LEMON 语法分析在嵌入式数据库设计中的应用

黄加喜，陈天煌，郑胜英，高庆锋
武汉理工大学计算机学院，湖北武汉(430063)
E-mail: jiaxi@163.com

摘　要：LEMON 是一款用 C 语言编写的 LALR(1)类型语法分析器生成器，LEMON 的主要功能是把由具有上下文无关的特殊语法写成一种语言文本，它短小精悍，功能齐全，非常适合在嵌入式数据库设计中应用。本文主要讨论了如何在嵌入式数据库设计的语法分析过程中使用 LEMON。

关键词：嵌入式数据库；LEMON；LALR (1)

1. 引言

高效安全的 SQL 编译器是一个优秀嵌入式数据库的核心和基础。SQL 作为关系数据库语言的国际标准，得到广泛应用。但对于 SQL 这样复杂庞大的语法，在嵌入式数据库中为之构建完善且可靠的语法分析器，具有一定的难度和工作量。语法分析方法可分为自底向上和自顶向下两大类，LALR(1) (LookAhead_LR)方法是 LR 方法的一种[1]，属于自底向上分析类。由 DeRemer 提出，它是介于 SLR(1)和 LR(1)之间的一种方法，具有 SLR(1)状态少的优点和 LR(1)的适用范围广的优点，主要思想来自这样一个事实：

1)LR(1)状态机从它们所表示的自动机角度看是等价的；

2)自动机可通过合并等价状态来减少状态个数。

LALR(1)状态机通常采用 LR(1)状态机来定义。假设[A ->a.ß]是 LR(1)项目，则称其中的 LR(0)项目部分 A->a.ß 为该项目的心。如果 LR(1)状态机的两个状态心相同，则称它们为同心状态。用 Core(S)表示状态 S 的心部分，SameCoreState(S)表示所有与 S 同心的状态号集合(包括 S)，Merge(SS)表示同心状态集 SS 中所有状态项目的合并。具体定义如下:

Core(S)={ LR0item| [LR0item,a]IS }

SameCoreState (S) ={ S'|Core(S')=Core(S') }

Merge(SS)={LR1item| LR1item∈S,S∈SS}

构造出 LALR(1)状态机后，若 LRSM 的每个状态中都无相互冲突的项目，则文法为 LALR(1)文法，可以进行 LALR(1)分析。

2. LEMON 简介

LEMON 是一款用 C 语言编写的 LALR(1)类型语法分析器生成器[2]，LEMON 的主要功能是把由具有上下文无关的特殊语法写成一种语言文本，先进行符号分析、语法分析，再转换成可直接对该语言进行编译的 C 程序。

2.1 LEMON 的构成

LEMON 语法分析生成器工作时，需要两个输入文件：

1) 语法文件(.y)。

2) 语法模版文件(leper.c)。

LEMON 自己已经附带一个设计完备的语法模版文件(leper.c)。所以使用 LEMON 只需要编写语法文件就可以生成 LALR(1)语法分析程序。LEMON 工作原理：

1) 首先编写语法文件。根据具体要求，需要仔细推敲、仔细考虑用上下文无关文法写成的语法文件，并且还附有当记号之间的关系符合其中某一巨型时如何进行处置的代码。

2) 调用 LEMON 程序，读入此语法文件。可以自动生成语法分析程序。

3) 将此语法分析程序加入到具体代码中去。词法分析器将字符串分解成一个一个的单词后，这些单词就会逐个送进语法分析器，语法分析器会按照语法文件中的约定，以句型作为“对号入座”的依据，进行正确的处理。图 1 给出了 LEMON 的工作过程。

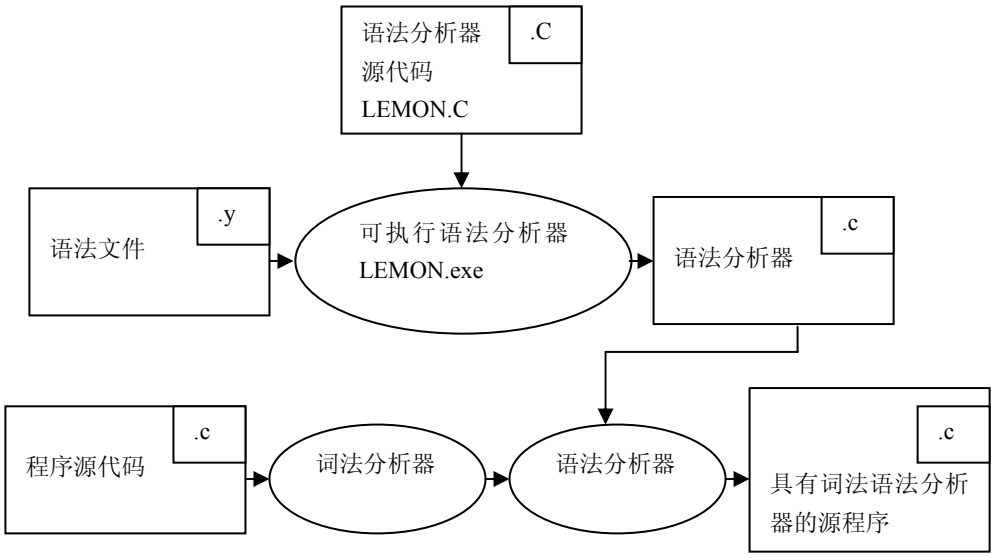

图 1 LEMON 的工作过程

2.2 LEMON 语法文件的语法

LEMON 的语法文件(.y)的主要目的是定义能为 LEMON 理解和产生语法分析器程序的一系列符号的次序排列，但是，.y 文件还需要有另外一些让 LEMON 程序能够理解和运用的附加信息[3]。因此，必须按照一定的规格和要求仔细撰写.y 文件。

1) 终结符和非终结符

在 LEMON 中，终结符的形式是由大写字母开头的一串英文字母，但习惯全用大写英文字母来书写终结符。而非终结符全用小写英文字母。

2) 产生式

语法文件的主体是一系列有序的产生式。每一个产生式的格式都是：左边由一个非终结符开头，然后是一个表示产生式定义的特殊组合符号“：=”，其右是一些列终结符和(或)非终结符，最后是一个黑句点“.”，此黑句点表示这条产生式的结束。在 LEMON 产生式的后面，加入由 C 语言写成的一段程序。当语法分析器对于这条产生式进行规约时，这段用 C 语言写成的程序就得到执行，即执行了一个与产生式相关联的动作。这段 C 语言必须放在相应产生式的黑句点之后，且应该放在花括号“{……}”对中。

例如：

```
createtable ::= CREATE TABLE ifnotexists(E) nm(Y) dbnm(Z). {
    mysqliteStartTable(pParse,&Y,&Z,0,0,E);}
```

3) 符号、产生式的优先级和结合规则

根据语法的具体情形，一个终结符可以具有左结合、右结合和不得结合三种结合规则，他们分别用 LEMON 中的特殊申明符来制定：

%left ：指定终结符具有左结合性。

%right：制定终结符具有右结合性。

%nonassoc：指定终结符不得在同一产生式中出现两次或两次以上。

优先级则用他们在语法文件中出现的前后次序来确定。出现在前面的符号的优先级低，出现在后面的符号的优先级高。

例如：

%left OR.

%left AND.

%right NOT.

%left IS MATCH LIKE_KW BETWEEN IN ISNULL NOTNULL NE EQ.

对于产生式的移进—规约冲突，一般处理时采用移进策略。对于产生式的规约—规约冲突，一般处理都是把放在前面的产生式作为首选产生式，从而用它来进行规约。但 LEMON 还有另外一些方法来控制语法分析中的二义性冲突。限于篇幅，此处不再叙述。

4) 特殊申明符

LEMON 有 20 个特殊申明符号[4]。每个特殊申明符号都以“%”字符开头，后面跟由小写英文字母和下划线组成的字符串：

如： %left %code %type %token_type %default_type %name

特殊申明符可以放在语法文件的任意处，但当为终结符指定优先级和结合性的时候，它们之间的顺序不能任意。

例如：%token_type 和%default_type

所有的终结符都必须具有唯一的一种数据类型，用%token_type 来制定终结符的数据类型。非终结符可以有自己的数据类型，可用%default_type 来为非终结符指定默认的统一数据类型。

3. LEMON 在嵌入式数据库中的应用

本文在分析了 LEMON 的语法结构之后，编写了一个嵌入式数据库的语法文件。

1) 首先利用特殊申明符指定终结符的数据类型、默认非终结符的数据类型和语法错误处理等。

```
%token_type {Token}
%default_type {Token}
%extra_argument {Parse *PParse}
%name mysqliteParser
%syntax_error
{
if( !PParse->Error )
{
if( TOKEN.error[0] )
{
        mysqliteErrorMsg(PParse, "NEAR    \"%T\":SYNTAX ERROR", &TOKEN);
}
else
{
        mysqliteErrorMsg(PParse, "Incomplete SQL ");
```

```
        }
        PParse->Error = 1;
      }
    }

    %stack_overflow
     {
      mysqliteErrorMsg(PParse, "Parser Stack Overflow");
      PParse->Error = 1;
    }
```

2) 包含头文件和定义结构体

```
    %include
    {
    #include "parse.h"
    struct ExprList
    {
      int nExpr;                /* Number of expressions on the list */
      int nAlloc;               /* Number of entries allocated below */
      int iECursor;             /* VDBE Cursor associated with this ExprList */
    struct ExprList_item
    {
        Expr *pExpr;            /* The list of expressions */
        char *zName;            /* Token associated with this expression */
        unsigned char sortOrder;          /* 1 for DESC or 0 for ASC */
        unsigned char isAgg;              /* True if this is an aggregate like count(*) */
        unsigned char done;               /* A flag to indicate when processing is finished
*/
      } *a;                     /* One entry for each expression */
    };
    struct IdList
    {
    struct IdList_item
    {
        char *zName;          /* Name of the identifier */
        int idx;              /* Index in some Table.aCol[] of a column named zName */
      } *a;
      int nId;              /* Number of identifiers on the list */
      int nAlloc;           /* Number of entries allocated for a[] below */
    };
    struct Token
    {
      const unsigned char *z;     /* Text of the token.  Not NULL-terminated! */
      unsigned dyn  : 1;          /* True for malloced memory, false for static */
      unsigned n    : 31;         /* Number of characters in this token */
    };
```

```
}
```

3) 接受输入的 SQL 命令

```
input ::= cmdlist.
cmdlist ::= cmdlist ecmd.
cmdlist ::= ecmd.
cmdx ::= cmd.                    { mysqliteFinishCoding(PParse); }
ecmd ::= SEMI.
ecmd ::= explain cmdx SEMI.
explain ::= .                    { mysqliteBeginParse(PParse, 0); }
```

其中 cmd 为 SQL 基本命令语句。

4) 创建表、删除表，查询、插入、更新、删除等功能的语法。

这里以创建表为例：

```
cmd ::= create_table create_table_args.
create_table ::= CREATE TABLE ifnotexists(E) nm(Y) dbnm(Z). {
    mysqliteStartTable(pParse,&Y,&Z,0,0,E);
}
%type ifnotexists {int}
ifnotexists(A) ::= .                    {A = 0;}
ifnotexists(A) ::= IF NOT EXISTS. {A = 1;}
create_table_args ::= LP columnlist RP(Y). {mysqliteEndTable(PParse,&Y,0);}
columnlist ::= columnlist COMMA column.
columnlist ::= column.
column(A) ::= columnid(X) type carglist.
{   A.z = X.z;   A.n = (PParse->LastToken.z-X.z) + PParse->LastToken.n;}
columnid(A) ::= nm(X).
 { mysqliteAddColumn(PParse,&X);   A = X;}
%type typetoken {Token}
type ::= .
type ::= typetoken(X).
                    {mysqliteAddColumnType(PParse,&X);}
typetoken(A) ::= typename(X).     {A = X;}
 typetoken(A) ::= typename(X) LP signed RP(Y).
 {   A.z = X.z;   A.n = &Y.z[Y.n] - X.z;}
typetoken(A) ::= typename(X) LP signed COMMA signed RP(Y).
{   A.z = X.z;   A.n = &Y.z[Y.n] - X.z;}
%type typename {Token}
typename(A) ::= ids(X).                 {A = X;}
typename(A) ::= typename(X) ids(Y). {A.z=X.z; A.n=Y.n+(Y.z-X.z);}
%type signed {int}
signed(A) ::= plus_num(X).      { A = atoi((char*)X.z); }
signed(A) ::= minus_num(X).     { A = -atoi((char*)X.z); }
%type nm {Token}
nm(A) ::= ID(X).                {A = X;}
nm(A) ::= STRING(X).            {A = X;}
nm(A) ::= JOIN_KW(X).           {A = X;}
```

```
%type dbnm {Token}
dbnm(A) ::= .                {A.z=0; A.n=0;}
dbnm(A) ::= DOT nm(X). {A = X;}
}
```

4. 语法分析器的接口

在编写完语法文件后，就可以执行语法分析器 LEMON.exe 生成语法分析器。语法分析器提供了接口供源程序调用，这样利用这些接口，就可以将语法分析程序加入到源程序中了。调用语法分析器接口的过程如下：

1) 建立语法分析器

(void) * PParser = mysqliteParserAlloc((mysqliteMalloc);

2) 调用词法分析器得到各个单词。

3) 为每一个单词调用 mysqliteParser (PParser, TokenID, TokenData, pParse)。

4) 当输入源到达末尾时，调用 mysqliteParser(PParser, 0, TokenData, pParse);通知语法分析器输入结束。

5) 最后调用 mysqliteParserFree(PParser, mysqliteFreeX); 回收语法分析器所占用的内存。

5. 结束语

讨论了 LEMON 的基本工作原理以及语法文件的编写。LEMON 短小功能齐全、安全、出错率低非常适合嵌入式数据库的设计。并针对嵌入式数据库的特点，将 LEMON 生成的语法分析程序应用到嵌入式数据库中，提升了其速度和稳定性。

参考文献

[1] 张敏,金茂忠. 一个高效的语法分析生成工具[J].微计算机信息,2005,21(8):47-52.
[2] 虞森林.LEMON 语法分析生成器[M].浙江:浙江大学出版社,2006: 9-25.
[3] KennethC.Louden. Compiler Construction Principlesand Practice[M].北京:机械工业出版社,2000.
[4] 范忠锋,刘坚.用词/语法分析器生成器实现软件系统的输入[J].计算机应用,2002,22(12).

The application of LEMON in Embedded database

Huang Jiaxi, Chen Tianhuang, Zheng Shengying, Gao Qingfeng
Wuhan University of Technology, WuHan, Hubei (430063)

Abstract

LEMON is LALR(1) parser generator. It takes a context free grammar and converts it into a subroutine that will parse a file using that grammar. It is faster, thread safe, and less prone to errors. It is very suitable to apply in the embedded database. This paper mainly discussed how to apply LEMON parser in embedded database.

Keywords embedded database, LEMON, LALR(1)