语言开发工具 Lex 和 Yacc 的分析与应用

丁英嘉　王能斌
(南京工学院)

ANALYSIS AND APPLICATIONS OF LANGUAGE IMPLEMENTATION TOOLS, LEX AND YACC

DING YINGJIA WANG NENGBIN
(*Nanjing Institute of Technology*)

Abstract In this paper, some results based on analysis and applications of two software tools Lex and Yacc are presented. Lex is a generator of lexical analyzer and Yacc is a generator of parser. Both are provided by UNIX. Making full use of these two tools, we have implemented Micro SQL, a relational data language used in DBMS NITDB. It is worthy to mention that the drawback which prohibited Lex from being used widespreadly has been overcome.

摘要 本文旨在向读者介绍对 UNIX 系统中的两个软件工具 Lex、Yacc 的分析结果和应用方法。Lex 是词法分析器的自动生成工具，Yacc 是语法分析器的自动生成工具。我们在关系数据语 MicroSQL 的实现过程中，灵活地利用了这两个工具，特别是解决了导致 Lex 被某些系统弃之不用的问题。

一、引　言

在许多计算机系统中，软件约占整个系统成本的 80%，而且，随着硬件成本的不断下降，这个比例还在上升。

显然，这种情况不能无限地发展下去，必须尽力提高程序员的工作效率和改进软件质量。一个重要的目标就是要开发、利用软件工具。

目前，软件工具的水平远不够理想，但仍有一些比较好的软件工具值得充分利用。如，UNIX 系统中的两个语言开发工具 Lex、Yacc。

本文旨在向读者介绍对这两个工具的分析结果和灵活应用的方法。应用背景是实现一个微机关系数据语言 MicroSQL，环境为 UNIX 第 7 版支持的微型机系统 Dual System

本文 1985 年 10 月 7 日收到。

二、对 Lex 和 Yacc 的分析

Lex (Lexical analyzer generater) 和 Yacc (Yet another compiler-compiler) 分别是一个词法分析器和语法分析器的自动生成工具。它既可单独使用，又可配合使用。这两个工具所基于的理论、主要功能、用法以及相互间的联系均概括于表 1。

表 1 Lex 和 Yacc(固定的名有下划线)

	Lex	Yacc
1. 基于的理论	正规式	LALR (1) 分析法
2. 命令形式	Lex filenam	Yacc filenam
3. 功能	filenam → [Lex] → Lex. YY.C 输入文件 输出文件	filenam → [Yacc] → y. tab. c 输入文件 输出文件
4. 输入文件规格	说明集 %% 词法规则集 %% 子程序集	说明集 %% 文法规则集 %% 子程序集
5. 输出文件的功能	词法分析器，C 程序模块 入口：yylex ( )	语法分析器，C 程序模块， 入口：yyparse( )
6. 说明集的组成及作用	• 全局变量或宏定义： % {C 全局变量或宏定义 %} 被定义的变量或宏可供所有动作(见下)使用。	• 开始符号说明： % start symbol • 终极符说明： % token name-list 不在此说明的每个名字都代表一个非终极符。 • 算术运算符的优先级与结合性说明： % left operator-list % right operator-list % nonassoc operator-list 同一行中运算符具有相同优先级，行与行之间优先级按顺序递增。 • 全局变量或宏定义： 同左
7. 规则集的组成	• 一条或多条词法规则 • 每条规则可分两部分： 正规式 {动作}	• 一条或多条文法规则 每条规则可分两部分： LALR(1) 产生式{语义动作}
8. 正规式或产生式的组成	正规式由正文字符和操作等组成[4]， 如正规式： [X－Z] 表示 X，Y，Z 中任一个 X* 表示 0 到多个 X， X+ 表示 1 到多个 X。	产生式形如： a: body; a 代表非终极符名， body 代表 0 到多个终极符或非终极符。

表1 （续）

	Lex	Yacc
9. 动作的组成与作用	• 当按正规式识别出一个单词时，对应的动作被执行。 • 动作由任意C语句组成，如：输入输出，调用子程序，返回被识别的单词类型等。 • 动作可对yytext中的内容作处理。yytext是一个字符数组，存有当前被识别的单词的字符串。	• 当按产生式归约时，对应的动作被执行。 • 动作可由任意C语句组成，如：修改全局变量，构造语法树，做语义检查等。 • 动作中可使用下述伪变量： $S, $1, $2, …, $n $$——用于存放动作的返回值(即产生式左边非终极符携带的值)。 S1, $2, …, $n——用于引用产生式右部分量的值(即先前的动作或词法分析器的返回值)。
10. 子程序集的组成	任意C子程序可供所有动作调用	同左
11. 规则二义性的处理	当输入能与多个正规式匹配时，处理办法是： (1) 最长的规则优先匹配； (2) 等长的规则中，先列出的规则优先匹配。	当文法规则集存在移进/归约冲突和归约/归约冲突时，处理办法是： (1) 利用说明集中的优先级说明； (2) 移进/归约冲突时，做移进； (3) 归约/归约冲突时，按先列出的规则进行归约。
12. 其它	Lex程序在识别模式的能力上超过理论上纯粹的正规式。 例如，Lex规则能识别少量周围上下文。 如正规式： ab/cd 匹配字符串ab仅当其后跟着cd。	一般语义动作和词法分析器返回的值被定义为整型。但用户也可自定义其它类型，包括联合类型。若定义联合类型，用户在说明集中应包括： % Union { body of union } 经这一说明后，语法分析器的值栈和全局变量yylval等都将具有这个联合类型，并且，该联合的成员名要与各终极符、非终极符相联系。与终极符联系的方法是： % token、% left 等后紧跟〈联合的某成员名〉，然后再跟终极符。 与非终极符联系的方法是：在说明集中增加 % type 〈联合的某成员名〉其后跟着有关非终极符。
13. 接口	yylex( ) 供 yyparse( ) 调用时，要返回整型值作为单词类号，还要往值栈填单词值(通过向全局变量yylval赋值。	yyparse( ) 中要调用 yylex( )，它可用lex生成，也可由用户手工构造。

对Lex和Yacc的分析重点放在Lex、Yacc的输出结构和输入与输出的关系上。限于篇幅，此处仅给出分析结果(详见[5])。至于Lex、Yacc所基于的算法、生成的词法、语法分析器的工作原理可参见[1,2]。

一个完整的Lex.YY.C结构如图1所示。

一个完整的y.tab.c结构如图2所示。

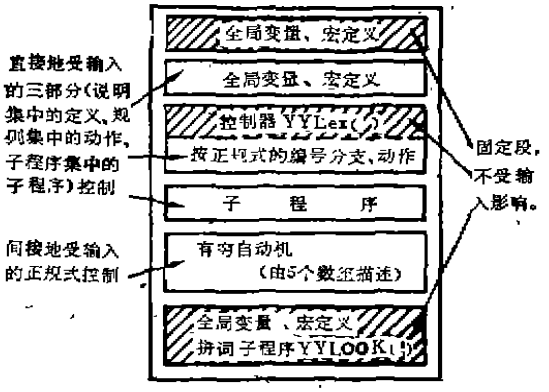

图1 Lex的输入与输出的关系

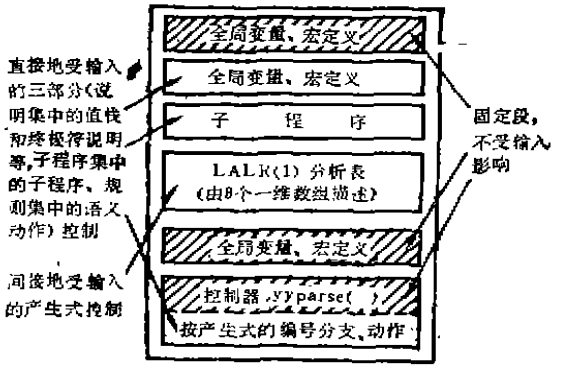

图2 Yacc的输入与输出的关系

Lex、Yacc的主要功能是分别生成随输入规则变化的有穷自动机和LALR(1)分析表、固定不变的词法分析器的控制器和语法分析器的控制器，并把输入的动作与控制器相联。而生成出来的有穷自动机和LALR(1)分析表能否正确识别某语言的语句，则要由Lex、Yacc的使用者所书写的规则来保证。衡量Lex、Yacc生成出来的词法、语法分析质量的一个重要方面，是看生成出来的有穷自动机和LALR(1)分析表的体积。根据分析，我们认为：Lex对有穷自动机所需占用的空间没有很好压缩，而Yacc这方面做得比较出色。当使用Lex时，注意力应放在节省空间上；当使用Yacc时，注意力应放在功能的正确性上。

三、Lex、Yacc在MicroSQL语言实现中的应用

在高档微机Dual System 83/20上，我们研制了一个多用户微机关系数据库管理系统NITDB，它支持的关系数据语言是MicroSQL[5]。在利用Lex和Yacc辅助实现MicroSQL的过程中，我们的目标不仅是要把这两个工具用起来，而且还要用得好，即生成出来的语法(词法)分析器的性能可与手工编制的相媲美。

用Lex、Yacc这两个工具，其关键在于根据具体的语言正确地设计出Lex、Yacc的输入文件。在这当中，文法规则集应占主导地位，因为Lex生成的程序要被Yacc生成的程序调用。

我们可以把文法规则集中每条规则的两部分((1) LALR(1)产生式，(2)语义动作)与语法分析器的两个功能((1)识别输入是不是该语言的一个语句，(2)构造语法树等中间形式，并作一些语义检查)对应起来。这样，设计MicroSQL语言的语法(词法)分析器的工作可分两步进行：

(1) 设计MicroSQL的产生式(正规式)并确认相应的语法(词法)分析器能正确地执行语法(词法)识别工作。

(2) 设计MicroSQL的中间形式，并为每个产生式配置形成中间形式的语义动作。

当使用Lex、Yacc时，需确定哪些被写成正规式哪些被写成LALR(1)产生式。我们把MicroSQL中的保留字、标识符、正整数、正实数、运算等以及分隔符等用正规式来描述，其它用产生式描述。例如，对定义关系模式的语句，有这样一个产生式：

cret: CREATE '␣' TABLE '␣' IDEN '(' f_d_l ')' ';'

其中：

f_d_l 为非终极符，是后继产生式的左部；

CREATE、TABLE、IDEN 是终极符，要在 Yacc 的说明集中用 % token 说明(见表 1(6))；

单个字符"␣"(空格)、"("、")""；"作为终极符，要用一对单引号括起来。

上述七个终极符的正规式及部分动作如下：

(1) CREATE {…, return (CREATE);}

(2) TABLE {…; return (TABLE);}

(3) [_A－Z a－z][_A－Z a－z 0－9] * {… return (IDEN);}

(4) [␣\t\n]+[注] {…; return('␣');}

(5) [␣\t\n]*\([␣\t\n] * {…; return('(');}

(6) [␣\t\n]*\)[␣\t\n] * {…; return(')');}

(7) [␣\t\n]*\; {…; return(';');}

[注] 规则(4)意为把多个空格、制表符\t、回车换行符 \n 压缩为一个空格传给语法分析器。

写出一个正确的正规式集并不困难。但对一个稍微复杂一点的语言，将其 BNF 改写成 Yacc 的产生式集往往会不满足 LALR(1)，即存在移进/归约和归约/归约冲突。如果只有个别移进/归约冲突还不要紧(见表 1(11))。倘若冲突较多，并且有归约/归约冲突，则生成出来的语法分析器就成问题。由于 MicroSQL 中的查询语句比较复杂，一开始写出的 MicroSQL 的产生式集含有不少冲突，并且还有归约/归约冲突。解决这一问题的办法是修改产生式集，把冲突特别是归约/归约冲突消去。经"调试"，使由 191 条产生式组成的 MicroSQL 的产生式集只含有 2 个移进/归约冲突，而 Yacc 自动处理移进/归约冲突的办法(优先移进)正与我们所期望的相一致。

从 MicroSQL 语言的三个部分——数据定义、数据控制和数据操纵中，抽出两个语句：

·关系模式的定义语句——create-table 语句

·查询语句——select-from-where 语句

分别作为 MicroSQL 的数据定义、数据控制和数据操纵部分的代表。create-table 语句的语法结构比较简单，没有复杂的递归运算等，语句所携带的描述信息相对固定，语法分析后的处理工作主要是把这些信息存入数据库的数据字典中，供以后 NITDB 内部或用户查阅。故将它的中间形式设计成信息表。select-from-where 语句的语法结构却比较复杂，有若干子句、布尔运算、查询块可嵌套，为了提高执行效率，对语法分析后的中间形式还要进行优化等。故将它的中间形式设计成语法树。

构造信息表、语法树的工作是分布在各产生式的语义动作中的。要构造出完整的信息表、语法树，词法分析器的返回值与产生式的动作之间、一产生式的动作与另一产生式的动作之间就要进行信息传递。因而需要发挥伪变量 $$, $1, $2, … 和全局变量 yylval (参见表 1(9)、(13))的作用。分析 Yacc 的输出文件 y. tab. c 可知：这些变量的背后都与一个深度为 150 的值栈(名为 yyv 的一维数组)相联系。yylval 可起到把单词的内部值传给值栈的媒介作用，引用 $1, $2… 相当于引用值栈中的内容，而对 $$ 赋值相当于在产生式归约、退栈后往栈顶填值。

对于我们要构造信息表、语法树的情况来说，需要利用值栈暂存单词值(有整型、实型、

字符串)、子树根(指针)等。靠 Yacc 自动把值栈定义成整型(参见表 1(12)) 是不够的。我们从节省空间、避免非终极符的类型混乱、同时又不给语法分析的后继处理带来困难出发，专门开辟一个字符缓存(1K)，把语法分析器的值栈定义成下述联合：

```
% union  {char * ssptr;
          struct sstree * sstr;
         }
```

其中，ssptr 是指向字符缓存的指针，sstr 是指向语法树的结点的指针。注意，全局变量 yylval 也为此种联合。对应于这样的值栈及字符缓存，MicroSQL 的正规式后的动作被安排成：每识别出一个单词，就把该单词的字符串存放到字符缓存中，并把指向缓存中该单词字符串的指针值和该单词的类型号传给语法分析器(通过对 yylval·ssptr 赋指针值和 return (类型号)。create-table 语句的信息表、语法树每个结点的结构用 C 语言定义如下：

```
struct ssrel                          /* creat-table 语句的信息表 */
  {char * ssrel-id;                   /* 关系名指针 */
   int sscol-no;                      /* 属性个数 */
   int ss-width;                      /* 元组长度 */
   int sskey-no;                      /* 关键字个数 */
   struct sscol
       {char * ssol-id;               /* 属性名指针 */
        int ss-type;                  /* 属性值的类型 */
        int ssk-flag;                 /* 是否作为关键字的标志 */
        int sscl-wth;                 /* 属性值的长度 */
       } ssc [50];                    /* 一个关系中属性个数可多达 50 个 */
  }
struct sstree                         /* 语法树的结点 */
  {char * ss-char;                    /* 指向字符缓存的指针 */
   short ss-type;                     /* 结点类型 */
         ⋮
   struct sstree * ssleft;            /* 指向左子树 */
   struct sstree * ssright;           /* 指向右子树 */
  }
```

MicroSQL 的中间形式设计好后，就可为每个产生式配置构造语法树等的语义动作。举例说明(见图 3)。

起初，Lex、Yacc 生成的 MicroSQL 词法、语法分析器，其功能符合我们的要求，但代码冗长，尤其是 Lex 产生的词法分析器达到 37K (C 代码)。这明显要比手工编制的大出一倍以上。经分析得知，这 37K 的程序模块中，一部分是我们自编的动作，另外主要是根据正规式生成的有穷自动机。因而，要使 Lex 生成的词法分析器体积小，就要使输给 Lex 的正规式少、动作精。但正规式不能轻易减少，否则会影响词法分析器的功能，而自编的动作只占 4.3K，再精炼也少不了多少。由此可以联想到，国外在 UNIX 上实现的关系数据库管理系

```
sel-cla: SELECT'_' se-ls
          {sspoint=ssmalloc( );                    [注 1]
          if(sspoint==NULL) YYERROR;               [注 2]
          sspoint→ss-char=$1;                      [注 3]
          sspoint→ss-type=100;                     [注 4]
          sspoint→ssleft=$3;                       [注 5]
          sspoint→ssright=NULL;
          $$=sspoint;                              [注 6]
         };
```

[注 1] ssmalloc ( ) 是在 Yacc 的输入文件的子程序集中定义的函数，功能是申请一个结点空间。

[注 2] 若没有结点空间，报错。

[注 3][注 4] 往刚申请来的结点里填指向字符缓存里的 SELECT 的指针值，填结点类型号(100)。

[注 5] 把 se-ls 子树根与新结点联系起来。

[注 6] 把新的子树根传给非终极符 se-ls。

图 3 MicroSQL 的一个产生式及语义动作(简化)

统 INGRES、MRS，为什么只利用 Yacc 而没利用 Lex。在微机内存空间不富裕的情况下，面对这过大的词法分析器，我们是否也得象 INGRES 那样，把Lex 冷落一旁，而重用手编?

当进一步对 MicroSQL 的词法规则，正规式集的书写规律以及 Lex 的输出内容作了分析以后，我们便找到了一种仍用 Lex 的解决办法。

一般，一个语言的保留字都是标识符的特例，若用手工构造词法分析器，常常是造一张保留字表，每当词法分析器识别出一个标识符时，需先查一下保留字表，看看它是否为保留字。用 Lex 生成词法分析器时，只需写出识别每一保留字的正规式和标识符的正规式，并把标识符的正规式放在所有保留字的正规式之后（参见表 1(11)），生成出来的词法分析器的功能就可与手工构造的一样。显然，Lex 提供的这种手段用起来很方便。然而，正是这一方便容易把用户迷住，使得生成出来的词法分析器比较庞大。

Lex 输出的有穷自动机的体积随输入的正规式增多而猛涨。一个语言往往会有不少保留字(MicroSQL 有 52 个)，若将每个保留字都作为正规式列于规则集中，正规式的数目就比较多，Lex 生成出来的词法分析器自然就小不了。但如果改成把识别保留字的工作压到标识符的正规式后的动作中去做，则大量的保留字规则可去掉，词法分析器的体积将会大大缩小而功能并不减弱。我们用这种方法，使 MicroSQL 的词法分析器由原先 37K 降至 17K (C 代码)。下列是 MicroSQL 的部分词法规则(改进前后对照给出)：

```
SELECT { COPY STRING[注]
        return (SELECT);
       }
FRO M { COPYSTRING
        return (FROM);
       }
[_A--Z a-z][_A-Z a-z 0-9] *
  {COPYSTRING
    return (IDEN);
  }
```

```
[_A-t a-z][_A-Z a-z 0-9]*
  {if (stromp(yytext, "SELECT") ==∅)
    {COPYSTRING return (SELECT); }
  if (stromp (yytext, "FRON") ==∅)
    {COPYSTRING
      return (FROM); }
     COPYSTRING
   return (IDEN);
   }
```

[注] COPYSTRING 是在说明集中定义的宏，功能是把单词的字符串拷贝到字符缓存中，并把当前字符缓存的指针值传给语法分析器。

(下转封三)