第五章 分析 SQL[1]

本章内容：

SQL 的要点概述
语法检查程序
语法分析程序
嵌入式 SQL
练习

SQL（代表“结构化查询语言”，通常发音为 sequel）是处理关系数据库最常用的语言[2]。首先，开发一个检测输入语法的 SQL 语法分析程序，但是不以它做任何事情。然后将它转换成嵌入在 C 程序中的 SQL 预处理程序。

这个语法分析程序使用 C.J.Date 编写的《A Guide to the SQL Standard》（第二版，Addison-Wesley 出版社 1989 年出版）一书中的 SQL 定义。Date 的描述采用 Backus-Naur 范式（BNF）．它是用于编写正规语言描述的标准形式。除了标点符号以外、yacc 的输入语法类似于 BNF，所以在许多地方，直译 BNF 得到相应的 yacc 规则就足够了。在多数情况下，使用与 Date 相同的符号名，但在有些地方为了符合 yacc 的语法规则而不能这样做。

SQL 的最终定义是标准的文档——ANSI X3.135-1989（它定义了 SQL 本身）和 ANSI X3.168-1989（它定义了在其他程序设计语言中嵌入 SQL 的方式）。

SQL 的要点概述

SQL 是用于关系数据库的专用语言。它处理数据库表中的数据，而不是处理内存中的数据（偶尔才引用内存）。

关系数据库

关系数据库是表的集合，类似于文件。每个表都包含行（row）和列（column），类似于记录和字段。表中的行没有任何特定的顺序。通过给出每列的名字和类型可以创建一组表：

```
CREATE SCHEMA
AUTHORIZATION JOHNL
```

[1] Jobn R.Levine, Tony Mason, Doug Brown 著，杨作梅，张旭东等译). 整理: qy8087@gmail.com
[2] SQL，是数据库的 Fortran，没有人特别喜欢它．这个语言丑陋而且特别，每个数据库都支持它，而且我们都用它。

```
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
)
```

这个语法是完全格式自由的，并且同一件事通常有几种不同的语法编写方式，注意给出的 PRIMARY KEY 说明符的两种不同的方式。（表中的主关键字 Primary key 是惟一指定一行的列或一组列。）图 5-1 展示了装载数据后刚刚创建的两个表。

Foods

name	*type*	*flavor*
peach	*fruit*	*sweet*
tomato	*fruit*	*savory*
lemon	*fruit*	*sour*
lard	*fat*	*bland*
cheddar	*fat*	*savory*

Courses

course	*flavor*	*sequence*
salad	*savory*	*1*
main	*savory*	*2*
dessert	*sweet*	*3*

图 5-1 两个关系表

使用数据库时，必须告诉数据库想要从表中得到什么。数据库算出如何得到它。一组想要的数据的规范说明就是查询（query）。例如，使用图 5-1 中的两个表，要得到水果列表，可以写成：

```
SELECT Foods.name, Foods.flavor
FROM Foods
WHERE Foods.type = “fruit”
```

响应是：

name	*flavor*
peach	*sweet*
tomato	*savory*
lemon	*sour*

也可以提出跨越多个表的问题。为了得到适合每餐的食物列表，可以写成：

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

响应是：

course	*name*	*flavor*	*type*
salad	*tomato*	*savory*	*fruit*
salad	*cheddar*	*savory*	*fat*
main	*tomato*	*savory*	*fruit*

当列出列名时，如果列名是明确的就可以省去表名。

处理关系

SQL 有丰富的表处理命令集合。用 SELECT、INSERT、UPDATE 和 DELETE 命令可以读取并写出独立的行。更常见的是，需要为一组行中的每个行做些事情。在这种情况下，SELECT 的不同变体定义了游标（cursor)，游标是能单步遍历一组行以便一次处理一行的一种文件指针。然后使用 OPEN 和 CLOSE 命令找到相应的行并使用 FETCH、UPDATECURRENT 和 DELETECURRENT 命令对这些行做一些操作。当进行事务处理时 COMMIT 和 ROLLBACK 命令完成或异常中止一组命令。

SELECT 语句有一个非常复杂的语法，可以使你寻找列中的值、比较列、做算术，并计算最小值、最大值、平均值和总和。

使用 SQL 的三种方法

在 SQL 的最初版本中，用户在文件中或直接在终端上键入命令并立即接收响应。有时用户仍然使用这种方式创建表并进行调试，但对大多数的应用，SQL 命令来源于程序内部而且结果被返回给那些程序。

将 SQL 与常规语言结合在一起的第一种途径是 SQL 模块语言，只需要在 SQL 中定义少许程序就可以从常规语言中调用 SQL 模块语言。例 5-1 定义了一个游标和使用它的三个过程。

例 5-1:SQL 模块语言的示例

```
MODULE LANGUAGE C AUTHORIZAIION JOHNL

DECLARE flav CURSOR FOR
    SELECT Foods.name, Foods.type
    FROM Foods
    WHERE Foods.flavor = myflavor
    -- myflavor is defined below

PROCEDURE open_flav
    SQLCODE
    myflavor CHAP(6) ;
    OPEN flav

PROCEDURE close_flav
    SQLCODE ;
    CLOSE flav

PROCEDURE get_flav
    SQLCODE
    myname CHAR(8)
    mytype CHAR(5) ;
    FETCH flav INTO myname, mytype
```

在 C 程序中，通过编写下面的代码可以使用模块中的这些例程

```
char flavor[6], name[8], type[5];

main()
{
    int icode;

    scanf(“%s”, flavor);
    open_flav(&icode, flavor);
    for(;;) {
        get_flav(&icode, name, type);
        if(icode != 0)
            break;
        printf("%8.8s %5.5s\n", name, type);
    }
    close_flav(&icode);
}
```

这种方法可以工作，但是存在使用的瓶颈，因为所编写的每条 SQL 语句都必须

打包到一个小例程。用户真正使用的方法是嵌入式 SQL，这种嵌入式 SQL 允许用户将 SQL 程序块正确地放置在自己的程序中。每条 SQL 语句都由“ EXEC SQL ”引入并且以分号结束。对宿主语言变量的引用前面以冒号引导。例 5-2 展示了以嵌入式 SQL 编写的同样的程序。

例 5-2：嵌入式 SQL 的示例

```
char flavor[6], name[8], type[5];
int SQLCODE; /* global status variable */

EXEC SQL DECLARE flav CURSOR FOR
    SELECT Foods.name, Foods.type
    FROM Foods
    WHERE Foods.flavor = :flavor;

main()
{
    scanf("%s", flavor);
    EXEC SQL OPEN flav;
    for(;;) {
        EXEC SQL FETCH flav INTO :name, :type;
        if(SQLCODE != 0)
            break;
        printf("%8.8s %5.5s\n", name, type);
    }
    EXEC SQL CLOSE flav;
}
```

为了编泽这个程序，用户通过 SQL 预处理程序来运行它，SQL 预处理程序将 SQL 代码转换成对 C 例程的调用，然后编译纯粹的 C 程序。本章后面将编写这样一个预处理程序的简单版本。

语法检查程序

用 yacc 编写语法检查程序很容易。试着分析一些程序，如果分析可以工作，说明语法正确。（如果语法分析程序中有错误校正，那么事情就不会这么简单。详细资料参见第九章。）为 SQL 构建一个语法检查程序，这个任务几乎全部是编写 yacc 语法完成的。因为 SQL 语法如此复杂，所以在附录十中将重新介绍整个 yacc 语法，及对语法中所有符号的交叉引用。

词法分析程序

首先，对于 SQL 使用的标记要使用词法分析程序。语法是自由格式的，忽略空

白（它仅分隔单词）。具有相当长的但却固定的保留字集合。其他的标记是很传统的：名字，字符串、数字和标点符号。注释是 Ada 风格的，从一对短划线开始到行尾。例 5-3 展示了 SQL 词法分析程序。

例 5-3：第一个 SQL 词法分析程序

```
%{
#include "sql1.h"
#include <string.h>

int lineno = 1;
void yyerror(char *s);
%}

%e 1200

%%

   /* 文字关键字标记 */

ADA    { return ADA; }
ALL    { return ALL; }
AND    { return AND; }
AVG    { return AMMSC; }
MIN    { return AMMSC; }
MAX    { return AMMSC; }
SUM    { return AMMSC; }
COUNT  { return AMMSC; }
ANY    { return ANY; }
AS     { return AS; }
ASC    { return ASC; }
AUTHORIZATION{ return AUTHORIZATION; }
BETWEEN      { return BETWEEN; }
BY     { return BY; }
C      { return C; }
CHAR(ACTER)? { return CHARACTER; }
CHECK     { return CHECK; }
CLOSE     { return CLOSE; }
COBOL     { return COBOL; }
COMMIT    { return COMMIT; }
CONTINUE  { return CONTINUE; }
CREATE    { return CREATE; }
CURRENT   { return CURRENT; }
CURSOR    { return CURSOR; }
DECIMAL   { return DECIMAL; }
```

```
DECLARE   { return DECLARE; }
DEFAULT   { return DEFAULT; }
DELETE    { return DELETE; }
DESC      { return DESC; }
DISTINCT  { return DISTINCT; }
DOUBLE    { return DOUBLE; }
ESCAPE    { return ESCAPE; }
EXISTS    { return EXISTS; }
FETCH     { return FETCH; }
FLOAT     { return FLOAT; }
FOR       { return FOR; }
FOREIGN   { return FOREIGN; }
FORTRAN   { return FORTRAN; }
FOUND     { return FOUND; }
FROM      { return FROM; }
GO[ \t]*TO   { return GOTO; }
GRANT     { return GRANT; }
GROUP     { return GROUP; }
HAVING    { return HAVING; }
IN        { return IN; }
INDICATOR { return INDICATOR; }
INSERT    { return INSERT; }
INT(EGER)?   { return INTEGER; }
INTO      { return INTO; }
IS        { return IS; }
KEY       { return KEY; }
LANGUAGE  { return LANGUAGE; }
LIKE      { return LIKE; }
MODULE    { return MODULE; }
NOT       { return NOT; }
NULL      { return NULLX; }
NUMERIC   { return NUMERIC; }
OF        { return OF; }
ON        { return ON; }
OPEN      { return OPEN; }
OPTION    { return OPTION; }
OR        { return OR; }
ORDER     { return ORDER; }
PASCAL    { return PASCAL; }
PLI       { return PLI; }
PRECISION { return PRECISION; }
PRIMARY   { return PRIMARY; }
PRIVILEGES   { return PRIVILEGES; }
PROCEDURE { return PROCEDURE; }
```

```
PUBLIC    { return PUBLIC; }
REAL      { return REAL; }
REFERENCES   { return REFERENCES; }
ROLLBACK  { return ROLLBACK; }
SCHEMA    { return SCHEMA; }
SELECT    { return SELECT; }
SET       { return SET; }
SMALLINT  { return SMALLINT; }
SOME      { return SOME; }
SQLCODE   { return SQLCODE; }
TABLE     { return TABLE; }
TO        { return TO; }
UNION     { return UNION; }
UNIQUE    { return UNIQUE; }
UPDATE    { return UPDATE; }
USER      { return USER; }
VALUES    { return VALUES; }
VIEW      { return VIEW; }
WHENEVER  { return WHENEVER; }
WHERE     { return WHERE; }
WITH      { return WITH; }
WORK      { return WORK; }

   /* 标点符号 */

"="    |
"<>"   |
"<"    |
">"    |
"<="   |
">="      { return COMPARISON; }

[-+*/:(),.;] { return yytext[0]; }

   /* 名称 */

[A-Za-z][A-Za-z0-9_]*  { return NAME; }

   /* 数字 */

[0-9]+ |
[0-9]+"."[0-9]* |
"."[0-9]* { return INTNUM; }
```

```
[0-9]+[eE][+-]?[0-9]+  |
[0-9]+"."[0-9]*[eE][+-]?[0-9]+ |
"."[0-9]*[eE][+-]?[0-9]+   { return APPROXNUM; }

   /* 字符串 */

'[^'\n]*' {
      int c = input();

      unput(c); /* just peeking */
      if(c != '\'') {
         return STRING;
      } else
         yymore();
   }

'[^'\n]*$ { yyerror("Unterminated string"); }

\n     lineno++;

[ \t\r]+  ;  /* 空白 */

"--".*$      ;  /* 注释 */

%%

void
yyerror(char *s)
{
   printf("%d: %s at %s\n", lineno, s, yytext);
}

main(int ac, char **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 */
```

词法分析程序从几个包含文件开始，特别是，*sql1.h*,它是由 yacc 生成的标记名定义文件（重新命名默认的 *y.tab.h*）。所有的保留字在语法分析程序中都为独立的标记，因为这是最容易做的事情。注意 CHARACTER 和 INTEGER 可以缩写成 CHAR 和 INT，而且 GOTO 可以写成一个或两个单词。保留字 AVG、MIN、MAX、SUM 和 COUNT 都转换成 **AMMSC** 标记；在 SQL 预处理程序中，使用标记值记忆它们各是哪个词。

接下来是标点符号标记，包括使用同一个模式匹配所有单个字符操作符的常用技巧。名字以字母开头并且由字母、数字和下划线组成。这种模式必须跟在所有的保留字后面，以保证这些保留字模式的高优先级。

SQL 定义精确的数字（它们具有小数点，但没有明确的指数），也可以定义近似数（它们有指数）。独立的模式可以区别这两种形式。

SQL 字符串包围在单引号内，使用一对引号代表字符串中的单引号。第一种字符串模式匹配引号中不包含嵌入式引号的字符串。它的动作程序使用 **input()**和 **unput()**取得下一个字符并查看它是否为另一个引号（意思是找到两个引号，而不是字符串的结尾）。如果这样的话，它使用 **yymore()**向标记附加另一个引用字符串。下一种模式捕获非终结的字符串并且在它看到时打印诊断结果。

最后几个模式跳过空白（当空白是换行符时计算行数），跳过注释，并且如果输入中出现无效的字符时会发出“ 抱怨 ”。

错误和主程序

yyerror()的这种版本报告当前的行号、当前的标记和错误消息。这个简单的程序通常用于报告有用错误的所有内容。因为它需要引用 **yytext**，并且只在词法分析程序的源文件中 **yytext** 才有定义，所以为了得到最好的可移植性，我们将它放入词法分析程序中。（lex 的不同版本将 **yytext** 定义为一个数组或一个指针，所以在其它地方引用它会破坏程序的可移植性。）

main()例程打开在命令行上命名的文件（若有的话），随后调用语法分析程序。当语法分析程序返回时，返回值报告分析成功还是失败。单独建立一个文件放置 **main()**函数，并声明 **yyin** 为一个外部“ *FILE* * ”，是可以很好地工作的，但我们仍然要在词法分析程序中放置 **main()**，因为 **yyin** 已经在这里定义了。

语法分析程序

SQL 语法分析程序比到目前为止所看到的所有语法分析程序都大，但是我们可以分段理解。

定义

例 5-4 展示了语法分析程序的定义部分

例 5-4：第一个 SQL 语法分析程序的定义部分

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

%token NAME
%token STRING
%token INTNUM APPROXNUM

    /* 操作符 */

%left OR
%left AND
%left NOT
%left <subtok> COMPARISON /* = <> < > <= >= */
%left '+' '-'
%left '*' '/'
%nonassoc UMINUS

    /* 文字关键字标记 */

%token ALL AMMSC ANY AS ASC AUTHORIZATION BETWEEN BY
%token CHARACTER CHECK CLOSE COMMIT CONTINUE CREATE CURRENT
%token CURSOR DECIMAL DECLARE DEFAULT DELETE DESC DISTINCT DOUBLE
%token ESCAPE EXISTS FETCH FLOAT FOR FOREIGN FOUND FROM GOTO
%token GRANT GROUP HAVING IN INDICATOR INSERT INTEGER INTO
%token IS KEY LANGUAGE LIKE MODULE NULLX NUMERIC OF ON
%token OPEN OPTION ORDER PRECISION PRIMARY PRIVILEGES PROCEDURE
%token PUBLIC REAL REFERENCES ROLLBACK SCHEMA SELECT SET
%token SMALLINT SOME SQLCODE SQLERROR TABLE TO UNION
%token UNIQUE UPDATE USER VALUES VIEW WHENEVER WHERE WITH WORK
%token COBOL FORTRAN PASCAL PLI C ADA
```

首先是值 union 类型的定义。字段 **intval** 和 **floatval** 处理整数和浮点数。（实际上，因为 SQL 的精确数字包括小数，后面我们会将所有的数值都存在 **floatval** 中。）字符串以 **strval** 返回，但在这个版本的语法分析程序中，我们还没有这么做。最

后，**subtok** 保存代表多个输入字符的标记的子标记代码，例如，AVG、MIN、MAX、SUM 和 COUNT，但在这个语法检查程序中，我们做也没有这么做。接下来是语法中使用的标记的定义。标记有 NAME、STRING、INTNUM 和 APPROXNUM，所有这些都可以在词法分析程序中看到，**%left** 声明设置所有操作符的优先级和结合规则。为了设置优先级，我们声明需要“ +-*/ ”文字标记。声明伪标记 UMINUS 是为了说明它只在的%prec 子句中使用的。

最后是 SQL 的所有保留字的标记定义。

最高层的规则

例 5-5 展示了语法分析程序的最高层的规则。

例 5-5：在第一个 SQL 语法分析程序中的最高层的规则

```
. . .
/* 模式定义语言 */
/* 说明其他"sql:"规则在后而的语法中出现 */
sql:   schema
       ;
       . . .
       /* 模块语言 */
sql:   module_def
       ;
       . . .
       /* 操纵语句 */

sql:   manipulative_statement
       ;
```

起始规则是 **sql_list**（几条 **sql** 规则），每条 sql 规则都是一种语句。有三种不同的语句：定义表的 **schema** 语句，模块定义语句 **module_def** 及包括所有语句（例如实际处理数据库的 OPEN、CLOSE 和 SELECT）的 **manipulative_statement**。在每个语法部分的开始放置 **sql** 规则。（yacc 不关心所有的具有相同左侧的规则是否一起出现在规范中，在本例中不一起出现是比较容易的，如果你这样做了，务必要包括解释你做了什么的注释。）

模式子语言

模式（schema）子语言定义数据库表。如例 5-6 所示，以 CREATE SCHEMA AUTHORIZATION “ user ”开始，后面跟着可选的模式元素列表。

例 5-6：模式子语言，上面部分

```
schema:
       CREATE SCHEMA AUTHORIZATION user opt_schema_element_list
   ;

opt_schema_element_list:
       /* 空 */
   |   schema_element_list
   ;

schema_element_list:
       schema_element
   |   schema_element_list schema_element
   ;

schema_element:
       base_table_def
   |   view_def
   |   privilege_def
   ;

user:     NAME
   ;
```

例 5-6 展示了最高层的模式定义规则。有一条说明 **user** 是一个 NAME 的规则。在语句构成上，可以在 **schema** 规则中直接使用 NAME，但是这条独立的规则为动作代码提供了便利的位置，这个动作代码用来验证所给出的用户名是可信的。

模式元素列表语法使用大量的规则，但是它们并不复杂 **opt_schema_element_list** 要么是 **schema_element_list**，要么什么也不是，**schema_element_list** 是一组 **schema_element**，而且 **schema_element** 是三种定义中的一种。我们尽量将这些规则简化为：

```
opt_schema_element_list:
       /* 空值 */
   |   opt_schema_element_list base_table_def
   |   opt_schema_element_list view_def
   |   opt_schema_element_list privilege_def
   ;
```

尽管在添加动作代码时，越复杂的版本越容易使用

基表

例 5-7 展示了基表语言

例 5-7：模式子语言，基表

```
base_table_def:
       CREATE TABLE table '(' base_table_element_commalist ')'
   ;

table:
       NAME
   |   NAME '.' NAME
   ;

base_table_element_commalist:
       base_table_element
   |   base_table_element_commalist ',' base_table_element
   ;

base_table_element:
       column_def
   |   table_constraint_def
   ;

column_def:
       column data_type column_def_opt_list
   ;

data_type:
       CHARACTER
   |   CHARACTER '(' INTNUM ')'
   |   NUMERIC
   |   NUMERIC '(' INTNUM ')'
   |   NUMERIC '(' INTNUM ',' INTNUM ')'
   |   DECIMAL
   |   DECIMAL '(' INTNUM ')'
   |   DECIMAL '(' INTNUM ',' INTNUM ')'
   |   INTEGER
   |   SMALLINT
   |   FLOAT
   |   FLOAT '(' INTNUM ')'
   |   REAL
   |   DOUBLE PRECISION
   ;

column_def_opt_list:
       /* 空值 */
   |   column_def_opt_list column_def_opt
```

```
   ;

column_def_opt:
       NOT NULLX
   |   NOT NULLX UNIQUE
   |   NOT NULLX PRIMARY KEY
   |   DEFAULT literal
   |   DEFAULT NULLX
   |   DEFAULT USER
   |   CHECK '(' search_condition ')'
   |   REFERENCES table
   |   REFERENCES table '(' column_commalist ')'
   ;

table_constraint_def:
       UNIQUE '(' column_commalist ')'
   |   PRIMARY KEY '(' column_commalist ')'
   |   FOREIGN KEY '(' column_commalist ')'
          REFERENCES table
   |   FOREIGN KEY '(' column_commalist ')'
          REFERENCES table '(' column_commalist ')'
   |   CHECK '(' search_condition ')'
   ;

column_commalist:
       column
   |   column_commalist ',' column
   ;

column:   NAME
   ;

literal:
       STRING
   |   INTNUM
   |   APPROXNUM
   ;
```

基表同样具有大量的语法，但是一旦你读这些部分，它并不复杂。基表定义 **base_table_def** 是“ CREATE TABLE ”，表名可以是简单的名字或由用户名限定的名字，而且括号中的基表元素的列表由逗号分开。每个 **base_table_element** 是一个列定义或一个表约束定义。

列定义 **column_def** 是列名、数据类型和可选列定义的选项。可能的数据类型列

表和列定义选项列表很长。因为有一个对 **data_type** 的引用，故而每列都有一个数据类型。这些标记中，一些是保留字，一些是数字，所以像 NUMERIC(5,2) 这样的类型匹配第 5 个 **data_type** 规则。对于列定义选项，**column_def_opt_list** 允许任意记录中零个或多个选项。这些选项规定列是否可以包含空（未定义的）值，规定值是否必须惟一，设置默认值，设置有效性检测条件，以及设置表间的一致性（REFERENCES）条件。后面定义了 **search_condition**，它处理语言的语法结构部分。

用标记 **NULLX** 表示保留字 NULL，因为 yacc 将所有的标记定义为 C 预处理程序符号，而符号 NULL 早已在 C 中有定义。同理，要避免将标记命名为 FILE、BUFSIZ 和 EOF，以及所有在标准 I/O 库中由符号已被占用的名字。

基表元素还可以是几种形式之一的表约束。SQL 语法在这里是冗余的，下面的两种形式等价：

```
thing CHAR(3) NOT NULL UNIQUE

thing CHAR(3),
       UNIQUE( some_name )
```

第一种形式被解析为具有‘ NOT NULL ”和“ UNIQUE ”的 **column_def**，每个都是 **column_def_opt_list** 中的 **column_def_opt**。第一种形式是后跟 **table_constraint_def** 的 **column_def**，每个都是 **base_table_element** 而且在 **base_table_element_list** 中两者组合在一起。

SQL 定义在 unique 的列中取值 NULL，但不一定必须如此明确地说明。它们在语法结构上是不同的，但动作代码可识别这两个形式是相等的，让 yacc 知道这些就够了。

视图定义

视图（view）是由查询定义的虚表，每次应用程序打开视图时，其内容都是执行查询的结果[3]。例如，可以在食物表中创建水果的视图：

```
CREATE VIEW fruits (frname, frflavor)
   AS SELECT Foods.name, Foods.flavor
   FROM Foods
   WHERE Foods.type = “fruit”
```

例 5-8 展示了模式视图定义的语法

例 5-8：模式视图定义

```
view_def:
```

[3]这有点过于简单化，因为在许多情况下，你可以将行写入视图并让数据库适当地更新基本表。

```
       CREATE VIEW table opt_column_commalist
       AS query_spec opt_with_check_option
   ;

opt_with_check_option:
       /* 空值 */
   |   WITH CHECK OPTION
   ;

opt_column_commalist:
       /* 空值 */
   |   '(' column_commalist ')'
   ;
```

视图定义是“ CREATE VIEW ”、表名、可选的列名列表（默认使用基表中的名字）、关键字“ AS ”、一种查询规范（稍后定义）和可选的检查选项，检查选项在向视图写入新行时控制错误检查的量。

特权定义

例 5-9 展示了特权定义子语言，位于视图定义子语言的最后。

例 5-9：模式特权定义

```
privilege_def:
       GRANT privileges ON table TO grantee_commalist
       opt_with_grant_option
   ;

opt_with_grant_option:
       /* 空值 */
   |   WITH GRANT OPTION
   ;

privileges:
       ALL PRIVILEGES
   |   ALL
   |   operation_commalist
   ;

operation_commalist:
       operation
   |   operation_commalist ',' operation
   ;
```

```
operation:
       SELECT
    |  INSERT
    |  DELETE
    |  UPDATE opt_column_commalist
    |  REFERENCES opt_column_commalist
    ;


grantee_commalist:
       grantee
    |  grantee_commalist ',' grantee
    ;

grantee:
       PUBLIC
    |  user
    ;
```

表或视图的主人可以给其他用户在他们的表上执行各种操作的权力，例如：

```
GRANT SELECT, UPDATE (address, telephone)
       ON employees TO PUBLIC
GRANT ALL ON foods TO tony, dale WITH GRANT OPTION

GRANT REFERENCES (flavor) ON Foods TO PUBLIC
```

WITH GRANT OPTION 允许授权人向其他用户再次授予他们的权力。当创建表时需要引用已有的表的键值列时，REFERENCES 给予它所需要的权力。另外，语法和 GRANT 语句的含义相当简单。

模块子语言

由于模块语言在实际应用中已经过时，所以这里不再详细介绍它。在附录十 SQL 语法的完整列表中可以找到它的 yacc 定义。

游标定义

为了在嵌入式 SQL 中应用，需要模块语言中的游标定义语句。例 5-10 展示了游标定义的语法.

例 5-10：游标定义

```
cursor_def:
```

```
       DECLARE cursor CURSOR FOR query_exp opt_order_by_clause
   ;

opt_order_by_clause:
       /* 空值 */
   |   ORDER BY ordering_spec_commalist
   ;

ordering_spec_commalist:   /* 定义排序次序 */
       ordering_spec
   |   ordering_spec_commalist ',' ordering_spec
   ;

ordering_spec:
       INTNUM opt_asc_desc        /* 通过列号 */
   |   column_ref opt_asc_desc    /* 通过列号 */
   ;

opt_asc_desc:
       /* 空值 */
   |   ASC
   |   DESC
   ;

cursor:      NAME
   ;

column_ref:
       NAME          /* 列名 */
   |   NAME '.' NAME/* table.col 或 range.col */
   |   NAME '.' NAME '.' NAME  /* user.table.col */
   ;
```

典型的游标定义如下：

```
DECLARE course_cur CURSOR FOR
       SELECT ALL
       FROM Courses
       ORDER BY sequence ASC
```

游标定义非常类似于视图定义、两者都是将名字与 SELECT 查询关联起来。区别是视图是驻留在数据库中的永久对象，而游标是驻留在应用程序中的临时对象。特别是，视图有自己的、不同于构建它们的表的特权（这是创建视图的主要原因）。你需要一个游标以在程序中读写数据；为了读写视图，在视图上需要游

标。而且，用于定义游标的查询表达式比用于定义视图的查询规范更通用。在下一节中，我们将看到 SELECT 语句与视图和游标的关系。

操纵子语言

SQL 的核心是操纵子语言（manipulation sublanguage)：搜索、阅读、插入、删除和更新行和表的命令。

有 11 种不同的操纵语句，如例 5-11 中的规则所示。

例 5-11：操纵子语言的上面部分

```
sql:      manipulative_statement
   ;

manipulative_statement:
       close_statement
   |   commit_statement
   |   delete_statement_positioned
   |   delete_statement_searched
   |   fetch_statement
   |   insert_statement
   |   open_statement
   |   rollback_statement
   |   select_statement
   |   update_statement_positioned
   |   update_statement_searched
   ;
```

尽管一些语句（特别是 **SELECT** 语句）涉及到数据库部分的大量工作，但一次只执行一条 SQL 语句。

简单的语句

简单的操纵语句如例 5-12 所示。

例 5-12 简单的操纵语句

```
open_statement:
       OPEN cursor
   ;

close_statement:
       CLOSE cursor
   ;
```

```
commit_statement:
       COMMIT WORK
   ;

rollback_statement:
       ROLLBACK WORK
   ;

delete_statement_positioned:
       DELETE FROM table WHERE CURRENT OF cursor
   ;
```

大部分操纵语句相当简单，例 5-12 全部展示了它们。OPEN 和 CLOSE 类似于打开和关闭文件。DELETE…WHERE CURRENT 删除游标标识的单个记录。

FETCH 语句是在程序中获取数据的主要方式。它的语法比前面的语句稍微复杂些，因为它要说明一列一列地放置数据的位置。

FETCH 语句

例 5-13 展示了 FETCH 的规则。

例 5-13：FETCH 语句

```
fetch_statement:
       FETCH cursor INTO target_commalist
   ;

target_commalist:
       target
   |   target_commalist ',' target
   ;

target:
       parameter_ref
   ;

parameter_ref:
       parameter
   |   parameter parameter
   |   parameter INDICATOR parameter
   ;

parameter:
```

```
    |   ':' NAME  /* 嵌入的参数 */
    ;
```

FETCH 由于所有可能的目标而变得复杂。每个目标是一个或两个参数，可选的第二个参数作为一个指示变量用来说明存储的数据是有效的还是空的。在嵌入式 SQL 中，参数是前面带有冒号的宿主语言变量名。在模块语言的一个过程中，参数还是模块头中作为参数声明的名字，但是在那种情况下，词法分析程序必须能将参数名与列名、区域名区分开，否则 yacc 会因为词法分析程序不能表明是哪个名字而产生许多移进/归约冲突。为了使语法检查程序保持相对简单，不考虑模块语言名字。在工作的示例中，词法分析程序查找符号表中的每个名字并返回一个不同的标记，例如、模块参数名 **MODPARAM**，需要添加一条规则：

```
parameter: MODPARAM;
```

INSERT 语句

INSERT 语句如例 5-14 所示。

例 5-14：INSERT 语句

```
insert_statement:
       INSERT INTO table opt_column_commalist values_or_query_spec
    ;

values_or_query_spec:
       VALUES '(' insert_atom_commalist ')'
    |  query_spec
    ;

insert_atom_commalist:
       insert_atom
    |  insert_atom_commalist ',' insert_atom
    ;

insert_atom:
       atom
    |  NULLX
    ;

atom:
       parameter_ref
    |  literal
    |  USER
    ;
```

用于向表中插入新行的 INSERT 语句有两个变量。在两种情况下都以表名和一个可选的列名作为参数。（我们能重新使用已经用在 CREATE VIEW 中的 **opt_column_commalist**。）接下来是一列值或一条查询规范。这列值是“ VALUE ”和由逗号分隔的一列 **insert_atom**。插入原子可以是 NULL、参数、文字字符串或数字或者意味着当前用户 ID 的“ USER ”。被定义的查询规范 **query__spec** 在数据库中选择现有的数据拷贝到当前的表中。

DELETE 语句

DELETE 语句从表中删除一行或多行。它的规则列在例 5-15 中

例 5-15：DELETE 语句

```
delete_statement_positioned:
       DELETE FROM table WHERE CURRENT OF cursor
   ;

delete_statement_searched:
       DELETE FROM table opt_where_clause
   ;

opt_where_clause:
       /* empty */
   |   where_clause
   ;

where_clause:
       WHERE search_condition
   ;
```

定位情况下删除游标处的行。搜索情况下删除由可选的 WHERE 子句标识的行，或者在没有 WHERE 子句的时候删除表中所有的行。WHERE 子句使用搜索条件（下面定义）来标识要删除的行。

UPDATE 语句

UPDATE 语句如例 5-16 所示。

例 5-16：UPDATE 语句

```
update_statement_positioned:
       UPDATE table SET assignment_commalist
       WHERE CURRENT OF cursor
   ;

assignment_commalist:
```

```
    |   assignment
    |   assignment_commalist ',' assignment
    ;

assignment:
        column '=' scalar_exp
    |   column '=' NULLX
    ;

update_statement_searched:
        UPDATE table SET assignment_commalist opt_where_clause
    ;
```

UPDATE 语句用于重写一行或多行。有两种形式，定位的和搜索的，与 DELETE 的两种形式相似。在两种情况下，由逗号分隔的赋值列表给相应行的列设置新值，值可以是 NULL 或标量表达式。

标量表达式

标量表达式如例 5-17 所示。

例 5-17：标量表达式

```
scalar_exp:
        scalar_exp '+' scalar_exp
    |   scalar_exp '-' scalar_exp
    |   scalar_exp '*' scalar_exp
    |   scalar_exp '/' scalar_exp
    |   '+' scalar_exp %prec UMINUS
    |   '-' scalar_exp %prec UMINUS
    |   atom
    |   column_ref
    |   function_ref
    |   '(' scalar_exp ')'
    ;

scalar_exp_commalist:
        scalar_exp
    |   scalar_exp_commalist ',' scalar_exp
    ;

function_ref:
        AMMSC '(' '*' ')'
    |   AMMSC '(' DISTINCT column_ref ')'
    |   AMMSC '(' ALL scalar_exp ')'
```

```
   |  AMMSC '(' scalar_exp ')'
   ;

scalar_exp_commalist:
       scalar_exp
   |  scalar_exp_commalist ',' scalar_exp
   ;
```

标量表达式类似于常规程序设计语言中的算术表达式。它们允许具有通用优先级的通用算术操作符。回忆一下使用%**left** 在语法的开头设置优先级的情形，而且%**prec** 赋予一元“ + ”和“ - ”最高的优先级。SQL 还有几个内置的函数。标记 AMMSC 是 AVG、MIN、MAX、SUM 或 COUNT 中任何一个的简写。这里的语法实际上比 SQL 允许的宽松。“ COUNT(*) ”计算所选择的集合中的行数，而且是允许参数“ * ”的惟一的地方。（动作代码必须检查 AMMSC 的标记值，并且如果它不是 COUNT 就会“ 抱怨 ”。）DISTINCT 意味着在执行函数之前删除重复值；它只允许列引用。其他函数可以采用标量引用。

为了更好地匹配允许的语法，就要使语法规则变得更加复杂，但是这种方法有两个优点：语法分析程序比较小而且快速；动作程序能发出更详细的消息（例如，“ MIN 不允许*参数 ”而不是“ 语法错误 ”）。

标量函数的定义是彻底的递归式，所以这些规则可以让你编写极其复杂的表达式，例如：

```
SUM( (p.age*p.age) / COUNT( p.age ) ) – AVG( p.age ) * AVG( p.age )
```

上述表达式计算表 **p** 中 **age** 列的算术方差（也许它非常慢）。

我们还定义了 **scalar_exp_commalist**——由逗号分隔的一列标量表达式，后面会用到。

SELECT 语句

SELECT 语句如例 5-18 所示。

例 5-18：SELECT 语句、查询规范和表达式

```
select_statement:
       SELECT opt_all_distinct selection
       INTO target_commalist
       table_exp
   |  //手工添加（faisal）
       SELECT opt_all_distinct selection
       table_exp
   ;
```

```
opt_all_distinct:
       /* empty */
    |  ALL
    |  DISTINCT
    ;

selection:
       scalar_exp_commalist
    |  '*'
    ;

query_exp:
       query_term
    |  query_exp UNION query_term
    |  query_exp UNION ALL query_term
    ;

query_term:
       query_spec
    |  '(' query_exp ')'
    ;

query_spec:
       SELECT opt_all_distinct selection table_exp
    ;
```

SELECT 语句从数据库中选择一行（可能得自不同表中大量不同的行）并使用 **table_exp**（在下一节进行定义）将它移入局部变量的集合，**table_exp** 是从数据库中选择表或子表的表值（table-valued）表达式。可选的 ALL 或 DISTINCT 用于保持或放弃重复的行。

查询表达式 **query_exp** 和查询规范 **query_spec**（也在例 5-18 中）类似表值的形式。查询规范几乎和 SELECT 语句具有相同的形式，但是没有 INTO 子句，因为查询规范是较大语句的一部分。查询表达式是几个查询规范的 UNION；查询的结果合并在一起。（规范必须都有相同数目和类型的列。）

表表达式

表表达式如例 5-19 所示。

例 5-19：表表达式

```
table_exp:
       from_clause
       opt_where_clause
```

```
       opt_group_by_clause
       opt_having_clause
   ;

from_clause:
       FROM table_ref_commalist
   ;

table_ref_commalist:
       table_ref
   |   table_ref_commalist ',' table_ref
   ;

table_ref:
       table
   |   table range_variable
   ;

range_variable:  NAME
   ;

where_clause:
       WHERE search_condition
   ;

opt_group_by_clause:
       /* 空值 */
   |   GROUP BY column_ref_commalist
   ;

column_ref_commalist:
       column_ref
   |   column_ref_commalist ',' column_ref
   ;

opt_having_clause:
       /* 空值 */
   |   HAVING search_condition
   ;
```

表表达式是赋予 SQL 动力的东西，因为它们可以让你随心所欲地定义复杂的表达式来正确地检索想要的数据。表表达式从强制性的 FROM 子句开始，后面跟着可选的 WHERE、GROUP BY 和 HAVING 子句。FROM 子句命名构建表达式的表的名称。可选的范围变量可以让你用一个表达式在分隔的内容中多次使用相

同的表，这个功能有时是很有用的，例如，经理人和职员都在各自的表中。WHERE 子句指定一个搜索条件来控制包含在表中的行，GROUP BY 户句根据普通的列值归组行，如果选择中包括类似 SUM 或 AVG 这样的函数时 GROUP BY 尤其有用，因为那时它们按组求和或求平均。HAVING 子句分组地应用搜索条件；例如，在供应者、零件名和价格表中，查询供应者供应的所有零件，这些供应者至少销售二种零件：

```
SELECT supplier
FROM p
GROUP BY supplier
HAVING COUNT(*) >= 3
```

搜索条件

例 5-20 定义了搜索条件的语法。

例 5-20：搜索条件

```
search_condition:
    |   search_condition OR search_condition
    |   search_condition AND search_condition
    |   NOT search_condition
    |   '(' search_condition ')'
    |   predicate
    ;

predicate:
        comparison_predicate
    |   between_predicate
    |   like_predicate
    |   test_for_null
    |   in_predicate
    |   all_or_any_predicate
    |   existence_test
    ;

comparison_predicate:
        scalar_exp COMPARISON scalar_exp
    |   scalar_exp COMPARISON subquery
    ;

between_predicate:
        scalar_exp NOT BETWEEN scalar_exp AND scalar_exp
    |   scalar_exp BETWEEN scalar_exp AND scalar_exp
    ;
```

```
like_predicate:
       scalar_exp NOT LIKE atom opt_escape
   |   scalar_exp LIKE atom opt_escape
   ;

opt_escape:
       /* 空値 */
   |   ESCAPE atom
   ;

test_for_null:
       column_ref IS NOT NULLX
   |   column_ref IS NULLX
   ;

in_predicate:
       scalar_exp NOT IN '(' subquery ')'
   |   scalar_exp IN '(' subquery ')'
   |   scalar_exp NOT IN '(' atom_commalist ')'
   |   scalar_exp IN '(' atom_commalist ')'
   ;

atom_commalist:
       atom
   |   atom_commalist ',' atom
   ;

all_or_any_predicate:
       scalar_exp COMPARISON any_all_some subquery
   ;

any_all_some:
       ANY
   |   ALL
   |   SOME
   ;

existence_test:
       EXISTS subquery
   ;
subquery:
       '(' SELECT opt_all_distinct selection table_exp ')'
   ;
```

搜索条件指定想要使用的组中的行。搜索条件是与 AND、OR 和 NOT 结合起来的谓词组合。有 7 种不同种类的谓词，谓词是人们愿意对数据库执行的操作的聚合。

COMPARISON 谓词比较两个标量表达式，或者一个标量表达式和一个子查询。回忆下，COMPARISON 标记是任意一种通用的比较操作符，例如“ = ”和“ <> ”。子查询（subquery）是一个递归的 SELECT 表达式，（在语义上而不是语法上）被限制为返回单列。

BETWEEN 谓词仅仅是一对比较的简写。下面两个谓词是等效的，例如：

```
p.age BETWEEN 21 and 65
p.age >= 21 AND p.age <= 65
```

LIKE 谓词执行一些字符串模式匹配、比较标量表达式与原子的操作，原子是文字字符串或字符串参数引用。与表达式进行比较的原子被看做一个简单的模式，类似 UNIX 命令解释程序（Shell）文件名模式。可选的 ESCAPE 子句可以让你在文件名模式中指定类似于“ \ ”的引号字符。

LIKE 谓词的左操作符必须是一个列引用，不是一般的表达式。这里使用一个标量表达式来规避 yacc 的局限性。LIKE 谓词更常用的语法是：

```
like_predicate:
       scalar_exp NOT LIKE atom opt_escape
   |   scalar_exp LIKE atom opt_escape
   ;
```

yacc 在谓词的内容中可以看到类似下面的东西：

```
Foods.flavor NOT . . .
```

当看到 NOT 时，它不能说明它是在 NOT BETWEEN 中还是在 NOT LIKE 中，所以它不能说明“ Food.flavor ”是 LIKE 谓词的 **column_ref** 还是 BETWEEN 的 **scalar_exp**。yacc 会用移进/归约冲突对此做出反应，因为它不能说明是否归约将 **column_ref** 转变为 **scalar_exp** 的规则。解决这个问题有两种方法。调整语法接受移进/归约冲突的归约端，两种情况都允许 **scalar_exp**，因为动作代码能容易地检测 NOT LIKE 的左操作符以确保它为列引用。（这样还为得到更好的错误消息提供了机会。）另一种可能性是词法修改。可以定义在词法分析程序中匹配两个单词的 NOTLIKE 标记：

```
NOT[ \t]+LIKE { return NOTLIKE; }
```

并且在 LIKE 谓词中使用：

```
like_predicate:
       scalar_exp NOTLIKE atom opt_escape
   |   scalar_exp LIKE atom opt_escape
   ;
```

这样就可解决这个问题，因为 yacc 一旦看到了 NOTLIKE 标记它就能说明它在分析 LIKE 谓词。但是修改词法是很讨厌的。（如果 NOT 和 LIKE 都在分离的行上，这种形式就会失败。如果在它们之间的空格中添加 “ \n ”，那么在这个词法动作中就需要检测是否有任何换行，如果有，更新 **lineno**，这会添加更多的混乱。）

对空值的测试正如它字面意思一样，也就是测试特定列的内容是否为空。在词法分析程序中使用标记名 NULLX 避免与 *stdio* NULL 符号发生冲突。

IN 谓词检测一个值是否为显式指定的或通过一个子查询指定的集合中的一个。显式形式等效于一组比较：

```
q.Name IN ( ‘Tom’, ‘Dick’, ‘Harry’ )
q.Name = ‘Tom’ OR q.Name = ‘Dick’ OR q.Name = ‘Harry’
```

ANY 或 ALL 谓词可以让你测试是表达式的任何一个还是所有的值使比较满足于子查询。它们有时是很有用的，但通常容易混淆：很难正确地书写 ANY 和 ALL 谓词。下面的例子检测表 **p** 的 **Name** 列中的名字与表 **q** 的 **name** 列中的名字匹配的所有名字：

```
p.Name = ALL (SELECT q.Name from q)
```

最后，存在测试可以测试是否有满足一些子查询的数据。

使用所有的谓词和子查询，可以创建无比复杂的查询和表表达式（该表达式执行时花费的时间也同样是惊人的）。然而，大多数 SQL SELECT 表达式都很简单，而执行复杂操作的能力也可以满足需要它的人们。

零碎的内容

例 5-21 只定义了一些在嵌入式 SQL 程序中使用的语句。

例 5-21：嵌入式 SQL 的条件

```
/* 嵌入的条件 */
sql:   WHENEVER NOT FOUND when_action
   |   WHENEVER SQLERROR when_action
   ;

when_action:  GOTO NAME
```

```
    |   CONTINUE
    ;
```

它们意味着，每当选择不检索任何数据（NOT FOUND）或一些其他错误(SQLERROR）的时候，程序将要么跳到主程序中的一个特定的标签，要么忽略这个条件。

使用语法检查程序

例 5-22 展示了 *Makefile*

例 5-22：SQL 语法检查程序的 Makefile

```
LEX = flex -I
YACC = byacc -dv
CFLAGS = -DYYDEBUG=1

all: sql1

sql1: sql1.o scn1.o
    ${CC} -o $@ sql1.o scn1.o

sql1.c sql1.h: sql1.y
    ${YACC} sql1.y
    mv y.tab.h sql1.h
    mv y.tab.c sql1.c
    mv y.output sql1.out

scn1.o: sql1.h
```

为了编译语法检查程序，合理地经由 lex 和 yacc 运行词法分析程序和扫描程序并与作为最终结果的 C 程序编译在一起。在这种情况下，使用 *make* 规则重新命名 lex 和 yacc 的输出来匹配输入文件。而且，使用 Berkely yacc 和 flex[4]，并定义自己的 **main()**和 **yyerror()**，所以不需要使用 lex 或 yacc 库。为了测试语法检查程序，可以检测 SQL 文件的完整性，或直接键入：

```
% sql1 sqlmod
SQL parse worked
% sql1
FETCH foo INTO
:a ,
b c, -- two names are legal
```

[4] AT&T lex 中存在的缺陷使它不能处理 SQL 词法分析程序，但是 lex 的所有的其他版本都会毫无问题地接受它。yacc 的所有版本都接受这个分析程序。

```
d e f -- but three aren't
4: syntax error at f
SQL parse failed
```

嵌入式 SQL

本章最后讨论将 SQL 语法检查程序转换为简单的嵌入式 SQL 预处理程序。假定有一个能够解释作为文本串传递的 SQL 语句的 SQL 实现。嵌入式 SQL 预处理程序只需要将 SQL 语句转换为 C 过程调用，这个 C 过程调用将 SQL 语句传递给解释例程。

嵌入式 SQL 比它看上去要复杂一点。词法分析程序必须在两个不同的状态下运行：只传递文本的正常状态和缓冲传递给解释程序的 SQL 语句的 SQL 模式。还需要处理参数引用，因为在这个已编译的程序中，解释程序没有将字符串“ :foo ”和变量 foo 关联在一起的方法。在词法分析程序中提取这个参数，用“ #N ”代替提到的变量 Nth，然后通过参数列表中的引用将所有提到的变量传递给解释程序。例如，嵌入式 SQL：

```
EXEC SQL FETCH flav INTO :name, :type
```

应该转换成 C 语言：

```
exec_sql( " FETCH flav INTO #1, #2 ", &name, &type );
```

这里重点强调对词法分析程序和语法分析程序的更改。完整的代码在附录十中。

对词法分析程序的更改

词法分析程序实际上有庞大的更改集合。例 5-23 展示了修改后的定义。

例 5-23：嵌入式词法分析程序中的定义

```
%{
#include "sql2.h"
#include <string.h>

int lineno = 1;
void yyerror(char *s);

    /* 保存 SQL 标记文本的宏 */
#define SV save_str(yytext)

    /* 保存文本并返回标记宏 */
#define TOK(name) { SV;return name; }
```

```
%}
%s SQL
```

我们已经定义了两个 C 宏，定义了调用 **save_str()**来保存当前标记的文本的 **SV**，以及保存标记文本并给语法分析程序返回一个标记的 **TOK()**。我们还添加了新的称为 **SQL** 的起始状态，将标准的 **INITIAL** 状态作为正常的非 SQL 的状态。

例 5-24 展示了修订的词法分析程序规则。

例 5-24：嵌入式词法分析程序规则

```
EXEC[ \t]+SQL{ BEGIN SQL; start_save(); }


   /* 文字关键字标记 */

<SQL>ALL     TOK(ALL)
<SQL>AND     TOK(AND)
<SQL>AVG     TOK(AMMSC)
   ...所有其他的保留字和标记...

   /* 名字 */
<SQL>[A-Za-z][A-Za-z0-9_]* TOK(NAME)

   /* 参数 */
<SQL>":"[A-Za-z][A-Za-z0-9_]* {
          save_param(yytext+1);
          return PARAMETER;
       }

   /* 数字 */

<SQL>[0-9]+  |
<SQL>[0-9]+"."[0-9]* |
<SQL>"."[0-9]*      TOK(INTNUM)

<SQL>[0-9]+[eE][+-]?[0-9]+ |
<SQL>[0-9]+"."[0-9]*[eE][+-]?[0-9]+ |
<SQL>"."[0-9]*[eE][+-]?[0-9]+ TOK(APPROXNUM)

   /* 字符串 */

<SQL>'[^'\n]*'   {
       int c = input();
```

```
       unput(c); /* 仅查看一下 */
       if(c != '\'') {
          SV;return STRING;
       } else
          yymore();
   }

<SQL>'[^'\n]*$   {  yyerror("Unterminated string"); }
<SQL>\n      { save_str(" ");lineno++; }
\n     { lineno++; ECHO; }

<SQL>[ \t\r]+save_str(" ");   /* 空白 */

<SQL>"--".*   ;   /* 注释 */

.      ECHO;  /* 随机的非 SQL 文本 */
%%
```

第一个新规则匹配“ EXEC SQL ”关键字并将扫描程序插入 SQL 状态。它还调用 **start_save()**初始化保存 SQL 命令的缓冲区。然后将“ <SQL> ”加在所有现有的标记规则的前面，这样它们就只在 SQL 模式中匹配，并改变动作为使用 **SV** 或 **TOK()**保存每个标记。因为我们需要区别对待参数和其他标记，所以为匹配冒号后跟有名字的参数添加一条新规则，并调用 **save_param()**保存参数引用。匹配换行和空格的每个 SQL 规则都保存单个空格；因为所有的空格都是等效的，保存单个空格可以使保存的字符串较短。最后，添加两条没有<SQL>前缀的规则，它们在不处于 SQL 模式时匹配和回送所有的字符。在用户子程序部分，添加将词法分析程序从 SQL 模式切换为 INITIAL 模式的小例程 **un_sql()**；这个例程必须在词法分析程序中，那是定义 BEGIN 宏的惟一的地方。

```
/* 离开 SQL 词法分析模式 */
un_sql()
{
   BEGIN INITIAL;
} /* un_sql */
```

对语法分析程序的更改

对语法分析程序的更改比对词法分析程序的更改要少得多。添力 PARAMETER 的%**token** 定义。向起始规则添加动作：

```
sql_list:
          sql ‘;’   { end_sql(); }
       |  sql_list sql ‘;’ { end_sql(); }
       ;
```

这些规则在每次分析完整的 SQL 语句时调用例程 **end_sql()**来切换 SQL 模式外部的词法分析程序。

因为目前参数有一个特殊的标记，更改 **parameter** 规则来引用这个新标记：

```
parameter:
          PARAMETER /* :name 在语法分析程序中处理*/
```

因为嵌入式 SQL 不使用模块语言，所以放弃模块语言的规则，只保留游标定义的规则，并使游标定义成为最高层的 SQL 语句：

```
sql:
       cursor_def
   ;
```

完成了——语法分析程序的其他方面没有更改。

辅助例程

嵌入式 SQL 文本支持例程的重要部分如例 5-25 所示。

例 5-25：嵌入式 SQL 文本支持例程的最重要部分

```
char save_buf[2000]    /* 用于 SQL 命令的缓冲区 */
char *savebp;          /* 当前缓冲区指针 */

#define NPARAM 20      /* 每个函数最大的参数 */
char *varnames[NPARAM];/* 参数的名字 */

/* 在 EXEC SQL 之后开始一个嵌入的命令 */
start_save(void);

/* 保存 SQL 标记 */
save_str(char *s);

/* 保存引用的参数 */
save_param(char *n);

/* SQL 命令结束，现在输出 */
end_sql(void);
```

我们编写了一些字符串处理例程，在它们被分析时缓冲并写出 SQL 命令。数据结构和入口点在例 5-25 中，而且整个文本在附录十中。在有很大的固定尺寸的字符缓冲区 **save_buf[]**中保存这些命令并使用字符指针 **savebp** 来跟踪缓冲区中

的当前位置。作为参数使用的每个变最名都保存在 **varnames[]**中。如果一个变量在同一个命令中使用两次，我们只保存一次。

当词法分析程序看到“ EXEC SQL ”时，程序 **start_save()**初始化缓冲指针。每个标记都用 **save_str()**保存，**sav_str()**将它的参数附加给 **save_buf**。参数引用由查找它的参数的 **save_param()**处理，如果 **varnames[]**中的变量名没有出现就输入它，然后保存“ #N ”形式的引用。

当语法分析程序看到整个 SQL 命令时，调用 **end_sql()**，写出对运行时解释程序 **exec_sql()**的调用。它将保存的缓冲区作为引用字符串来传递．必要时将它拆分成行，而且还传递参数表中每个变量的地址。最后，它调用词法分析程序例程 **un_sql()**让词法分析程序离开 SQL 状态。所有的输出都进入 **yyout**（默认的 lex 输出流），正像词法分析程序中的 **ECHO** 语句通过非 SQL 代码传递一样。

使用预处理程序

修改 Makefile，以便在辅助程序中与词法分析程序和语法分析程序相连接。因为没有更改 **main** 程序，只更改了它的消息（一个纯粹的装饰性更改），所以用运行语法检查程序的同样的方式运行预处理程序。

例 5-26 展示了在例 5-2 中的嵌入式 SQL 上运行预处理程序的结果。

例 5-26：来自嵌入式 SQL 预处理程序的输出

```
char flavor[6], name[8], type[5];
int SQLCODE;     /* 全局状态变量 */

exec_sql(“DECLARE flav CURSOR FOR SELECT Foods.name, Foods.type FROM
Foods WHERE Foods.flavor = #1”, &flavor);

main()
{
   scanf(“$s”, flavor);
   exec_sql(“OPEN flav”);
   for(;;) {
      exec_sql(“FETCH flav INTO #1, #2”, &name, &type);
      if(SQLCODE != 0)
         break;
      printf(“%8.8s %5.5s\n”, name, type);
   }
   exec_sql(“CLOSE flav”);
}
```

练习

1. 在几个方面，SQL 语法分析程序接受比 SQL 本身允许的更宽松的语法。例如语法分析程序接受无效的标量表达式 “ MIN(*) ”，并且把任何表达式都作为 LIKE 谓词的左操作数来接受，尽管操作数必须是列引用。调整语法检查程序以诊断这些错误的输入。可以更改语法或添加动作代码来检查表达式。两种方法都尝试一下，看看哪种更容易，以及哪种方法可以给出更好的诊断。

2. 将语法分析程序转换成 SQL 交叉引用程序，读取一组 SQL 语句并产生一个报告，这个报告展示定义每个名字的地方和引用这些名字的地方。

3. （术语工程）修改嵌入式 SQL 翻译程序以接到你的系统上真正的数据库。