it together, putting a pointer to the application instance data into the scanner instance via `yylex_init_extra` and storing the pointer to the scanner instance into `p.scaninfo`. Then it allocates a fresh symbol table, and it's ready to start parsing.

In this simple example, each time it calls the parser, if the parser returns an AST, it immediately evaluates the AST and frees it.

```
int
main()
{
  struct pcdata p = { NULL, 0, NULL };

  /* set up scanner */
  if(yylex_init_extra(&p, &p.scaninfo)) {
    perror("init alloc failed");
    return 1;
  }

  /* allocate and zero out the symbol table */
  if(!(p.symtab = calloc(NHASH, sizeof(struct symbol)))) {
    perror("sym alloc failed");
    return 1;
  }

  for(;;) {
    printf("> ");
    yyparse(&p);
    if(p.ast) {
      printf("= %4.4g\n", eval(&p, p.ast));
      treefree(&p, p.ast);
      p.ast = 0;
    }
  }
}
```

MAKEFILE FOR PURE APPLICATIONS

This Makefile is slightly more complex than the ones in previous chapters, because the `purecalc` scanner and parser each use a header file created from the other.

```
CFLAGS = -g

all:      purewc purecalc

purewc: purewc.lex.o
        cc -g -o $@ purewc.lex.c

purewc.lex.c: purewc.l
        flex -opurewc.lex.c purewc.l

purecalc:       purecalc.lex.o purecalc.tab.o purecalcfuncs.o
        cc -g -o $@ purecalc.tab.o purecalc.lex.o purecalcfuncs.o -lm

purecalc.lex.o: purecalc.lex.c purecalc.tab.h purecalc.h

purecalc.tab.o: purecalc.tab.c purecalc.lex.h purecalc.h

purecalc.lex.c purecalc.lex.h: purecalc.l
        flex -opurecalc.lex.c purecalc.l

purecalc.tab.c purecalc.tab.h: purecalc.y
        bison -vd purecalc.y
```

PUSH AND PULL PARSERS

Bison has an experimental *push parse* option that turns the flow of control inside out. A regular *pull* parser starts up and repeatedly calls `yylex` to "pull" each token into the parser. In a *push* parser, you create a `yypstate` parser state, and then you call the parser for each token, passing it the token and the state to "push" the token into the parser. The parser does what it can with the token, shifting, reducing, and calling any action routines, and returns after each token. Push parsers are usually also reentrant and are intended to be called from event routines in GUIs and the like. Since they're experimental, the details are likely to change; therefore consult the bison manual for the current calling sequence.

There's no flex push scanner at this point. Each call to a push scanner would pass it a chunk of input text, which it would process and turn it into tokens. Since the flow of control in a push scanner would be inside out relative to a pull scanner, each action rather than returning would call the push parser, passing it the current token and value. When the scanner ran out of input, it would remember where it was, either saving the current position in the scanning automaton or just backing up to the end of the last token recognized, saving the remaining text for next time, and returning to the caller.

There's no reason that one couldn't modify flex to do this. It's open source, so if you're interested, you can do it!

GLR Parsing

WITH GREAT POWER COMES GREAT RESPONSIBILITY

A big reason that parser generators such as yacc and bison became popular is that they create parsers that are much more reliable than handwritten parsers. If you feed a grammar to bison and it has no conflicts, you can be completely sure that the language that the generated parser accepts is exactly the one described by the grammar. It won't have any of the holes that handwritten parsers tend to have, particularly when diagnosing erroneous input. If you use precedence declarations sparingly to resolve conflicts in known situations, expression grammars, and if/then/else, you can still be sure that your parser is handling the language as you think it is.

On the other hand, if you use GLR parsing, you can hand any grammar to bison, and it will create a parser that parses something, resolving the conflicts at parse time. But the more conflicts it has, the less likely it is that the language it's parsing is the language you want, and the less likely it is that your parser will resolve the conflicts the way you want. Before switching to GLR, be sure you understand why your grammar has the conflicts you're expecting GLR to handle and that you understand how you are resolving them. Otherwise, you risk the embarrassing situation of finding out much later that your parser gives up unexpectedly when it runs into a conflict you didn't anticipate or that because of an incorrect conflict resolution, the language it's parsing isn't quite the one you wanted.

GLR parsers can in theory be extremely slow, since running N parses in parallel is roughly N times as slow as a single parse, and a particularly ambiguous grammar could split on each token. Useful GLR grammars typically have only a few ambiguities that are resolved within a few tokens, so the performance is adequate.

A normal bison LALR parser doesn't have to deal with shift/reduce or reduce/reduce conflicts, since any conflicts were resolved one way or the other when the parser was built. (See Chapter 8.) But when a GLR parser encounters a conflict, it conceptually splits and continues both possible parses, with each parser consuming the input tokens in parallel. When there are several conflicts, it can create a tree of partial parses, splitting each time there is a conflict.

If the grammar is actually unambiguous and it just needs more lookahead than the single token that LALR(1) offers, most of the parses will come to a point when they can't match the next input token and will fail. Bison silently discards a failing parse and continues so long as there's at least one other still active. If all possible parses fail, bison reports an error in the usual way. For grammars like this, a GLR parser works very much like a regular LALR parser, and you need only add a few lines to tell it to use the GLR parser and tell it how many conflicts to expect.

On the other hand, if the grammar really is ambiguous, the parser will reach states where there are two or more possible reductions of rules with the same LHS symbol, and it has to decide what to do. If you know that it should always use the same rule, you can put `%dprec N` tags in each of the rules to set the precedence among them. If all of the rules in an ambiguous reduction have `%dprec`, the parser reduces the rule with the highest `N`. Your other option is to use `%merge`, which tells it to call a

routine you write that examines the results of all the rules and "merges" the results into the value to use for the LHS symbol.

While a GLR parser is handling multiple possible parses, it remembers what reductions it would make for each parse but doesn't call the action routines. When it resolves the ambiguity, either by having all but one of the parses fail or by `%dprec` tags, it then calls the action routines and catches up. Normally this makes no difference, but if your parser feeds back information to the scanner, setting start states or flags that the scanner tests, you may have hard-to-diagnose bugs since the parser won't be setting the states or flags when it logically would do so.

GLR Version of the SQL Parser

The SQL parser in Chapter 4 has a few lexical hacks to deal with the limits of LALR parsers. We'll take the hacks out and use a GLR parser instead. One hack made `ONDUPLICATE` a single token because of a lookahead limitation. The other made `NOT EXISTS` a single token that was a variant of `EXISTS` because of ambiguity in the expression grammar. This version of the scanner simply takes out those hacks and makes `EXISTS`, `ON`, and `DUPLICATE` ordinary keyword tokens.

```
EXISTS  { return EXISTS; }
ON      { return ON; }
DUPLICATE { return DUPLICATE; }
```

In the parser, the grammar becomes more straightforward with `ON DUPLICATE` as separate tokens and a separate rule for `NOT EXISTS`.

```
opt_ondupupdate: /* nil */
   | ON DUPLICATE KEY UPDATE insert_asgn_list { emit("DUPUPDATE %d", $5); }
   ;
    ...
expr: ...
   | NOT expr                       { emit("NOT"); }
   | EXISTS '(' select_stmt ')'     { emit("EXISTS 1"); }
   | NOT EXISTS '(' select_stmt ')' { emit("EXISTS 0"); }
   ;
```

The next step is to run the grammar through bison, using the `-v` switch to create a bison listing, and to see what it says. In this case, it says there were 2 shift/reduce conflicts and 59 reduce/reduce:

```
State 249 conflicts: 1 shift/reduce
State 317 conflicts: 1 shift/reduce
State 345 conflicts: 59 reduce/reduce
```

Before throwing the switch to GLR, it's important to be sure that the conflicts are the ones we were expecting, so we look at those three states in the listing file:

```
state 249

   55 join_table: table_reference STRAIGHT_JOIN table_factor .
   56           | table_reference STRAIGHT_JOIN table_factor . ON expr

    ON  shift, and go to state 316

    ON        [reduce using rule 55 (join_table)]
    $default  reduce using rule 55 (join_table)

state 317

   54 join_table: table_reference opt_inner_cross JOIN table_factor . opt_join_co

    ON     shift, and go to state 377
    USING  shift, and go to state 378

    ON        [reduce using rule 70 (opt_join_condition)]
    $default  reduce using rule 70 (opt_join_condition)

    opt_join_condition  go to state 379
    join_condition      go to state 380

state 345

  263 expr: EXISTS '(' select_stmt ')' .
  264     | NOT EXISTS '(' select_stmt ')' .

    NAME              reduce using rule 263 (expr)
    NAME              [reduce using rule 264 (expr)]

    ... 57 more reduce/reduce conflicts ...

    ')'               reduce using rule 263 (expr)
    ')'               [reduce using rule 264 (expr)]
    $default          reduce using rule 263 (expr)
```

States 249 and 317 are indeed limited lookahead for `ON`, and state 345 is the ambiguity of treating `NOT EXISTS` as one operator or two. (The large number of conflicts is because the token that follows `NOT EXISTS` can be any token that's valid in or after an expression.) Having confirmed that the conflicts are the expected ones, we add three lines to the definition section, one to make it a GLR parser and the other two to tell bison how many conflicts to expect. If you change the grammar and the number of conflicts changes, bison will fail, which is a good thing; then you can go back and be sure the new set of conflicts is still the expected one.

```
%glr-parser
%expect 2
%expect-rr 59
```

With these additions, the parser will now build, and it mostly works. Here it correctly parses two statements that use `ON` and `ON DUPLICATE`, but

we haven't quite finished with the expression ambiguity:

```
insert into foo select a from b straight_join c on d;
rpn: NAME a
rpn: TABLE b
rpn: TABLE c
rpn: NAME d
rpn: JOIN 128
rpn: SELECT 0 1 1
rpn: INSERTSELECT 0 foo
rpn: STMT

insert into foo select a from b straight_join c on duplicate key update x=y;
rpn: NAME a
rpn: TABLE b
rpn: TABLE c
rpn: JOIN 128
rpn: SELECT 0 1 1
rpn: NAME y
rpn: ASSIGN x
rpn: DUPUPDATE 1
rpn: INSERTSELECT 0 foo
rpn: STMT

select not exists(select a from b);
rpn: NAME a
rpn: TABLE b
rpn: SELECT 0 1 1
Ambiguity detected.
Option 1,
  expr -> <Rule 247, tokens 2 .. 9>
    NOT <tokens 2 .. 2>
    expr -> <Rule 263, tokens 3 .. 9>
      EXISTS <tokens 3 .. 3>
      '(' <tokens 4 .. 4>
      select_stmt <tokens 5 .. 8>
      ')' <tokens 9 .. 9>

Option 2,
  expr -> <Rule 264, tokens 2 .. 9>
    NOT <tokens 2 .. 2>
    EXISTS <tokens 3 .. 3>
    '(' <tokens 4 .. 4>
    select_stmt <tokens 5 .. 8>
    ')' <tokens 9 .. 9>

1: error: syntax is ambiguous
SQL parse failed
```

Bison produces excellent diagnostics in GLR parsers. Here we can see the two possible parses: the first treating `NOT EXISTS` as separate operators, and the second treating it as one operator. This problem is easily fixed with `%dprec` since the one-operator version is always the one we want:

```
expr: ...
   | NOT expr { emit("NOT"); }  %dprec 1
   ...
   | NOT EXISTS '(' select_stmt ')'  { emit("EXISTS 0"); } %dprec 2
```

Now the parser works correctly. The other way to resolve ambiguous parses is to provide your own function that takes the results of both rules and returns the result to be used as the value of the reduced rule. The arguments to the function and its value are of type `YYSTYPE`, which is the name that flex gives to the union created from `%union` declarations. If there are multiple reduce/reduce conflicts, you need a separate function for each one you want to resolve yourself. Each rule involved has a `%merge` tag, and all of the tags have to be the same for the rules involved in a conflict to be resolved:

```
%{
YYSTYPE exprmerge(YYSTYPE x1, YYSTYPE x2);
%}
expr: ...
   | NOT expr { emit("NOT"); } %merge <exprmerge>
   ...
   | NOT EXISTS '(' select_stmt ')'  { emit("EXISTS 0"); } %merge <exprmerge>
```

It's hard to come up with an application of `%merge` that isn't either contrived or extremely complicated. Since the merge function is called after all of the possible rules have been reduced, the values for rules have to be ASTs or something similar that contain all of the necessary information about all of the possible parses. The function would look at the ASTs and probably pick one to return and throw the other away, remembering to free its allocated storage first. The classic ambiguous syntax handled by GLR parsers is C++ declarations, which can be syntactically identical to an assignment with a typecast, but since the resolution rule is that anything that can be a declaration is a declaration, `%dprec` handles it just fine with no merging needed.

In many cases, GLR parsers are overkill, and you're better off tweaking your scanner and parser to run in a regular LALR parser, but if you have a predefined input language that just isn't LALR, GLR can be a lifesaver.

C++ Parsers

Bison can create parsers in C++. Although flex appears to be able to create C++, scanners, the C++ code doesn't work.[21] Fortunately, C scanners created by flex compile under C++ and it is not hard to use a flex C scanner with a bison C++ parser, which is what we'll do in the next example.

All bison C++ parsers are reentrant, so bison creates a class for the parser. As with reentrant C parsers, the programmer can create as many in-

stances as needed and usually passes in per-instance application data kept in a separate class.

Every time you create a C++ parser, bison creates four class headers; `location.hh` and `position.hh` to define the location structure, `stack.hh` for the internal parser stack, and the header for the parser itself. The first three always have the same contents; the last has specific values from the parser, as its C equivalent does. The parser header includes the other files, so they're separate files mostly; therefore they can be included by the lexer and other modules that need to handle locations. (One could ask why they didn't just make those three files standard library include files.) Creating the parser header is mandatory, since the generated C++ source includes it, although you still have to tell bison to do so.

A C++ Calculator

C++ parsers are somewhat more complex than C parsers, so to keep the code manageable, the example is based on the very simple calculator from Chapter 1. To make it slightly more interesting, the calculator can work in any radix from 2 to 10, with the radix being stored in the per-parser application context.

The application class `cppcalc_ctx` is defined in the header file `cppcalc-ctx.hh`, shown in Example 9-6.

Example 9-6. Application context class for the C++ calculator cppcalc-ctx.hh

```
class cppcalc_ctx {
public:
  cppcalc_ctx(int r) { assert(r > 1 && r <= 10); radix = r; }

  inline int getradix(void) { return radix; }

private:
  int radix;

};
```

C++ Parser Naming

Unless otherwise instructed, bison creates the parser in the `yy` namespace, specifically in a class called `parser`. The parser itself is a class method called `parse`, which you call after creating an instance of the class. The namespace can be changed by the declaration `%define namespace`, and the class can be changed by `%define parser_class_name`. In this example we change the class name to `cppcalc`. The class contains the private data for a parser. It also has some debugging methods enabled by defining the preprocessor symbol `YYDEBUG`, notably `set_debug_level(N)`, which turns on parser tracing if `N` is nonzero.

A C++ Parser

The C++ parser, shown in Example 9-7, uses some new declarations.

Example 9-7. C++ calculator parser cppcalc.yy

```
/* C++ version of calculator */

%language "C++"
%defines
%locations

%define parser_class_name "cppcalc"

%{
#include <iostream>
using namespace std;
#include "cppcalc-ctx.hh"
%}

%parse-param { cppcalc_ctx &ctx }
%lex-param   { cppcalc_ctx &ctx }

%union {
       int ival;
};

/* declare tokens */
%token <ival> NUMBER
%token ADD SUB MUL DIV ABS
%token OP CP
%token EOL

%type <ival> exp factor term

%{
extern int yylex(yy::cppcalc::semantic_type *yylval,
       yy::cppcalc::location_type* yylloc,
       cppcalc_ctx &ctx);

void myout(int val, int radix);
%}

%initial-action {
 // Filename for locations here
 @$.begin.filename = @$.end.filename = new std::string("stdin");
}
%%
%%
```

The parser uses `%language` to declare that it's written in C++ rather than in C, `%defines` to create the header file, `%locations` to put code to handle locations into the parser, and `%define` to call the class `cppcalc` rather than the default parser. Then some C++ code includes the `iostream` library and the context header described earlier.

The `%parser-param` and `%lex-param` declarations are the ones we met in the section Pure Parsers in Bison, and we define an extra argument to the parser (the parser's class constructor) and to `yylex`. In this example we're not using a reentrant lexer, but if we were, the parameter to the lexer would have to be a flex scanner context `yyscan_t` as it was in the previous example. It would use the same trick of storing it in a field in the parser parameter, which is now an instance of the context class.

We declare a `%union`, which works the same way it does in C. In this simple case it has one member, the integer value of an expression. Following that are the token and nonterminal declarations.

C++ AND %UNION

Although C allows a union to include structures, C++ doesn't permit a union to include class instances, so you can't use a class directly in a `%union`. Pointers to class instances are fine and are quite common in C++ parsers. Since the class instances pointed to will be dynamically allocated with `new`, remember that in each action where RHS symbols have class pointer values, each pointer value has to be saved somewhere it can be referenced later or has to be deleted. If you use parser error recovery, use `%destructor` declarations to tell bison how to free values that are discarded during error recovery. Otherwise, the program will have storage leaks.

These are the same rules that apply to `malloc`ed C structures, as in the AST we built in Chapter 3.

For some reason, bison doesn't create the declaration of `yylex` that C++ requires, so we do it manually here. Its arguments are the same as in the pure C example: pointers to the token value, token location, and the `lex-param` context pointer. In a C++ parser, token values and locations have the types `semantic_type` and `location_type`, both members of the parser class. We also define `myout`, an output routine defined later that prints values using a particular radix.

In a C parser that uses locations, if you want to report the filename, you have to add your own code to do so, as in section More Sophisticated Locations with Filenames, but the C++ version of locations fixes that oversight, adding a `filename` string field to the begin and end positions in each location. The `%initial-action` sets the filename for the initial location in `$@`, which is the location that's passed to the scanner. That is all we need since the scanner in this example reads only one file and never updates the location filename.

```
   // bison rules for the C++ parser
  calclist: /* nothing */
   | calclist exp EOL { cout << "= "; myout(ctx.getradix(), $2); cout << "\n> "; }
   | calclist EOL { cout <<  "> "; } /* blank line or a comment */
   ;
```

```
exp: factor
 | exp ADD factor { $$ = $1 + $3; }
 | exp SUB factor { $$ = $1 - $3; }
 | exp ABS factor { $$ = $1 | $3; }
 ;

factor: term
 | factor MUL term { $$ = $1 * $3; }
 | factor DIV term { if($3 == 0) {
                        error(@3, "zero divide");
                        YYABORT;
                     }
                     $$ = $1 / $3; }
 ;

term: NUMBER
 | ABS term { $$ = $2 >= 0? $2 : - $2; }
 | OP exp CP { $$ = $2; }
 ;
%%
```

The rules in the parser are the same as they were in Chapter 1, with the action code changed from C to C++. Note that the rule that prints a top-level `calclist` expression now calls `myout`, passing it the radix fetched from the `ctx` structure. The rule for division now has a zero divide test, calling parser class member function `error`, which replaces `yyerror`. (Because of a bug in the C++ parser skeleton, it always calls `error` with a location argument even if you don't use `%location`. The easiest work-around is always to use locations in a C++ parser.)

```
// C++ code section of parser
main()
{
  cppcalc_ctx ctx(8);      // work in octal today

  cout << "> ";

  yy::cppcalc parser(ctx); // make a cppcalc parser

  int v = parser.parse();  // and run it

  return v;
}

// print an integer in given radix
void
myout(int radix, int val)
{
  if(val < 0) {
    cout << "-";
    val = -val;
  }
  if(val > radix) {
    myout(radix, val/radix);
    val %= radix;
```

```
  }
  cout << val;
}

int
myatoi(int radix, char *s)
{
  int v = 0;

  while(*s) {
    v = v*radix + *s++ - '0';
  }
  return v;
}

namespace yy {
  void
  cppcalc::error(location const &loc, const std::string& s) {
     std::cerr << "error at "  << loc << ": " << s << std::endl;
  }
}
```

Unlike a C pure parser, a C++ pure parser requires that you first create an instance of the parser then call it. Hence, the main program creates a `ctx` structure with an appropriate radix, creates an instance of `yy::cppcalc` called `parser` using that context, and then calls the `parse` method to do the actual parsing.

Two helper routines, `myout` and `myatoi`, do radix to binary conversion, and finally we define `yy::error`, the error routine analogous to `yyerror`. For some reason, bison declares `error` as a private member function in the parser class, which means you can't call it from elsewhere; in particular, you can't call it from the scanner. The bison manual suggests that `yy::error` call the real error routine, which is defined in a context visible to the scanner and is probably the best workaround. Notice, incidentally, that the error routine outputs the location of the error using the normal C++ `<<` operator. This works because the `location` class defines a variety of operators including output formatting.

Interfacing a Scanner with a C++ Parser

The flex scanner, shown in Example 9-8, is written in C++-compatible C.

Example 9-8. C++ calculator scanner cppcalc.l

```
/* recognize tokens for the C++ calculator and print them out */

%option noyywrap
%{
# include <cstdlib>

# include "cppcalc-ctx.hh"
# include "cppcalc.tab.hh"
```

```
#define YY_DECL int yylex(yy::cppcalc::semantic_type *yylval, \
    yy::cppcalc::location_type *yylloc, cppcalc_ctx &ctx)

// make location include the current token
# define YY_USER_ACTION  yylloc->columns (yyleng);

typedef yy::cppcalc::token token;
extern int myatoi(int radix, char *s); // defined in the parser
%}
%%
```

The declaration part of the scanner includes the standard C library and the header files for the context and parser classes. It defines `YY_DECL` to declare the calling sequence for `yylex` to match what the parser expects, and it defines `YY_USER_ACTION`, the macro invoked before the action for each token, to set the location based on the length of the token. This is the same trick we did in Chapter 8, but the code is much shorter since C++ locations have a method that does what we want.

The parser token numbers are defined in the `token` member of the parser class, so a `typedef` for the plain name `token` will make the token values easier to type.

```
 // rules for C++-compatible scanner
%{
     // start where previous token ended
     yylloc->step ();
%}

"+"     { return token::ADD; }
"-"     { return token::SUB; }
"*"     { return token::MUL; }
"/"     { return token::DIV; }
"|"     { return token::ABS; }
"("     { return token::OP; }
")"     { return token::CP; }
[0-9]+  { yylval->ival = myatoi(ctx.getradix(), yytext); return token::NUMBER; }

\n      { yylloc->lines(1); return token::EOL; }

  /* skip over comments and whitespace */
"//".*  |
[ \t]   {  yylloc->step (); }

.       { printf("Mystery character %c\n", *yytext); }
%%
```

The code at the beginning of the rules section is copied near the beginning of `yylex`. The `step` method sets the beginning of the location equal to the end, so the location now points to the end of the previous token. (An alternative would be what we did in Chapter 8, tracking the line and column in local variables and copying them into the location for each

token, but this takes advantage of predefined methods on C++ locations to make the code shorter.)

The action code prefixes the token names with `token::` since the token names are now parser class members. The action for a newline uses the `lines` method to update the location line number, the action for comments and whitespace invokes `step` since they don't return from the scanner, and the previous `step` is invoked only when `yylex` has returned and is called again.

The last catchall rule prints an error message. In the original C version of the scanner it called `yyerror`, but since this scanner isn't part of the C++ parser class, it can't call the parser `error` routine. Rather than write glue routines to allow the various parts of the program to call the same error reporting routine, for simplicity we just call `printf`.

Should You Write Your Parser in C++ ?

As should be apparent by now, the C++ support in bison is nowhere near as mature as the C support, which is not surprising since it's about 30 years newer. The fact that `%union` can't include class instances can require some extra work, and the less than seamless integration between C++ bison and C flex requires careful programming, particularly if they need to share significant data structures accessed from C in the scanner and C++ in the parser or if they need to have the scanner read its input using C stdio while the rest of the program uses C++ library I/O. A good object design would wrap a class around the application context (`ctx` in this example), the parser, and probably the scanner to present a unified interface to the rest of the program.

Nonetheless, C++ bison parsers do work, and the design of the parser class is a reasonable one. If you're integrating your parser into a larger C++ project or if you want to use C++ libraries that don't have C equivalents, a C++ parser can work well.

JAVA AND BEYOND

Bison currently (2009) has experimental support for parsers written in Java. By the time you read this, it may support other languages as well. The Java support is modeled on that for C++, with adjustments for the Java environment, which has no preprocessor or unions, but does have garbage collection. Since the details are likely to have changed, consult the bison manual for the current Java interface.

Exercises

1. Modify the parser in the pure calculator to parse one statement at a time and return without using `YYACCEPT`. You'll probably want to change the scanner so it returns a zero token at end-of-line rather than using `EOL`.

2. Does the GLR version of the SQL parser accept the same language as the original version? Come up with an example that would be accepted and one that wouldn't. (Hint: Try putting comments between tokens that usually just have a space between them.)

---

[20] If you pass multiple arguments to the parser, each one has to be in a separate `%parse-param`. If you put two arguments in the same `%parse-param`, bison won't report an error but the second parameter will disappear.

---

[21] This is confirmed by the guy who wrote it. It will probably be fixed eventually, but it turned out to be surprisingly hard to design a good C++ interface for flex scanners.