

PL/0 编译器实验报告

一、实验目的

1. 理解编译器的工作机制，理清词法、语法、语义各个步骤编译器的工作内容，掌握编译器的构造方法
2. 掌握词法分析器的生成工具 LEX、语法分析器的生成工具 bison 的用法
3. 熟悉 PL/0 语言的编译原理和过程，理解中间代码类 pcode 和编译运行的关系，了解类 pcode 代码的运行机制

二、实验内容

1. PL/0 语言简介

- a. PL/0 语言是 Pascal 语言的子集
 - 数据类型只有整型
 - 标识符的有效长度是 10，以字母开头的字母数字串
 - 数最多 14 位
 - 过程无参，可嵌套（最多三层），可递归调用
 - 变量的作用域同 Pascal，常量为全局的
- b. 语句类型：
 - 赋值语句：if...then..., while...do..., read, write, call
 - 复合语句：begin...end
 - 说明语句：const..., var..., procedure...
- c. 13 个保留字：
 - If, then, while, do, read, write, call, begin, end, const, var, procedure, odd

2. PL/0 语言的 EBNF 范式

- $\langle \text{程序} \rangle ::= \langle \text{分程序} \rangle.$
- $\langle \text{分程序} \rangle ::= [\langle \text{常量说明部分} \rangle][\langle \text{变量说明部分} \rangle][\langle \text{过程说明部分} \rangle]\langle \text{语句} \rangle$
- $\langle \text{常量说明部分} \rangle ::= \text{CONST} \langle \text{常量定义} \rangle \{, \langle \text{常量定义} \rangle\};$
- $\langle \text{常量定义} \rangle ::= \langle \text{标识符} \rangle = \langle \text{无符号整数} \rangle$
- $\langle \text{无符号整数} \rangle ::= \langle \text{数字} \rangle \{ \langle \text{数字} \rangle \}$
- $\langle \text{变量说明部分} \rangle ::= \text{VAR} \langle \text{标识符} \rangle \{, \langle \text{标识符} \rangle\};$
- $\langle \text{标识符} \rangle ::= \langle \text{字母} \rangle \{ \langle \text{字母} \rangle | \langle \text{数字} \rangle \}$

- $\langle \text{过程说明部分} \rangle ::= \langle \text{过程首部} \rangle \langle \text{分程序} \rangle \{ \langle \text{过程说明部分} \rangle \};$
- $\langle \text{过程首部} \rangle ::= \text{PROCEDURE} \langle \text{标识符} \rangle;$
- $\langle \text{语句} \rangle ::= \langle \text{赋值语句} \rangle | \langle \text{复合语句} \rangle | \langle \text{条件语句} \rangle | \langle \text{当型循环语句} \rangle | \langle \text{过程调用语句} \rangle | \langle \text{读语句} \rangle | \langle \text{写语句} \rangle | \langle \text{空} \rangle$
- $\langle \text{赋值语句} \rangle ::= \langle \text{标识符} \rangle := \langle \text{表达式} \rangle$
- $\langle \text{复合语句} \rangle ::= \text{BEGIN} \langle \text{语句} \rangle \{ \langle \text{语句} \rangle \} \text{END}$
- $\langle \text{条件} \rangle ::= \langle \text{表达式} \rangle \langle \text{关系运算符} \rangle \langle \text{表达式} \rangle | \text{ODD} \langle \text{表达式} \rangle$
- $\langle \text{条件语句} \rangle ::= \text{IF} \langle \text{条件} \rangle \text{THEN} \langle \text{语句} \rangle$
- $\langle \text{表达式} \rangle ::= [+ | -] \langle \text{项} \rangle \{ \langle \text{加法运算符} \rangle \langle \text{项} \rangle \}$
- $\langle \text{项} \rangle ::= \langle \text{因子} \rangle \{ \langle \text{乘法运算符} \rangle \langle \text{因子} \rangle \}$
- $\langle \text{因子} \rangle ::= \langle \text{标识符} \rangle | \langle \text{无符号整数} \rangle | (\langle \text{表达式} \rangle)$
- $\langle \text{加法运算符} \rangle ::= + | -$
- $\langle \text{乘法运算符} \rangle ::= * | /$
- $\langle \text{关系运算符} \rangle ::= = | \# | < | < = | > | > =$
- $\langle \text{当型循环语句} \rangle ::= \text{WHILE} \langle \text{条件} \rangle \text{DO} \langle \text{语句} \rangle$
- $\langle \text{过程调用语句} \rangle ::= \text{CALL} \langle \text{标识符} \rangle$
- $\langle \text{读语句} \rangle ::= \text{READ} (\langle \text{标识符} \rangle \{ \langle \text{标识符} \rangle \})$
- $\langle \text{写语句} \rangle ::= \text{WRITE} (\langle \text{表达式} \rangle \{ \langle \text{表达式} \rangle \})$
- $\langle \text{字母} \rangle ::= a | b | \dots | X | Y | Z$
- $\langle \text{数字} \rangle ::= 0 | 1 | \dots | 8 | 9$

3. YACC 简介

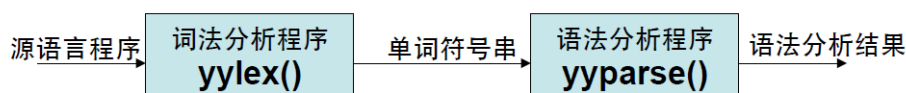
YACC = Yet Another Compiler Compiler

YACC是一个语法分析程序的自动产生系统

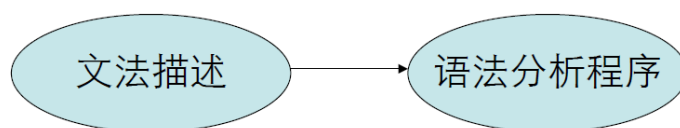
YACC源程序 \longrightarrow YACC \longrightarrow parser_tab.c文件

源语言程序 \longrightarrow yyparse()函数 \longrightarrow 语法分析结果

词法分析程序与语法分析程序的关系



YACC的工作原理：



YACC的处理能力：可以用LALR(1)文法表示的上下文无关文法。

4. 类 pcode 代码简介

- a. 目标代码类 pcode 是一种栈式机的汇编语言
 - 栈式机系统结构：没有累加器和寄存器，只有存储栈指针，所有运算都在栈顶
 - 指令格式：f l a
 - f: 功能码
 - l: 层次差（标识符引用层减去定义层）
 - a: 根据不同的指令有所区别
- b. 指令功能表：

LIT 0 a	将常数值取到栈顶，a为常数值
LOD l a	将变量值取到栈顶，a为偏移量，l 为层差
STO l a	将栈顶内容送入某变量单元中，a为偏移量，l 为层差
CAL l a	调用过程，a为过程地址，l 为层差
INT 0 a	在运行栈中为被调用的过程开辟a个单元的数据区
JMP 0 a	无条件跳转至a地址
JPC 0 a	条件跳转，当栈顶布尔值非真则跳转至a地址，否则顺序执行
OPR 0 0	过程调用结束后，返回调用点并退栈
OPR 0 1	栈顶元素取反
OPR 0 2	次栈顶与栈顶相加，退两个栈元素，结果值进栈
OPR 0 3	次栈顶减去栈顶，退两个栈元素，结果值进栈
OPR 0 4	次栈顶乘以栈顶，退两个栈元素，结果值进栈
OPR 0 5	次栈顶除以栈顶，退两个栈元素，结果值进栈
OPR 0 6	栈顶元素的奇偶判断，结果值在栈顶
OPR 0 7	
OPR 0 8	次栈顶与栈顶是否相等，退两个栈元素，结果值进栈
OPR 0 9	次栈顶与栈顶是否不等，退两个栈元素，结果值进栈
OPR 0 10	次栈顶是否小于栈顶，退两个栈元素，结果值进栈
OPR 0 11	次栈顶是否大于等于栈顶，退两个栈元素，结果值进栈
OPR 0 12	次栈顶是否大于栈顶，退两个栈元素，结果值进栈
OPR 0 13	次栈顶是否小于等于栈顶，退两个栈元素，结果值进栈
OPR 0 14	栈顶值输出至屏幕
OPR 0 15	屏幕输出换行
OPR 0 16	从命令行读入一个输入置于栈顶

5. 实验内容

(1) 词法部分

- 1) 用 flex 工具生成一个 PL/0 语言的词法分析程序，对 PL/0 语言的源程序进行扫描，识别出单词符号的类别，输出各种符号的信息。
- 2) 输入：PL/0 源程序
- 3) 输出：把单词符号分为下面六类，然后按单词符号出现顺序依次输出各单词符号的种类和出现在源程序中的位置（行数和列数）
 - K 类（关键字）
 - I 类（标识符）
 - C 类（常量）
 - O 类（算符）
 - D 类（界符）
 - T 类（其他）
- 4) 实验环境
 - Windows & C 或 C++
 - 词法分析器生成工具：flex

(2) 语法部分

- 1) 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
 - 输出：
 - 按归约顺序用到的语法规则
 - 语法单位的层次结构关系
- 2) 实验环境
 - Windows & C
 - 语法分析器生成工具：bison

(3) 语义部分

- 1) 在语法分析的基础上，在 bison 工具生成的 .y 文件中，进行静态（类型检查和作用域分析）和动态（根据产生式进行翻译）的语义分析，构造符号表，在相应的语法规约规则后添加适当的语义动作，生成可以用 interpret.c 程序直接读取并运行的类 pcode 代码。
 - 输入：PL/0 源程序
 - 输出：源程序对应的类 pcode 代码
- 2) 实验环境
 - Windows & C
 - 语法分析器生成工具：bison
 - 类 pcode 代码解释器 interpret.c 程序

三、实验步骤

1. 词法部分

(1) 思路说明:

根据词法部分的要求, 将对应的词法定义规则写到.1 文件中。定义不区分大小写的 13 个保留字 (之后增加了保留字 else 的功能, 共 14 个)、标识符、常量 (正负整数和浮点数)、算符、界符等的识别规则和动作, 并将相应内容输出到词法输出文件中。

(2) 识别规则:

1. Keyword
[iI][fF]|[tT][hH][eE][nN]|[eE][lL][sS][eE]|[wW][hH][iI][lL][eE]|[dD][oO][rR][eE][aA][dD]|[wW][rR][iI][tT][eE]|[cC][aA][lL][lL]|[bB][eE][gG][iI][nN]|[eE][nN][dD]|[cC][oO][nN][sS][tT][vV][aA][rR]|[pP][rR][oO][cC][eE][dD][uU][rR][eE]|[oO][dD][dD]|[cC][aA][sS][eE]|[eE][nN][dD][cC][aA][sS][eE]
2. Identifier [A-Za-z][A-Za-z0-9]*
3. Zero [0]
4. PossiInt ([+]?[1-9][0-9]*)
5. NegatiInt ([-]?[0-9]+)
6. Float {Zero}|{PossiInt}|{NegatiInt}(. [0-9]+)
7. Constant {PossiInt}|{NegatiInt}|{Zero}
8. Operator [\[\]\^\\-*\+\?\{\}\\"\\(\)\|\|\/\\$\<\>\#\=\:]|:=|<=|>=
9. Delimiter [\,\;\.\:]
10. Space (\)
11. Tab (\t)
12. Other [^ {Keyword}{Identifier}{Constant}{Operator}{Delimiter}]

(3) 对代码的说明

代码分为四个部分: 声明、辅助定义、识别规则和用户子程序

- 声明部分定义了要用到的全局变量;
- 辅助定义定义了 Keyword, Identifier 等正规式, 以便后续的代码更加易读。(尽管 PL/0 语言中数据类型只有整型, 也同时定义了识别正负浮点数的正规式并通过了测试, 可供功能拓展);
- 识别规则则是给出了对相关正规式识别后的一系列操作, 包括输出到 txt 文件和错误处理等;
- 用户子程序主要是调用 yylex() 函数, 并增加 yywrap() 函数 (由于不增加时在编译 lex.yy.c 文件过程中会报错, 故增加直接返回 1 的 yylex() 函数);

对于未匹配的字符, 程序会识别为 T 类, 同时对于长度超过限定的标识符和常量会进行报错。

(4) 操作命令:

- a. 将 flex.exe 文件、写好的 lex 程序和待输入的 PL/0 程序源代码放到同一个文件夹
- b. 在 Windows 系统下打开 cmd, 进入当前文件夹
- c. 在 cmd 下输入命令 flex lex.l 将 lex 程序用 flex 工具生成 lex.yy.c 文件, 在 C 编译环境下对该文件进行编译, 生成 lex.yy.exe 文件
- d. 输入命令 lex.yy.exe < max.pl0 对 max.pl0 文件进行词法分析

2. 语法部分

(1) 思路说明:

对于第一个输出, 按规约顺序输出用到的语法规则, 思路比较简明, `yyparse` 函数执行规约的时候会按照程序本身的顺序进行规约, 即直接在对应产生式规约后执行的动作中加上将该产生式输出到输出文件的操作即可。对于第二个输出, 建立一个栈, 在词法每次进行匹配的时候, 将匹配到的终结符入栈。同时建立一个树, 左节点表示孩子节点, 右节点表示兄弟节点。在语法文件进行规约的时候, 在规约动作中, 将产生式右边的对应的终结符或者非终结符出栈, 并建立新节点作为产生式左部的非终结符, 将产生式右部的第一个符号作为产生式左部的孩子节点, 并以此将右部其他节点作为其左边节点的兄弟节点, 最后将产生式左部的节点压入栈中。程序执行完成规约后, 对建立的树进行先序遍历, 对应层次越深缩进越多, 以此来表示层次结构。

(2) 实验步骤:

a. 将 `flex.exe` 文件、`bison` 相关程序文件、写好的 `lex` 程序、写好的语法的 `.y` 程序和待输入的 `PL/0` 程序源代码放到同一个文件夹

b. 在当前目录下打开 `cmd`

c. 在 `cmd` 下输入命令 `flex lex.l` 将 `lex` 程序用 `flex` 工具生成 `lex.yy.c` 文件, 输入命令 `bison Syntax.y -d` 将 `.y` 程序用 `bison` 工具生成 `Syntax.tab.c` 和 `Syntax.tab.h` 文件(提前将 `Syntax.tab.h` 文件在 `lex.l` 声明部分中引用), 并在生成的 `Syntax.tab.c` 中的 `/*Shift the lookahead token.*/` 注释下方添加 `Process(key);` 一行, 用于在向前读取一个符号之前将上一个符号进行 `Process` 函数中的相关操作(详见实验结果部分)。

d. 在 `cmd` 下用 `gcc` 命令将 `Syntax.tab.c` 和 `lex.yy.c` 文件联合编译:

```
gcc -o Syntax.tab.exe Syntax.tab.c lex.yy.c
```

e. 输入命令 `Syntax.tab.exe < test-syn.pl0` 对 `test-syn.pl0` 文件进行对应输出

(3) 对代码的说明

下面对部分相关代码进行说明:

```
ProceDec      :ProceHead subProg SEMI {
                fprintf(fi, "ProceDec -> ProceHead subProg ProceDec;\n");
                Reduce("ProceDec", 3);
            }
|ProceDec ProceHead subProg SEMI {
                fprintf(fi, "ProceDec -> ProceDec ProceHead subProg;\n");
                Reduce("ProceDec", 4);
            }
```

- 上图是过程并列的部分产生式定义, 采用左递归方式, 对应规约动作为输出该产生式, 并调用 `Reduce` 函数进行出栈入栈和连接节点操作

```
if(!strcmpi(yytext, "BEGIN")){
    strcpy(key, "BEGIN");
    // Process(key);
    return _BEGIN_;
}
```

- 上图为词法.1 文件中的代码, 匹配到 `BEGIN` 终结符后, 将该终结符压入栈中(由于 `bison` 会提前多看一个符号, 所以将具体的 `Process`

函数即入栈操作放到了 Syntax.tab.c 中移进操作之前), 并将其 return 到语法分析程序中进行处理

```
void Reduce(char* name, int num){
    elem t[num];
    for (int i = 0; i < num; i++){
        t[i] = stack_pop();
    }
    Node* n = (Node*)malloc(sizeof(Node));
    n -> data = name;
    n -> left = NULL;
    n -> right = NULL;
    left_insert(n, t[num-1]);
    for (int i = num-1; i > 0; i--){
        right_insert(t[i], t[i-1]);
    }
    stack_push(n);
}
```

- 上图是语法规约动作中对应的规约操作, 即产生式左的节点入栈, 右边的节点进行连接

```
void PreOrderTravel(Node* T, int k){
    if(T==NULL) return;
    fprintf(fh, "%d:|\t", k);
    for(int i=0; i<k-1; i++) fprintf(fh, "|\t");
    fprintf(fh, "%s\n", T->data);
    PreOrderTravel(T->left, k+1);
    PreOrderTravel(T->right, k);
}
```

- 上图是进行先序遍历的函数, k 表示递归的层数, T 是传进来的节点, 最终规约完后只剩 Program 一个节点, 即树的根节点, 从其开始先序遍历即可得到对应的层次结构

3. 语义部分

(1) 思路说明:

语义部分要求大致可以分为两个部分完成: 符号表和生成类 pcode 代码。

对于符号表, 记录了所有常量、变量、过程名的相关信息, 以及相应的还有 display 表。用结构体数组实现符号表, 其中包含了各个标识符的类型、名字、层次等信息, 具体结构如下:

```
typedef struct{
    int kind;           //const = 1, var = 2, proc = 3
    char* name;
    int val;            //如果为-2则为过程名, -1为未赋值的变量
    int level;          //所在层次
    int addr;           //变量的偏移地址
    int previous;       //指向的下一个地址
}symbol;

#define MAX_SYMBOL_TABLE_SIZE 50
typedef symbol* sym;

typedef struct{
    int top;
    symbol index[MAX_SYMBOL_TABLE_SIZE];
}symbol_table;
```

其中 level 标记了标识符所在的层次，主程序所在的层次为 0；kind 记录了标识符的类型；val 存储变量的值，为进行区分，未赋值的变量为-1，过程名为-2；addr 只有变量使用，记录了变量在运行栈中相对于及地址的偏移量，从静态连、动态链、返回地址之后开始，即从 3 开始。

在 bison 中使用 union 结构可以定义非终结符和终结符的类型，便于在识别具体的语法规则后相应的语义动作中进行值的传递。定义的类型如下：

```
%union{
    struct{
        int val;
        char *str;
    }var;
    int val;
}

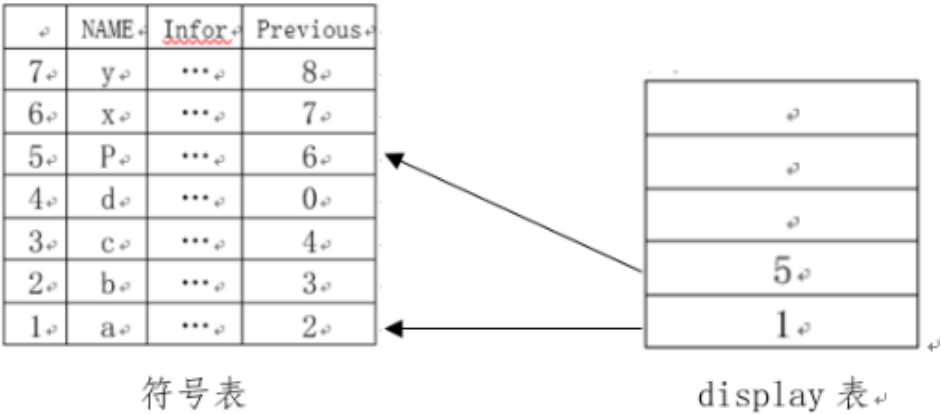
%start Program
%token <var> IDENTIFIER
%token <val> CONSTANT
%token <var> RELOP
%token COMMA SEMI DOT LPAREN RPAREN ASSIGN MINUS COLON
%token CONST VAR PROCEDURE _BEGIN_ END IF THEN ELSE ODD WHILE DO CALL READ WRITE CASE ENDCASE
%type <val> Factor Term Expr
%left PLUS MINUS
%left TIMES DIVIDE
%nonassoc LOWER_THAN_ELSE
%nonassoc ELSE
```

上图也给出了 .y 文件中定义的各个终结符和非终结符及其类型，包括左结合的优先性和用 %nonassoc 定义的 else 的优先性（强于 if then），在具体的产生中做如下定义：

```
CondStm      :IF Condition BeforeThen THEN Statement %prec LOWER_THAN_ELSE {
               |IF Condition BeforeThen THEN Statement ELSE BeforeElseDo Statement{
```

具体识别过程中则会优先判断 if...then...后是否有 else 关键字，有的话则优先进行规约。

在对应的规约位置后的语义动作处进行相关操作。声明部分先对标识符进行是否定义过的判断（故过程和变量不能同名），即静态语义检查，未定义过则将其名字、值、层次等信息写入符号表。栈式的 display 表最底处加入主程序定义部分开始的位置，即 1，过程定义处将过程名加入到符号表中，并将过程名在符号表中的位置写到 display 表的栈顶，同时写入符号表的时候将过程及其层次和当前类 pcode 的位置写入一个过程栈，便于在之后 call 过程的时候进行调用。



下面对类 pcode 各条代码在对应规约规则后的语义动作中具体生成方法进行介绍。类 pcode 生成函数 gen 三个参数对应为 f, l, a。

LIT: 将常量取到栈顶, 识别到常量则直接将其值生成 LIT 代码, 若引用之前定义过的常量名, 则在符号表中找到其值进行生成。

```
gen(LIT, 0, $1);
```

```
gen(LIT, 0, symtable.index[find_sign($1.str)].val);
```

LOD: 将 1 层差, 偏移量为 a 的变量取到栈顶; 在识别到标识符后取值。

```
gen(LOD, now_Level - symtable.index[find_sign($1.str)].level, symtable.index[find_sign($1.str)].addr);
```

STO: 将栈顶值送到 1 层差, 偏移量为 a 的变量单元。在赋值语句和 read 语句规约后生成。

```
gen(STO, now_Level - symtable.index[find_sign($3.str)].level, symtable.index[find_sign($3.str)].addr);
```

CAL: 调用层差为 1, 位置为 a 的过程。对应过程位置到过程栈中根据过程名寻到, 故过程名字不能重复。

```
gen(CAL, now_Level - prostack[findpro($2.str)].l, prostack[findpro($2.str)].pos - 1);
```

INT: 用于过程开始时开辟 a 个变量的数据区。在声明部分之后, 具体语句开始之前生成, 查符号表和 display 表得到当前层的常量变量个数, 前 3 个数据区为静态链、动态链和返回地址, 故实际开辟大小为 3+常量变量个数。

```
gen(INT, 0, symtable.top - display_stack[display_top] + 3);
```

JMP: 无条件跳转至第 a 条类 pcode 代码。在应无条件跳转的位置先记录位置, 然后生成 JMP, 0, 0 的代码等待之后回填。在 while 循环中, 每层循环记录循环开始位置, 然后在循环结束时生成跳转到循环开始时位置的 JMP 代码, 然后循环位置栈顶减一。

```
gen(JMP, 0, 0);
```

```
gen(JMP, 0, whilepos[whiletop--]);
```

JPC: 条件跳转, 当栈顶值非真则跳转到 a 地址, 否则顺序执行。所有进行条件判断的位置记录位置并生成 JPC, 0, 0 的代码, 在条件为非时应该执行的语句之前进行回填。

```
gen(JPC, 0, 0);
```

OPR: 根据 a 具体的值进行相应的操作。在程序或过程结束规约的时候生成 OPR, 0, 0 结束程序, 在相应加减乘除 read 和 write 等操作规约时生成对应 a 值的 OPR 代码。

```
gen(OPR, 0, 0);
```

```
Expr RELOP Expr {  
    if(strcmp($2.str, "=") == 0) gen(OPR, 0, 8);  
    if(strcmp($2.str, "#") == 0) gen(OPR, 0, 9);  
    if(strcmp($2.str, "<") == 0) gen(OPR, 0, 10);  
    if(strcmp($2.str, ">=") == 0) gen(OPR, 0, 11);  
    if(strcmp($2.str, ">") == 0) gen(OPR, 0, 12);  
    if(strcmp($2.str, "<=") == 0) gen(OPR, 0, 13);
```

对于回填的处理, 设置了一个回填栈, 在每次需要回填的位置先记录要回填的位置并压入栈中, 到了规约要回填的时候把当前类 pcode 代码的行数 code_line 回填到栈中记录的位置, 并将栈顶位置出栈。

```
bkpchpos[++bkpchpos_top] = code_line; //填入回填栈等待回填  
gen(JPC, 0, 0);
```

```
Backpatch(bkpchpos[bkpchpos_top--], code_line);
```

对于条件语句, 带 else 的条件语句优先级比单纯的 if else 高。在对条

件进行判断之后，随后记录回填位置并生成 JPC 代码，在 then 后的语句执行完之后进行回填。如果是带 else 的条件语句，在 then 执行完之后，把当前 code_line 的下一行先回填至条件判断的假出口，回填栈顶出栈后再记录当前位置为回填位置，在 else 后的语句执行完之后再进行回填。

```
CondStm      :IF Condition BeforeThen THEN Statement %prec LOWER_THAN_ELSE {
                Backpatch(bkpchpos[bpchpos_top--], code_line);
                fprintf(fi, "CondStm -> IF Condition THEN Statement\n");
                Reduce("CondStm", 4);
            }
            [IF Condition BeforeThen THEN Statement ELSE BeforeElseDo Statement{
                Backpatch(bkpchpos[bpchpos_top--], code_line);
                fprintf(fi, "Statement -> IF Condition THEN Statement ELSE Statement\n");
                Reduce("CondStm", 6);
            }
            ;
BeforeThen    :{
                bkpchpos[++bpchpos_top] = code_line; //填入回填栈等待回填
                gen(JPC, 0, 0);
            }
            ;
BeforeElseDo:{
                Backpatch(bkpchpos[bpchpos_top--], code_line+1); //回填条件错误跳转的地址
                bkpchpos[++bpchpos_top] = code_line; //填入回填栈等待回填
                gen(JMP, 0, 0);
            }
            ;
```

对于 while 循环语句，相似的，设置了记录循环位置的栈 whilepos，在循环开始时记录循环所在的 code_line 位置，条件判断之后记录回填位置并生成 JPC 代码，在循环条件的真出口语句执行完之后，先生成 JMP 回记录的循环初始位置的代码，再将循环的假出口进回填。

```
WhilelpStm    :WHILE BeforeCond Condition DO BeforeCondDo Statement {
                gen(JMP, 0, whilepos[whiletop--]);
                Backpatch(bkpchpos[bpchpos_top--], code_line);
                fprintf(fi, "WhilelpStm -> WHILE Condition DO Statement\n");
                Reduce("WhilelpStm", 4);
            }
            ;
BeforeCond     :{
                whilepos[++whiletop] = code_line;
            }
            ;
BeforeCondDo:{
                bkpchpos[++bpchpos_top] = code_line; //填入回填栈等待回填
                gen(JPC, 0, 0);
            }
            ;
```

对于拓展功能 case 语句，语法定义对于给出的标识符，和 casebody 中的每个常量进行条件比较，如果相等则执行后面的语句。在 casehead 的时候记录标识符，在常量之后记录位置等待回填，每一条 case 比较完之后进行回填，并将记录的标识符重新生成 LOD 代码。到了 endcase 规约的时候对最后一次 case 的 LOD 进行回撤，即 code_line--，结束 case 语句。

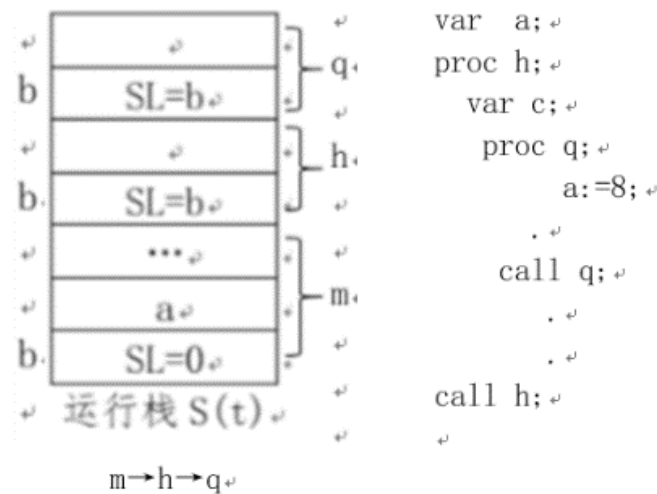
```
CaseStm       :CaseHead CaseBody ENDCASE {
                // Backpatch(bkpchpos[bpchpos_top--], code_line);
                code_line--; //回撤最后一次LOD指令
                fprintf(fi, "CaseStm -> CaseHead CaseBody ENDCASE\n");
                Reduce("CaseStm", 3);
            }
            ;
CaseHead       :CASE IDENTIFIER COLON {
                if(find_sign($2.str) == -1){ //变量没有被定义
                    printf("CaseHead error: '%s' has not been declared!\n", $2.str);
                    exit(1);
                }
                caseid = strdup($2.str);
                gen(LOD, now_level - symtable.index[find_sign($2.str)].level, symtable.index[find_sign($2.str)].addr);
                fprintf(fi, "CaseHead -> CASE IDENTIFIER COLON\n");
                Reduce("CaseHead", 3);
            }
            ;
```

```

CaseBody :CaseBody AfterConst COLON Statemt {
    Backpatch(bkpchpos[bpchpos_top--], code_line);
    gen(LOD, now_level - symtable.index[find_sign(caseid)].level, symtable.index[find_sign(caseid)].addr);
    fprintf(fi, "CaseBody -> CaseBody CONSTANT COLON Statemt\n");
    Reduce("CaseBody", 4);
}
AfterConst COLON Statemt {
    Backpatch(bkpchpos[bpchpos_top--], code_line);
    gen(LOD, now_level - symtable.index[find_sign(caseid)].level, symtable.index[find_sign(caseid)].addr);
    fprintf(fi, "CaseBody -> CONSTANT COLON Statemt\n");
    Reduce("CaseBody", 3);
}
;
AfterConst :CONSTANT {
    gen(LIT, 0, $1);
    gen(OPR, 0, 0);
    bkpchpos[++bpchpos_top] = code_line; //填入回栈栈等待回填
    gen(JPC, 0, 0);
}
;

```

对于生成的类 pcode 代码，interpret.c 文件会从.txt 文件中读取类 pcode 代码并进行解释，用到的 base 函数示意图如下：



q 引用 m 的变量时层次差 l 为 2，所以需寻找 m 的基地址 b。由 q 的 SL 找到 h 的基地址 b 再由 h 的 SL 找到 m 的基地址 b。

具体实现方式如下：

```

int base(int level, int *s, int b){
    int tempb = b;
    while(level > 0){
        tempb = s[tempb];
        level --;
    }
    return tempb;
}

```

对于传入的参数和运行栈指针 s，在层次差范围内，依次根据上一层次的基地址找上上层的基地址，最后得到所要寻找层的基地址并返回。

四、实验结果

1. 词法部分

执行 flex test.1 (语法和语义部分更名为 Word.1) 命令后未报错如下：

```

\Compiling Principle\Lab1>flex test.1
\Compiling Principle\Lab1>

```

对 lex.yy.c 进行编译，并执行 lex.yy.exe < case_test.pl0:

```
\Compiling Principle\Lab1>flex test.1
\Compiling Principle\Lab1>lex.yy.exe < case_test.pl0
\Compiling Principle\Lab1>
```

case_test.pl0 作为样例 pl0 代码所示如下，在其末尾添加了不符合规定的标识符和各种常量和符号用于测试不同情况下的输出情况。

```
1  var n, m, i, j, digit(1:3);
2
3
4  procedure init;
5  begin
6      i := 1;
7      while i < 4 do
8          begin
9              digit(i) := 0;
10             i := i + 1;
11         end;
12 end;
13
14 procedure parse;
15 var tmp;
16 begin
17     tmp := n;
18     i := 0;
19     while n != 0 do
20         begin
21             m := n / 10;
22             i := i + 1;
23             digit(i) := n - m * 10;
24             n := m;
25         end;
26     n := tmp;
27 end;
28
29 procedure sumdigit;
30 begin
31     m := digit(1) * digit(1) * digit(1) +
32         digit(2) * digit(2) * digit(2) +
33         digit(3) * digit(3) * digit(3);
34 end;
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53 var input, a;
54
55 procedure P;
56 begin
57     read(input);
58     case input:
59         1: a := 0
60         2: a := 1
61         3: a := 2;
62     endcase;
63 end;
64
65 call P.
66
67
68 +1321
69 a:=+2-5
70 a:=2-6
71 a:= 2 - 6
72 +2 + 6
73 -2 + 6
74 -5 - 2
75 +1 + +6
76 0
77 1
78 -1
79 123
80 -123
81 &
82 _
83 @
84 :::
85 a12121
86 a12132131321321
87 9adsasdgasdgasdga454adsg2sd
88 adsasdg3asdgasdga454adsg2sd
89 123456789123456789
90 1/0
```

其最终输出到 txt 文件中的结果如下:

```
1  var : K, (2, 1)
2  n : I, (2, 4)
3  , : D, (2, 5)
4  m : I, (2, 6)
5  , : D, (2, 7)
6  i : I, (2, 8)
7  , : D, (2, 9)
8  j : I, (2, 10)
9  , : D, (2, 11)
10 digit : I, (2, 12)
11 ( : O, (2, 17)
12 1 : C, (2, 18)
13 : : T, (2, 19)
14 3 : C, (2, 20)
15 ) : O, (2, 21)
16 ; : D, (2, 22)
17 procedure : K, (4, 1)
18 init : I, (4, 10)
19 ; : D, (4, 14)
20 begin : K, (5, 1)
21 i : I, (6, 1)
22 := : O, (6, 2)
23 1 : C, (6, 4)
24 ; : D, (6, 5)
25 while : K, (7, 1)
26 i : I, (7, 6)
27 < : O, (7, 7)
28 4 : C, (7, 8)
29 do : K, (7, 9)
30 begin : K, (8, 1)
31 digit : I, (9, 1)
32 ( : O, (9, 6)
33 i : I, (9, 7)
34 ) : O, (9, 8)
35 := : O, (9, 9)
36 0 : C, (9, 11)
37 ; : D, (9, 12)
38 i : I, (10, 1)
249 2 : C, (71, 4)
250 - : O, (71, 5)
251 6 : C, (71, 6)
252 +2 : C, (72, 1)
253 + : O, (72, 3)
254 6 : C, (72, 4)
255 -2 : C, (73, 1)
256 + : O, (73, 3)
257 6 : C, (73, 4)
258 -5 : C, (74, 1)
259 - : O, (74, 3)
260 2 : C, (74, 4)
261 +1 : C, (75, 1)
262 + : O, (75, 3)
263 +6 : C, (75, 4)
264 0 : C, (76, 1)
265 1 : C, (77, 1)
266 -1 : C, (78, 1)
267 123 : C, (79, 1)
268 -123 : C, (80, 1)
269 & : T, (81, 1)
270 _ : T, (82, 1)
271 @ : T, (83, 1)
272 : : T, (84, 1)
273 : : T, (84, 2)
274 : : T, (84, 3)
275 a12121 : I, (85, 1)
276 Error!Expected a shorter IDENTIFIER!(86, 1)
277 9 : C, (87, 1)
278 Error!Expected a shorter IDENTIFIER!(87, 2)
279 Error!Expected a shorter IDENTIFIER!(88, 1)
280 Error!Expected a shorter CONSTANT!(89, 1)
281 1 : C, (90, 1)
282 / : O, (90, 2)
283 0 : C, (90, 3)
284 01 : C, (91, 1)
285 ab : I, (91, 3)
286
```


2. 语法部分

语法部分在词法 Word.l 文件中增加了新的内容, 需要用 flex 对 .l 文件重新编译。在 bison 指令中添加 -d 用于生成 .l 中引用的 Syntax.tab.h 头文件 (此外若执行 bison 命令后有移进规约冲突的报错, 可添加 -v 生成可读的 .output 文件, 里面记录了所有的状态信息, 方便查看各种移进规约冲突等的位置)。

在使用命令 `bison -d Syntax.y` 生成 `Syntax.tab.c` 文件后, 查找 `/* Shift the lookahead token. */` 的注释, 在其下面添加一行 `Process(key);` 代码, 再用 `gcc -o Syntax.tab.exe Syntax.tab.c lex.yy.c` 命令进行联合编译 (注: 由于 bison 识别符号会自动提前 lookahead 一个符号, 若在词法 Word.l 中对应的识别规则后的动作中进行 Process 函数的操作会造成生成语法层次结构的混乱, 故在 lookahead 之前进行 Process 函数的操作)。

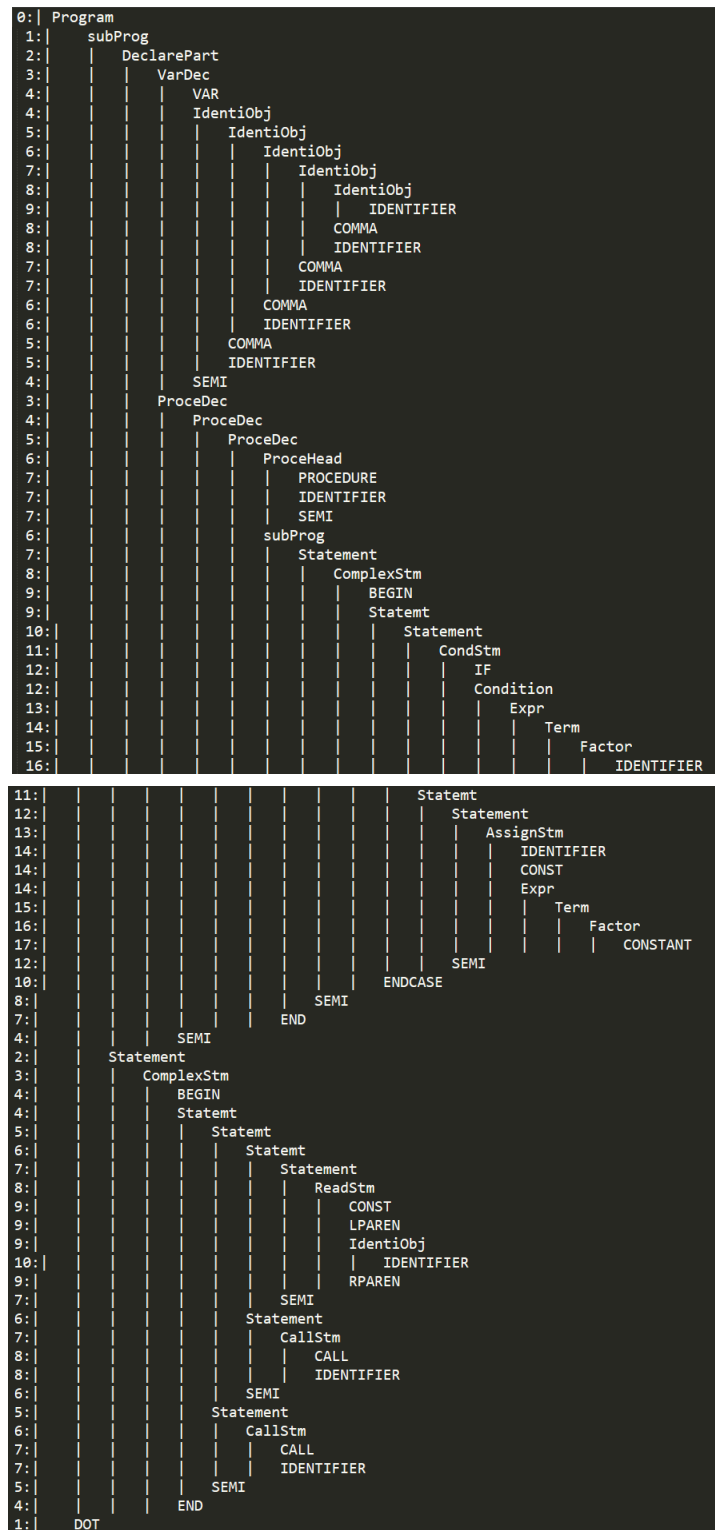
```
\Compiling Principle\Lab2>flex Word.l
\Compiling Principle\Lab2>bison -d Syntax.y
\Compiling Principle\Lab2>gcc -o Syntax.tab.exe Syntax.tab.c lex.yy.c
\Compiling Principle\Lab2>Syntax.tab.exe < test-syn.pl0
```

用 test-syn.pl0 文件进行测试, 使用命令 `Syntax.tab.exe < test-syn.pl0`, 按规约顺序输出的语法规则如下:

```
1  IdentiObj -> IDENTIFIER
2  IdentiObj -> IdentiObj, IDENTIFIER
3  IdentiObj -> IdentiObj, IDENTIFIER
4  IdentiObj -> IdentiObj, IDENTIFIER
5  IdentiObj -> IdentiObj, IDENTIFIER
6  VarDec -> VAR IdentiObj;
7  ProceHead -> PROCEDURE IDENTIFIER;
8  Factor -> IDENTIFIER
9  Term -> Factor
10 Expr -> Term
11 Factor -> CONSTANT
12 Term -> Factor
13 Expr -> Term
14 Condition -> Expr RELOP Expr
15 Factor -> IDENTIFIER
16 Term -> Factor
17 Expr -> Term
18 ExprObj -> Expr
19 WriteStm -> WRITE(ExprObj)
20 Statement -> WriteStm
21 Statemt -> Statement;
22 Factor -> IDENTIFIER
23 Term -> Factor
24 Expr -> Term
25 Factor -> CONSTANT
26 Term -> Factor
27 Expr -> Expr - Term
28 AssignStm -> IDENTIFIER := Expr
29 Statement -> AssignStm
30 Statemt -> Statemt Statement;

236 Statemt -> Statement;
237 CaseBody -> CaseBody CONSTANT COLON Statemt
238 Factor -> CONSTANT
239 Term -> Factor
240 Expr -> Term
241 AssignStm -> IDENTIFIER := Expr
242 Statement -> AssignStm
243 Statemt -> Statement;
244 CaseBody -> CaseBody CONSTANT COLON Statemt
245 CaseStm -> CaseHead CaseBody ENDCASE
246 Statement -> CaseStm
247 Statemt -> Statemt Statement;
248 ComplexStm -> _BEGIN_ Statemt Statement END
249 Statement -> ComplexStm
250 subProg -> DeclarePart Statement
251 ProceDec -> ProceDec ProceHead subProg;
252 DeclarePart -> VarDec ProceDec
253 IdentiObj -> IDENTIFIER
254 ReadStm -> READ(IdentiObj)
255 Statement -> ReadStm
256 Statemt -> Statement;
257 CallStm -> CALL IDENTIFIER
258 Statement -> CallStm
259 Statemt -> Statemt Statement;
260 CallStm -> CALL IDENTIFIER
261 Statement -> CallStm
262 Statemt -> Statemt Statement;
263 ComplexStm -> _BEGIN_ Statemt Statement END
264 Statement -> ComplexStm
265 subProg -> DeclarePart Statement
266 Program -> subProg.
```

语法单位的层次结构关系如下, 行首的数字表示节点 (递归) 的层数:



3. 语义部分

和语法部分一样，需要在编译生成.tab.c 文件中/* Shift the lookahead token. */的注释下面添加一行 Process(key); 联合编译后生成 out.exe 可执行文件，执行 out.exe < 会在 pcode.txt 中输出生成的类 pcode 代码。下面是用于测试的 p10 代码，里面包含了递归调用、过程并列、过程嵌套、if then 条件语句、if then else 条件语句、while 循环语句、case 语句和常规的读写运算等操作语句。

```

1  var n, a, b, c, max;      23      sum := 0;      45
2  procedure rec;           24      if n > 0 then      46 procedure I;
3  begin                    25      if n < 10 then      47 var a, input;
4      if n # 0 then        26      while n > 0 do      48 begin
5          begin            27      begin
6              write(n);    28      sum := sum + a;    49 read(input);
7              n := n - 1;  29      n := n - 1;      50 a := 0;
8              call rec;    30      end;              51 case input:
9          end;            31      write(sum);        52 1: a:= 8; write(input);
10 end;                    32      call R;            53 2: a:= 5; write(input);
11                          33      end;              54 3: write(input);
12 procedure P;            34 begin                    55 endcase;
13     const x = 99;        35 read(a);                56 write(a);
14     procedure Q;         36 read(b);                57 end;
15     var a, b, n, sum;    37 read(c);                58
16     procedure R;         38 if a > b then max := a  59 begin
17     begin                39 else max := b;          60 read(n);
18         write(x);        40 if max > c then max := max  61 write(n);
19     end;                 41 else max := c;          62 call rec;
20     begin                42 write(max);              63 call P;
21     read(a);             43 call Q;                  64 call I;
22     read(n);             44 end;                            65 end.

```

输出的类 pcode 代码如下:

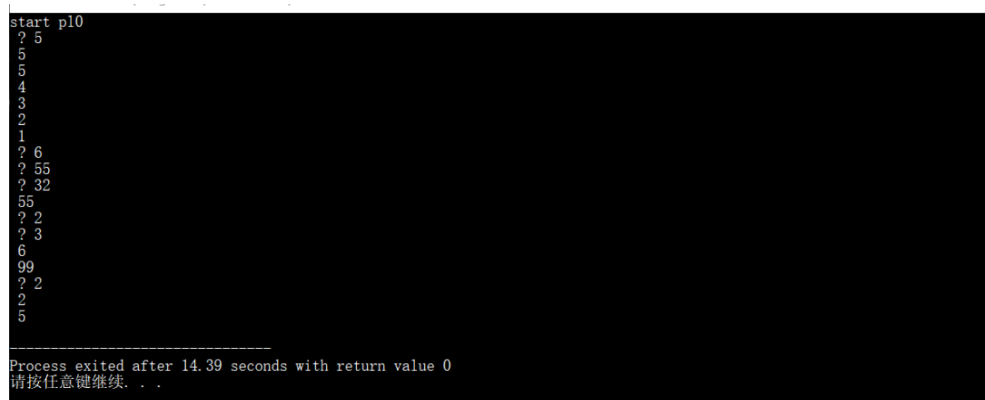
```

1  JMP 0 128 36 LOD 0 5 71 LOD 1 4 106 LOD 0 4
2  JMP 0 2 37 LIT 0 10 72 LOD 1 5 107 OPR 0 14
3  INT 0 3 38 OPR 0 10 73 OPR 0 13 108 OPR 0 15
4  LOD 1 3 39 JPC 0 52 74 JPC 0 76 109 LOD 0 4
5  LIT 0 0 40 LOD 0 5 75 LOD 1 5 110 LIT 0 2
6  OPR 0 9 41 LIT 0 0 76 STO 1 7 111 OPR 0 8
7  JPC 0 15 42 OPR 0 12 77 LOD 1 7 112 JPC 0 117
8  LOD 1 3 43 JPC 0 52 78 LOD 1 6 113 LIT 0 5
9  OPR 0 14 44 LOD 0 6 79 OPR 0 12 114 STO 0 3
10 OPR 0 15 45 LOD 0 3 80 JPC 0 82 115 LOD 0 4
11 LOD 1 3 46 OPR 0 2 81 LOD 1 7 116 OPR 0 14
12 LIT 0 1 47 STO 0 6 82 STO 1 7 117 OPR 0 15
13 OPR 0 3 48 LOD 0 5 83 LOD 1 7 118 LOD 0 4
14 STO 1 3 49 LIT 0 1 84 LOD 1 6 119 LIT 0 3
15 CAL 1 1 50 OPR 0 3 85 OPR 0 13 120 OPR 0 8
16 OPR 0 0 51 STO 0 5 86 JPC 0 88 121 JPC 0 124
17 JMP 0 57 52 JMP 0 39 87 LOD 1 6 122 LOD 0 4
18 JMP 0 24 53 LOD 0 6 88 STO 1 7 123 OPR 0 14
19 JMP 0 19 54 OPR 0 14 89 LOD 1 7 124 OPR 0 15
20 INT 0 3 55 OPR 0 15 90 OPR 0 14 125 LOD 0 3
21 LIT 0 99 56 CAL 0 18 91 OPR 0 15 126 OPR 0 14
22 OPR 0 14 57 OPR 0 0 92 CAL 0 17 127 OPR 0 15
23 OPR 0 15 58 INT 0 4 93 OPR 0 0 128 OPR 0 0
24 OPR 0 0 59 OPR 0 16 94 JMP 0 94 129 INT 0 8
25 INT 0 7 60 STO 1 4 95 INT 0 5 130 OPR 0 16
26 OPR 0 16 61 OPR 0 16 96 OPR 0 16 131 STO 0 3
27 STO 0 3 62 STO 1 5 97 STO 0 4 132 LOD 0 3
28 OPR 0 16 63 OPR 0 16 98 LIT 0 0 133 OPR 0 14
29 STO 0 5 64 STO 1 6 99 STO 0 3 134 OPR 0 15
30 LIT 0 0 65 LOD 1 4 100 LOD 0 4 135 CAL 0 1
31 STO 0 6 66 LOD 1 5 101 LIT 0 1 136 CAL 0 16
32 LOD 0 5 67 OPR 0 12 102 OPR 0 8 137 CAL 0 93
33 LIT 0 0 68 JPC 0 70 103 JPC 0 108 138 OPR 0 0
34 OPR 0 12 69 LOD 1 4 104 LIT 0 8
35 JPC 0 52 70 STO 1 7 105 STO 0 3 139

```

执行 interpret.c 解释程序。p10 程序内容为：先输入 1 个数字并将其输出；调用 rec 函数，输入 1 个数，将该数依次减 1 输出；调用 P 过程，输入 3 个数并将最大值输出，其中调用 Q 过程输入 2 个数并用加法方式循环计算两数之积，其中再调用 Q 过程输出 P 过程中定义的常量值；调用 I 过程根据输入的数 case 操作修改值并输出输入的数和操作执行后的 a 变量。具体

执行过程如下：



```
start pl0
? 5
5
? 6
6
? 5
5
? 4
4
? 3
3
? 2
2
? 1
1
? 6
6
? 55
55
? 32
32
? 55
55
? 2
2
? 3
3
? 3
3
? 6
6
? 99
99
? 2
2
? 2
2
? 5
5

-----
Process exited after 14.39 seconds with return value 0
请按任意键继续. . .
```

至此，整个 PL/0 编译器已经实现，包括词法、语法、语义三个部分，生成类 pcode 代码后由 interpret.c 程序读取并解释运行，对程序进行了多次调试和测试，运行正常。如果遇到异常可尝试修改 interpret.c 程序中 STACKSIZE 的大小及 .y 文件中各个栈或数组的大小。

五、实验体会

词法实验是关于编译器制作中的第一步，即词法分析器。通过词法实验，理解了编译器的工作机制，掌握了词法分析器的生成工具 LEX 的用法，熟悉了正规式的匹配规则和相关指令。

语法实验是关于编译器制作中的第二步，即语法分析器。通过语法实验，进一步理解了编译器的工作机制，尤其是语法处理过程中的规约规则和其执行顺序，掌握了语法分析器的生成工具 bison 的用法，同时对产生式和规约的层次结构有了更深的理解。

语义实验是关于编译器制作中的第三步，即语义分析器。通过语义实验，在前两次实验的基础上，将整个编译器的三个部分综合在了一起，进一步加深了编译器的尤其是语义生成中间代码部分的理解，也熟悉了类 pcode 代码和解释器运行的简单实现方式。整个实验下来，掌握了自己实现编译器的方式，完成了自定义语法的编译实现，对编译器工作和实现方式有了的理解和实践经验。