flex & bison by John Levine

Chapter 4. Parsing SQL

SQL (which stands for Structured Query Language and is usually pronounced *sequel*) is the most common language used to handle relational databases.[15] We'll develop a SQL parser that produces a compact tokenized version of SQL statements.

This parser is based on the version of SQL used in the popular MySQL open source database. MySQL actually uses a bison parser to parse its SQL input, although for a variety of reasons this parser isn't based on mySQL's parser but rather is based on the description of the language in the manual.

MySQL's parser is much longer and more complex, since this pedagogical example leaves out many of the less heavily used parts. MySQL's parser is written in an odd way that uses bison to generate a C parser that's compiled by the C++ compiler, with a handwritten C++ lexer. There's also the detail that its license doesn't allow excerpting in a book like this one. But if you're interested, it's the file `sql/sql_yacc.yy`, which is part of the source code at http://dev.mysql.com/downloads/mysql/5.1.html.

The ultimate definitions for SQL are the standards documents published by ANSI and [illegible] documents that define the way to embed SQL in other programming languages and in XML.

A Quick Overview of SQL

SQL is a special-purpose language for relational databases. Rather than manipulating data in memory, it manipulates data in database tables, referring to memory only incidentally.

Relational Databases

A *database* is a collection of *tables*, which are analogous to files. Each table contains


```
CREATE TABLE Foods (
        name CHAR(8) NOT NULL,
        type CHAR(5),
        flavor    CHAR(6),
        PRIMARY KEY ( name )
)

CREATE TABLE Courses (
        course      CHAR(8) NOT NULL PRIMARY KEY,
        flavor      CHAR(6),
        sequence INTEGER
)
```

The syntax is completely free-format, and there are often several different syntactic ways to write the same thing—notice the two different ways we gave the `PRIMARY KEY` specifier. (The primary key in a table is a column, or set of columns, that uniquely specifies a row.) Table 4-1 shows the two tables we just created after loading in data.

Table 4-1. Two relational tables

Foods					**Courses**		
name	*type*	*flavor*			*course*	*flavor*	*sequence*
peach	fruit	sweet			salad	savory	1
tomato	fruit	savory			main	savory	2
lemon	fruit	sour			dessert	sweet	3
lard	fat	bland					
cheddar	fat	savory					

SQL implements what's known as a *tuple calculus*, where *tuple* is relational-ese for a record, which is an ordered list of fields or expressions. To use a database, you tell the database what tuples you want it to extract from your data. It's up to the database to figure out how to get it from the tables it has. (That's the calculus part.) The specification of a set of desired data is a *query*. For example, using the two tables in Table 4-1, to get a list of fruits, you would say the following:

```
SELECT name, flavor
```


Table 4-2. SQL response table

name	flavor
peach	sweet
tomato	savory
lemon	sour

You can also ask questions spanning more than one table. To get a list of foods suitable to each course of the meal, you say the following:

```
SELECT course, name, Foods.flavor, type
FROM   Courses, Foods

WHERE  Courses.flavor = Foods.flavor
```

The response is shown in Table 4-3.

Table 4-3. Second SQL response table

course	name	flavor	type
salad	tomato	savory	fruit
salad	cheddar	savory	fat
main	tomato	savory	fruit
main	cheddar	savory	fat
dessert	peach	sweet	fruit

When listing the column names, we can leave out the table name if the column name is unambiguous.

Manipulating Relations

Three Ways to Use SQL

In the original version of SQL, users typed commands into a file or directly at the terminal and received responses immediately. People still sometimes use it this way for creating tables and for debugging, but for the vast majority of applications, SQL commands come from inside programs, and the results are returned to those programs. The SQL standard defines a "module language" to embed SQL in a variety of programming languages, but MySQL avoids the issue by using subroutine calls for communication between a user program and the database, and it doesn't use the module language at all.

Since the syntax of SQL is so large, we have reproduced the entire grammar in one place in the Appendix A, with a cross-reference for all of the symbols in the grammar.

SQL to RPN

Our tokenized version of SQL will use a version of Reverse Polish Notation (RPN), familiar to users of HP calculators. In 1920, Polish logician Jan Łukasiewicz[16] realized that if you put the operators before the operands in logical expressions, you don't need any parentheses or other punctuation to describe the order of evaluation:

```
(a+b)*c         * + a b c
a+(b*c)         + a * b c
```



```
(a+b)*c         a b + c *
a+(b*c)         a b c * +
```

On a computer, RPN has the practical advantage that it is very easy to interpret using a stack. The computer processes each token in order. If it's an operand, it pushes the token on the stack. If it's an operator, it pops the right number of operands off the stack, does the operation, and pushes the result. This trick is very well known and has been used since 1954 to build software and hardware that interprets RPN code using a stack.

RPN has two other advantages for compiler developers. One is that if you're using a


```
expr: NAME          { emit("NAME %s", $1); }
    | INTNUM        { emit("NUMBER %d", $1); }
    | expr '+' expr { emit("ADD"); }
    | expr '-' expr { emit("SUB"); }
    | expr '*' expr { emit("MUL"); }
    | expr '/' expr { emit("DIV"); }
```

When it parses `a+2*3`, it emits `NAME a, NUMBER 2, NUMBER 3, MUL, ADD`. This lovely property comes directly from the way a LALR parser works, pushing the symbols for partially parsed rules on its internal stack and then at the end of each rule popping the symbols and pushing the new LHS symbol, which is a sequence of operations just the same as what an RPN interpreter does.

The other advantage is that it is very easy to turn a string of RPN tokens into an AST, and vice versa. To turn RPN into an AST, you run through the RPN pushing each operand and, for each operator, pop the operands, build an AST tree node with the operands and operator, and then push the address of the new tree node. When you're done, the stack will contain the root of the AST. To go the other way, you do a depth-first walk of the AST. Starting from the root of the AST, at each node you visit the subnodes (by recursively calling the tree-walking subroutine) and then emit the operator for the node. At leaf nodes, you just emit the operand for that node.

Classic RPN has a fixed number of operands for each operator, but we're going to relax the rules a little and have some operators that take a variable number of operands, with the number as part of the operator. For example:


```
rpn: NAME a
rpn: NAME b
rpn: NAME c
rpn: TABLE d
rpn: SELECT 3
```

The `3` in the `SELECT` tells the RPN interpreter that the statement is selecting three things, so after it pops the table name, it should take the three field names off the stack. I've written interpreters for RPN code, and this trick makes the code a lot simpler than the alternative of using extra tree-building operators to combine the variable number of operands into one before handing the combined operand to the main operator.

The Lexer

a MySQL extension also allowing C comments.

Example 4-1. MySQL lexer

```
/*
 * Scanner for mysql subset
 * $Header: /usr/home/johnl/flnb/RCS/ch04.tr,v 1.7 2009/05/19 18:28:27 johnl
 */

%option noyywrap nodefault yylineno case-insensitive
%{
#include "pmysql.tab.h"
#include <stdarg.h>
#include <string.h>

void yyerror(char *s, ...);

int oldstate;

%}

%x COMMENT
%s BTWMODE

%%
```

The lexer, shown in Example 4-1, starts with a few include files, notably `pmysql.tab.h`, the token name definition file generated by bison. It also defines two start states, an exclusive `COMMENT` state used in C-style comments and an inclusive `BTWMODE` state used in a kludge to deal with a SQL expression that has its own idea of the keyword AND

Scanning SQL Keywords

SQL has a lot of keywords:

```
  /* keywords */

ADD     { return ADD; }
ALL     { return ALL; }
ALTER   { return ALTER; }
ANALYZE { return ANALYZE; }

  /* Hack for BETWEEN ... AND ...
```

```
     * return special AND token if BETWEEN seen
     */
```



```
AUTO_INCREMENT  { return AUTO_INCREMENT; }
BEFORE  { return BEFORE; }
BETWEEN { BEGIN BTWMODE; return BETWEEN; }
INT8|BIGINT     { return BIGINT; }
BINARY  { return BINARY; }
BIT     { return BIT; }
BLOB    { return BLOB; }
BOTH    { return BOTH; }
BY      { return BY; }
CALL    { return CALL; }
CASCADE { return CASCADE; }
CASE    { return CASE; }
CHANGE  { return CHANGE; }
CHAR(ACTER)?    { return CHAR; }
CHECK   { return CHECK; }
COLLATE { return COLLATE; }
COLUMN  { return COLUMN; }
COMMENT { return COMMENT; }
CONDITION       { return CONDITION; }
CONSTRAINT      { return CONSTRAINT; }
CONTINUE        { return CONTINUE; }
CONVERT { return CONVERT; }
CREATE  { return CREATE; }
CROSS   { return CROSS; }
CURRENT_DATE    { return CURRENT_DATE; }
CURRENT_TIME    { return CURRENT_TIME; }
CURRENT_TIMESTAMP       { return CURRENT_TIMESTAMP; }
CURRENT_USER    { return CURRENT_USER; }
CURSOR  { return CURSOR; }
DATABASE        { return DATABASE; }
DATABASES       { return DATABASES; }
DATE    { return DATE; }
DATETIME        { return DATETIME; }
```



```
DAY_MICROSECOND { return DAY_MICROSECOND; }
DAY_MINUTE      { return DAY_MINUTE; }
DAY_SECOND      { return DAY_SECOND; }
NUMERIC|DEC|DECIMAL     { return DECIMAL; }
DECLARE { return DECLARE; }
DEFAULT { return DEFAULT; }
DELAYED { return DELAYED; }
DELETE  { return DELETE; }
DESC    { return DESC; }
DESCRIBE        { return DESCRIBE; }
DETERMINISTIC   { return DETERMINISTIC; }
DISTINCT        { return DISTINCT; }
DISTINCTROW     { return DISTINCTROW; }
DIV     { return DIV; }
```

```
FLOAT8|DOUBLE   { return DOUBLE; }
```



```
ENUM { return ENUM; }
ESCAPED { return ESCAPED; }
EXISTS  { yylval.subtok = 0; return EXISTS; }
NOT[ \t\n]+EXISTS       { yylval.subtok = 1; return EXISTS; }
EXIT    { return EXIT; }
EXPLAIN { return EXPLAIN; }
FETCH   { return FETCH; }
FLOAT4? { return FLOAT; }
FOR     { return FOR; }
FORCE   { return FORCE; }
FOREIGN { return FOREIGN; }
FROM    { return FROM; }
FULLTEXT        { return FULLTEXT; }
GRANT   { return GRANT; }
GROUP   { return GROUP; }
HAVING  { return HAVING; }
HIGH_PRIORITY   { return HIGH_PRIORITY; }
HOUR_MICROSECOND        { return HOUR_MICROSECOND; }
HOUR_MINUTE     { return HOUR_MINUTE; }
HOUR_SECOND     { return HOUR_SECOND; }
IF      { return IF; }
IGNORE  { return IGNORE; }
IN      { return IN; }
INFILE  { return INFILE; }
INNER   { return INNER; }
INOUT   { return INOUT; }
INSENSITIVE     { return INSENSITIVE; }
INSERT  { return INSERT; }
INT4?|INTEGER   { return INTEGER; }
INTERVAL        { return INTERVAL; }
INTO    { return INTO; }
IS      { return IS; }
ITERATE { return ITERATE; }
```



```
INDEX|KEY       { return KEY; }
KEYS    { return KEYS; }
KILL    { return KILL; }
LEADING { return LEADING; }
LEAVE   { return LEAVE; }
LEFT    { return LEFT; }
LIKE    { return LIKE; }
LIMIT   { return LIMIT; }
LINES   { return LINES; }
LOAD    { return LOAD; }
LOCALTIME       { return LOCALTIME; }
LOCALTIMESTAMP  { return LOCALTIMESTAMP; }
LOCK    { return LOCK; }
LONG    { return LONG; }
```

```
LONGBLOB        { return LONGBLOB; }
```



```
MEDIUMTEXT      { return MEDIUMTEXT; }
MINUTE_MICROSECOND      { return MINUTE_MICROSECOND; }
MINUTE_SECOND   { return MINUTE_SECOND; }
MOD     { return MOD; }
MODIFIES        { return MODIFIES; }
NATURAL { return NATURAL; }
NOT     { return NOT; }
NO_WRITE_TO_BINLOG      { return NO_WRITE_TO_BINLOG; }
NULL    { return NULLX; }
NUMBER  { return NUMBER; }
ON      { return ON; }
ON[ \t\n]+DUPLICATE { return ONDUPLICATE; } /* hack due to limited lookahead
OPTIMIZE        { return OPTIMIZE; }
OPTION  { return OPTION; }
OPTIONALLY      { return OPTIONALLY; }
OR      { return OR; }
ORDER   { return ORDER; }
OUT     { return OUT; }
OUTER   { return OUTER; }
OUTFILE { return OUTFILE; }
PRECISION       { return PRECISION; }
PRIMARY { return PRIMARY; }
PROCEDURE       { return PROCEDURE; }
PURGE   { return PURGE; }
QUICK   { return QUICK; }
READ    { return READ; }
READS   { return READS; }
REAL    { return REAL; }
REFERENCES      { return REFERENCES; }
REGEXP|RLIKE    { return REGEXP; }
RELEASE { return RELEASE; }
RENAME  { return RENAME; }
REPEAT  { return REPEAT; }
```



```
REQUIRE { return REQUIRE; }
RESTRICT        { return RESTRICT; }
RETURN  { return RETURN; }
REVOKE  { return REVOKE; }
RIGHT   { return RIGHT; }
ROLLUP  { return ROLLUP; }
SCHEMA  { return SCHEMA; }
SCHEMAS { return SCHEMAS; }
SECOND_MICROSECOND      { return SECOND_MICROSECOND; }
SELECT  { return SELECT; }
SENSITIVE       { return SENSITIVE; }
SEPARATOR       { return SEPARATOR; }
SET     { return SET; }
SHOW    { return SHOW; }
```

```
INT2|SMALLINT   { return SMALLINT; }
```



```
SQLSTATE        { return SQLSTATE; }
SQLWARNING      { return SQLWARNING; }
SQL_BIG_RESULT  { return SQL_BIG_RESULT; }
SQL_CALC_FOUND_ROWS     { return SQL_CALC_FOUND_ROWS; }
SQL_SMALL_RESULT        { return SQL_SMALL_RESULT; }
SSL     { return SSL; }
STARTING        { return STARTING; }
STRAIGHT_JOIN   { return STRAIGHT_JOIN; }
TABLE   { return TABLE; }
TEMPORARY       { return TEMPORARY; }
TERMINATED      { return TERMINATED; }
TEXT    { return TEXT; }
THEN    { return THEN; }
TIME    { return TIME; }
TIMESTAMP       { return TIMESTAMP; }
INT1|TINYINT    { return TINYINT; }
TINYTEXT        { return TINYTEXT; }
TO      { return TO; }
TRAILING        { return TRAILING; }
TRIGGER { return TRIGGER; }
UNDO    { return UNDO; }
UNION   { return UNION; }
UNIQUE  { return UNIQUE; }
UNLOCK  { return UNLOCK; }
UNSIGNED        { return UNSIGNED; }
UPDATE  { return UPDATE; }
USAGE   { return USAGE; }
USE     { return USE; }
USING   { return USING; }
UTC_DATE        { return UTC_DATE; }
UTC_TIME        { return UTC_TIME; }
UTC_TIMESTAMP   { return UTC_TIMESTAMP; }
VALUES? { return VALUES; }
VARCHAR(ACTER)? { return VARCHAR; }
VARYING { return VARYING; }
WHEN    { return WHEN; }
WHERE   { return WHERE; }
WHILE   { return WHILE; }
WITH    { return WITH; }
WRITE   { return WRITE; }
XOR     { return XOR; }
YEAR    { return YEAR; }
YEAR_MONTH      { return YEAR_MONTH; }
ZEROFILL        { return ZEROFILL; }
```

All of the reserved words are separate tokens in the parser, because it is the easiest

the `&&` logical-and operator, except in the SQL operator `BETWEEN ... AND`:

```
IF(a && b, ...)      normally these mean the same thing
IF(a AND b, ...)

... WHERE a BETWEEN c AND d, ... except here
```

There's a variety of ways to deal with problems like this, but lexical special cases are often the easiest.

Also note that the phrases `NOT EXISTS` and `ON DUPLICATE` are recognized as single tokens; this is to avoid shift/reduce conflicts in the parser because of other contexts where `NOT` and `ON` can appear. To remember the difference between `EXISTS` and `NOT EXISTS`, the lexer returns a value along with the token that the parser uses when generating the token code. These two don't actually turn out to be ambiguous, but parsing them needs more than the single-token lookahead that bison usually uses. We revisit these in Chapter 9 where the alternate GLR parser can handle them directly.

Scanning Numbers

Numbers come in a variety of forms:

```
   /* numbers */

-?[0-9]+                { yylval.intval = atoi(yytext); return INTNUM; }

-?[0-9]+"."[0-9]* |
```



```
-?[0-9]+E[-+]?[0-9]+     |
-?[0-9]+"."[0-9]*E[-+]?[0-9]+ |
-?"."[0-9]+E[-+]?[0-9]+ { yylval.floatval = atof(yytext) ;
                                  return APPROXNUM; }
    /* booleans */
TRUE    { yylval.intval = 1; return BOOL; }
UNKNOWN { yylval.intval = -1; return BOOL; }
FALSE   { yylval.intval = 0; return BOOL; }

   /* strings */

'(\\.|''|[^'\n])*'    |
\"(\\.|\"\"|[^"\n])*\"  { yylval.strval = strdup(yytext); return STRING; }
```

```
   /* bit strings */

0B[01]+      |
B'[01]+'     { yylval.strval = strdup(yytext); return STRING; }
```

SQL numbers are similar to the numbers we've seen in previous chapters. The rules to scan them turn them into C integers or doubles and store them in the token values. Boolean values are true, false, and unknown, so they're recognized as reserved words and returned as variations on a `BOOL` token.

SQL strings are enclosed in single quotes, using a pair of quotes to represent a single quote in the string. MySQL extends this to add double-quoted strings, and `\x` escapes within strings. The first two string patterns match valid, quoted strings that don't extend past a newline and return the string as the token value, remembering to make a copy since the value in `yytext` doesn't stay around.[17] The next two patterns catch unterminated strings and print a suitable diagnostic.

The next four patterns match hex and binary strings, each of which can be written in two ways. A more realistic example would convert them to binary, but for our purposes we just return them as strings.

Scanning Operators and Punctuation

Operators and punctuation can be captured with a few patterns:


```
"&&"           { return ANDOP; }
"||"           { return OR; }

"="     { yylval.subtok = 4; return COMPARISON; }
"<=>"   { yylval.subtok = 12; return COMPARISON; }
">="    { yylval.subtok = 6; return COMPARISON; }
">"     { yylval.subtok = 2; return COMPARISON; }
"<="    { yylval.subtok = 5; return COMPARISON; }

"<"     { yylval.subtok = 1; return COMPARISON; }
"!="    |
"<>"    { yylval.subtok = 3; return COMPARISON; }
```

```
"<<"    { yylval.subtok = 1; return SHIFT; }
">>"    { yylval.subtok = 2; return SHIFT; }
```


character operators with the same pattern. MySQL has the usual range of comparison operators, which are all treated as one `COMPARISON` operator with the token value telling which one. We'll see later that this doesn't work perfectly, since the = token is used in a few places where it's not a comparison, but we can work around it.

Scanning Functions and Names

The last pieces to capture are functions and names:

```
        /* functions */

SUBSTR(ING)?/"(" { return FSUBSTRING; }
TRIM/"("         { return FTRIM; }
DATE_ADD/"("    { return FDATE_ADD; }
DATE_SUB/"("    { return FDATE_SUB; }

         /* check trailing context manually */
COUNT    { int c = input(); unput(c);
           if(c == '(') return FCOUNT;
           yylval.strval = strdup(yytext);
           return NAME; }

        /* names */

[A-Za-z][A-Za-z0-9_]*   { yylval.strval = strdup(yytext);
                          return NAME; }

`[^`/\\.\n]+`           { yylval.strval = strdup(yytext+1);
                          yylval.strval[yyleng-2] = 0;
                          return NAME; }
```



```
        /* user variables */
@[0-9a-z_.$]+ |
@\"[^"\n]+\" |
@`[^`\n]+` |
@'[^'\n]+' { yylval.strval = strdup(yytext+1); return USERVAR; }

@\"[^"\n]*$ |
@`[^`\n]*$ |
@'[^'\n]*$ { yyerror("unterminated quoted user variable %s", yytext); }
```

Standard SQL has a small fixed list of functions whose names are effectively keywords,

context, by peeking at the next character with `input()` and `unput()`. This is less elegant than a trailing context pattern but has the advantage that your code can decide at runtime whether to do the test and what token to report back.

Names start with a letter and are composed of letters, digits, and underscores. The pattern to match them has to follow all of the reserved words, so the reserved word patterns take precedence. When a name is recognized, the scanner returns a copy of it. Names can also be quoted in backticks, which allow arbitrary characters in names. The scanner returns a quoted name the same way as an unquoted one, stripping off the backticks. The next pattern catches a missing close backtick, by matching a string that starts with a backtick, and runs to the end of the line.

User variables are a MySQL extension to standard SQL and are variables that are part of a user's session rather than part of a database. Their names start with an `@` sign and can use any of three quoting techniques to include arbitrary characters.

We also have some patterns to catch unclosed quoted user variable names. They end with `\n` rather than `$` to avoid a "dangerous trailing context" warning from flex; a `$` at the end of a pattern is equivalent to `/\n`, and multiple patterns with trailing context that share an action turn out to be inefficient to handle. In this case, since we didn't have any other plans for the `\n`, we just make the pattern match the newline, but if that were a problem, the alternative would just be to copy the action code separately for each of the three patterns.

Comments and Miscellany


```
#.*                ;
"--"[ \t].*        ;

"/*"               { oldstate = YY_START; BEGIN COMMENT; }
<COMMENT>"*/"      { BEGIN oldstate; }
<COMMENT>.|\n      ;
<COMMENT><<EOF>> { yyerror("unclosed comment"); }

        /* everything else */
[ \t\n]            /* whitespace */
.                  { yyerror("mystery character '%c'", *yytext); }



%%
```

```
%%
```


end of the input file.

The Parser

The SQL parser, shown in Example 4-2, is larger than any of the parsers we've seen up to this point, but we can understand it in pieces.

Example 4-2. MySQL subset parser

```
/*
 * Parser for mysql subset
 * $Header: /usr/home/johnl/flnb/RCS/ch04.tr,v 1.7 2009/05/19 18:28:27 johnl
 */
%{
#include <stdlib.h>
#include <stdarg.h>
#include <string.h>

void yyerror(char *s, ...);
void emit(char *s, ...);
%}

%union {
        int intval;
        double floatval;
        char *strval;
        int subtok;
}
```


one for `yyerror()`, which is the same as in Chapter 3, and one for `emit()`, the routine used to emit the RPN code, which takes a `printf`-style format string and arguments.

The `%union` has four members, all of which we met in the lexer: integer and float numeric values, a pointer to copies of strings, and `subtok` for tokens that have subtypes. Since `intval` and `subtok` are both integers, the parser would work just as well if we'd used a single field for both, but separating them helps document the two different purposes, numeric value and subtype, that the token value is used for.

```
        /* names and literal values */
```

```
%token <strval> NAME
%token <strval> STRING
```



```
%token <strval> USERVAR

       /* operators and precedence levels */

%right ASSIGN

%left OR
%left XOR
%left ANDOP
%nonassoc IN IS LIKE REGEXP
%left NOT '!'
%left BETWEEN
%left <subtok> COMPARISON /* = <> < > <= >= <=> */
%left '|'
%left '&'
%left <subtok> SHIFT /* << >> */
%left '+' '-'
%left '*' '/' '%' MOD
%left '^'
%nonassoc UMINUS
```

Next come token declarations, matching the tokens used in the lexer. Like C, MySQL has a dauntingly large number of precedence levels, but bison has no trouble handling them if you can define them. The `COMPARISON` and `SHIFT` tokens are both declared here to have `subtok` values where the lexer returns the particular operator or shift direction.

Next comes a long list of reserved words. Some of these are duplicates of tokens already defined. Bison doesn't object to duplicate token declarations, and it's convenient to have one master alphabetical list of all the reserved word tokens. The full list of tokens is in the cross-reference in the Appendix A, so here we just show a representative part

of the list. Note the special definition of `EXISTS`, which can correspond to `EXISTS` or `NOT EXISTS` when the lexer reads the input.

```
%token ADD
%token ALL
 ...
%token ESCAPED
%token <subtok> EXISTS /* NOT EXISTS or EXISTS */
 ...
 /* functions with special syntax */
%token FSUBSTRING
%token FTRIM
%token FDATE_ADD FDATE_SUB
%token FCOUNT
```

```
%token FCOUNT
```


the way we generate the RPN code in the parser, these values are either bitmasks where a nonterminal matches a set of options or else a count where the nonterminal matches a list of items of variable length.

```
%type <intval> select_opts select_expr_list
%type <intval> val_list opt_val_list case_list
%type <intval> groupby_list opt_with_rollup opt_asc_desc
%type <intval> table_references opt_inner_cross opt_outer
%type <intval> left_or_right opt_left_or_right_outer column_list
%type <intval> index_list opt_for_join

%type <intval> delete_opts delete_list
%type <intval> insert_opts insert_vals insert_vals_list
%type <intval> insert_asgn_list opt_if_not_exists update_opts update_asgn_li
%type <intval> opt_temporary opt_length opt_binary opt_uz enum_list
%type <intval> column_atts data_type opt_ignore_replace create_col_list

%start stmt_list

%%
```

The Top-Level Parsing Rules

```
stmt_list: stmt ';'
  | stmt_list stmt ';'
  ;
```

The top level is just a list of statements with each terminated by a semicolon, roughly

define an alternative, or several alternatives, for stmt.

SQL Expressions

Before we define the syntax for specific statements, we'll define the syntax of MySQL expressions, which are an extended version of the expressions familiar from languages like C and Fortran.

```
   /**** expressions ****/

expr: NAME          { emit("NAME %s", $1); free($1); }
```

```
    | NAME '.' NAME { emit("FIELDNAME %s.%s", $1, $3); free($1); free($3); }
```


The simplest expressions are variable names and constants. Since a name in a SQL expression is usually a column name in a table, a name can also be qualified as `table.name` if there are several tables in the statement that use the same field name, which is quite common when the fields are used with common values to link tables together. Other simple expressions are user variables starting with an `@` sign (dealt with in the lexer and not visible here) and constant strings, fixed and floating numbers, and boolean values. In each case, the code just emits an RPN statement for the item. For the items returned from the lexer as strings, it then frees the string created by the lexer to avoid storage leaks. In a more realistic parser, names would probably be entered into a symbol table rather than passed around as strings.

```
expr: expr '+' expr { emit("ADD"); }
    | expr '-' expr { emit("SUB"); }
    | expr '*' expr { emit("MUL"); }
    | expr '/' expr { emit("DIV"); }
    | expr '%' expr { emit("MOD"); }
    | expr MOD expr { emit("MOD"); }
    | '-' expr %prec UMINUS { emit("NEG"); }
    | expr ANDOP expr { emit("AND"); }
    | expr OR expr { emit("OR"); }
    | expr XOR expr { emit("XOR"); }
    | expr '|' expr { emit("BITOR"); }
    | expr '&' expr { emit("BITAND"); }
    | expr '^' expr { emit("BITXOR"); }
    | expr SHIFT expr { emit("SHIFT %s", $2==1?"left":"right"); }
    | NOT expr { emit("NOT"); }
    | '!' expr { emit("NOT"); }
```



```
       /* recursive selects and comparisons thereto */
    | expr COMPARISON '(' select_stmt ')' { emit("CMPSELECT %d", $2); }
    | expr COMPARISON ANY '(' select_stmt ')' { emit("CMPANYSELECT %d", $2);
    | expr COMPARISON SOME '(' select_stmt ')' { emit("CMPANYSELECT %d", $2);
    | expr COMPARISON ALL '(' select_stmt ')' { emit("CMPALLSELECT %d", $2);
    ;


expr:  expr IS NULLX     { emit("ISNULL"); }
    |  expr IS NOT NULLX { emit("ISNULL"); emit("NOT"); }
    |  expr IS BOOL      { emit("ISBOOL %d", $3); }
    |  expr IS NOT BOOL  { emit("ISBOOL %d", $4); emit("NOT"); }
```

```
    | USERVAR ASSIGN expr { emit("ASSIGN @%s", $1); free($1); }
```


Unary and binary expressions are straightforward and just emit the code for the appropriate operator. Comparisons also emit a subcode to tell what kind of comparison to do. (The subcodes are bit-encoded, where 1 means less than, 2 means greater than, and 4 means equal.)

SQL permits recursive `SELECT` statements where an internal `SELECT` returns a list of values that an external condition checks. If the internal `SELECT` can return multiple values, it can check whether `ANY` or `ALL`/`SOME` of the comparisons succeed. Although this can produce very complex statements, parsing it is simple since it just refers to `select_stmt`, defined later, for the internal `SELECT`. The RPN code emitted is the code for the expression to compare, then the code for the `SELECT`, and then an operator `CMPSELECT`, `CMPANYSELECT`, or `CMPALLSELECT` to say that this is a comparison of the preceding expression and `SELECT`.

SQL has some postfix operators including `IS NULL`, `IS TRUE`, and `IS FALSE`, as well as negated versions of them such as `IS NOT FALSE`. (Remember that `BOOL` is a boolean constant, `TRUE`, `FALSE`, or `UNKNOWN`.) Rather than coming up with RPN codes for the negated versions, we just emit a `NOT` operator to reverse the result of the test.

Next comes a MySQL extension to standard SQL: Internal assignments to user variables. These use a `:=` assignment operator, returned from the lexer as an `ASSIGN` token, to avoid ambiguity with the equality comparison operator.

The syntactically unusual `BETWEEN ... AND` operator tests a value against two limits. It needed a lexical hack, described earlier, because of the ambiguity between the `AND` in this operator and the logical operation `AND`. (Like all hacks, this one isn't totally satisfactory, but it will do.) Since bison's precedence rules normally use the precedence of the rightmost token in a rule, we need a `%prec` to tell it to use `BETWEEN`'s precedence.

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

opt_val_list: /* nil */ { $$ = 0 }
    | val_list
    ;

expr: expr IN '(' val_list ')'        { emit("ISIN %d", $4); }
    | expr NOT IN '(' val_list ')'    { emit("ISIN %d", $5); emit("NOT"); }
    | expr IN '(' select_stmt ')'     { emit("CMPANYSELECT 4"); }
```

```
    | expr NOT IN '(' select_stmt ')' { emit("CMPALLSELECT 3"); }
```


or val_lists in the MySQL manual). In Chapter 3 we built trees to manage multiple expressions, but RPN makes the job considerably easier. Since an RPN interpreter evaluates each RPN value onto its internal stack, an operator that takes multiple values needs only to know how many values to pop off the stack. In our RPN code, such operators include an expression count.

This means the bison rules to parse the variable-length lists need only maintain a count of how many expressions they've parsed, which we keep as the value of the list's LHS symbol, in this case `val_list`. A single element list has length 1, and at each stage, a multi-element list has one more element than its sublist. There are some constructs where the list of values is optional, so an `opt_val_list` is either empty, with a count value of zero, or a `val_list` with a count value of whatever the `val_list` had. (Remember the default action `$$ = $1` for rules with no explicit action.)

Once we have the lists, we can parse the `IN` and `NOT IN` operators that test whether an expression is or isn't in a list of values. Note that the emitted code includes the count of values. SQL also has a variant form where the values come from a `SELECT` statement. For these statements, `IN` and `NOT IN` are equivalent to `= ANY` and `!= ALL`, so we emit the same code.

Functions

SQL has a limited set of functions that MySQL greatly extends. Parsing normal function calls is very simple, since we can use the `opt_val_list` rule and the RPN is `CALL` with the number of arguments, but the parsing is made much more complex by several functions that have their own quirky optional syntax.

```
  /* regular functions */
expr: NAME '(' opt_val_list ')' {  emit("CALL %d %s", $3, $1); free($1); }
   ;

  /* functions with special syntax */
expr: FCOUNT '(' '*' ')' { emit("COUNTALL") }
   | FCOUNT '(' expr ')' { emit(" CALL 1 COUNT"); }

expr: FSUBSTRING '(' val_list ')'              {  emit("CALL %d SUBSTR", $3
   | FSUBSTRING '(' expr FROM expr ')'         {  emit("CALL 2 SUBSTR"); }
   | FSUBSTRING '(' expr FROM expr FOR expr ')' {  emit("CALL 3 SUBSTR"); }

   | FTRIM '(' val_list ')'                    { emit("CALL %d TRIM", $3);
   | FTRIM '(' trim_ltb expr FROM val_list ')'  { emit("CALL 3 TRIM"); }
```

```
    ;
```



```
expr: FDATE_ADD '(' expr ',' interval_exp ')' { emit("CALL 3 DATE_ADD"); }
   |  FDATE_SUB '(' expr ',' interval_exp ')' { emit("CALL 3 DATE_SUB"); }
   ;

interval_exp: INTERVAL expr DAY_HOUR { emit("NUMBER 1"); }
   | INTERVAL expr DAY_MICROSECOND   { emit("NUMBER 2"); }
   | INTERVAL expr DAY_MINUTE        { emit("NUMBER 3"); }
   | INTERVAL expr DAY_SECOND        { emit("NUMBER 4"); }
   | INTERVAL expr YEAR_MONTH        { emit("NUMBER 5"); }
   | INTERVAL expr YEAR              { emit("NUMBER 6"); }
   | INTERVAL expr HOUR_MICROSECOND  { emit("NUMBER 7"); }
   | INTERVAL expr HOUR_MINUTE       { emit("NUMBER 8"); }
   | INTERVAL expr HOUR_SECOND       { emit("NUMBER 9"); }
   ;
```

We handle five functions with special syntax here, `COUNT`, `SUBSTRING`, `TRIM`, `DATE_ADD`, and `DATE_SUB`. `COUNT` has a special form, `COUNT(*)`, used to efficiently count the number of records returned by a `SELECT` statement, as well as a normal form that counts the number of different values of an expression. We have one rule for the special form, which emits a special `COUNTALL` operator, and a second rule for the regular form, which emits a regular function call. `SUBSTRING` is a normal substring operator taking the original string, where to start, and how many characters to take. It can either use the regular call syntax or use reserved words `FROM` and `FOR` to delimit the arguments. There's a rule for each form, all generating similar code since it's the same two or three arguments. `TRIM` similarly can use normal syntax or special syntax like `TRIM(LEADING 'x' FROM a)`. Again, we parse each form and generate rules. The keywords `LEADING`, `TRAILING`, and `BOTH` turn into the integer values 1 through 3 passed as the first argument in a three-argument form. `DATE_ADD` and `DATE_SUB` add or subtract a scaled number of time periods to a date. The special syntax accepts a long list of scaling types, which again turn into integers passed to the functions.

Bison really shines when handling this kind of complex syntax for two reasons: one is that you can generally just write down rules like these as you need and they'll work, but more important, since bison will diagnose any ambiguous grammar, you know that if it doesn't report conflicts, you haven't accidentally broken some other part of the parser.

Other expressions

We wrap up the expression grammar with a grab bag of special cases.

```
expr: CASE expr case_list END           { emit("CASEVAL %d 0", $3); }
   |  CASE expr case_list ELSE expr END { emit("CASEVAL %d 1", $3); }
```



```
    ;

expr: expr LIKE expr { emit("LIKE"); }
   | expr NOT LIKE expr { emit("LIKE"); emit("NOT"); }
   ;

expr: expr REGEXP expr { emit("REGEXP"); }
   | expr NOT REGEXP expr { emit("REGEXP"); emit("NOT"); }
   ;

expr: CURRENT_TIMESTAMP { emit("NOW") };
   | CURRENT_DATE       { emit("NOW") };
   | CURRENT_TIME       { emit("NOW") };
   ;

expr: BINARY expr %prec UMINUS { emit("STRTOBIN"); }
   ;
```

The `CASE` statement comes in two forms. In the first, `CASE` is followed by a value that is compared against a list of test values with an expression value for each test, and an optional `ELSE` default, as in `CASE a WHEN 100 THEN 1 WHEN 200 THEN 2 ELSE 3 END`. The other is just a list of conditional expressions, as in `CASE WHEN a=100 THEN 1 WHEN a=200 THEN 2 END`. We have a rule `case_list` that builds up a list of `WHEN`/`THEN` expression pairs and then uses it in four variants of `CASE`, each of the two versions with and without `ELSE`. The RPN is `CASEVAL` or `CASE` for the versions with or without an initial value, with a count of `WHEN`/`THEN` pairs and 1 or 0 if there's an `ELSE` value. The `LIKE` and `REGEXP` operators do forms of pattern matching. They're basically binary operators except that they permit a preceding `NOT` to reverse the sense of the test. Finally, there are three versions of the keyword for the current time, as well as a unary `BINARY` operator that coerces an expression to be treated as binary rather than text data.

Select Statements

By far the most complex statement in SQL is `SELECT`, which retrieves data from SQL tables and summaries and manipulates it. We deal with it first because it will use several subrules that we can reuse when parsing other statements.

```
    /* statements: select statement */

stmt: select_stmt { emit("STMT"); }
    ;
```

```
select_stmt: SELECT select_opts select_expr_list     simple select with no tables
```



```
;
```

The first rule says that a `select_stmt` is a kind of statement, and it emits an RPN `STMT` as a delimiter between statements. The syntax of `SELECT` lists the expressions that SQL needs to calculate for each record (aka tuple) it retrieves, lists an optional (but usual) `FROM` with the tables containing the data for the expressions, and lists optional qualifiers such as `WHERE`, `GROUP BY`, and `HAVING` that limit, combine, and sort the records retrieved. Each qualifier has its own rules.

```
opt_where: /* nil */
   | WHERE expr { emit("WHERE"); };

opt_groupby: /* nil */
   | GROUP BY groupby_list opt_with_rollup
                              { emit("GROUPBYLIST %d %d", $3, $4); }
;

groupby_list: expr opt_asc_desc
                              { emit("GROUPBY %d",  $2); $$ = 1; }
   | groupby_list ',' expr opt_asc_desc
                              { emit("GROUPBY %d",  $4); $$ = $1 + 1; }
   ;

opt_asc_desc: /* nil */ { $$ = 0; }
   | ASC                { $$ = 0; }
   | DESC               { $$ = 1; }
    ;

opt_with_rollup: /* nil */  { $$ = 0; }
   | WITH ROLLUP  { $$ = 1; }
   ;

opt_having: /* nil */
   | HAVING expr { emit("HAVING"); };

opt_orderby: /* nil */
   | ORDER BY groupby_list { emit("ORDERBY %d", $3); }
   ;

opt_limit: /* nil */ | LIMIT expr { emit("LIMIT 1"); }
  | LIMIT expr ',' expr             { emit("LIMIT 2"); }
  ;

opt_into_list: /* nil */
```

```
    | INTO column_list { emit("INTO %d", $2); }
```


Some of the options, `WHERE`, `GROUPBY`, and `HAVING`, take a fixed number of expressions, while `LIMIT` takes either one or two expressions. These each have straightforward rules to match the option and its expression(s), and they emit an RPN operator to say what to do with the expressions.

`GROUP BY` and `ORDER BY` take a list of expressions, usually column names, each optionally followed by `ASC` or `DESC` to set the sort order. The `groupby_list` rule makes a counted list of expressions, emitting a `GROUPBY` operator with an operand for the sort order. The `GROUP BY` and `ORDER BY` rules then emit `GROUPBYLIST` and `ORDERBY` operators with the count and, for `GROUP BY`, a flag to say whether to use the `WITH ROLLUP` option, which adds some extra summary fields to the result.

The `INTO` operator takes a plain list of names, which we call a `column_list`, that is a list of field names into which to store the selected data. `INTO` isn't used very often, but we'll reuse `column_list` several other places later where the syntax has a list of column names.

Select options and table references

Now we handle the initial options and the main list of expressions in a `SELECT`.

```
select_opts:                          { $$ = 0; }
| select_opts ALL
   { if($1 & 01) yyerror("duplicate ALL option"); $$ = $1 | 01; }
| select_opts DISTINCT
   { if($1 & 02) yyerror("duplicate DISTINCT option"); $$ = $1 | 02; }
| select_opts DISTINCTROW
   { if($1 & 04) yyerror("duplicate DISTINCTROW option"); $$ = $1 | 04; }
| select_opts HIGH_PRIORITY
   { if($1 & 010) yyerror("duplicate HIGH_PRIORITY option"); $$ = $1 | 010;
| select_opts STRAIGHT_JOIN
   { if($1 & 020) yyerror("duplicate STRAIGHT_JOIN option"); $$ = $1 | 020;
| select_opts SQL_SMALL_RESULT
   { if($1 & 040) yyerror("duplicate SQL_SMALL_RESULT option"); $$ = $1 | 04
| select_opts SQL_BIG_RESULT
   { if($1 & 0100) yyerror("duplicate SQL_BIG_RESULT option"); $$ = $1 | 010
| select_opts SQL_CALC_FOUND_ROWS
   { if($1 & 0200) yyerror("duplicate SQL_CALC_FOUND_ROWS option"); $$ =
   $1 | 0200; }
    ;
```

```
select_expr_list: select_expr { $$ = 1; }
```



```
opt_as_alias: AS NAME { emit ("ALIAS %s", $2); free($2); }
  | NAME              { emit ("ALIAS %s", $1); free($1); }
  | /* nil */
  ;
```

The options are flags that affect the way that a SELECT is handled. The rules about what options are compatible with each other are too complex to encode into the grammar, so we just accept any set of options and build up a bitmask of them, which also lets us diagnose duplicate options. (When options can occur in any order, there's no good way to prevent duplicates in the grammar, and it's generally easy to detect them yourself as we do here.)

The SELECT expression list is a comma-separated list of expressions, each optionally followed by an AS clause to give the expression a name to use to refer to it elsewhere in the SELECT statement. We emit an ALIAS operator in the RPN. As a special case, * means all of the fields in the source records, for which we emit SELECTALL.

SELECT table references

The most complex and powerful part of SELECT, and the most powerful part of SQL, is the way it can refer to multiple tables. In a SELECT, you can tell it to create conceptual joined tables built from data stored in many actual tables, either by explicit joins or by recursive SELECT statements. Since tables can be rather large, there are also ways to give it hints about how to do the joining efficiently.

```
table_references:    table_reference { $$ = 1; }
    | table_references ',' table_reference { $$ = $1 + 1; }
    ;

table_reference:  table_factor
  | join_table
;

table_factor:
    NAME opt_as_alias index_hint { emit("TABLE %s", $1); free($1); }
  | NAME '.' NAME opt_as_alias index_hint { emit("TABLE %s.%s", $1, $3);
                              free($1); free($3); }
  | table_subquery opt_as NAME { emit("SUBQUERYAS %s", $3); free($3); }
  | '(' table_references ')' { emit("TABLEREFERENCES %d", $2); }
  ;
```

```
opt_as: AS
```



```
  | table_reference STRAIGHT_JOIN table_factor
                  { emit("JOIN %d", 200); }
  | table_reference STRAIGHT_JOIN table_factor ON expr
                  { emit("JOIN %d", 200); }
  | table_reference left_or_right opt_outer JOIN table_factor join_condition
                  { emit("JOIN %d", 300+$2+$3); }
  | table_reference NATURAL opt_left_or_right_outer JOIN table_factor
                  { emit("JOIN %d", 400+$3); }
  ;

opt_inner_cross: /* nil */ { $$ = 0; }
   | INNER { $$ = 1; }
   | CROSS  { $$ = 2; }
;

opt_outer: /* nil */  { $$ = 0; }
   | OUTER {$$ = 4; }
   ;

left_or_right: LEFT { $$ = 1; }
    | RIGHT { $$ = 2; }
    ;

opt_left_or_right_outer: LEFT opt_outer { $$ = 1 + $2; }
   | RIGHT opt_outer  { $$ = 2 + $2; }
   | /* nil */ { $$ = 0; }
   ;

opt_join_condition: /* nil */
   | join_condition ;

join_condition:
    ON expr { emit("ONEXPR"); }
    | USING '(' column_list ')' { emit("USING %d", $3); }
    ;

index_hint:
     USE KEY opt_for_join '(' index_list ')'
                  { emit("INDEXHINT %d %d", $5, 10+$3); }
   | IGNORE KEY opt_for_join '(' index_list ')'
                  { emit("INDEXHINT %d %d", $5, 20+$3); }
   | FORCE KEY opt_for_join '(' index_list ')'
                  { emit("INDEXHINT %d %d", $5, 30+$3); }
   | /* nil */
   ;

opt_for_join: FOR JOIN { $$ = 1; }
   | /* nil */ { $$ = 0; }
```

```
        ;
```



```
        ;
```

Although the grammar for the table sublanguage is long, it's not all that complex, consisting mostly of lists of items and a lot of optional clauses. Each `table_reference` can be a `table_factor` (which is a plain table, a nested `SELECT`, or a parenthesized list) or else a `join_table`, an explicit join. A plain table reference is the name of the table, with or without the name of the database that contains it; an optional `AS` clause to give an alias name (a table can usefully appear more than once in the same `SELECT`, and this makes it possible to tell which instance an expression refers to); and an optional hint about which indexes to use, described in a moment.

A nested `SELECT` is a `SELECT` statement in parentheses, which must have a name assigned, although the `AS` before the name is optional. A `table_factor` can also be a parenthesized list of `table_references`, which can be useful when creating joins.

Each `table_factor` can also take an index hint. A SQL table can have indexes on any combination of fields, which makes it faster to do searches based on those fields. Each index has a name, typically something like `foo_index` for a field `foo`. Normally MySQL uses the appropriate indexes automatically, but you can also override its choice of indexes by `USE KEY`, `FORCE KEY`, or `IGNORE KEY`.

A join specifies the way to combine two groups of tables. Joins come in a variety of flavors that change the order in which the table are matched up, specify what to do with records in one group that don't match any records in the other group, and specify other details. Every join also explicitly or implicitly specifies the fields to use to match up the tables, in a variety of syntaxes, for example:

```
SELECT * FROM a JOIN b on a.foo=b.bar
SELECT * FROM a JOIN b USING (foo) a.foo=b.foo
```

In a `NATURAL` join, the join matches on fields with the same name, and in a regular join, if there are no fields listed, it creates a cross-product, joining every record in the first group with every record in the second group. In this latter case, the result is usually whittled down by a `WHERE` or `HAVING` clause. For all the various sorts of joins, we emit a `JOIN` operator with subfields describing the exact kind of join.

Note the separate rules `table_factor` and `table_reference`. They're separate to set the

want. In this case, we could have made everything a `table_reference` and used precedence to resolve the ambiguity, but this syntax comes directly from the SQL standard, and there seemed to be no reason to change it.

Delete Statement

Once we have the `SELECT` statement under control, the other data manipulation statements are easy to parse. `DELETE` deletes records from a table, with the records to delete chosen using a `WHERE` clause identical to the `WHERE` clause in a `SELECT` or chosen from a group of tables also specified the same as in a `SELECT`.

```
   /* statements: delete statement */

stmt: delete_stmt { emit("STMT"); }
   ;

   /* single table delete */
delete_stmt: DELETE delete_opts FROM NAME
    opt_where opt_orderby opt_limit
                  { emit("DELETEONE %d %s", $2, $4); free($4); }
;

delete_opts: delete_opts LOW_PRIORITY { $$ = $1 + 01; }
   | delete_opts QUICK { $$ = $1 + 02; }
   | delete_opts IGNORE { $$ = $1 + 04; }
   | /* nil */ { $$ = 0; }
   ;
```

The `DELETE` statement reuses several rules we wrote for `SELECT`: `opt_where` for an optional `WHERE` clause, `opt_orderby` for an optional `ORDER BY` clause, and `opt_limit` for an optional `LIMIT` clause. Since the rules for each of those clauses emits its own RPN, we only have to write rules for some keywords specific to `DELETE`, `QUICK`, and `IGNORE`, and for the `DELETE` statement itself.

```
   /* multitable delete, first version */
delete_stmt: DELETE delete_opts
    delete_list
    FROM table_references opt_where
            { emit("DELETEMULTI %d %d %d", $2, $3, $5); }

delete_list: NAME opt_dot_star { emit("TABLE %s", $1); free($1); $$ = 1; }
```

```
    | delete_list ',' NAME opt_dot_star
```



```
delete_stmt: DELETE delete_opts
     FROM delete_list
     USING table_references opt_where
               { emit("DELETEMULTI %d %d %d", $2, $4, $6); }
;
```

There are two different syntaxes for multitable `DELETE`s, to be compatible with various other implementations of SQL. One lists the tables followed by `FROM` and the `table_references`; the other says `FROM`, the list of tables, `USING`, and the `table_references`. Bison deals easily with these variants, and we emit the same RPN for both. The `delete_list` has a little optional "syntactic sugar," letting you specify the table from which records are to be deleted as `name.*`, as well as plain `name`, to remind readers that all of the fields in each record are deleted.

Insert and Replace Statements

The `INSERT` and `REPLACE` statements add records to a table. The only difference between them is that if the primary key fields in a new record have the same values as an existing record, `INSERT` fails with an error unless there's an `ON DUPLICATE KEY` clause, while `REPLACE` replaces the existing record. `INSERT`, like `DELETE`, has two equivalent variant forms to insert new data, and it has a third form that inserts records created by a `SELECT`.

INSERT INTO a(b,c) values (1,2),(3,DEFAULT)

```
   /* statements: insert statement */

stmt: insert_stmt { emit("STMT"); }
   ;

insert_stmt: INSERT insert_opts opt_into NAME
     opt_col_names
     VALUES insert_vals_list
     opt_ondupupdate { emit("INSERTVALS %d %d %s", $2, $7, $4); free($4) }
   ;

opt_ondupupdate: /* nil */
   | ONDUPLICATE KEY UPDATE insert_asgn_list { emit("DUPUPDATE %d", $4); }
   ;

insert_opts: /* nil */ { $$ = 0; }
```

```
    | insert_opts LOW_PRIORITY { $$ = $1 | 01 ; }
```



```
    ;

opt_col_names: /* nil */
    | '(' column_list ')' { emit("INSERTCOLS %d", $2); }
    ;

insert_vals_list: '(' insert_vals ')' { emit("VALUES %d", $2); $$ = 1; }
    | insert_vals_list ',' '(' insert_vals ')' { emit("VALUES %d", $4); $$ =

insert_vals:
      expr { $$ = 1; }
    | DEFAULT { emit("DEFAULT"); $$ = 1; }
    | insert_vals ',' expr { $$ = $1 + 1; }
    | insert_vals ',' DEFAULT { emit("DEFAULT"); $$ = $1 + 1; }
    ;
```

The first form specifies the name of the table and the list of fields to be provided (all of them if not specified), then specifies `VALUES`, and finally specifies lists of values. This form can insert multiple records, so the rule `insert_vals` matches the fields for one record enclosed in parentheses and `insert_vals_list` matches multiple comma-separated sets of fields. Each field value can be an expression or the keyword `DEFAULT`. There are a few optional keywords to control the details of the insert.

The `opt_ondupupdate` rule handles the `ON DUPLICATE` clause, which gives a list of fields to change if an inserted record would have had a duplicate key. Since the syntax is `SET field=value` and `=` is scanned as a `COMPARISON` operator, we accept `COMPARISON` and check in our code to be sure that it's an equal sign and not something else.[18] Note that `ONDUPLICATE` is one token; in the lexer we treat the two words as one token to avoid ambiguity with `ON` clauses in nested `SELECT`s.

```
      INSERT INTO a SET b=1, c=2

insert_stmt: INSERT insert_opts opt_into NAME
     SET insert_asgn_list
     opt_ondupupdate
      { emit("INSERTASGN %d %d %s", $2, $6, $4); free($4) }
    ;

insert_asgn_list:
      NAME COMPARISON expr
        { if ($2 != 4) { yyerror("bad insert assignment to %s", $1); YYERROR;
        emit("ASSIGN %s", $1); free($1); $$ = 1; }
```

```
    | NAME COMPARISON DEFAULT
```



```
        { if ($4 != 4) { yyerror("bad insert assignment to %s", $1); YYERROR;
                    emit("DEFAULT"); emit("ASSIGN %s", $3); free($3); $$ = $1 +
    ;
```

The second form uses an assignment syntax similar to the one for `ON DUPLICATE`. We have to check that the `COMPARISON` is really an =. If not, we produce an error message by calling `yyerror()`, and then we tell the parser to start error recovery with `YYERROR`. (In this version of the parser there's no error recovery, but see Chapter 8.) This form uses same optional `ON DUPLICATE` syntax at the end of the statement, so we use the same rule.

```
   INSERT into a(b,c) SELECT x,y FROM z where x < 12
insert_stmt: INSERT insert_opts opt_into NAME opt_col_names
    select_stmt
    opt_ondupupdate { emit("INSERTSELECT %d %s", $2, $4); free($4); }
  ;
```

The third form of `INSERT` uses data from a `SELECT` statement to create new records. All of the pieces of this statement are the same as syntax we've seen before, so we write only the one rule and reuse subrules for the pieces.

Replace statement

The syntax of the `REPLACE` statement is just like `INSERT`, so the rules for it are the same too, changing `INSERT` to `REPLACE` and renaming the top-level rules.

```
   /** replace just like insert **/
stmt: replace_stmt { emit("STMT"); }
   ;

replace_stmt: REPLACE insert_opts opt_into NAME
     opt_col_names
     VALUES insert_vals_list
     opt_ondupupdate { emit("REPLACEVALS %d %d %s", $2, $7, $4); free($4) }
   ;

replace_stmt: REPLACE insert_opts opt_into NAME
    SET insert_asgn_list
    opt_ondupupdate
     { emit("REPLACEASGN %d %d %s", $2, $6, $4); free($4) }
   ;
```

Update Statement

The UPDATE statement changes fields in existing records. Again, its syntax lets us reuse rules from previous statements.

```
/** update **/
stmt: update_stmt { emit("STMT"); }
   ;

update_stmt: UPDATE update_opts table_references
    SET update_asgn_list
    opt_where
    opt_orderby
opt_limit { emit("UPDATE %d %d %d", $2, $3, $5); }
;

update_opts: /* nil */ { $$ = 0; }
   | insert_opts LOW_PRIORITY { $$ = $1 | 01 ; }
   | insert_opts IGNORE { $$ = $1 | 010 ; }
   ;

update_asgn_list:
     NAME COMPARISON expr
     { if ($2 != 4) { yyerror("bad update assignment to %s", $1); YYERROR; }
         emit("ASSIGN %s", $1); free($1); $$ = 1; }
   | NAME '.' NAME COMPARISON expr
       { if ($4 != 4) { yyerror("bad update assignment to %s", $1); YYERROR;
         emit("ASSIGN %s.%s", $1, $3); free($1); free($3); $$ = 1; }
   | update_asgn_list ',' NAME COMPARISON expr
       { if ($4 != 4) { yyerror("bad update assignment to %s", $3); YYERROR;
         emit("ASSIGN %s.%s", $3); free($3); $$ = $1 + 1; }
   | update_asgn_list ',' NAME '.' NAME COMPARISON expr
       { if ($6 != 4) { yyerror("bad update  assignment to %s.$s", $3, $5);
          YYERROR; }
         emit("ASSIGN %s.%s", $3, $5); free($3); free($5); $$ = 1; }
   ;
```

UPDATE has its own set of options in the update_opts rule. The list of assignments after SET is similar to the one in INSERT, but it allows qualified table names since you can update more than one table at a time, and it doesn't have the default option in INSERT, so we have a similar but different update_asgn_list. INSERT uses the same opt_where and opt_orderby to limit and sort the records updated.

This ends the list of data manipulation statements in our SQL subset. MySQL has

structure of databases and tables.

```
    /** create database **/

stmt: create_database_stmt { emit("STMT"); }
    ;

create_database_stmt:
      CREATE DATABASE opt_if_not_exists NAME
        { emit("CREATEDATABASE %d %s", $3, $4); free($4); }
    | CREATE SCHEMA opt_if_not_exists NAME
        { emit("CREATEDATABASE %d %s", $3, $4); free($4); }
    ;

opt_if_not_exists:  /* nil */ { $$ = 0; }
    | IF EXISTS
        { if(!$2) { yyerror("IF EXISTS doesn't exist"); YYERROR; }
                          $$ = $2; /* NOT EXISTS hack */ }
    ;
```

`CREATE DATABASE`, or the equivalent `CREATE SCHEMA` statement, makes a new database in which you can then create tables. It has one optional clause, `IF NOT EXISTS`, to prevent an error message if the database already exists. Recall that we did a lexical hack to treat `IF NOT EXISTS` and `IF EXISTS` as the same token in expressions. In this case, only `IF NOT EXISTS` is valid, so we test in the action code and complain and tell the parser it's a syntax error if it's the wrong one.

Create Table

The `CREATE TABLE` statement rivals `SELECT` in its length and number of options, but its syntax is much simpler since nearly all of the syntax is just declaring the type and attribute of each column in the table.

We start with six versions of `create_table_statement`. There are three pairs that differ only in `NAME` or `NAME.NAME` for the name of the table. The first pair is the normal version with an explicit list of columns in `create_col_list`. The other two create and populate a table from a `SELECT` statement, with one including a list of column names and the other defaulting to the column names from the `SELECT`.

```
    /** create table **/
stmt: create_table_stmt { emit("STMT"); }
```

```
    ;
```



```
    '(' create_col_list ')' { emit("CREATE %d %d %d %s.%s", $2, $4, $9, $5, $
                          free($5); free($7); }
    ;

create_table_stmt: CREATE opt_temporary TABLE opt_if_not_exists NAME
    '(' create_col_list ')'
create_select_statement { emit("CREATESELECT %d %d %d %s", $2, $4, $7, $5);
     ;

create_table_stmt: CREATE opt_temporary TABLE opt_if_not_exists NAME
    create_select_statement { emit("CREATESELECT %d %d 0 %s", $2, $4, $5); fr
     ;

create_table_stmt: CREATE opt_temporary TABLE opt_if_not_exists NAME '.' NAM
    '(' create_col_list ')'
    create_select_statement  { emit("CREATESELECT %d %d 0 %s.%s", $2, $4, $5,
                              free($5); free($7); }
     ;

create_table_stmt: CREATE opt_temporary TABLE opt_if_not_exists NAME '.' NAM
    create_select_statement { emit("CREATESELECT %d %d 0 %s.%s", $2, $4, $5,
                          free($5); free($7); }
     ;

opt_temporary:   /* nil */ { $$ = 0; }
    | TEMPORARY { $$ = 1;}
    ;
```

The heart of a `CREATE DATABASE` statement is the list of columns, or more precisely the list of `create_definitions`, which includes both columns and indexes. The indexes can be the `PRIMARY KEY`, which means that it's unique for each record; a regular `INDEX` (also called `KEY`); or a `FULLTEXT` index, which indexes individual words in the data. Each of those takes a list of column names, for which we once again reuse the `column_list` rule we defined for `SELECT`.

```
create_col_list: create_definition { $$ = 1; }
     | create_col_list ',' create_definition { $$ = $1 + 1; }
     ;

create_definition: PRIMARY KEY '(' column_list ')'    { emit("PRIKEY %d", $4
     | KEY '(' column_list ')'             { emit("KEY %d", $3); }
     | INDEX '(' column_list ')'           { emit("KEY %d", $3); }
     | FULLTEXT INDEX '(' column_list ')' { emit("TEXTINDEX %d", $4); }
```

```
      | FULLTEXT KEY '(' column_list ')'    { emit("TEXTINDEX %d", $4); }
```


the column, the data type, and the optional attributes, such as whether the column can contain null values, what its default value is, and whether it's a key. (Declaring a column to be a key is equivalent to creating an index on the column.) For the attributes, we emit an `ATTR` operator for each one and count the number of attributes. The code here doesn't check for duplicates, but we could do so by making the value of `column_atts` a structure with both a count and a bitmask and checking the bitmask as we did earlier in `SELECT` options.

```
create_definition: { emit("STARTCOL"); } NAME data_type column_atts
                   { emit("COLUMNDEF %d %s", $3, $2); free($2); }

column_atts: /* nil */ { $$ = 0; }
    | column_atts NOT NULLX             { emit("ATTR NOTNULL"); $$ = $1 + 1;
    | column_atts NULLX
    | column_atts DEFAULT STRING
        { emit("ATTR DEFAULT STRING %s", $3); free($3); $$ = $1 + 1; }
    | column_atts DEFAULT INTNUM
        { emit("ATTR DEFAULT NUMBER %d", $3); $$ = $1 + 1; }
    | column_atts DEFAULT APPROXNUM
        { emit("ATTR DEFAULT FLOAT %g", $3); $$ = $1 + 1; }
    | column_atts DEFAULT BOOL
        { emit("ATTR DEFAULT BOOL %d", $3); $$ = $1 + 1; }
    | column_atts AUTO_INCREMENT
        { emit("ATTR AUTOINC"); $$ = $1 + 1; }
    | column_atts UNIQUE '(' column_list ')'
        { emit("ATTR UNIQUEKEY %d", $4); $$ = $1 + 1; }
    | column_atts UNIQUE KEY { emit("ATTR UNIQUEKEY"); $$ = $1 + 1; }
    | column_atts PRIMARY KEY { emit("ATTR PRIKEY"); $$ = $1 + 1; }
    | column_atts KEY { emit("ATTR PRIKEY"); $$ = $1 + 1; }
    | column_atts COMMENT STRING
        { emit("ATTR COMMENT %s", $3); free($3); $$ = $1 + 1; }
    ;
```

The syntax for the data type is long but not complicated. Many of the types allow the number of characters or digits to be specified, so there's an `opt_length` that takes one or two length values. (We encode them into one number here; a structure would have been more elegant.) Other options say whether a number is unsigned or displayed filled with zeros, whether a string is treated as binary data, and, for text, what character set and collation rule it uses. Those last two are specified as strings from a large set of language and collation systems, but for our purposes we just accept any string. With these auxiliary rules, we can now parse the long list of MySQL data types. Again we encode the data type into a number, and again a structure would be more elegant, but


```
opt_length: /* nil */ { $$ = 0; }
   | '(' INTNUM ')' { $$ = $2; }
   | '(' INTNUM ',' INTNUM ')' { $$ = $2 + 1000*$4; }
   ;

opt_binary: /* nil */ { $$ = 0; }
   | BINARY { $$ = 4000; }
   ;

opt_uz: /* nil */ { $$ = 0; }
   | opt_uz UNSIGNED { $$ = $1 | 1000; }
   | opt_uz ZEROFILL { $$ = $1 | 2000; }
   ;

opt_csc: /* nil */
   | opt_csc CHAR SET STRING { emit("COLCHARSET %s", $4); free($4); }
   | opt_csc COLLATE STRING { emit("COLCOLLATE %s", $3); free($3); }
   ;

data_type:
     BIT opt_length { $$ = 10000 + $2; }
   | TINYINT opt_length opt_uz { $$ = 10000 + $2; }
   | SMALLINT opt_length opt_uz { $$ = 20000 + $2 + $3; }
   | MEDIUMINT opt_length opt_uz { $$ = 30000 + $2 + $3; }
   | INT opt_length opt_uz { $$ = 40000 + $2 + $3; }
   | INTEGER opt_length opt_uz { $$ = 50000 + $2 + $3; }
   | BIGINT opt_length opt_uz { $$ = 60000 + $2 + $3; }
   | REAL opt_length opt_uz { $$ = 70000 + $2 + $3; }
   | DOUBLE opt_length opt_uz { $$ = 80000 + $2 + $3; }
   | FLOAT opt_length opt_uz { $$ = 90000 + $2 + $3; }
   | DECIMAL opt_length opt_uz { $$ = 110000 + $2 + $3; }
   | DATE { $$ = 100001; }
   | TIME { $$ = 100002; }
   | TIMESTAMP { $$ = 100003; }
   | DATETIME { $$ = 100004; }
   | YEAR { $$ = 100005; }
   | CHAR opt_length opt_csc { $$ = 120000 + $2; }
   | VARCHAR '(' INTNUM ')' opt_csc { $$ = 130000 + $3; }
   | BINARY opt_length { $$ = 140000 + $2; }
   | VARBINARY '(' INTNUM ')' { $$ = 150000 + $3; }
   | TINYBLOB { $$ = 160001; }
   | BLOB { $$ = 160002; }
   | MEDIUMBLOB { $$ = 160003; }
   | LONGBLOB { $$ = 160004; }
   | TINYTEXT opt_binary opt_csc { $$ = 170000 + $2; }
   | TEXT opt_binary opt_csc { $$ = 171000 + $2; }
   | MEDIUMTEXT opt_binary opt_csc { $$ = 172000 + $2; }
```

```
    | LONGTEXT opt_binary opt_csc { $$ = 173000 + $2; }
```



```
    ;
```

The other version of a `CREATE` uses a `SELECT` statement preceded by some optional keywords and an optional meaningless `AS`.

```
create_select_statement: opt_ignore_replace opt_as select_stmt { emit("CREAT
    ;

opt_ignore_replace: /* nil */ { $$ = 0; }
    | IGNORE { $$ = 1; }
    | REPLACE { $$ = 2; }
    ;
```

User Variables

The last statement we parse is a `SET` statement, which is a MySQL extension that sets user variables. The assignment can use either `:=`, which we call `ASSIGN`, or a plain `=` sign, checking as always to be sure it's not some other comparison operator.

```
    /**** set user variables ****/

stmt: set_stmt { emit("STMT"); }
    ;

set_stmt: SET set_list ;

set_list: set_expr | set_list ',' set_expr ;

set_expr:
      USERVAR COMPARISON expr { if ($2 != 4) { yyerror("bad set to @%s", $1)
                 emit("SET %s", $1); free($1); }
     | USERVAR ASSIGN expr { emit("SET %s", $1); free($1); }
     ;
```

That ends our SQL syntax. MySQL has many, many other statements, but these give a reasonable idea of what's involved in parsing them.

The Parser Routines

Finally, we have a few support routines. The `emit` routine just prints out the RPN. In a

debugging a grammar as complex as this one.

```
%%
void
emit(char *s, ...)
{
  extern yylineno;

  va_list ap;
  va_start(ap, s);

  printf("rpn: ");
  vfprintf(stdout, s, ap);
  printf("\n");
}

void
yyerror(char *s, ...)
{
  extern yylineno;

  va_list ap;
  va_start(ap, s);

  fprintf(stderr, "%d: error: ", yylineno);
  vfprintf(stderr, s, ap);
  fprintf(stderr, "\n");
}

main(int ac, char **av)
{
  extern FILE *yyin;

  if(ac > 1 && !strcmp(av[1], "-d")) {

    yydebug = 1; ac--; av++;
  }

  if(ac > 1 && (yyin = fopen(av[1], "r")) == NULL) {
    perror(av[1]);
    exit(1);
  }

  if(!yyparse())
    printf("SQL parse worked\n");
  else
    printf("SQL parse failed\n");
```

```
} /* main */
```


compiles them together. The dependencies take care of generating and compiling the scanner, including the bison-generated header file.

```
# Makefile for pmysql
CC = cc -g
LEX = flex
YACC = bison
CFLAGS = -DYYDEBUG=1

PROGRAMS5 = pmysql

all:    ${PROGRAMS5}

# chapter 5

pmysql: pmysql.tab.o pmysql.o
        ${CC} -o $@ pmysql.tab.o pmysql.o

pmysql.tab.c pmysql.tab.h:      pmysql.y
        ${YACC} -vd pmysql.y

pmysql.c:       pmysql.l
        ${LEX} -o $*.c $<

pmysql.o:       pmysql.c pmysql.tab.h

.SUFFIXES:    .pgm .l .y .c
```

Exercises

1. In several places, the SQL parser accepts more general syntax than SQL itself permits. For example, the parser accepts any expression as the left operand of a `LIKE` predicate, although that operand has to be a column reference. Fix the parser to diagnose these erroneous inputs. You can either change the syntax or add action code to check the expressions. Try both to see which is easier and which gives better diagnostics.

2. Turn the parser into a SQL cross-referencer, which reads a set of SQL statements and produces a report showing for each name where it is defined and where it is referenced.

3. (Term project.) Modify the embedded SQL translator to interface to a real

[17] MySQL actually accepts multiline strings, but we're keeping this example simple.

[18] This could fairly be considered a kludge, but the alternative would be to treat = separately from the other comparison operators and add an extra rule every place a comparison can occur, which would result in more code.