

pl0 基础部分实验报告

一. 任务分工

我们小组本来一共 4 人，分别为梁聪、罗李媛、雷珣思、方林涛，罗李媛和雷珣思复制要求（1）、（4）和（5）中 `return` 的实现，方林涛负责要求（3），梁聪负责要求（2）和（5）中的剩余项目。

实际工作中，在我们都互相有 qq 联系的情况下，方林涛从未参与小组见面讨论，也未接受我们的 github 邀请链接，即未完成任何实际工作。最终罗李媛和雷珣思任务不变，梁聪完成（2）、（3）和（5）中剩余项目。

二. 设计框架

- ① 词法分析：复合符号的词法如下

id \rightarrow letter(letter|digit)*

number \rightarrow digit(digit)*

letter \rightarrow A|B|C|...|Z|a|b|...|z

digit $\rightarrow 0|1|\dots|9$

其中 id 中的 letter 区分大小写。

- ② 语法分析: Procedure := Block “.”

$$\text{Block} := \{(\text{"const"} \text{ (SingleInitial | ArrayInitial)})$$

```
{"," (SingleInitial | ArrayInitial)} ";"
```

```
| ("var" (id | Array){"," (id | Array)} ";")
```

```
|("procedure" id "(" Arguments ")" Block ";")}
```

Statement

SingleInitial := id "=" (number | const)

```
ArrayInitial := Array "=" "{" (number | const) "," (number | const) "}"
```

Array := id "[" (number | const) "]" { "[" (number | const) "]" }

$$\text{Arguments} := \text{id} \{“,” \text{id}\}$$

不带双引号的 `const` 表示已定义的常量或常量数组中的一个元素

Statement := Chunk | Top_expr ";" | "exit" ";" | "return" [Top_expr] ";"

Print statement	If statement	For statement	While statement
-----------------	--------------	---------------	-----------------

Chunk := ("begin" | "{" {Statement} ("end" | "}"))

If_statement := "if" "(" Top_expr ")" Statement ["else" Statement]

For_statement := "for" "(" Top_expr ";" Top_expr ";" Top_expr ")" Statement

While_statement := "while" "(" Top_expr ")" Statement

Print_statement := "print" "(" [Top_expr {"," Top_expr}] ")" ";"

$$\text{Top_expr} := \text{id} \text{ ":=" } \text{Top_expr} \mid \text{Or_expr}$$
$$\text{Or_expr} := \text{Or_expr} \text{ " || " } \text{And_expr} \mid \text{And_expr}$$
$$\text{And_expr} := \text{And_expr} \text{ ``\&\&' ' } \text{Bitor_expr} \mid \text{Bitor_expr}$$
$$\text{Bitor_expr} := \text{Bitor_expr} \mid \text{Bitxor_expr} \mid \text{Bitxor_expr}$$
$$\text{Bitxor_expr} := \text{Bitxor_expr} \wedge \text{Bitand_expr} \mid \text{Bitand_expr}$$
$$\text{Bitand_expr} := \text{Bitand_expr} \text{ ``\&'' } \text{Rel_expr} \mid \text{Rel_expr}$$
$$\text{Rel_expr} := \text{Expression } ("=" \mid "!=" \mid ">" \mid ">=" \mid "<" \mid "<=") \text{Expression}$$
$$\text{Expression} := \text{Term } (+ \mid -) \text{Term}$$

```

Term := Factor ("*" | "/" | "%") Factor
Factor := id | Array | number | "(" Top_expr ")"
        | ("-" | "!" | "~") Factor
        | ProcedureCall
ProcedureCall := id "(" [Top_expr {"," Top_expr}] ")"

```

③ 语法特点说明：

- 整个程序以 “.” 结束，子过程以 “;” 结束，这保留了最初版本 p10 的特点；
- 修改原有 block 使得可以读入任意顺序、数量不超过符号表长度的符号定义，所有子过程均认为有返回值，不显式使用返回语句时默认返回值为 0；
- 对数组维度和常量的 initializer 可以使用数字，也可以使用已定义的常量；
- Statement 中去掉了赋值语句，赋值被作为表达式放入下一层 (Top_expr)，新添加 for 循环语句、程序退出 (exit) 语句、过程返回 (return) 语句、表达式 (Top_expr) 语句、输出 (print) 语句，修改原有的 if 语句、while 语句和 begin-end 语句；
- “return;” 返回 0 值；
- 由 “{” 包裹起来的 0 至多条语句被视为语句块，begin 等同于 “{” end 等同于 “}” 以向旧版本 p10 兼容，可用内含 0 条语句的 “{” 或 begin-end 代表空语句；
- 所有条件判断以非 0 为 true，0 为 false，所有表达式均有 1 个返回值（放在运行栈栈顶）；
- 赋值表达式（见 top_expr）支持右递归；
- 不进行数组越界检查，当下标 [...] 数量少于定义时维数时默认之后的下标为 0（等同于[0]），调用常量数组时只能以数字或常量作下标；
- 赋值时不再自动输出，需要显式使用 print 语句；

④ 亮点：

- ❖ 空语句不是 ‘;’ 而是 “{” 或 begin-end；
- ❖ 实现了常量数组，可用已定义常量代替数字作为 initializer；
- ❖ 以 “return;” 简化了 “return 0;”，以 “{” 简化 begin-end，简化掉 “then”；
- ❖ 添加了位运算符 ‘&’, ‘|’, ‘~’, ‘^’；

三. 具体实现

(1) 添加注释功能

参照老师的改法，在 getsym() 中读入除法符号时对下一个字符进行判断，如果是 ‘/’ 则跳过当前行，是 ‘*’ 则跳过后续符号直到按顺序连续读入 ‘*’ 和 ‘/’。

```

165     else if (ch == '/')
166     {
167         getch();
168         if (ch == '/') { cc = 11; getch(); getsym(); } // line comment
169         else if (ch == '*') { // block-comment
170             getch();
171             while (1) {
172                 if (ch == '*') {
173                     getch();
174                     if (ch == '/') break;
175                 }
176                 getch();
177             }
178             getch();
179             getsym();
180         }
181         else sym = SYM_SLASH;
182     }

```

(2) 拓展“条件”

添加新的运算符需要在 `pl0.h` 的 `enum symtype` 中加入新符号，然后将单字符运算符和其枚举类型对应加入 `csym` 和 `ssym`，多字符运算符则需要在 `getsym()` 中为其增加“else if”条目。在将不等“<>”替换为“!=”后，运算符 `&&`、`||`、`!` 都需要在 `getsym()` 中添加“else if”条目，而我新增了一些位运算符和括号。

```
int ssym[NSYM + 1] =
{
    SYM_NULL, SYM_PLUS, SYM_MINUS, SYM_TIMES, SYM_SLASH,
    SYM_LPAREN, SYM_RPAREN, SYM_EQU, SYM_COMMA, SYM_PERIOD, SYM_SEMICOLON,
    SYM_BITXOR, SYM_MOD, SYM_BITNOT, SYM_LBRKET, SYM_RBRKET, SYM_BEGIN, SYM_END
};

char csym[NSYM + 1] =
{
    ' ', '+', '-', '*', '/', '(', ')', '=', '!', ',', '.', ';', '^', '$', '~', '[', ']', '{', '}'
};
```

```
else if (ch == '!')
{
    getch();
    if (ch == '=')
    {
        sym = SYM_NEQ;    // !=
        getch();
    }
    else
    {
        sym = SYM_NOT;    // !
    }
}
else if (ch == '|') //2017-09-24
{
    getch();
    if (ch == '|') { sym = SYM_OR; getch(); }
    else
    {
        sym = SYM_BITOR;
    }
}
else if (ch == '&') //2017-09-24
{
    getch();
    if (ch == '&') { sym = SYM_AND; getch(); }
    else
    {
        sym = SYM_BITAND;
    }
}
else if (ch == '/')
```

进行条件判断时数据位于运行栈栈顶，将 `interpret` 语句中对 JPC 的解释修改为

```
case JPC:
    if (stack[top] == 0)
        pc += i.a-1;
    top--;
    break;
```

即可，修改后可将所有表达式（top_expr）作为条件。

条件作短路判断时，产生条件的是 Bitor_expr 及更下级的表达式，进行判断动作的是 And_expr 或 Or_expr，因此在判断时条件已位于栈顶。只需在 And_expr 和 Or_expr 中用局部变量记录待回填的跳转指令就行，其中 And_expr 中判断栈顶为 0 则跳，Or_expr 判断栈顶非 0 则跳。

```
900 void or_expr(symset fsys)
901 {
902     symset set;
903     cxlist cx0, p;
904     int cx1, flag = 0;
905
906     cx0 = p = (cxlist)malloc(sizeof(struct cxnode));
907     p->next = NULL;
908     set = uniteset(fsys, createset(SYM_OR, SYM_NULL));
909
910     and_expr(set);
911     while (sym == SYM_OR) //2017.10.24 , 2017.10.26
912     {
913         flag = 1;
914
915         gen(OPR, 0, 14); // stack[top]=!stack[top];
916         p = p->next = (cxlist)malloc(sizeof(struct cxnode));
917         p->next = NULL;
918         p->cx = cx;
919         gen(JPC, 0, 0); // short path
920
921         getsym();
922         and_expr(set);
923     } // while
924
925     if (flag)
926     {
927         gen(OPR, 0, OPR_NOT);
928         gen(OPR, 0, OPR_NOT);
929         cx1 = cx;
930         gen(JMP, 0, 0); //if did not JPC, avoid restoring
931         while (cx0->next)
932         {
933             p = cx0;
934             cx0 = cx0->next;
935             free(p);
936             code[cx0->cx].a = cx - (cx0->cx);
937         }
938         free(cx0);
939         gen(LIT, 0, 1); //if JPC, restore 1
940         code[cx1].a = cx - cx1;
941     }
942     destroyset(set);
943 } // or_expr
```

And_expr 的短路逻辑和上图的主要区别就是没有 915 行的取反操作。其中记录短路跳转指令号的结构体如下：

```
typedef struct cxnode{
    int cx;
    struct cxnode *next;
}*cxlist;
```


(3) 添加数组

使用宏定义 `#define MAXDIM 100` // maximum array dimensions 规定最大维度，在符号表中加入 `int *dim` 域使用整型数组存储维界信息，将符号表中记录常量数值的 `int value` 换成 `int *value` 数组。

```
190 typedef struct
191 {
192     char name[MAXIDLEN + 1];
193     int kind;
194     int *value;
195     short empty; //no use
196     int *dim; //dim of const array
197 } comtab;
198
199 comtab table[TXMAX];
200
201 typedef struct
202 {
203     char name[MAXIDLEN + 1];
204     int kind;
205     short level;
206     short address;
207     short prodn;
208     int *dim; //dim of var array
209 } mask;
```

在定义常量和变量时判断下一个符号，如果是 '[' 则调用 `arraylength()` 读入数组维度定义，计算维界与总空间大小。td 为全局变量。

```
int *td; //temporary dimensions
```

```
232 int arraylength()
233 { //calculate and return length of an array when declare
234     int i = 0, sum = 1, k = 0;
235     symset set, set1;
236
237     while(sym == SYM_LBRKET) { // '['
238         getsym();
239         if(k >= MAXDIM)
240         {
241             error(37); //Too many dimensions
242             set1 = createset(SYM_PLUS, SYM_MINUS, SYM_MOD, SYM_AND, SYM_BECOMES, SYM_EQU, SYM_NULL);
243             test(set1, facbegsys, 0);
244             destroyset(set1);
245             break;
246         }
247
248         if(sym == SYM_NUMBER || sym == SYM_IDENTIFIER) {
249             if(sym == SYM_IDENTIFIER)
250             {
251                 if ((i = position(id)) == 0)
252                 {
253                     error(11); // Undeclared identifier.
254                 }
255                 else if(table[i].kind == ID_CONSTANT)
256                 {
257                     num = constvalue(i);
258                 }
259             }
260         }
261     }
```

```

259     else
260     {
261         error(29); //There must be an number or const in dimension declaration
262         getsym();
263         if(sym == SYM_RBRKET) //'']'
264         {
265             getsym();
266         }
267         else
268         {
269             error(22); //Missing ')' or ']'
270         }
271         continue;
272     }
273 }
274
275 td[k++] = num;
276
277 getsym();
278 if(sym == SYM_RBRKET) //'']'
279 {
280     getsym();
281 }
282 else
283 {
284     error(22); //Missing ')' or ']'
285 }
286 }
287
288 else
289 {
290     error(29); //There must be an number or const in declaration
291
292     set1 = createset(SYM_LBRKET, SYM_NULL);
293     set = createset(SYM_IDENTIFIER, SYM_BEGIN, SYM_EQU, SYM_NULL);
294     test(set1, set, 0);
295     destroyset(set1);
296     destroyset(set);
297 }
298 while(k--)
299 {
300     i = td[k];
301     td[k] = sum;
302     //printf("td[%d] = %d\n", k, sum);
303     sum *= i;
304 }
305 return sum;
306 } //arraylength

```

变量数组定义时读完维度就可以加入符号表，常量数组读完维度还要读入以“{”包裹的初始化数据(initializers)存放于全局数组 value 中,对于多维数组仍然采用一维排列的 initializers,若 initializers 个数不足以填满常量数组，则以 0 填充，定义语法参考如下：

```

const K = 4;
const k[3][2][K] = {2, 3, K, 5, 6, 7, 8, 9, 10};

```

对 enter 也做了一定修改。

```

void enter(int kind)
{
    mask* mk;

    tx++;
    strcpy(table[tx].name, id);
    *id = '\0'; //reset to empty char array
    table[tx].kind = kind;
    switch (kind)

```

```

{
    case ID_CONSTANT:
        table[tx].value = value;
        value = NULL;
        table[tx].dim = td;
        td = (int*)malloc(MAXDIM*sizeof(int));
        break;
    case ID_VARIABLE:
        mk = (mask*)&table[tx];
        mk->level = level;
        mk->address = dx;
        mk->dim = td;
        td = (int*)malloc(MAXDIM*sizeof(int));
        break;
    case ID_PROCEDURE:
        mk = (mask*)&table[tx];
        mk->level = level;
        break;
} // switch
-} // enter

```

变量空间分配的 dx 在 vardeclaration()里按 arraylength()返回的数组大小修改。

(4) 参数传递

过程（即函数）定义时必须在 id 后面紧跟 “()”，内含 0 至多个以 ‘,’ 分隔的形参。先读入形参再进入子过程的 block。

```

1380         if( sym == SYM_PROCEDURE )
1381         {
1382             int k;
1383             char argumentID[TXMAX][MAXIDLEN+1]={0},proceID[MAXIDLEN+1]={0};
1384
1385             getsym();
1386
1387             if( sym == SYM_IDENTIFIER )
1388             {
1389                 enter(ID_PROCEDURE);
1390             }
1391             else
1392             {
1393                 error(4);
1394             }
1395
1396             savedTx = tx;    //critical
1397             level++;        //critical
1398
1399             getsym();
1400             if( sym == SYM_LPAREN )
1401                 getsym();
1402
1403             else error(16);
1404
1405             prodn = 0;
1406             if(sym == SYM_IDENTIFIER ){
1407                 do{
1408                     strcpy(argumentID[prodn++], id); //记录参数名
1409                     getsym();
1410                     if( sym == SYM_COMMA)
1411                         getsym();
1412
1413                 }while(sym == SYM_IDENTIFIER); //处理完所有参数
1414             }

```



```

1415 //printf("prodn = %d\n",prodn);
1416 k = prodn;
1417 while(k)
1418 {
1419     dx = -k; //b指向静态链,则实参dx为负数
1420     strcpy(id, argumentID[prodn-(k--)]);
1421     enter(ID_VARIABLE);
1422 }
1423
1424 if( sym == SYM_RPAREN )
1425     getsym();
1426 else error(22); //缺少右括号
1427
1428 set1 = createset(SYM_SEMICOLON, SYM_NULL);
1429 set = uniteset(set1, fsys);
1430 block(set);
1431 destroyset(set1);
1432 destroyset(set);
1433 tx = savedTx;
1434 level--;
1435
1436 if (sym == SYM_SEMICOLON)
1437 {
1438     getsym();
1439 }
1440 else
1441 {
1442     error(5); // Missing ',' or ';'.
1443 }
1444 } // if
1445 dx = block_dx; //restore dx after handling procedure declaration!

```

其中将实参的偏移地址依次置为负值，最终过程的空间分配（LIT）只需考虑局部变量。

将原有“call”关键字删除，以“id(...)”进行过程调用，如果实参与形参数量不匹配则报错。Factor()读到过程 id 则调用 proceCall()，先依次计算实参放在栈顶，然后压静态链、动态链和 pc，然后进入被调过程。原 CAL 指令被拆成 DIP 和 CAL 两条指令，为后续更复杂功能作准备。

```

995 int prodn;
996 ///////////////////////////////////////////////////
997 void proceCall(mask* mk,symset fsys){ // procedure call, critical
998     int j;
999     symset set,set1;
1000     //printf("proceID = %s, prodn = %d\n",mk->name,mk->prodn);
1001     set1 = createset(SYM_RPAREN,SYM_COMMA,SYM_NULL);
1002     set = uniteset(set1,fsys);
1003
1004     if( sym == SYM_LPAREN ){
1005         j = 0;
1006         do{
1007             getsym();
1008
1009             if(sym == SYM_RPAREN)
1010             {
1011                 break;
1012             }
1013             top_expr(set);
1014             j++;
1015
1016         }while(sym == SYM_COMMA && j <= mk->prodn);
1017         if(sym == SYM_RPAREN && j <= mk->prodn)
1018         {
1019             getsym();
1020         }
1021         else
1022         {
1023             error(22); //missing ')'
1024         }
1025     }

```



```

1026         else
1027             error(16); //'(' expected
1028
1029
1030         destroyset(set);
1031         destroyset(set1);
1032         if(j != mk->prodn)
1033         {
1034             error(34); //"The number of actual parameters and virtual paramet
1035             if(j > mk->prodn){
1036                 set = createset(SYM_RPAREN,SYM_NULL);
1037                 set1 = createset(SYM_RPAREN,SYM_SEMICOLON,SYM_END,SYM_NULL);
1038                 test(set,set1,0);
1039                 destroyset(set);
1040                 destroyset(set1);
1041             }
1042         }
1043
1044         gen(DIP,level-mk->level,mk->prodn);
1045         gen(CAL, 0, mk->address);
1046     } //proccall

```

(5) exit、return、for 语句和 else 子句

“exit;”产生指令“EXT 00”将 pc 置零并输出 `printf("Exit Program");`。

“return”先把返回值放栈顶，然后利用记录当前所在过程形参的全局变量 `prodn`，将栈顶元素放在第一个实参的位置然后调整 `top`、`b` 和 `pc`。

Statement 中：

```

1262     else if (sym == SYM_RET) {
1263         getsym();
1264         if (sym == SYM_SEMICOLON) { //return; 则把0放在被调用的栈顶，然后再放到原栈栈顶
1265             gen(LIT, 0, 0);
1266             gen(OPR, prodn, OPR_RET);
1267             getsym();
1268         }
1269         else { //return 1;return 1+x;return fact(n-1);
1270             top_expr(fsyz); //调用后的结果存在栈顶
1271             gen(OPR, prodn, OPR_RET);
1272             if (sym != SYM_SEMICOLON)
1273                 error(10); // missing ';'.
1274             else
1275                 getsym();
1276         }
1277     }

```

Interpret 中：

```

case OPR_RET: //put the return value in stack[b-i.l],since the argument isn't useful anymore.
    pc = stack[b+2];
    stack[b-i.l] = stack[top];
    //printf("ret %d from [%d] to [%d]\n",stack[top],top,b-i.l);
    top = b-i.l;
    b = stack[top + i.l + 1];
    break;

```

else 子句很简单，直接在处理 if 时最后回填条件转移指令（JPC）时判断下一个符号就行。参考 c 语言的语法，我没有单独做 elif，使用 else if 可以达到相同效果。

```
if (sym == SYM_ELSE) //2017.10.22
{
    gen(JMP, 0, 0);
    code[cx1].a = cx-cx1; //code[cx1] = JPC, 0, cx-cx1
    cx1 = cx - 1; //cx1 = JMP,0,0
    getsym();
    statement(fsys);
    code[cx1].a = cx-cx1;
}
else
{
    code[cx1].a = cx-cx1; //cx1 = JPC,0,0
}
```

For 语句处理时用临时空间将 (top_expr ; top_expr ; top_expr) 中第三个 top_expr 产生的语句保存，最后处理完循环体内的 statement 后再将保存的语句接在其后就行。实现 For 主要的工作量是修改程序整体的语法结构，将赋值语句改成赋值表达式，以及将 JMP 和 JPC 都改成相对地址跳转。

```
1143     else if (sym == SYM_FOR) //2017.10.25
1144     { // for statement
1145         int i;
1146
1147         getsym();
1148         if( sym != SYM_LPAREN)
1149         {
1150             error(16); // '(' expected
1151         }
1152         else
1153         {
1154             getsym();
1155             set1 = createset(SYM_SEMICOLON, SYM_RPAREN, SYM_NULL);
1156             set = uniteset(set1, fsys);
1157             top_expr(set);
1158             if(sym != SYM_SEMICOLON)
1159             {
1160                 error(10); // ';' expected
1161                 test(fsys,set1,28); //Incomplete 'for' statement.
1162             }
1163             else
1164             {
1165                 getsym();
1166                 gen(POP, 0, 0);
1167                 cx1 = cx; //come back here after loop
1168                 top_expr(set);
1169                 if(sym != SYM_SEMICOLON)
1170                 {
1171                     error(10); // ';' expected
1172                     test(fsys,set1,28); //Incomplete 'for' statement.
1173                 }
1174             }
1175             else
1176             {
1177                 getsym();
1178
1179                 cx2 = cx; //if false, skip loop
1180                 gen(JPC, 0, 0);
1181
1182                 cx3 = cx; //beginning of codes that need move
1183                 top_expr(set);
```

```

1183         if(sym == SYM_RPAREN)
1184         {
1185             getsym();
1186         }
1187         else
1188         {
1189             error(22);
1190         }
1191         destroyset(set);
1192         destroyset(set1);
1193         gen(POP, 0, 0);
1194
1195         instruction temp[CXMAX];
1196         for(i = cx3; i<cx; i++)
1197         {
1198             temp[i-cx3]=code[i];
1199         }
1200         cx = cx3;
1201         cx3 = i-cx3; //the length of temp
1202         statement(fsys);
1203         for(i = 0; i<cx3; i++)
1204         {
1205             code[cx++]=temp[i];
1206         }
1207         cx1 = cx1-cx; //offset
1208         gen(JMP, 0, cx1); //come back to condition
1209         code[cx2].a = cx-cx2; //destination of false condition
1210     }
1211 }
1212 }
1213 } // sym == SYM_FOR

```

(6) 其他

print(top_expr ,top_expr , ... , top_expr)将其中表达式的值按序带间隔地输出，若为空括号则换行。
加入的位运算符优先级与 c 语言相同，暂未设计自增（++）自减（--）运算符。

四．测试样例与运行结果

```

var i,j;
const K = 2, k[3][4][K]={2,3,4,K,6,7,8,9,10};

```

```

procedure add(a,b)
{
    return a+b;
};

```

```

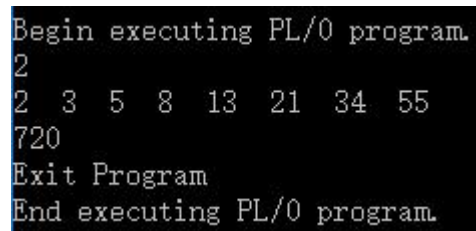
procedure fibo()
var a[10];
{
    a[0]:=a[1]:=1;
    for(i:=2;i<10;i:=i+1)
        print(a[i]:=add(a[i-1],a[i-2]));
};

```

```
procedure factorial(n)
{
    if(n=1)
        return 1;
    else
        return n*factorial(n-1);
};

{
    print(k[0][1][1]);
    print();
    fibo();
    print();
    print(factorial(k[0][K]));
    print();
    exit;
    print(9);
}.
```

运行结果截图：



```
Begin executing PL/0 program
2
2 3 5 8 13 21 34 55
720
Exit Program
End executing PL/0 program
```

The screenshot shows the output of a PL/0 program. It starts with 'Begin executing PL/0 program', followed by the number '2' on a new line. Then, a sequence of Fibonacci numbers is printed: '2 3 5 8 13 21 34 55'. This is followed by the factorial of 6, '720'. The program then prints 'Exit Program' and finally 'End executing PL/0 program'.