实验三 PL/0语言语义分析器

Fan Xinyan

2019.6.9

一、实验目的

通过C语言编写PL/0编译程序，将PL/0语言程序转换为类Pcode代码，然后用interpret.c文件的程序检查类Pcode代码的正确性。

二、实验要求

（一）基本要求

对PL/0源程序，要求输出类Pcode代码，并用interpret.c文件的程序检查类Pcode代码的正确性。

（二）扩展要求

自定义for循环语句。并能够对for语句进行语义分析，输出对应的类Pcode代码，并检查正确性。

三、实验环境

语法分析生成工具：bison、flex

编程语言：C

操作系统：macOS

四、实验回顾

（一）词法分析

1、for语句的设计

代码1 - FOR语句

```
1  for ID := NUMBER to NUMBER do sentence          /*默认循环变量自动+1*/
2  for ID := NUMBER downto NUMBER do sentence      /*默认循环变量自动 - 1*/
```

2、单词分类

对于PL/0源程序，将程序中的单词分为六类，然后按照单词符号出现的顺序依次输出各个单词符号的种类和出现在源程序的位置（行数和列数）。

PL/0语言中单词的种类说明见表1。

表1 单词符号种类

单词种类	类别表示	例子
关键词	K类	基本单词有13个：if、then、while、do、read、write、call、procedure、begin、end、const、var、odd、for、to、downto
标识符	I类	长度小于11的变量名、过程名。字母开头的字母和数字的组合。
常量	C类	长度小于15的整数。
算符	O类	“+"、"-"、"*"、"/"、"#"、"<"、">"、"="、"<>"、"#"、">="、"<="、":="
界符	D类	"("、")"、","、":"、";"、".
其他（非法符、空白符）	T类	空白字符和非法单词

3、正规式

正规式的设计如下所示。

代码2 - 正规式

```
keyword (if)|(do)|(end)|(var)|(odd)|(then)|(read)|(call)|(while)|(begin)|(break)|(const)|(write)|
(procedure)|(for)|(to)|(downto)
num [1-9][0-9]*|[0-9]
identifier [a-zA-Z_][a-zA-Z0-9_]*
delimiters ;|,|:|\(|\)|\.
operator (\+)|(-)|(\*)|(\/)|(:=)|(>)|(<)|(>=)|(<=)|(=)|(#)
wrong_id [0-9][a-z0-9A-Z]*
notes  (\/\/).*(\n)
newline [\n\r]
```

4、识别规则

程序识别到不同种类的单词进行不同的操作。

代码3 - 识别规则

```
/*识别关键词*/
{keyword} {
    if(strcmp(yytext,"if") == 0){return IF;}
    else if(strcmp(yytext,"do") == 0){return DO;}
    ……}
/*识别标识符*/
{identifier} {
    add_var(yytext); //添加到符号表
    yylval = iCurIndex;
    return IDENTIFIER;
}
/*识别数字*/
{num} {
    yylval = atoi(yytext);
    return NUMBER;
}
/*识别算符*/
{operator} {
    yylval = *yytext;
    return *yytext;
}
/*识别界符*/
{delimiters} {
    yylval = *yytext;
    return *yytext;
}
```

（二）语法分析

1、文法设计

根据PL/0语言的EBNF范式设计文法。

表2 EBNF范式与文法

EBNF范式	文法
<程序>::=<分程序>.	program -> subprogram.
<分程序> ::= [<常量说明部分>][<变量说明部分>][<过程说明部分>]<语句>	subprogram -> constInstruction variableInstruction procedureInstruction sentence
<常量说明部分> ::= CONST<常量定义>{,<常量定义>};	constInstruction -> CONST constDeclaration; | NULL
<常量定义> ::= <标识符>=<无符号整数>	constDeclaration -> ID = NUMBER | ID = NUMBER , constDeclaration

EBNF范式	文法
<变量说明部分> ::= VAR<标识符>{,<标识符>}	variableInstruction -> VAR ID variableInstructionSuf ; | NULL variableInstructionSuf -> , ID variableInstructionSuf | NULL
<过程说明部分> ::= <过程首部><分程序>{;<过程说明部分>};	procedureInstruction -> procedureHeader subprogram ; procedureInstruction | NULL
<过程首部> ::= PROCEDURE<标识符>;	procedureHeader -> PROCEDURE ID ;
<语句> ::= <赋值语句>|<复合语句>|<条件语句>|<当型循环语句>|<for循环语句>|<过程调用语句>|<读语句>|<写语句>|<空>	sentence -> assignment | compound | conditional | whileLoop | forLoop | procedureCall | readSent | writeSent | NULL
<赋值语句> ::= <标识符>:=<表达式>	assignment -> ID := expression
<复合语句> ::= BEGIN<语句>{;<语句>}END	multsentence -> sentence ; mulsentence | sentence compound -> BEGIN multsentence END
<条件语句> ::= IF<条件>THEN<语句>	conditional -> IF condition THEN sentence
<条件> ::= <表达式><关系运算符><表达式>|ODD<表达式>	condition -> expression relationop expression | ODD expression
<表达式> ::= [+|-]<项>{<加法运算符><项>}	expression -> item addItem | plusOp item addItem addItem -> plusOp item addItem
<项> ::= <因子>{<乘法运算符><因子>}	item -> factor mulItem mulItem -> mulOp factor mulItem
<因子> ::= <标识符>|<无符号整数>|'('<表达式>')'	factor -> ID | NUMBER | (expression)
<加法运算符> ::= +|-	plusOp -> + | -
<乘法运算符> ::= *|/	mulOp -> * | /
<关系运算符> ::= =|#|<|<=|>|>=	relationop -> =|#|<|<=|>|>=
<当型循环语句> ::= WHILE<条件>DO<语句>	whileLoop -> WHILE condition DO sentence
<for循环语句> ::= FOR<赋值语句>TO<数字>DO<语句> | FOR<赋值语句>DOWNTO<数字>DO<语句>	forLoop -> FOR assignment TO number DO sentence | FOR assignment DOWNTO number DO sentence
<过程调用语句> ::= CALL<标识符>	procedureCall -> CALL ID
<读语句> ::= READ'('<标识符>{,<标识符>}')'	readSent -> READ (readpara) readpara -> ID
<写语句> ::= WRITE'('<表达式>{,<表达式>}')'	writeSent -> WRITE (writepara) writepara -> expression , writepara

2、终结符与优先级说明

程序中定义格式：

代码4 - 终结符定义与优先级

```
%token IF THEN WHILE DO READ WRITE CALL BEGINN END CONST VAR PROCEDURE ODD
FOR TO DOWNTO
%token IDENTIFIER NUMBER
%token ASSIGNMENT
%token ENDD    0    "end of file”

%left "<=" ">=" '>' '<' '=' '#'
%left '+' '-'
%left '*' '/'
%left '(' ‘)'
```

3、核心代码

（1）移进

在文法中，当编译程序遇到终结符时，需要为该终结符申请新的Node并压入栈。具体函数如下，第一个参数表示该节点的类型，第二个参数表示该节点在源程序中的字符表示，第三个参数表示这个符号的数值大小。

代码5 - SHIFT

```
void shift(int type,char* id,int number){
    Node* a;
    a = (Node*)malloc(sizeof(Node));
    a->value = number;
    a->type = type;
    if(id == NULL) {
        a->id = NULL;
    } else {
            a->id = (char*)malloc(ID_MAX_SIZE);
            strcpy(a->id, id);
    }
    a->child = NULL;
    a -> childcnt = 0;
    push(a);
}
```

在代码语法规则定义中，具体表示如下，以文法“procedureHeader -> PROCEDURE ID ;”为例：

代码6 - 移进

```
procedureHeader : "procedure" "id" ";"    {
                                          shift(PROCEDURE_KEY,$1,0);
                                          shift(IDENTI,$2,0);
                                          shift(SEMI_OP,";",0);
                                          };
```

（2）归约

在文法中，当编译程序遇到符合某一文法的候选式时，就需要将栈顶中满足文法右部类型的符号弹出栈，并将左部符号设为一个新的节点，即为这些被弹出符号的父节点，并压入栈中。具体函数如下。其中第一个参数表示左部符号的类型，第二个参数表示孩子的个数（右部符号的个数），第三个节点表示孩子的类型（右部符号的类型）。

代码7 - REDUCE

```
void reduce(int type,int child_type_cnt,int child_type[TYPE_MAXN]){
    Node* father;
    father = (Node*)malloc(sizeof(Node));
    father -> value = 0;
    father -> id = NULL;
    father -> type = type;

    int cnt = 0;
    int tmptop = stop;
    fflush(stdout);

    while(tmptop > 0 && inarray(stack[tmptop-1]->type, child_type_cnt, child_type) == 1){
        tmptop --;
        cnt ++;
    }
    father -> child = (Node**)malloc(cnt * sizeof(Node*));
    int idx = 0;

    while(!empty() && inarray(top() -> type, child_type_cnt, child_type) == 1 ){
        father -> child[idx] = (Node*)malloc(sizeof(Node));
        node_copy(father -> child[idx], stack[stop-1]);
        pop();
        idx ++;
    }
    father -> childcnt = idx;
    push(father);
}
```

在代码语法规则定义中，具体表示如下，以文法“procedureHeader -> PROCEDURE ID ;”为例：

代码8 - 归约

```
procedureHeader : "procedure" "id" ";"  {
                                        shift(PROCEDURE_KEY,$1,0);
                                        shift(IDENTI,$2,0);
                                        shift(SEMI_OP,";",0);
                                        child_tp_cnt = 3;
                                        child_tp[0] = PROCEDURE_KEY;
                                        child_tp[1] = IDENTI;
                                        child_tp[2] = SEMI_OP;
                                        reduce(PROCEDURE_HEADER,child_tp_cnt,child_tp);
                                        };
```

五、实验内容

本次语义实验的代码共分为三部分：“userdef.h”“lex.l”“parse.y”。其中“userdef.h”为标识符、符号表、pcode指令、程序栈结构体的声明。“lex.l”为词法分析程序，识别出关键词、标识符、数字、界符、运算符等传入语法分析程序。“parse.y”为语法语义分析程序，根据PL/0的EBNF范式定义了文法规则，并对不同的语句进行翻译。

（一）数据结构

1、符号栈

编译时维护一个符号栈，当一个过程中声明var变量时，将其名字、类型、所在过程是第几层、上一层过程在过程栈的位置加入符号栈中；当过程调用结束后，把该过程的符号弹出栈。

代码9 - 符号表

```
struct sign{ //符号表结构体
    char name[11]; //名字
    int type; //类型
    int layer; //层数
    int previous; //上一层过程
};
struct sign *sign_stack;
```

2、过程栈

编译时维护一个过程栈，当程序遇到一个过程时，需要记录该过程的名字、位置、包含的var变量个数。同时维护一个记录过程编号的栈，当开始一个过程编译时，压入栈中；执行结束后，将该过程编号弹出。

代码10 - 过程栈

```
struct procedure_pos{  //记录程序位置
    char name[11]; //程序名字
    int pos; //位置
    int var_number; //变量个数
};
struct procedure_pos * procedure_position;
int procedure_stack[stack_size]; //过程编号栈
```

3、类Pcode代码

按照Pcode代码的指令结构，记录每条指令所在的行号、f值、l值、a值。

代码11 - 类PCODE代码

```
struct code_struct{  //代码结构
    int line; //行数
    int f;
    int l;
    int a;
};
struct code_struct *mycode;
```

（二）翻译

1、初始化

在编译程序之前，对display栈和过程栈进行初始化；在程序结束后返回程序结束的指令。

代码12 - 初始化

```
program :
            {
            display_top++;
            display_stack[display_top] = 1;
            PushProcedurePosition("program",0); //改变procedure_position
            cur_pro_ct = pro_ct-1;
            procedure_stack[procedure_stack_point] = pro_ct-1;
            procedure_stack_point++;
            }/*在程序开始之前对display栈和程序栈进行初始化*/
    subprogram '.' ENDD   {
            printf("\nSuccessfully Compiled\n0 error(s) 0 warning(s)\n");
            AddPcode(8,0,0);
            };
```

2、开辟空间

当编译到过程结束后时，需要为该过程开辟3+var变量数量的空间。

代码13 - 开辟空间

```
/*子程序*/
subprogram :
    constInstruction
    variableInstruction
    procedureInstruction {
        procedure_position[cur_pro_ct].pos = total_line; //记录程序的位置
        AddPcode(5,0,3+procedure_position[cur_pro_ct].var_number);
        }/*在procedure的位置中记录对应行数，并且记下PCODE INT 0 3+var num 在运行栈中为被调用的区域开辟A个单元的区域*/
    sentence
;
```

3、说明语句的翻译

说明语句分为三种类型：const常量说明语句、var变量说明语句、procedure过程说明语句。对于const说明语句，需要检查标识符是否重复定义，在符号表中记录变量名称和对应的值；对于var说明语句，需要检查标识符是否重复定义，在符号表中记录变量的名字，以及更新过程中记录的变量的个数；对于procedure说明语句，首先是过程首部，需要检查标识符是否重复定义，将这个过程名放入sign栈中记录，同时将这个过程也放在display栈顶，过程栈也需要记录，其次对于过程结束后，需要退栈，清空该过程的相关信息，同时加上指令OPR 0 0代表结束。

（1）const说明语句的翻译

代码14 - CONST语句的翻译

```
/*常量说明部分*/
constInstruction :
    CONST  constDeclaration ';'
|    /*empty*/
;
/*常量定义*/
constDeclaration :
    IDENTIFIER '=' NUMBER   {
                    if(CheckSignIsDeclare(strMem[$1].sMark) == 1){
                       PushSignStack(strMem[$1].sMark,$3);
                    }
                    }
|    IDENTIFIER '=' NUMBER   {
                    if(CheckSignIsDeclare(strMem[$1].sMark) == 1){
                       PushSignStack(strMem[$1].sMark,$3);
                    }
                    }
    ',' constDeclaration
;
```

（2）var说明语句的翻译

代码15 - VAR语句的翻译

```
/*变量说明部分*/
variableInstruction :
    VAR IDENTIFIER  {
            if(CheckSignIsDeclare(strMem[$2].sMark) == 1){
               PushSignStack(strMem[$2].sMark,-1)procedure_position[cur_pro_ct].var_number++;
            }}
    variableDeclaration ';'
|    /*empty*/;
/*变量定义，记录变量，同时更新变量的个数*/
variableDeclaration :
    ',' IDENTIFIER  {
            if(CheckSignIsDeclare(strMem[$2].sMark) == 1){
               PushSignStack(strMem[$2].sMark,-1);
               procedure_position[cur_pro_ct].var_number++;//当前程序变量数目+1
            }}
    variableDeclaration
|    /*empty*/;
```

（3）procedure说明语句的翻译

代码16 - PROCEDURE语句的翻译

```
/*过程说明部分*/
procedureInstruction :
    procedureHeader  subprogram  ';' {
       cur_layer--;procedure_stack_point--; cur_pro_ct = procedure_stack[procedure_stack_point-1];
       procedure_stack[procedure_stack_point] = -1; /*清空栈*/
       PopSignStack(); /*把该过程的ID清空*/
       AddPcode(8,0,0);
       } /*表示一个程序结束，需要将程序退栈，同时加上指令OPR 0 0代表结束*/
    procedureInstruction
|    /*empty*/;
/*过程首部，定义一个新的过程，需要将这个过程放入sign栈中记录，同时将这个过程也放在
display栈顶，过程栈也需要记录*/
procedureHeader :
    PROCEDURE IDENTIFIER  ';'    {
        if(CheckSignIsDeclare(strMem[$2].sMark) == 1){
           PushSignStack(strMem[$2].sMark,-2);
           display_top++; display_stack[display_top] = sign_top+1;
           cur_layer++; PushProcedurePosition(strMem[$2].sMark,total_line+1);
           cur_pro_ct = pro_ct-1; /*在一个过程开始时，记录这个过程的编号*/
           procedure_stack[procedure_stack_point] = pro_ct - 1; procedure_stack_point++;
        } } ;
```

4、赋值语句的翻译

首先需要查找标识符是否存在，若不存在返回报错信息；若存在，执行指令 STO l a将栈顶内容送入某单元变量中，l为层差，a为偏移量。

代码17 - 赋值语句的翻译

```
/*赋值语句*/
assignmentSentence :
    IDENTIFIER  {
            int pos_flag = FindSign(strMem[$1].sMark); //找sign是否存在，返回在sign_stack的位置
            if(pos_flag > 0){
               if(sign_stack[pos_flag].type != -1){//-1表示为var类型的标识符
                  printf("in line %d illegal name\n",code_line);
                  exit(1);
               }
            }}
    ASSIGNMENT expression  {
            int pos_flag = FindSign(strMem[$1].sMark);
            AddPcode(3,cur_layer-sign_stack[pos_flag].layer,CaculateA(pos_flag));/*STO l a*/
            }
;
```

5、控制流语句的翻译

（1）if语句的翻译

首先需要在code_address_stack记录JPC条件跳转指令的位置，在if语句编译结束后用ChangePcode函数回填该指令的跳转地址。其次在条件判断语句中，根据不同的语句执行不同的指令操作。

代码18 - IF的翻译

```
/*条件语句*/
condition :
    expression  relationOp  expression {AddPcode(8,0,$2);}  /*执行指令OPR 0 X*/
|     ODD  expression {AddPcode(8,0,6);}  /*指令OPR 0 6，栈顶奇偶元素判断*/;
/*执行 JPC 0 a条件跳转，并且记录代码地址；then后面改变代码地址*/
conditionSentence :
    IF  condition   {
              AddPcode(7,0,-1); /*JPC条件跳转，栈顶布尔值非真跳转到地址a，条件不满足时转*/
              code_address_stack[code_address_stack_point] = total_line-1; /*记录条件判断的位置*/
              code_address_stack_point++;
              }
    THEN sentence  {/*条件满足*/
              code_address_stack_point--;
              int target_line = code_address_stack[code_address_stack_point];
              code_address_stack[code_address_stack_point]  = -1;
              ChangePcode(target_line,7,0,total_line);
             };
```

（2）while语句的翻译

首先需要在code_address_stack记录JPC条件跳转指令的位置，在while语句编译结束后用ChangePcode函数回填该指令的跳转地址。其次维护一个栈loop_stack，记录循环语句执行结束后JMP无条件跳转回来的指令的位置。

代码19 - WHILE的翻译

```
/*while循环语句*/
whileLoop :
   WHILE  {
                loop_stack[loop_stack_point] = total_line; /*记录JMP回来的位置*/
                loop_stack_point++;
                }
   condition {
                AddPcode(7,0,-1); /*JPC 要跳转的地址还没有，先存为-1，后面有ChangePcode()*/
                code_address_stack[code_address_stack_point] = total_line-1; /*记录条件判断的位置，方便后面ChangePcode找到要修改的位置*/
                code_address_stack_point++;
                }
   DO sentence    {
                loop_stack_point--;
                AddPcode(6,0,loop_stack[loop_stack_point]); /*JMP 无条件跳转至a地址*/
                loop_stack[loop_stack_point] = -1;
                code_address_stack_point--;
                int target_line = code_address_stack[code_address_stack_point];
                ChangePcode(target_line,7,0,total_line);
                code_address_stack[code_address_stack_point]  = -1;
                };
```

（3）call语句的翻译

首先需要判断过程名之前有没有定义，如果有找到这个过程的位置，将它加入代码中。CAL l a 调用过程 a为调用地址，l为层差。

代码20 - CALL的翻译

```
/*过程调用语句*/
procedureCall :
      CALL IDENTIFIER {
                int pos_flag = FindSign(strMem[$2].sMark);
                if(pos_flag > 0){
                   if(sign_stack[pos_flag].type==-2 || sign_stack[pos_flag].type == -3){
                      int target_pos = FindProcedurePosition(strMem[$2].sMark); /*根据procedure的名字来找位置*/
                      AddPcode(4,cur_layer-sign_stack[pos_flag].layer,target_pos);
                   }else{
                      printf("error in line %d illegal name\n",code_line); exit(1);
                   }
                } };
```

（4）for语句的翻译

以循环变量+1的for 语句文法为例。首先需要在code_address_stack记录JPC条件跳转指令的位置，在for语句编译结束后用ChangePcode函数回填该指令的跳转地址。其次维护一个栈loop_stack，记录循环语句执行结束后JMP无条件跳转回来的指令的位置。接着在语句执行中需要默认加上循环变量+1的指令。

代码21 - FOR的翻译

```
/*for循环语句*/
forLoop :
    FOR IDENTIFIER ASSIGNMENT NUMBER   {
                int pos_flag = FindSign(strMem[$2].sMark); /*返回ID的地址*/
                if(pos_flag > 0){ /*ID不存在，报错*/
                  if(sign_stack[pos_flag].type == -1){} /*ID为var类型*/
                  else{
                    printf("error in line %d illegal input\n",code_line);
                    exit(1);
                  }
                }
                AddPcode(1,0,$4);/*LIT 0 a 把NUMBER的值取到栈顶*/
                AddPcode(3,cur_layer-sign_stack[pos_flag].layer, CaculateA(pos_flag));
                loop_stack[loop_stack_point] = total_line;
                loop_stack_point++;
               }
    TO NUMBER      {
            int pos_flag = FindSign(strMem[$2].sMark); /*找到ID的地址*/
            AddPcode(2,cur_layer-sign_stack[pos_flag].layer, CaculateA(pos_flag));
            AddPcode(1,0,$7);/*LIT*/
            AddPcode(8,0,13);/*OPR < */
            AddPcode(7,0,-1);/*JPC*/
            code_address_stack[code_address_stack_point] = total_line-1;
            code_address_stack_point++;
            }
    DO  sentence   {
            int pos_flag = FindSign(strMem[$2].sMark);
            AddPcode(2,cur_layer-sign_stack[pos_flag].layer, CaculateA(pos_flag));
            AddPcode(1,0,1);/*把常数1取到栈顶*/
            AddPcode(8,0,2);/*模拟循环变量+1的操作*/
            AddPcode(3,cur_layer-sign_stack[pos_flag].layer, CaculateA(pos_flag));
            loop_stack_point--; /*前面有++，退一层，找到本层循环的入口地址*/
            AddPcode(6,0,loop_stack[loop_stack_point]); /*JMP 0 a 无条件跳转至循环开始的位置*/
            loop_stack[loop_stack_point] = -1;  /*该层循环结束，清空栈*/
            code_address_stack_point--;
            int target_line = code_address_stack[code_address_stack_point];  /*需修改代码的地址*/
            ChangePcode(target_line,7,0,total_line);   /*修改地址*/
            code_address_stack[code_address_stack_point]  = -1;  /*清空栈*/
            }
```

6、读写语句的翻译

（1）读语句

循环读入标识符的输入，首先需要查找这个标识符是否存在，如果存在，执行OPR 0 16从屏幕中读入一个输入，之后执行STO L A将其送入对应的变量。

代码22 - 读语句的翻译

```
/*读写语句*/
readPara :
    IDENTIFIER  {
            int pos_flag = FindSign(strMem[$1].sMark);
            if(pos_flag>0){
              if(sign_stack[pos_flag].type == -1){
                AddPcode(8,0,16); /*从命令行读入一个并放入栈顶*/
                AddPcode(3,cur_layer-sign_stack[pos_flag].layer,CaculateA(pos_flag));/*栈顶值放ID*/
              } else {
                printf("error in line  %d  change a const parameter\n",code_line);
                exit(1);
              }
            } }
|    IDENTIFIER  {
            int pos_flag = FindSign(strMem[$1].sMark);
            if(pos_flag>0){
              if(sign_stack[pos_flag].type == -1){
                AddPcode(8,0,16); /*从命令行读入一个并放入栈顶*/
                AddPcode(3,cur_layer-sign_stack[pos_flag].layer,CaculateA(pos_flag));
              } else {
                printf("error in line  %d  change a const parameter\n",code_line);
                exit(1);
              }}}
    ',' readPara;
readSentence :
    READ '(' readPara ')' ;
```

（2）写语句

输出参数为表达式循环，执行OPR 0 14 & OPR 0 15，栈顶值输出到屏幕，屏幕输出换行。

代码23 - 写语句的翻译

```
writePara :
    expression  {
            AddPcode(8,0,14);
            AddPcode(8,0,15); }
|    expression  {
            AddPcode(8,0,14);
            AddPcode(8,0,15);}
    ',' writePara;
writeSentence :
    WRITE '(' writePara ')' ;
```

（三）核心代码

1、符号的操作

（1）int CheckSignIsDeclare(char *tag_name)

看定义的标识符之前有没有定义过。

代码24 - CHECKSIGNISDECLARE

```
int CheckSignIsDeclare(char *tag_name){
    int bias = display_stack[display_top]; //栈顶偏差
    while(bias <= sign_top){
        if(strcmp(tag_name,sign_stack[bias].name) == 0){
            printf("error!!! line %d: declare repeatedly \n",code_line);
            exit(1);
        }
        if(sign_stack[bias].previous == 0){
            break;
        }
        bias ++;
    }
    return 1;
}
```

（2）int FindSign(char *tag_name)

查找符号是否存在。

代码25 - FINDSIGN

```
int FindSign(char *tag_name){
    int display_bias = display_top;
    int cur_bias;
    while(display_bias > 0){
        cur_bias = display_stack[display_bias];
        while(cur_bias <= sign_top){
            if(strcmp(tag_name,sign_stack[cur_bias].name) == 0){//找到了
                return cur_bias;
            }
            if(sign_stack[cur_bias].previous==0){ //代表找到这个程序的底部
                break;
            }
            cur_bias++;
        }
        display_bias --;
    }
    printf("error!!! line %d: no such parameter \n",code_line); //没找到
    exit(1);
}
```

(3) void PopSignStack()

清空符号栈。将该过程的所有的符号退栈，并且display中最上面的程序也需要退出来。

代码26 - POPSIGNSTACK

```
void PopSignStack(){
    int bias = display_stack[display_top]-1;
    for(int i=sign_top; i>bias;--i){  //退sign stack
        sign_stack[i].type = -10;
        sign_stack[i].layer = -1;
        sign_stack[i].previous = -1;
        for(int j=0;j<11;++j){
            sign_stack[i].name[j] = '\0';
        }
    }
    sign_top = bias;
    sign_stack[sign_top].previous = 0;

    sign_stack[sign_top].type = -3;
    display_stack[display_top] = 0;  //退display stack
    display_top--;
}
```

(4) void PushSignStack(char *s,int t)

符号入栈。s为符号，t为符号类型。

代码27 - PUSHSIGNSTACK

```
void PushSignStack(char *s,int t){
    sign_top++;
    for(int i=0;i<11;++i){
        sign_stack[sign_top].name[i] = '\0';
    }
    strcpy(sign_stack[sign_top].name,s);
    sign_stack[sign_top].type = t;
    sign_stack[sign_top].layer = cur_layer;
    sign_stack[sign_top].previous = 0;
    if(sign_top > 1){  //更新previous的值
        int pre_bias = sign_top-1;
        if(sign_stack[pre_bias].type > -2){
            sign_stack[pre_bias].previous = sign_top;
        }
        else if(sign_stack[pre_bias].type == -3){
            sign_stack[pre_bias].previous = sign_top;
        }
    }
}
```

2、过程的操作

(1) void PushProcedurePosition(char *c,int p)

将程序的位置记录在procedure_position中。

代码28 - PUSHPROCEDUREPOSITION

```
void PushProcedurePosition(char *c,int p){
    for(int i = 0; i < 11; ++i){
        procedure_position[pro_ct].name[i] = '\0';
    }
    strcpy(procedure_position[pro_ct].name, c);
    procedure_position[pro_ct].pos = p;
    procedure_position[pro_ct].var_number = 0;
    pro_ct ++;  //过程个数加一
}
```

(2) int FindProcedurePosition(char *s)

找到过程s的位置。

代码29 - FINDPROCEDUREPOSITION

```
int FindProcedurePosition(char *s){
    for(int i=cur_pro_ct; i<pro_ct; ++i){
        if(strcmp(procedure_position[i].name,s) == 0){

            return procedure_position[i].pos;
        }
    }
    printf("error in func FindProcedurePosition\n");  //没找到
    exit(1);
}
```

3、类Pcode代码的操作

(1) void AddPcode(int thef, int thel, int thea)

添加类PCODE代码。

代码30 - ADDPCODE

```
void AddPcode(int thef, int thel, int thea){
    mycode[total_line].line = total_line;
    mycode[total_line].f = thef;
    mycode[total_line].l = thel;
    mycode[total_line].a = thea;
    total_line++;
}
```

(2) void ChangePcode(int theline, int thef, int thel, int thea)

将第theline行代码改为所指的。

代码31 - CHANGEPCODE

```
void ChangePcode(int theline, int thef, int thel, int thea){
    mycode[theline].f = thef;
    mycode[theline].l = thel;
    mycode[theline].a = thea;
}
```

(3) int CaculateA(int pos_flag)

计算偏移量。

代码32 - CACULATEA

```
int CaculateA(int pos_flag){
    int target_flag = pos_flag;
    if(pos_flag >= 1){
        while(target_flag >= 1){
            if(sign_stack[target_flag].type <= -2 || sign_stack[target_flag].type >= 0){
                return pos_flag-target_flag + 2;
            }
            target_flag--;
        }
        return pos_flag+2;
    }
    else{
        printf("error\n");
        exit(1);
    }
}
```

六、实验结果

（一）命令行

将PL/0源程序存在test.txt文件中，完整PL/0源程序请看附录A。在MacOS系统下程序执行命令如下：

执行命令

```
/*编译PL/0程序，输出类Pcode代码到pcode.txt中*/
bison -d parse.y
flex lex.l
gcc -o out lex.yy.c parse.tab.c
./out
输入测试文件的名称
/*执行编译结果*/
gcc interpret.c
./a.out
```

（二）测试程序

1、for.txt

测试程序如下所示：

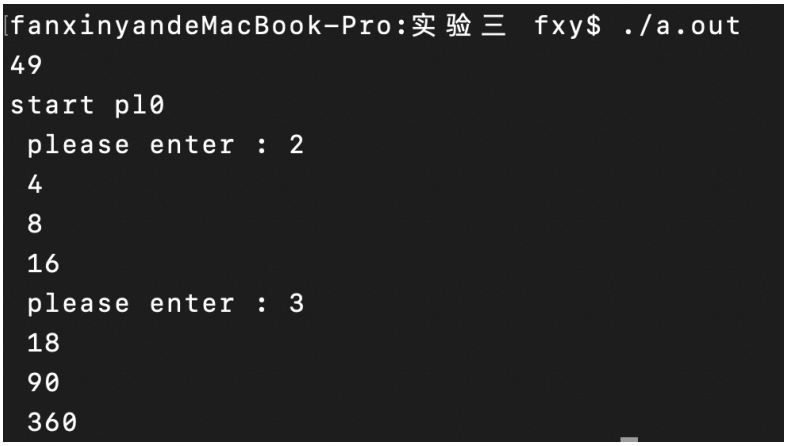
FOR测试程序

```
1  var n,a, b, c, max;
2  procedure for_test;
3  var i, n;
4  begin
5          read(n);
6          for i := 1 to 3 do
7                  begin
8                          n := 2 * n;
9                          write(n);
10                 end;
11         read(n);
12         for i := 5 downto 3 do
13                 begin
14                         n := (i + 1) * n;
15                         write(n);
16                 end;
17 end;
18 begin
19         call for_test;
20 end.
```

程序执行后的类Pcode代码为：

类PCODE代码													
1	(0)	JMP	0	46	(17)	OPR	0	15	(34)	LOD	0	4
2	(1)	JMP	0	2	(18)	LOD	0	3	(35)	OPR	0	4
3	(2)	INT	0	5	(19)	LIT	0	1	(36)	STO	0	4
4	(3)	OPR	0	16	(20)	OPR	0	2	(37)	LOD	0	4
5	(4)	STO	0	4	(21)	STO	0	3	(38)	OPR	0	14
6	(5)	LIT	0	1	(22)	JMP	0	7	(39)	OPR	0	15
7	(6)	STO	0	3	(23)	OPR	0	16	(40)	LOD	0	3
8	(7)	LOD	0	3	(24)	STO	0	4	(41)	LIT	0	1
9	(8)	LIT	0	3	(25)	LIT	0	5	(42)	OPR	0	3
10	(9)	OPR	0	13	(26)	STO	0	3	(43)	STO	0	3
11	(10)	JPC	0	23	(27)	LOD	0	3	(44)	JMP	0	27
12	(11)	LIT	0	2	(28)	LIT	0	3	(45)	OPR	0	0
13	(12)	LOD	0	4	(29)	OPR	0	11	(46)	INT	0	8
14	(13)	OPR	0	4	(30)	JPC	0	45	(47)	CAL	0	2
15	(14)	STO	0	4	(31)	LOD	0	3	(48)	OPR	0	0
16	(15)	LOD	0	4	(32)	LIT	0	1				
17	(16)	OPR	0	14	(33)	OPR	0	2				

用interpret.c程序执行后的结果为：



可以看到正确的类Pcode代码。当程序输入2时，结果返回4、8、16；当程序输入3时，结果返回18、90、360，符合PL/0程序的执行结果。

2、SumOfn!.txt

测试程序如下所示：

SUMOFN!测试程序

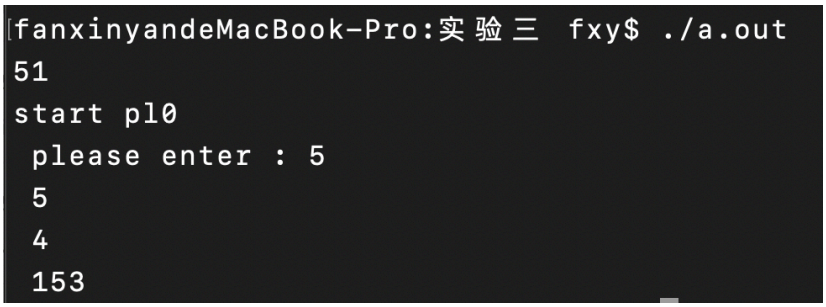
```
1  var n, m, fact, sum;
2  procedure factorial;
3  begin
4     if m > 0 then
5        begin
6           fact := fact * m;
7           m := m - 1;
8           call factorial;
9        end;
10 end;
11 begin
12    read(n);
13    sum := 0;
14    while n > 0 do
15       begin
16          m := n;
17          fact := 1;
18          call factorial;
19          if fact > 20 then write (n);
20          sum  := sum + fact;
21          n := n - 1;
22       end;
23    write(sum);
24 end.
```

程序执行后的类Pcode代码为：

类PCODE代码

1	(0)	JMP	0	17	(17)	INT	0	7	(34)	JPC	0	38
2	(1)	JMP	0	2	(18)	OPR	0	16	(35)	LOD	0	3
3	(2)	INT	0	3	(19)	STO	0	3	(36)	OPR	0	14
4	(3)	LOD	1	4	(20)	LIT	0	0	(37)	OPR	0	15
5	(4)	LIT	0	0	(21)	STO	0	6	(38)	LOD	0	6
6	(5)	OPR	0	12	(22)	LOD	0	3	(39)	LOD	0	5
7	(6)	JPC	0	16	(23)	LIT	0	0	(40)	OPR	0	2
8	(7)	LOD	1	5	(24)	OPR	0	12	(41)	STO	0	6
9	(8)	LOD	1	4	(25)	JPC	0	47	(42)	LOD	0	3
10	(9)	OPR	0	4	(26)	LOD	0	3	(43)	LIT	0	1
11	(10)	STO	1	5	(27)	STO	0	4	(44)	OPR	0	3
12	(11)	LOD	1	4	(28)	LIT	0	1	(45)	STO	0	3
13	(12)	LIT	0	1	(29)	STO	0	5	(46)	JMP	0	22
14	(13)	OPR	0	3	(30)	CAL	0	2	(47)	LOD	0	6
15	(14)	STO	1	4	(31)	LOD	0	5	(48)	OPR	0	14
16	(15)	CAL	1	2	(32)	LIT	0	20	(49)	OPR	0	15
17	(16)	OPR	0	0	(33)	OPR	0	12	(50)	OPR	0	0

用interpret.c程序执行后的结果为：



可以看到正确的类Pcode代码。当程序输入5时，结果返回5、4（阶乘>20的数）、153（5的阶乘和），符合PL/0程序的执行结果。