实验三 用 Flex 和 Bison 构造扩充的 PL/0 语言的编译器

软件 31 罗鹏魁 2003010655

一、实验目的及要求

实现满足以下EBNF描述的PL/0语言的编译器

```
<程序>          ::=<分程序>.
<分程序>        ::=[<常量说明部分>][<变量说明部分>][<过程说明部分>]<语句>
<常量说明部分>  ::= CONST <常量定义> { , <常量定义> } ;
<常量定义>      ::= <标识符> = <无符号整数>
<无符号整数>    ::=<数字>{<数字>}
* <变量说明部分>   ::= VAR <变量声明> { , <变量声明> } ;
* <变量声明>       ::= <标识符> | <标识符> '(' <数组界> : <数组界> ')'
* <数组界>         ::= <标识符> | <无符号整数>
<标识符>        ::= <字母>{<字母>|<数字>}
<过程说明部分>  ::= <过程首部><分程序> ; {<过程说明部分>}
<形式参数>      ::= <标识符>
<过程首部>      ::= PROCEDURE <标识符> [ '(' <形式参数>{, <形式参数>} ')' ] ;
<语句>          ::= <赋值语句> | <复合语句> | <条件语句> | <当型循环语句>
                    |<重复语句> | <过程调用语句> | <读语句> | <写语句> | ε
<赋值语句>      ::= <变量引用> := <表达式>
* <变量引用>       ::= <标识符> | <标识符> '(' <表达式> ')'

<复合语句>      ::= BEGIN <语句>{; <语句>} END
* <条件语句>       ::= IF <条件> THEN <语句> [ELSE <语句>]
<条件>          ::= <表达式><关系运算符><表达式> | ODD<表达式>
<表达式>        ::= [+|-]<项>{<加法运算符><项>}
<项>            ::= <因子>{ <乘法运算符><因子> }
* <因子>           ::= <变量引用> | <无符号整数> | '(' <表达式> ')'
<加法运算符>    ::= + | -
<乘法运算符>    ::= * | /
<关系运算符>    ::= = | # | < | <= | > | >=
<当型循环语句>  ::= WHILE <条件> DO <语句>
* <重复语句>       ::= REPEAT <语句> UNTIL <条件>
<过程调用语句>  ::= CALL  <标识符>['('<传值参数> {,<传值参数> }')']
<传值参数>      ::= <表达式>
<读语句>        ::= READ '(' <变量引用> { , <变量引用> } ')'
<写语句>        ::= WRITE '(' <表达式> { , <表达式> } ')'
<字母>          ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|
                    w|x|y|z|A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|
                    S|T|U|V|W|X|Y|Z
<数字>          ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

二、主要设计

（一）词法分析脚本 pl3.l 的设计

略：基本上沿用第二次实验的脚本。

（二）语法分析脚本 pl3.y 的设计

语法分析脚本较上次实验有很大变动，因为加入属性文法之后，会凭空引入很多的冲突，大部分移进-规约冲突可以利用 bison 的冲突默认处理机制解决（虽然这好像不被老师允许）。但是还有很多冲突无法利用默认规则正确解决，主要集中在 if-then-else 和 while-do 上，其中 if-then-else 主要是悬挂二义性的问题。实验二中我是用指定优先级的方法来消除 if-else 悬挂二义性的，但是跟属性文法结合出现了一时没法解决的冲突。为解决这两个冲突，我从 bison 手册中得到了启发，把语句 statement 分成两大类：CloseStatement 和 OpenStatement，见下面的代码：

```
 99  statement:
100      Cstmt
101      |Ostmt
102      ;
103  Cstmt:
104      var_ref COLEQ expression                    {asgn_stmt($<name>1);}
105      |IF cond THEN tmp_jpc Cstmt tmp_else ELSE Cstmt {code[$<val>6].a=cx; }
106      |BEGINSYM statement_list ENDSYM
107      |CALL id arguments                          {call_stmt($<name>2,$<val>3); }
108      |READ '(' read_list ')'                     { }
109      |WRITE '(' write_list ')'                   {gendo(OPR, 0, 15); }
110      |WHILE tmp_cx cond tmp_jpc DO Cstmt         {gendo(JMP,0,$<val>2);code[$<val>4].a=cx; }
111      |REPEAT tmp_cx statement UNTIL cond         {gendo(JPC,0,$<val>2);}
112      |
113      ;
114  Ostmt:
115      IF cond THEN tmp_jpc statement                  {code[$<val>4].a=cx; }
116      |IF cond THEN tmp_jpc Cstmt tmp_else ELSE Ostmt {code[$<val>6].a=cx; }
117      |WHILE tmp_cx cond tmp_jpc DO Ostmt             {gendo(JMP,0,$<val>2); code[$<val>4].a=cx;}
118      ;
```

OpenStatement是专为if-then设立的语句，它指的是所有不以else结尾的if语句，这种语句有三类，一类是IF condition THEN statement（要避免无穷递归，这个statement必须是非if的CloseStatement）；一类是IF condition THEN CloseStatement ELSE OpenStatement；最后一类是WHILE condition DO OpenStatement，因为while语句是后向开放的，仍有可能以then语句结束。OpenStatement的结尾最后是要由CloseStatement来定义的。

CloseStatement是排除掉OpenStatement之外的常规statement，除了read, write, call, complex, repeat, assignment语句之外，还包括以else子句结尾的if-then-else语句，和以else字句结尾的while语句。

举例：对于可能产生悬挂二义性的语句“IF cond1 THEN IF cond2 THEN stmt1 ELSE stmt2”（stmt1和stmt2都是闭合的），应用的正确规则（略去属性文法）依次为：

statement -> IF cond THEN statement

statement -> Cstmt

Cstmt -> IF cond THEN Cstmt ELSE Cstmt

仔细跟踪可知整个过程没有任何冲突，并且其他的规则应用序列均不能正确分析。

（三）符号表和虚拟机设计

基本上按照老师提供的框架，完成了符号表的操作和虚拟机的操作。一些在 pl3.y 中频繁用到的功能也写成函数放在 table.c 中供调用。

以下是一些代码片断：

```
typedef struct
{
    ident_t name;
    int     kind;
    int     val, level, adr, para, beginindex, endindex, size;
} tableitem;
tableitem   table[IDMAXCOUNT];
```

```
typedef struct
{
    int f, l, a;
}instruction;
instruction code[CODEMAXCOUNT];
```

```
char name[][5]=
{
    {"lit"},
    {"opr"},
    {"lod"},
    {"sto"},
    {"cal"},
    {"int"},
    {"jmp"},
    {"jpc"},
    {"lodx"},
    {"stox"},
    {"pop"}
};
```

```
i=code[p++];
switch (i.f)
{
case LIT:
    t=t+1; s[t]=i.a; break;
case LOD:
    t=t+1; s[t]=s[base(i.l, s, b)+i.a]; break;
case STO:
    s[base(i.l, s, b)+i.a]=s[t]; t=t-1; break;
case JMP:
    p=i.a; break;
case JPC:
    if (s[t]==0) p=i.a; t=t-1; break;
case LODX:  /*load a element from array*/
    s[t]=s[base(i.l, s, b)+i.a+s[t]]; break;
case STOX:  /*store value to array element*/
    s[base(i.l, s, b)+s[t-1]]=s[t]; t=t-2;  break;
case CAL:
    s[t+1]=base(i.l, s, b); s[t+2]=b; s[t+3]=p;temp=t+3; b=t+1; p=i.a; break;
case INI:
    t=t+i.a; break;
case POP:
    t-=i.a; break;
case OPR:
```

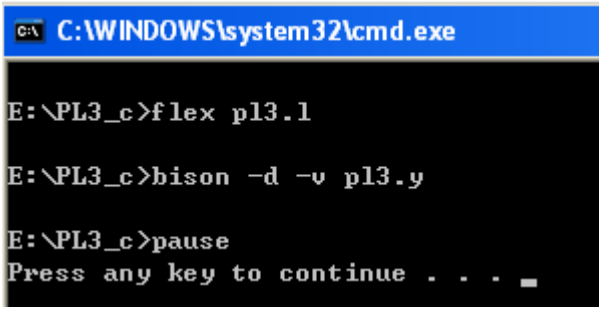
其中 LODX 和 STOX 是存取数组元素的指令。

```
case OPR:
    switch(i.a)
    {
        case 0:
            t=b-1; p=s[t+3]; b=s[t+2]; break;
        case 1:
            s[t]=-s[t]; break;
        case 2:
            t=t-1; s[t]=s[t] + s[t+1]; break;
        case 3:
            t=t-1; s[t]=s[t] - s[t+1]; break;
        case 4:
            t=t-1; s[t]=s[t] * s[t+1]; break;
        case 5:
            t=t-1; s[t]=s[t] / s[t+1]; break;
        case 6:
            s[t]=(s[t] % 2 == 1); break;
        case 8:
            t=t-1; s[t]=(s[t] == s[t+1]); break;
        case 9:
            t=t-1; s[t]=(s[t] != s[t+1]); break;
        case 10:
            t=t-1; s[t]=(s[t]<s[t+1]); break;
        case 11:
            t=t-1; s[t]=(s[t]>=s[t+1]); break;
        case 12:
            t=t-1; s[t]=(s[t]>s[t+1]); break;
        case 13:
            t=t-1; s[t]=(s[t]<=s[t+1]); break;
        case 14:
            printf(" %d", s[t]); t=t-1; break;
        case 15:
            printf("\n"); break;
        case 16:
            t=t+1; printf(" >> "); scanf("%d", &s[t]); break;
    }
```

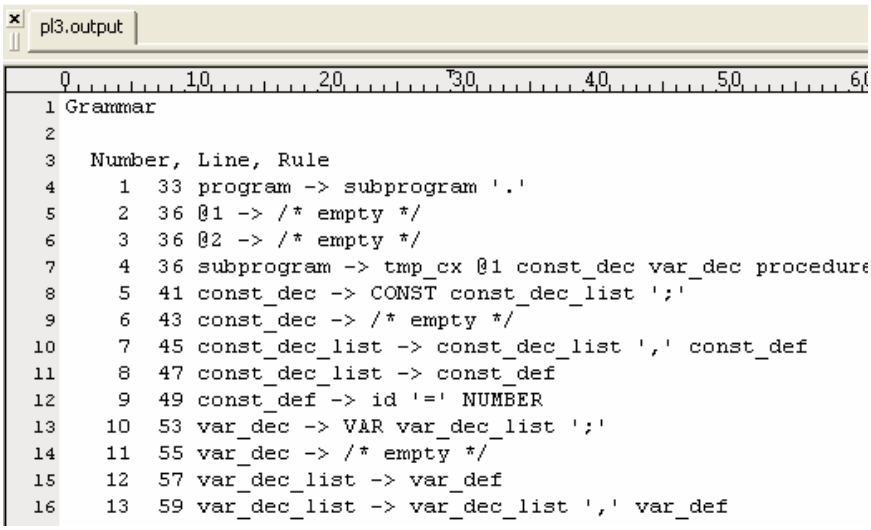
三、编译及测试

（一）PL/0 编译器的生成

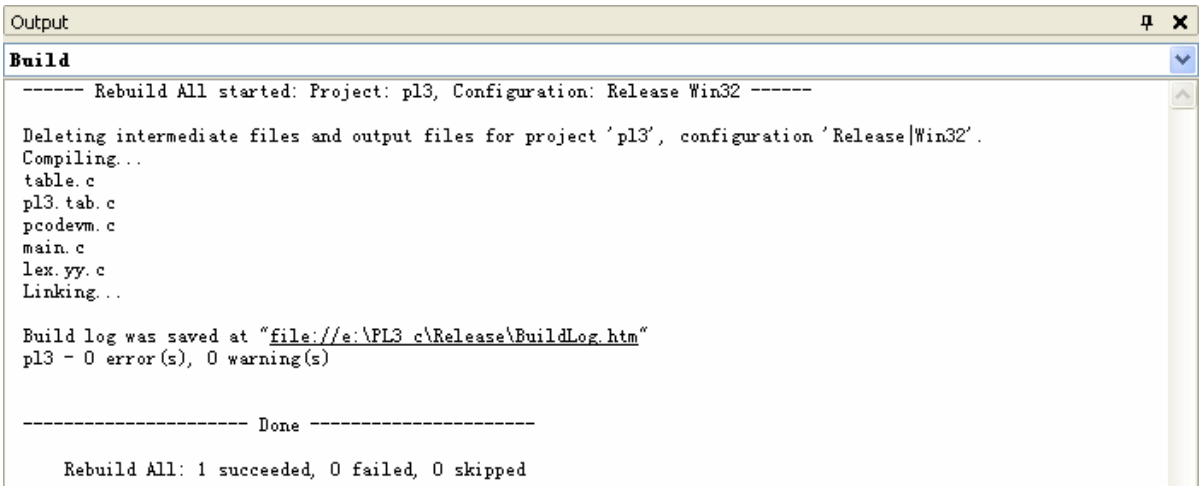
成功根据脚本生成 lex.yy.c 和 pl3.tab 源代码：



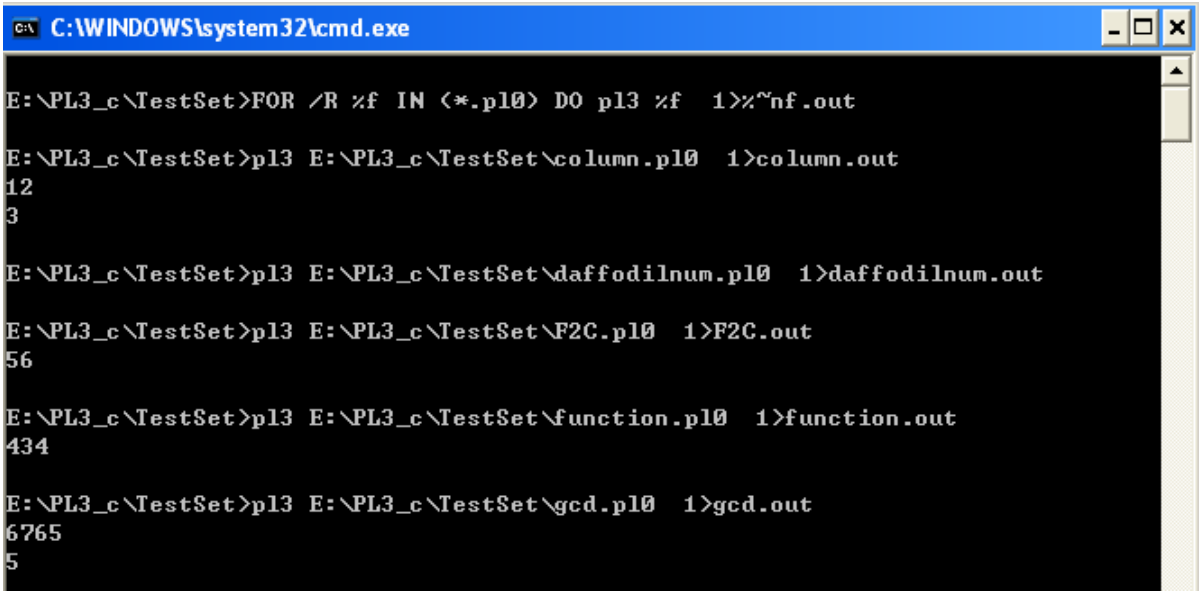
这是 pl3.output 文件输出情况：

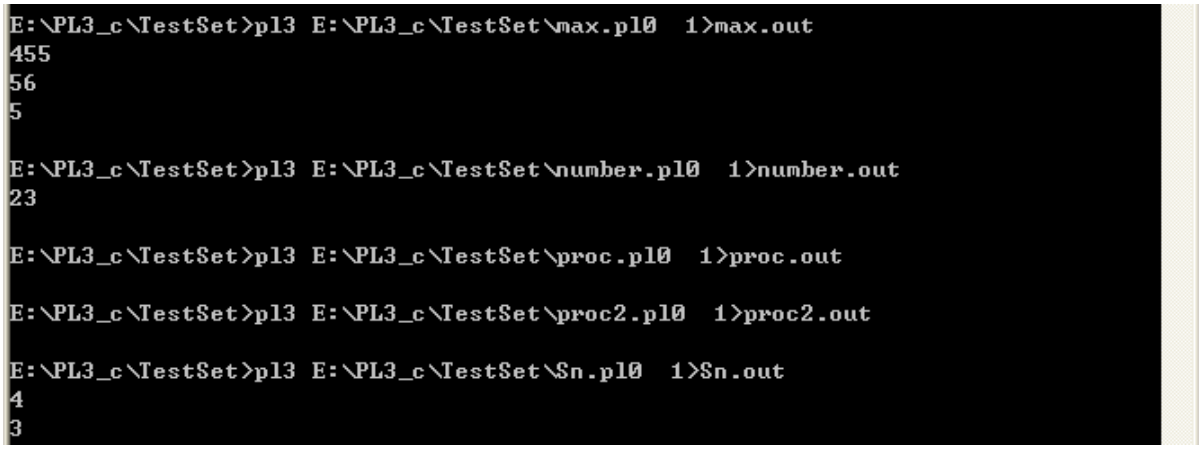


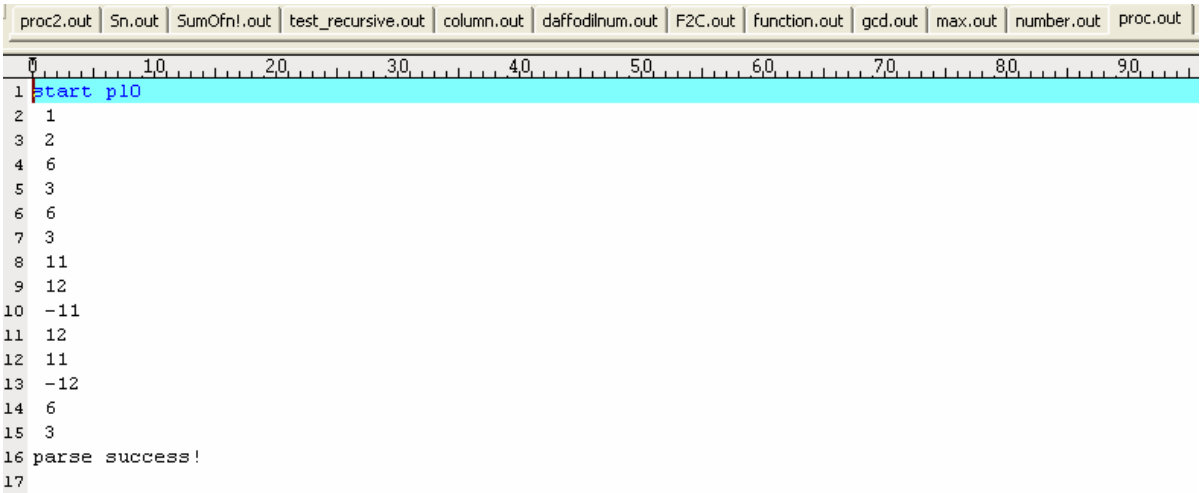
工程在 Windows XP + Visual C++7.0 环境中成功编译连接，得到满足脚本描述的 PL/0 编译程序 pl3.exe



（二）生成的 PL/0 编译器的测试







可以看出，老师提供的所用代码均正确通过测试。