实验：语法分析器

周炎亮 2018202196 信息学院

一、实验概述

1. 实验内容

用bison工具生成一个PL/0语言的语法 分析程序，对PL/0源程序进行语法分析。

输入：

pl/0源程序，要求测试的pl0源文件包含

支持的所有语法成分（含扩展成分）

嵌套过程（三层）

并列过程

输出：

按归约顺序用到的语法规则序列（输出到文件）

语法树（或能表示语法单位的层次结构关系的其他形式）

2. 实验环境：

语法分析器生成工具：bison

编程语言：C

二、实验思路

1. 改造EBNF范式

在上一次词法分析器的实验中提供了EBNF范式的资料，也就是PL/0语法的规约规则，但提供的资料中EBNF的部分范式无法直接满足yacc语法的规则，因此需要对其进行改造。

(1). {}的改造

在EBNF范式中，{}之间的部分代表可重复（即出现0次及以上），而yacc语法中没有表示“可重复”的符号，因此需要将可重复的部分用**递归**来表示。在查资料时发现，在yacc中使用右递归的方法会在所有元素进栈后才进行规约，而如果使用左递归则栈中的元素不会超过三个，为了节省内存空间，修改后的递归形式一律使用左递归。

eg: `<复合语句> ::= BEGIN<语句>{;<语句>}END`

可以看到上述范式中<语句>是可重复的且至少出现一次，对其进行改造后便成为如下范式：

`<复合语句> ::= BEGIN<语句集>END`

`<语句集> ::= <语句集>;<语句>|<语句>`

(2). []的改造

在EBNF范式中，[]之间的部分代表可选（即出现0次或1次），而yacc语法中也没有表示"可选"的规则，虽然枚举是一种方法，但在同一条规约式中若[]部分过多会产生很多枚举方案，过于复杂。因此可以使用如下方法：在[]之间部分的非终结符A规约式中加入一条A→ε的式子，便可在原式中将[]去掉。

eg: `<分程序> ::= [<常量说明部分>][<变量说明部分>][<过程说明部分>]<语句>`

上述规约式中有三个可重复的部分，如果将其全部列举会产生$2^3 = 8$中组合，过于繁琐，因此要用上述方法将[]消去。以<常量说明部分>为例，在对<常量说明部分>原先的规约式进行其他的改造（如去掉{}）后，现有的规约式为：`<常量说明部分> ::= CONST<常量定义集>;`。通过加上一条`<常量说明部分> ::= <空>`的规则后，便可成为以下形式：

`<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<常量定义集>;|<空>`

(3). 扩展语法的加入

数组

定义数组在定义时的形式为`标识符(整型，整型)`的形式，其中两个整型分别表示下界和上界，又定义数组元素在被调用时的形式为`标识符(整型)`或`标识符(标识符)`。因为数组元素在本质上也是标识符，因此可以将其放入标识符的规约式中。又因为在定义过程和调用过程时的数组表现形式不同，因此又将标识符分为两类：定义过程标识符和（普通的）标识符。

`<定义过程标识符> ::= IDENTIFIER'('INTEGER,INTEGER')'|IDENTIFIER`

`<标识符> ::= IDENTIFIER'('IDENTIFIER')'| IDENTIFIER'('INTERGER')'|IDENTIFIER`

其中IDENTIFIER和INTEGER分别为标识符和整型，因为直接是由词法分析器识别的，因此可以当作终结符处理。

字符型/浮点型

字符型和浮点型的加入只需注意两个部分：定义过程和调用过程。

对于定义过程，定义char为定义字符串的关键字，定义real为定义浮点型的关键字，因此只需在var的部分将其插入即可：

`<变量说明> ::= VAR<定义过程标识符集>|CHAR<定义过程标识符集>|REAL<定义过程标识符集>`

而调用过程，即直接调用字符串和浮点型而非标识符，可以借鉴的是直接调用整型的部分，该部分出现在表达式的规约中，原本的规约式为：`<因子> ::= <标识符>|<无符号整数>|'('<表达式>')'`，因此只需要将字符串STRING和浮点型FLOAT插入到右边即可：

`<因子> ::= '('<表达式>')'|<标识符>|INTEGER|FLOAT|STRING`

for型循环语句

定义for型循环语句的形式为：`<for型循环语句> := FOR<赋值语句>TO<表达式>[STEP<无符号整数>]DO<语句>`

因此yacc可接受的的语法规则即为：

`<for型循环语句> := FOR <赋值语句> TO <表达式> STEP INTEGER DO<语句>| FOR <赋值语句> TO <表达式> DO<语句>`

repeat型循环语句

定义repeat型循环语句为：`<repeat型循环语句> ::= REPEAT<语句集>UNTIL<条件>`

可以注意到repeat型循环语句中间使用的是<语句集>（即<语句>的重复）而非<语句>，这是因为如while和for的中间部分是以begin和end开头结尾的<复合语句>，而<复合语句>可以归约成<语句>，因此可以直接使用<语句>，但根据上次词法实验中下发的PL/0的文件发现repeat和until之间没有begin和end，因此使用<语句集>来代替<复合语句>构成的<语句>

(4). 改进后的EBNF范式

```
<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<常量定义集>;|<空>
<常量定义集> ::= <常量定义集>,<常量定义>|<常量定义>
<常量定义> ::= IDENTIFIER:=INTEGER
<变量说明部分> := <变量说明部分集>| <空>
<变量说明部分集> := <变量说明部分集><变量说明>;|<变量说明>;
<变量说明> ::= VAR<定义过程标识符集>|CHAR<定义过程标识符集>|REAL<定义过程标识符集>
<定义过程标识符集> ::= <定义过程标识符集>,<定义过程标识符>|<定义过程标识符>
<定义过程标识符> ::= IDENTIFIER’(’INTEGER,INTEGER’)’|IDENTIFIER
<过程说明部分> ::= <过程说明部分集>|<空>
<过程说明部分集> ::= <过程说明部分集><过程首部><分程序>;|<过程首部><分程序>;
<过程首部> ::= PROCEDURE IDENTIFIER;
<语句集> ::= <语句集>;<语句>|<语句>
<语句> ::= <赋值语句>|<复合语句>|<条件语句>|<当型循环语句>|<repeat型循环语句>|<for型循环语句>|<过程调用语句>|<读语句>|<写语句>|<空>
<赋值语句> ::= <标识符>:=<表达式>
<标识符> ::= IDENTIFIER’(‘IDENTIFIER’)’|IDENTIFIER’(‘INTERGER’)’|IDENTIFIER
<复合语句> ::= BEGIN<语句集>END
<条件> ::= <表达式><关系运算符><表达式>|ODD<表达式>
<条件语句> ::= IF<条件>THEN<语句>
<表达式> ::= <加法运算符><项集>|<项集>
<项集> ::= <项集><加法运算符><因子集>|<因子集>
<因子集> ::= <因子集><乘法运算符><因子>| <因子>
<因子> ::= ’(‘<表达式>’)’|<标识符>|INTEGER|FLOAT|STRING
<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>
<repeat型循环语句> ::= REPEAT<语句集>UNTIL<条件>
<for型循环语句> := FOR <赋值语句> TO <表达式> STEP INTEGER DO<语句>| FOR <赋值语句> TO <表达式> DO<语句>
<过程调用语句> ::= CALL IDENTIFIER
<读语句> ::= READ’(‘<标识符集>’)’
<标识符集> ::= <标识符集>,<标识符>|<标识符>
<写语句> ::= WRITE’(‘<表达式集>’)’
<表达式集> ::= <表达式集>,<表达式>|<表达式>
```

2. 语法分析部分代码

(1). 辅助定义

在改进EBNF范式后，便可根据现有范式进行yacc部分代码的编写，首先在辅助定义中定义终结符和非终结符等一系列变量：

```
%union{
         struct Node *token_p;
}
%type <token_p> program p_program con_des var_dess var_des var_d proc_des states
con_defs con_def def_ids def_id proc_defs proc_head
%type <token_p> state assign_st comp_st if_st while_st repeat_st for_st
procedure_st read_st write_st
%type <token_p> id ifs expr terms factors factor exprs ids
%token <token_p> IF THEN WHILE DO READ WRITE CALL BEGIN_ END CONST VAR PROCEDURE
ODD FOR TO STEP REPEAT UNTIL CHAR REAL
%token <token_p> INTEGER FLOAT STRING IDENTIFIER OPERATOR SEM COMMA EQUAL COLON
%left <token_p> ADDSUB
%left <token_p> MULTIDIV
%left <token_p> DOT LP RP
```

其中Node结构体对应语法树中的一个结点，为了方便之后语法树的构建，其定义和使用会在之后提到。

(2). 语法规则

之后便是根据改进的EBNF范式编写语法规则段。

因为输出文件要有“按归约顺序用到的语法规则序列”，因此每一次的规约结束后，都输出该条规约式使用的规约规则。

又因为最后还要输出语法树，因此每一次都使用Node结构体即相关函数构建语法树。因为yacc使用的是自底向上的LALR文法，因此在规约式右部规约成左部时，在语法树中的对应体现即为右部的每个终结符或非终结符作为子结点连接到作为父结点的左部非终结符之下。所以每次规约完后，都要为左部非终结符使用newNode函数新建一个结点（因为在这条规则之前，左部的结点还未曾出现过），再用insert函数将右部的每个结点作为子结点插入到父结点之下，最后将该父结点赋值给左部。

代码如下：

```
program : p_program DOT {printf("<程序> ::= <分程序>.\n");
                         p=newNode("program");insert(p,$1);insert(p,$2);$$=p;}
    ;
p_program : con_des var_des proc_des state
            {printf("<分程序> ::= <常量说明部分><变量说明部分集><过程说明部分><语句
>\n");

p=newNode("p_program");insert(p,$1);insert(p,$2);insert(p,$3);insert(p,$4);$$=p;
}
    ;
con_des : CONST con_defs SEM {printf("<常量说明部分> ::= CONST<常量定义集>;\n");

p=newNode("con_des");insert(p,$1);insert(p,$2);insert(p,$3);$$=p;}
    | {printf("<常量说明部分> ::= <空>\n");
        p=newNode("NULL");$$=p;}
    ;
con_defs : con_defs COMMA con_def {printf("<常量定义集> ::= <常量定义集>,<常量定义
>\n");

p=newNode("con_defs");insert(p,$1);insert(p,$2);insert(p,$3);$$=p;}
    | con_def {printf("<常量定义集> ::= <常量定义>\n");
                p=newNode("con_defs");insert(p,$1);$$=p;}
    ;
```

```
con_def : IDENTIFIER EQUAL INTEGER {printf("<常量定义> ::=
IDENTIFIER:=INTEGER\n");

p=newNode("con_def");insert(p,$1);insert(p,$2);insert(p,$3);$$=p;}
    ;
var_des : var_dess{printf("<变量说明部分> := <变量说明部分集
>\n");p=newNode("var_des");insert(p,$1);$$=p;}
    | {printf("<变量说明部分> ::= <空>\n");
        p=newNode("NULL");$$=p;}
    ;
var_dess : var_dess var_d SEM{printf("<变量说明部分集> ::= <变量说明部分集><变量说明
>;\n");

p=newNode("var_dess");insert(p,$1);insert(p,$2);insert(p,$3);$$=p;}
    | var_d SEM{printf("<变量说明部分集> ::= <变量说明>;\n");
                p=newNode("var_dess");insert(p,$1);insert(p,$2);$$=p;}
var_d : VAR def_ids {printf("<变量说明部分> ::= VAR<定义过程标识符集>;\n");

p=newNode("var_des");insert(p,$1);insert(p,$2);$$=p;}
    | CHAR def_ids {printf("<变量说明部分> ::= CHAR<定义过程标识符集>;\n");
                        p=newNode("var_des");insert(p,$1);insert(p,$2);$$=p;}
    | REAL def_ids {printf("<变量说明部分> ::= REAL<定义过程标识符集>;\n");
                        p=newNode("var_des");insert(p,$1);insert(p,$2);$$=p;}
    ;
def_ids : def_ids COMMA def_id {printf("<定义过程标识符集> ::= <定义过程标识符集>,<定
义过程标识符>\n");

p=newNode("def_ids");insert(p,$1);insert(p,$2);insert(p,$3);$$=p;}
    | def_id {printf("<定义过程标识符集> ::= <定义过程标识符>\n");
                p=newNode("def_ids");insert(p,$1);$$=p;}
    ;
def_id : IDENTIFIER LP INTEGER COLON INTEGER RP
            {printf("<定义过程标识符> ::= IDENTIFIER'('INTEGER,INTEGER')'\n");

p=newNode("def_id");insert(p,$1);insert(p,$2);insert(p,$3);insert(p,$4);insert(p
,$5);insert(p,$6);$$=p;}
    | IDENTIFIER {printf("<定义过程标识符> ::= IDENTIFIER\n");
                    p=newNode("def_id");insert(p,$1);$$=p;}
    ;
proc_des : proc_defs {printf("<过程说明部分> ::= <过程说明部分集>\n");
                        p=newNode("proc_des");insert(p,$1);$$=p;}
    | {printf("<过程说明部分> ::= <空>\n");p=newNode("NULL");$$=p;}
    ;
proc_defs : proc_defs proc_head p_program SEM
            {printf("<过程说明部分集> ::= <过程说明部分集><过程首部><分程序>;\n");

p=newNode("proc_defs");insert(p,$1);insert(p,$2);insert(p,$3);insert(p,$4);$$=p;
}
    | proc_head p_program SEM {printf("<过程说明部分集> ::= <过程首部><分程序>;\n");

p=newNode("proc_defs");insert(p,$1);insert(p,$2);insert(p,$3);$$=p;}
    ;
proc_head : PROCEDURE IDENTIFIER SEM {printf("<过程首部> ::= PROCEDURE
IDENTIFIER;\n");

p=newNode("proc_head");insert(p,$1);insert(p,$2);insert(p,$3);$$=p;}
    ;
```

```
states : state {printf("<语句集> ::= <语句
>\n");p=newNode("states");insert(p,$1);$$=p;}
    | states SEM state {printf("<语句集> ::= <语句集>;<语句>\n");

p=newNode("states");insert(p,$1);insert(p,$2);insert(p,$3);$$=p;}
    ;
state : assign_st {printf("<语句> ::= <赋值语句
>\n");p=newNode("state");insert(p,$1);$$=p;}
    | comp_st {printf("<语句> ::= <复合语句
>\n");p=newNode("state");insert(p,$1);$$=p;}
    | if_st {printf("<语句> ::= <条件语句
>\n");p=newNode("state");insert(p,$1);$$=p;}
    | while_st {printf("<语句> ::= <当型循环语句
>\n");p=newNode("state");insert(p,$1);$$=p;}
    | repeat_st {printf("<语句> ::= <REPEAT型循环语句
>\n");p=newNode("state");insert(p,$1);$$=p;}
    | for_st {printf("<语句> ::= <FOR型语句
>\n");p=newNode("state");insert(p,$1);$$=p;}
    | procedure_st {printf("<语句> ::= <过程调用语句
>\n");p=newNode("state");insert(p,$1);$$=p;}
    | read_st {printf("<语句> ::= <读语句
>\n");p=newNode("state");insert(p,$1);$$=p;}
    | write_st {printf("<语句> ::= <写语句
>\n");p=newNode("state");insert(p,$1);$$=p;}
    | {printf("<语句> ::= <空>\n");p=newNode("NULL");$$=p;}
    ;
assign_st : id EQUAL expr {printf("<赋值语句> ::= <标识符>:=<表达式>\n");

p=newNode("assign_st");insert(p,$1);insert(p,$2);insert(p,$3);$$=p;}
    ;

id : IDENTIFIER LP IDENTIFIER RP {printf("<标识符> ::=
IDENTIFIER'('IDENTIFIER')'\n");

p=newNode("id");insert(p,$1);insert(p,$2);insert(p,$3);insert(p,$4);$$=p;}
    | IDENTIFIER LP INTEGER RP {printf("<标识符> ::= IDENTIFIER'('INTEGER')'\n");

p=newNode("id");insert(p,$1);insert(p,$2);insert(p,$3);insert(p,$4);$$=p;}
    | IDENTIFIER {printf("<标识符> ::=
IDENTIFIER\n");p=newNode("id");insert(p,$1);$$=p;}
    ;
comp_st : BEGIN_ states SEM END {printf("<复合语句> ::= BEGIN<语句集>;END\n");

p=newNode("comp_st");insert(p,$1);insert(p,$2);insert(p,$3);insert(p,$4);$$=p;}
    | BEGIN_ proc_des END {printf("<复合语句> ::= BEGIN<过程说明部分>END\n");

p=newNode("comp_st");insert(p,$1);insert(p,$2);insert(p,$3);$$=p;}
    ;
ifs : expr OPERATOR expr {printf("<条件> ::= <表达式><关系运算符><表达式>\n");

p=newNode("ifs");insert(p,$1);insert(p,$2);insert(p,$3);$$=p;}
    | ODD expr {printf("<条件> ::= ODD<表达式
>\n");p=newNode("ifs");insert(p,$1);insert(p,$2);$$=p;}
    ;
if_st : IF ifs THEN state {printf("<条件语句> ::= IF<条件>THEN<语句>\n");

p=newNode("if_st");insert(p,$1);insert(p,$2);insert(p,$3);insert(p,$4);$$=p;}
    ;
```

```
expr : ADDSUB terms {printf("<表达式> ::= <加法运算符><项集
>\n");p=newNode("expr");insert(p,$1);insert(p,$2);$$=p;}
    | terms {printf("<表达式> ::= <项集>\n");p=newNode("expr");insert(p,$1);$$=p;}
    ;
terms : terms ADDSUB factors {printf("<项集> ::= <项集><加法运算符><因子集>\n");

p=newNode("terms");insert(p,$1);insert(p,$2);insert(p,$3);$$=p;}
    | factors {printf("<项集> ::= <因子集
>\n");p=newNode("terms");insert(p,$1);$$=p;}
    ;
factors : factors MULTIDIV factor {printf("<因子集> ::= <因子集><乘法运算符><因子
>\n");

p=newNode("factors");insert(p,$1);insert(p,$2);insert(p,$3);$$=p;}
    | factor {printf("<因子集> ::= <因子
>\n");p=newNode("factors");insert(p,$1);$$=p;}
    ;
factor : LP expr RP {printf("<因子> ::= '('<表达式
>')'\n");p=newNode("factor");insert(p,$1);insert(p,$2);insert(p,$3);$$=p;}
    | id {printf("<因子> ::= <标识符>\n");p=newNode("factor");insert(p,$1);$$=p;}
    | INTEGER {printf("<因子> ::=
INTEGER\n");p=newNode("factor");insert(p,$1);$$=p;}
    | FLOAT {printf("<因子> ::= FLOAT\n");p=newNode("factor");insert(p,$1);$$=p;}
    | STRING {printf("<因子> ::=
STRING\n");p=newNode("factor");insert(p,$1);$$=p;}
    ;

while_st : WHILE ifs DO state {printf("<当型循环语句> ::= WHILE<条件>DO<语句>\n");

p=newNode("while_st");insert(p,$1);insert(p,$2);insert(p,$3);insert(p,$4);$$=p;}
    ;
repeat_st : REPEAT states UNTIL ifs {printf("<repeat型循环语句> ::= REPEAT<语句
>UNTIL<条件>\n");

p=newNode("repeat_st");insert(p,$1);insert(p,$2);insert(p,$3);insert(p,$4);$$=p;
}
    ;
for_st : FOR assign_st TO expr STEP INTEGER DO state
        {printf("<for型循环语句> := FOR <赋值语句> TO <表达式> STEP INTEGER DO<语句
>\n");

p=newNode("for_st");insert(p,$1);insert(p,$2);insert(p,$3);insert(p,$4);insert(p
,$5);insert(p,$6);insert(p,$7);insert(p,$8);$$=p;}
    | FOR assign_st TO expr DO state
        {printf("<for型循环语句> := FOR <赋值语句> TO <表达式> DO<语句>\n");

p=newNode("for_st");insert(p,$1);insert(p,$2);insert(p,$3);insert(p,$4);insert(p
,$5);insert(p,$6);$$=p;}
    ;
procedure_st : CALL IDENTIFIER {printf("<过程调用语句> ::= CALL
IDENTIFIER\n");p=newNode("procedure_st");insert(p,$1);insert(p,$2);$$=p;}
    ;
read_st : READ LP ids RP {printf("<读语句> ::= READ'('<标识符集
>')'\n");p=newNode("read_st");insert(p,$1);insert(p,$2);insert(p,$3);insert(p,$4
);$$=p;}
    ;
ids : ids COMMA id {printf("<标识符集> ::= <标识符集>,<标识符
>\n");p=newNode("ids");insert(p,$1);insert(p,$2);$$=p;}
```

```
    | id {printf("<标识符集> ::= <标识符>\n");p=newNode("ids");insert(p,$1);$$=p;}
    ;
write_st : WRITE LP exprs RP {printf("<写语句> ::= WRITE'('<表达式集>')'\n");

p=newNode("write_st");insert(p,$1);insert(p,$2);insert(p,$3);insert(p,$4);$$=p;}
    ;
exprs : exprs COMMA expr {printf("<表达式集> ::= <表达式集>,<表达式
>\n");p=newNode("exprs");insert(p,$1);insert(p,$2);insert(p,$3);$$=p;}
    | expr {printf("<表达式集> ::= <表达式
>\n");p=newNode("exprs");insert(p,$1);$$=p;}
```

(3). 用户子程序

用户子程序段定义的有检测出错的并报错的yyerror函数和主函数main。main函数放到最后讲解，此处解释yyerror函数。

因为yacc程序中，如果在语法段的规约时出错，就会调用yyerror函数，故yyerror主要是用于debug。因此，每次出错后，都向目标输出文件输出"Error."信息，并新建一个输出文件stderr，在其中打印由词法分析部分传输的行号和列号，便于确定出错位置。

代码如下：

```
void yyerror(char* s)
{
     FILE* errdir=NULL;
     errdir=fopen("stderr","w");
     if(fout!=NULL)
     fprintf(fout,"Error.");
     fprintf(errdir,"line %d num %d error.\n",num_lines,num_chars-yyleng);
     fclose(fout);
     fclose(errdir);
     exit(1);
}
```

3. 语法树部分代码

(1). Node结构体

对于一棵树中的每个结点，最基本的信息是要有父结点p和子结点child，因为在本代码中最后树的遍历是从根节点开始的，因此父结点的信息可以省略。此外，因为语法树的一个结点不一定只有一个子结点，因此还要有记录子结点的兄弟结点brother。另外，还要有记录该结点对应的终结符或非终结符名称的label。为了便于之后使用.dot文件生成可视的语法树，还要有一个dotname变量。

代码如下：

```
struct Node
{
    char label[20];
    char dotname[20];
    struct Node *brother;
    struct Node *child;
};
```

(2). newNode函数

每次新建一个结点前，词法分析器或语法分析器会向newNode函数传输该结点的label信息。利用malloc函数成功新建结点后（不成功则报错退出），为Node中的一系列变量赋初值即可。

代码如下：

```
struct Node* newNode (char* node_name)
{
    struct Node *p=(struct Node*)malloc(sizeof(struct Node));
    if (p==NULL)
    {
        printf("Error:out of memory.\n");
        exit(1);
    }
    strncpy(p->label,node_name,20);
    p->brother=NULL;
    p->child=NULL;
    return p;
}
```

(3). insert函数

每次向一个父结点插入一个子结点时，都可能出现两种情况：父结点已有子结点和父结点未有子结点。若父结点未有子结点，则直接将其child结点赋值为当前结点即可；否则，从父结点的子结点出发，寻找所有brother结点直至为NULL，并将该子结点的brother赋值为当前结点

代码如下：

```
void insert(struct Node *parent,struct Node *child)
{

    if (child==NULL)
        return;
    if(parent->child==NULL)
    {
        parent->child=child;
    }
    else
    {
        struct Node *Child;
        Child=parent->child;
        while(Child->brother!=NULL)
            Child=Child->brother;
        Child->brother=child;
    }
}
```

(4). printtree函数

此函数用以打印一个类语法树的结构，是在得知使用graphviz工具之前使用的函数。此函数从根结点出发，每行打印一个结点的深度即其label信息，并根据结点在语法树中的深度赋予缩进。

本函数使用深度遍历的方法，递归调用自己，优先打印出子结点，再打印出其兄弟结点。

代码如下：

```
int space =0;
void printtree(struct Node *root, FILE *stream) {
  int i;
  if (root == NULL)
    return;
```

```
  for (i = 0; i < space; i++)
    fprintf(stream," ");
  fprintf(stream, "%d %s\n", space,root->label);

  if (root->child != NULL) {
    space++;
    printtree(root->child, stream);
    space--;
  }
  if(root->brother != NULL){
    printtree(root->brother, stream);
  }
}
```

(5). treedot函数

本函数用以根据语法树信息生成相应.dot文件，便于之后使用graphviz工具生成可视化的语法树。

因为.dot文件对应的边的信息是如`a->b`的形式，因此只需要用类似printtree的方式递归调用自己，遍历每个结点并输出父结点的label->子结点的label的信息即可。

代码如下：

```
void treedot(struct Node *root,FILE *output)
{
    struct Node *Child=root->child;
    while(Child!=NULL)
    {
        fprintf(output,"%s->%s;\n",root->label,Child->label);
        treedot(Child,output);
        Child=Child->brother;
    }
}
```

然而，因为语法树中终结符和非终结符的种类有限，因此同一个终结符或非终结符对应的label信息是一样的。虽然其可能在语法树的不同层次和位置中，但是在graphviz构图时会将同一个label信息的结点都视为一个结点从而构建一个错误的语法树，因此需要改进本代码。

在查找资料后发现，graphviz工具还能支持如以下格式的.dot文件：

```
    a [label="Birth of George Washington"];
    b [label="Main label"];
    a-> b;
```

虽然最后的构图的确是a指向b的边，但在图中原本的a和b被其对应的label信息所代替。因此本代码中也可以将每个结点的label信息放入[]中，而为每个结点定义一个新的名称，故需引入一个dotname变量。

对于每个结点，先将label的值赋值给dotname。判断其dotname是否被记录，若未被记录，则将其记录，并定义其记录的被次数为1；否则，将其被记录的次数加一，并在dotname后面加上该数值。此方法能保证每个结点的dotname都不同，最后只需输出`dotname [label="label"]`的形式即可。

定义一个二维字符数组tmp存放label的记录，定义一维整型数组存放对应label的出现次数，在定义一个初始化为0的size变量记录已被记录的label种类数。每次从0开始遍历size个数的tmp字符串，若label被匹配到，则将对应的num[i]值加一，并在dotname后加上该数值。如果最后没有匹配到，则size值后移一位，并将label的值赋值给tmp[size]。

又因为graphviz不支持名称为符号的字符，因此对于运算符即界符等信息在一开始要为其赋值英文版本的dotname。

最终代码如下：

```
void treedot(struct Node *root,FILE *output)
{
    struct Node *Child=root->child;
    while(Child!=NULL)
    {
        //printf("%s\n",Child->label);
        char name[20];
        strncpy(name,Child->label,20);
        if(!strcmp(name,"+")||!strcmp(name,"-"))
            strncpy(Child->dotname,"ADDSUB",20);
        else if(!strcmp(name,"*")||!strcmp(name,"/"))
            strncpy(Child->dotname,"MULTIDIV",20);
        else if(!strcmp(name,":="))
            strncpy(Child->dotname,"EQUAL",20);
        else if(!strcmp(name,"."))
            strncpy(Child->dotname,"DOT",20);
        else if(!strcmp(name,","))
            strncpy(Child->dotname,"COMMA",20);
        else if(!strcmp(name,";"))
            strncpy(Child->dotname,"SEM",20);
        else if(!strcmp(name,"("))
            strncpy(Child->dotname,"LP",20);
        else if(!strcmp(name,")"))
            strncpy(Child->dotname,"RP",20);
        else if(!strcmp(name,":"))
            strncpy(Child->dotname,"COLON",20);
        else if(!strcmp(name,"#")||!strcmp(name,"
<")||!strcmp(name,">")||!strcmp(name,"
<=")||!strcmp(name,">=")||!strcmp(name,"="))
            strncpy(Child->dotname,"OPERATOR",20);
        else
            strncpy(Child->dotname,Child->label,20);

        int i;
        for(i=0;i<size;i++)
        {
            if(!strcmp(tmp[i],Child->dotname))
            {
                num[i]++;
                char s[10];
                sprintf(s,"%d",num[i]);
                //itoa(num[i],s,10);
                strcat(Child->dotname,s);
                break;
            }
        }
        if(i==size)     //未匹配到
        {
            strncpy(tmp[i],Child->dotname,20);
            num[i]=0;
            size++;
        }
```

```
        if(Child->label[0]=='\"')   //处理字符串类型的label
        {
            char s[20];
            strncpy(s,Child->label,strlen(Child->label)-1);
            fprintf(output,"%s [label=\"\\%s\\\"\"];\n",Child->dotname,s);
        }
        else
            fprintf(output,"%s [label=\"%s\"];\n",Child->dotname,Child->label);
        fprintf(output,"%s->%s;\n",root->dotname,Child->dotname);
        treedot(Child,output);
        Child=Child->brother;
    }
}
```

4. 词法分析部分代码

词法分析器部分代码基本与上次一致，不过上次只是返回了6种不同的部分，这次针对不同的终结符都有不同的返回。并且为了方便debug，每次识别出一个终结符都会打印其行号列号和值，这样和语法分析器在规约时交替输出，在语法分析器出错时能更好定位出错的地方。

同时，和语法分析一样，每次识别出一个终结符都要为其用newNode函数分配一个结点信息。最后再返回对应的终结符的值。

(1).正规定义

```
key
if|then|while|do|read|write|call|begin|end|const|var|procedure|odd|for|to|step|r
epeat|until|char|real
letter [A-Za-z]
digit [0-9]
integer [+-]?{digit}+
float [+-]?{digit}+(\.{digit}+)?((E|e)[+-]?{digit}+)?
string \"[^\n^\"]+\"
id ({letter}|{digit}|_)*
array {id}\({digit}+":"{digit}+\)
operator "#"|\<|\>|"<="|">="|"="
other [^ \n\r\t{key}{id}{integer}{float}{string}{operator}]
```

(2). 翻译部分

```
"\n" {num_lines++;num_chars=1;}
[ ] {num_chars++;}
"\t" {num_chars+=4;}
if {printf("%d,%d:%s
IF\n",num_lines,num_chars,yytext);num_chars+=yyleng;yylval.token_p=newNode("IF")
;return IF;}
then {printf("%d,%d:%s
THEN\n",num_lines,num_chars,yytext);num_chars+=yyleng;yylval.token_p=newNode("TH
EN");return THEN ;}
while {printf("%d,%d:%s
WHILE\n",num_lines,num_chars,yytext);num_chars+=yyleng;yylval.token_p=newNode("W
HILE");return WHILE;}
do {printf("%d,%d:%s
DO\n",num_lines,num_chars,yytext);num_chars+=yyleng;yylval.token_p=newNode("DO")
;return DO;}
```

```
read {printf("%d,%d:%s
READ\n",num_lines,num_chars,yytext);num_chars+=yyleng;yylval.token_p=newNode("RE
AD");return READ;}
write {printf("%d,%d:%s
WRITE\n",num_lines,num_chars,yytext);num_chars+=yyleng;yylval.token_p=newNode("W
RITE");return WRITE;}
call {printf("%d,%d:%s
CALL\n",num_lines,num_chars,yytext);num_chars+=yyleng;yylval.token_p=newNode("CA
LL");return CALL;}
begin {printf("%d,%d:%s
BEGIN_\n",num_lines,num_chars,yytext);num_chars+=yyleng;yylval.token_p=newNode("
BEGIN");return BEGIN_;}
end {printf("%d,%d:%s
END\n",num_lines,num_chars,yytext);num_chars+=yyleng;yylval.token_p=newNode("END
");return END;}
const {printf("%d,%d:%s
CONST\n",num_lines,num_chars,yytext);num_chars+=yyleng;yylval.token_p=newNode("C
ONST");return CONST;}
var {printf("%d,%d:%s
VAR\n",num_lines,num_chars,yytext);num_chars+=yyleng;yylval.token_p=newNode("VAR
");return VAR;}
procedure {printf("%d,%d:%s
PROCEDURE\n",num_lines,num_chars,yytext);num_chars+=yyleng;yylval.token_p=newNod
e("PROCEDURE");return PROCEDURE;}
odd {printf("%d,%d:%s
ODD\n",num_lines,num_chars,yytext);num_chars+=yyleng;yylval.token_p=newNode("ODD
");return ODD;}
for {printf("%d,%d:%s
FOR\n",num_lines,num_chars,yytext);num_chars+=yyleng;yylval.token_p=newNode("FOR
");return FOR;}
to {printf("%d,%d:%s
TO\n",num_lines,num_chars,yytext);num_chars+=yyleng;yylval.token_p=newNode("TO")
;return TO;}
step {printf("%d,%d:%s
STEP\n",num_lines,num_chars,yytext);num_chars+=yyleng;yylval.token_p=newNode("ST
EP");return STEP;}
repeat {printf("%d,%d:%s
REPEAT\n",num_lines,num_chars,yytext);num_chars+=yyleng;yylval.token_p=newNode("
REPEAT");return REPEAT;}
until {printf("%d,%d:%s
UNTIL\n",num_lines,num_chars,yytext);num_chars+=yyleng;yylval.token_p=newNode("U
NTIL");return UNTIL;}
char {printf("%d,%d:%s
CHAR\n",num_lines,num_chars,yytext);num_chars+=yyleng;yylval.token_p=newNode("CH
AR");return CHAR;}
real {printf("%d,%d:%s
REAL\n",num_lines,num_chars,yytext);num_chars+=yyleng;yylval.token_p=newNode("RE
AL");return REAL;}
{integer} {printf("%d,%d:%s INTEGER\n",num_lines,num_chars,yytext);
            num_chars+=yyleng;
            if(yyleng>10)
                fprintf(yyout,"Warning: integer %s\'s length is longer than
10.\n",yytext);
            yylval.token_p=newNode(yytext);
            return INTEGER;}
{id}    {printf("%d,%d:%s IDENTIFIER\n",num_lines,num_chars,yytext);
        num_chars+=yyleng;
        if(yyleng>10)
```

```
            fprintf(yyout,"Warning: identifier %s\'s length is longer than
10.\n",yytext);
        if(!('a'<=yytext[0]&&yytext[0]<='z'||'A'<=yytext[0]&&yytext[0]<='Z'))
            fprintf(yyout,"Invalid identifier %s.\n",yytext);
        yylval.token_p=newNode(yytext);
        return IDENTIFIER;}
{float} {printf("%d,%d:%s FLOAT\n",num_lines,num_chars,yytext);
    num_chars+=yyleng;
    if(yyleng>15)
        fprintf(yyout,"Warning: float %s\'s length is longer than
15.\n",yytext);
    yylval.token_p=newNode(yytext);
    return FLOAT;}
{string} {printf("%d,%d:%s
STRING\n",num_lines,num_chars,yytext);num_chars+=yyleng;yylval.token_p=newNode(y
ytext);return STRING;}
\+|\- {printf("%d,%d:%s
ADDSUB\n",num_lines,num_chars,yytext);num_chars+=yyleng;yylval.token_p=newNode(y
ytext);return ADDSUB;}
\*|\/ {printf("%d,%d:%s
MULTIDIV\n",num_lines,num_chars,yytext);num_chars+=yyleng;yylval.token_p=newNode
(yytext);return MULTIDIV;}
":=" {printf("%d,%d:%s
EQUAL\n",num_lines,num_chars,yytext);num_chars+=yyleng;yylval.token_p=newNode(yy
text);return EQUAL;}
"." {printf("%d,%d:%s
DOT\n",num_lines,num_chars,yytext);num_chars+=yyleng;yylval.token_p=newNode(yyte
xt);return DOT;}
"," {printf("%d,%d:%s
COMMA\n",num_lines,num_chars,yytext);num_chars+=yyleng;yylval.token_p=newNode(yy
text);return COMMA;}
";" {printf("%d,%d:%s
SEM\n",num_lines,num_chars,yytext);num_chars+=yyleng;yylval.token_p=newNode(yyte
xt);return SEM;}
":" {printf("%d,%d:%s
COLON\n",num_lines,num_chars,yytext);num_chars+=yyleng;yylval.token_p=newNode(yy
text);return COLON;}
\( {printf("%d,%d:%s
LP\n",num_lines,num_chars,yytext);num_chars+=yyleng;yylval.token_p=newNode(yytex
t);return LP;}
\) {printf("%d,%d:%s
RP\n",num_lines,num_chars,yytext);num_chars+=yyleng;yylval.token_p=newNode(yytex
t);return RP;}
{operator} {printf("%d,%d:%s
OPERATOR\n",num_lines,num_chars,yytext);num_chars+=yyleng;yylval.token_p=newNode
(yytext);return OPERATOR;}
{other} {printf("%d,%d:%s
OTHER\n",num_lines,num_chars,yytext);num_chars+=yyleng;fprintf(yyout,"Invalid
character %s.\n",yytext);}
```

5. main函数

位于语法分析部分的main函数接受两个输入：PL/0语法的文件名和要输出类语法树的文件名。在成功打开PL/0文件后，main函数调用yyparse，在yyparse执行时会将“按归约顺序用到的语法规则序列”以及词法分析器识别出的终结符信息打印在屏幕上。之后main函数会调用printtree将类语法树打印到输出文件中。再自动生成一个tree.dot文件，调用treedot函数构建.dot文件。

代码如下：

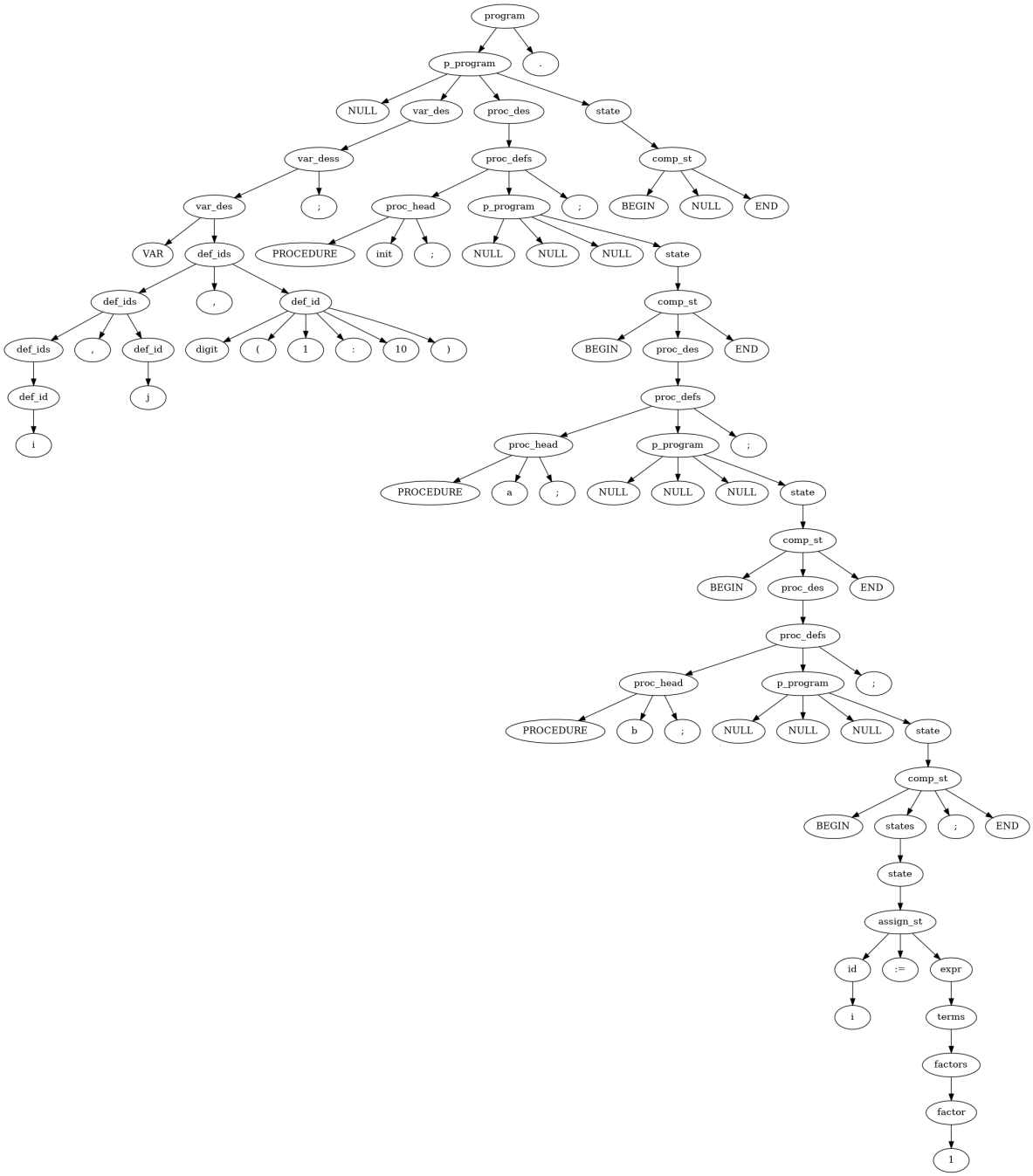
```
int main(int argc,char *argv[])
{
     FILE* fin=NULL;
     extern FILE* yyin;
     fin=fopen(argv[1],"r");
     fout=fopen(argv[2],"w");
     if(fin==NULL)
     {
         printf("cannot open reading file.\n");
         return -1;
     }
     yyin=fin;
     yyparse();
     //printTree(p,fout);
     printtree(p,fout);
     strncpy(p->dotname,p->label,20);
     FILE *output=fopen("tree.dot","w");
     fprintf(output,"digraph g {\n");
     fprintf(output,"%s [label=\"%s\"];\n",p->dotname,p->label);
     treedot(p,output);
     fprintf(output,"}");
     fclose(fin);
     fclose(fout);
     fclose(output);
     return 0;
}
```

三、实验结果截图

以下是一个三层嵌套的文件：

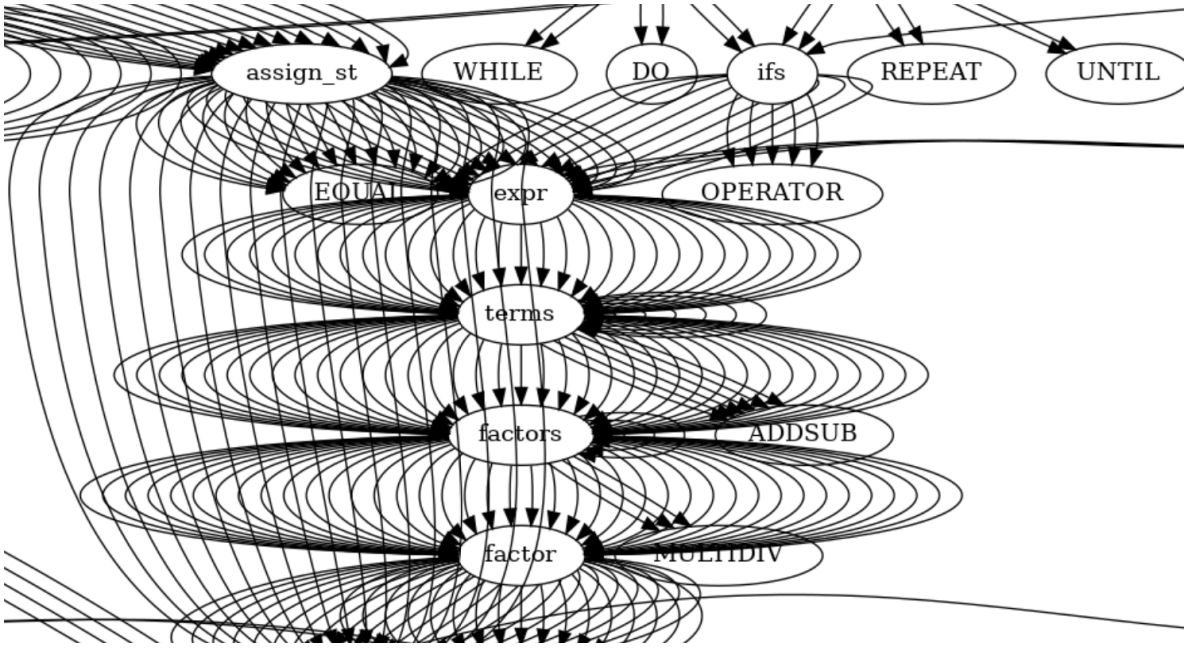
```
var i, j, digit(1:10);

procedure init;
begin
    procedure a;
    begin
        procedure b;
        begin
            i := 0;
        end;
    end;
end;

begin

end.
```

用graphviz工具生成对应语法树如下：



四、实验中碰到的问题及解决

1. 在编写语法分析的语法规则段时，因为自己重新构建了EBNF范式以及加入了扩展部分，因此多次出现了移进-规约冲突和规约-规约冲突，之后反复调试将其解决。
2. 前面提到的失败的语法树部分截图如下（整张图过大）：


assign_st
WHILE
DO
ifs
REPEAT
UNTIL
EQUAL
expr
OPERATOR
terms
factors
ADDSUB
factor
MULTIDIV