厦門大學


# 软件学院

## 《编译技术》大作业报告

题　　目　PL/0 编译器的设计与实现

姓　　名　陈澄

学　　号　32420212202930

班　　级　软工三班

实验时间　2024/6/6

**2024　年　6 月 6 日**

# 1 实验要求

设计并实现一个 PL/0 语言的编译器，能够将 PL/0 语言翻译成 P-code 语言（具体语言描述见《编译原理》（第 3 版），清华大学出版社，王生原等编著）。

# 2 实验步骤

1.编写头文件 pl0.h。

定义关键字个数，名字表容量，number 的最大位数，符号的最大长度，地址上界，最大允许嵌套声明层数，最多的虚拟机代码数。

```
#define norw 13     /* 关键字个数 */
#define txmax 100   /* 名字表容量 */
#define nmax 14     /* number的最大位数 */
#define al 10       /* 符号的最大长度 */
#define amax 2047   /* 地址上界*/
#define levmax 3    /* 最大允许过程嵌套声明层数 [0,  levmax]*/
#define cxmax 500   /* 最多的虚拟机代码数 */
```

定义所有可以被识别的符号

```
/* 符号 */
enum symbol {
    nul,        ident,    number,   plus,     minus,
    times,      slash,    oddsym,   eql,      neq,
    lss,        leq,      gtr,      geq,      lparen,
    rparen,     comma,    semicolon, period,  becomes,
    beginsym,   endsym,   ifsym,    thensym,  whilesym,
    writesym,   readsym,  dosym,    callsym,  constsym,
    varsym,     procsym,
};
#define symnum 32
```

定义名字表中的类型

```
/* 名字表中的类型 */
enum object {
    constant,
    variable,
    procedur,
    array       //add
};
```

定义虚拟机中的代码

```
/* 虚拟机代码 */
enum fct {
    lit,     opr,     lod,
    sto,     cal,     inte,
    jmp,     jpc,
};
#define fctnum 8
```

定义其他需要的全局变量

```
FILE* fas;  /* 输出名字表 */
FILE* fa;   /* 输出虚拟机代码 */
FILE* fa1;  /* 输出源文件及其各行对应的首地址 */
FILE* fa2;  /* 输出结果 */
bool listswitch;     /* 显示虚拟机代码与否 */
bool tableswitch;    /* 显示名字表与否 */
char ch;             /* 获取字符的缓冲区，getch 使用 */
enum symbol sym;     /* 当前的符号 */
char id[al+1];       /* 当前ident，多出的一个字节用于存放0 */
int num;             /* 当前number */
int cc, ll;          /* getch使用的计数器，cc表示当前字符(ch)的位置 */
int cx;              /* 虚拟机代码指针，取值范围[0，cxmax-1]*/
char line[81];       /* 读取行缓冲区 */
char a[al+1];        /* 临时符号，多出的一个字节用于存放0 */
struct instruction code[cxmax]; /* 存放虚拟机代码的数组 */
char word[norw][al];         /* 保留字 */
enum symbol wsym[norw];      /* 保留字对应的符号值 */
enum symbol ssym[256];       /* 单字符的符号值 */
char mnemonic[fctnum][5];    /* 虚拟机代码指令名称 */
bool declbegsys[symnum];     /* 表示声明开始的符号集合 */
bool statbegsys[symnum];     /* 表示语句开始的符号集合 */
bool facbegsys[symnum];      /* 表示因子开始的符号集合 */
```

```
FILE* fin;
FILE* fout;
char fname[al];
int err; /* 错误计数器 */
```

定义名字表的结构和名字表

```
/* 名字表结构 */
struct tablestruct
{
    char name[al];      /* 名字 */
    enum object kind;   /* 类型: const, var, array or procedure */
    int val;            /* 数值, 仅const使用 */
    int level;          /* 所处层, 仅const不使用 */
    int adr;            /* 地址, 仅const不使用 */
    int size;           /* 需要分配的数据区空间, 仅procedure使用 */
};

struct tablestruct table[txmax]; /* 名字表 */
```

定义部分函数的异常退出

```
/* 当函数中会发生fatal error时, 返回-1告知调用它的函数, 最终退出程序 */
#define getsymdo                      if(-1 == getsym()) return -1
#define getchdo                       if(-1 == getch()) return -1
#define testdo(a, b, c)               if(-1 == test(a, b, c)) return -1
#define gendo(a, b, c)                if(-1 == gen(a, b, c)) return -1
#define expressiondo(a, b, c)         if(-1 == expression(a, b, c)) return -1
#define factordo(a, b, c)             if(-1 == factor(a, b, c)) return -1
#define termdo(a, b, c)               if(-1 == term(a, b, c)) return -1
#define conditiondo(a, b, c)          if(-1 == condition(a, b, c)) return -1
#define statementdo(a, b, c)          if(-1 == statement(a, b, c)) return -1
#define constdeclarationdo(a, b, c)   if(-1 == constdeclaration(a, b, c)) return -1
#define vardeclarationdo(a, b, c)     if(-1 == vardeclaration(a, b, c)) return -1
```

定义所有需要编写的函数接口

```
void error(int n);
int getsym();
int getch();
void init();
int gen(enum fct x, int y, int z);
int test(bool* s1, bool* s2, int n);
int inset(int e, bool* s);
int addset(bool* sr, bool* s1, bool* s2, int n);
int subset(bool* sr, bool* s1, bool* s2, int n);
int mulset(bool* sr, bool* s1, bool* s2, int n);
int block(int lev, int tx, bool* fsys);
void interpret();
int factor(bool* fsys, int* ptx, int lev);
int term(bool* fsys, int* ptx, int lev);
int condition(bool* fsys, int* ptx, int lev);
int expression(bool* fsys, int* ptx, int lev);
int statement(bool* fsys, int* ptx, int lev);
void listcode(int cx0);
int vardeclaration(int* ptx, int lev, int* pdx);
int constdeclaration(int* ptx, int lev, int* pdx);
int position(char* idt, int tx);
void enter(enum object k, int* ptx, int lev, int* pdx);
int base(int l, int* s, int b);
```

2.编写主函数

```
int main()
{
    bool nxtlev[symnum];

    printf("Input pl/0 file?   ");
    scanf("%s", fname);     /* 输入文件名 */

    fin = fopen(fname, "r");

    if (fin)
    {
        printf("List object code?(Y/N)");   /* 是否输出虚拟机代码 */
        scanf("%s", fname);
        listswitch = (fname[0]=='y' || fname[0]=='Y');

        printf("List symbol table?(Y/N)");  /* 是否输出名字表 */
        scanf("%s", fname);
        tableswitch = (fname[0]=='y' || fname[0]=='Y');

        fa1 = fopen("fa1.tmp", "w");
        fprintf(fa1,"Input pl/0 file?   ");
        fprintf(fa1,"%s\n",fname);

        init();     /* 初始化 */

        err = 0;
        cc = cx = ll = 0;
        ch = ' ';

        if(-1 != getsym())
        {
            fa = fopen("fa.tmp", "w");
            fas = fopen("fas.tmp", "w");
            addset(nxtlev, declbegsys, statbegsys, symnum);
            nxtlev[period] = true;

            if(-1 == block(0, 0, nxtlev))   /* 调用编译程序 */
            {
                fclose(fa);
                fclose(fa1);
                fclose(fas);
                fclose(fin);
                printf("\n");
                return 0;
            }
            fclose(fa);
            fclose(fa1);
            fclose(fas);

            if (sym != period)
            {
                error(9);
            }

            if (err == 0)
            {
                fa2 = fopen("fa2.tmp", "w");
                interpret();    /* 调用解释执行程序 */
                fclose(fa2);
            }
            else
            {
                printf("Errors in pl/0 program");
            }
        }

        fclose(fin);
    }
    else
    {
        printf("Can't open file!\n");
    }

    printf("\n");
    return 0;
}
```

3.初始化符号集合、保留字和虚拟机指令集

```
void init()
{
    int i;

    /* 设置单字符符号 */
    for (i=0; i≤255; i++)
    {
        ssym[i] = nul;
    }
    ssym['+'] = plus;
    ssym['-'] = minus;
    ssym['*'] = times;
    ssym['/'] = slash;
    ssym['('] = lparen;
    ssym[')'] = rparen;
    ssym['='] = eql;
    ssym[','] = comma;
    ssym['.'] = period;
    ssym['#'] = neq;
    ssym[';'] = semicolon;

    /* 设置保留字名字,按照字母顺序，便于折半查找 */
    strcpy(&(word[0][0]), "begin");
    strcpy(&(word[1][0]), "call");
    strcpy(&(word[2][0]), "const");
    strcpy(&(word[3][0]), "do");
    strcpy(&(word[4][0]), "end");
    strcpy(&(word[5][0]), "if");
    strcpy(&(word[6][0]), "odd");
    strcpy(&(word[7][0]), "procedure");
    strcpy(&(word[8][0]), "read");
    strcpy(&(word[9][0]), "then");
    strcpy(&(word[10][0]), "var");
    strcpy(&(word[11][0]), "while");
    strcpy(&(word[12][0]), "write");

    /* 设置保留字符号 */
    wsym[0] = beginsym;
    wsym[1] = callsym;
    wsym[2] = constsym;
    wsym[3] = dosym;
    wsym[4] = endsym;
    wsym[5] = ifsym;
    wsym[6] = oddsym;
    wsym[7] = procsym;
    wsym[8] = readsym;
    wsym[9] = thensym;
    wsym[10] = varsym;
    wsym[11] = whilesym;
    wsym[12] = writesym;

    /* 设置指令名称 */
    strcpy(&(mnemonic[lit][0]), "lit");
    strcpy(&(mnemonic[opr][0]), "opr");
    strcpy(&(mnemonic[lod][0]), "lod");
    strcpy(&(mnemonic[sto][0]), "sto");
    strcpy(&(mnemonic[cal][0]), "cal");
    strcpy(&(mnemonic[inte][0]), "int");
    strcpy(&(mnemonic[jmp][0]), "jmp");
    strcpy(&(mnemonic[jpc][0]), "jpc");

    /* 设置符号集 */
    for (i=0; i<symnum; i++)
    {
        declbegsys[i] = false;
        statbegsys[i] = false;
        facbegsys[i] = false;
    }

    /* 设置声明开始符号集 */
    declbegsys[constsym] = true;
    declbegsys[varsym] = true;
    declbegsys[procsym] = true;

    /* 设置语句开始符号集 */
    statbegsys[beginsym] = true;
    statbegsys[callsym] = true;
    statbegsys[ifsym] = true;
    statbegsys[whilesym] = true;
    statbegsys[readsym] = true;
    statbegsys[writesym] = true;

    /* 设置因子开始符号集 */
    facbegsys[ident] = true;
    facbegsys[number] = true;
    facbegsys[lparen] = true;
}
```

4.编写用数组实现的集合运算函数

```c
/*
* 用数组实现集合的集合运算
*/
int inset(int e, bool* s)
{
    return s[e];
}

int addset(bool* sr, bool* s1, bool* s2, int n)
{
    int i;
    for (i=0; i<n; i++)
    {
        sr[i] = s1[i]||s2[i];
    }
    return 0;
}

int subset(bool* sr, bool* s1, bool* s2, int n)
{
    int i;
    for (i=0; i<n; i++)
    {
        sr[i] = s1[i]&&(!s2[i]);
    }
    return 0;
}

int mulset(bool* sr, bool* s1, bool* s2, int n)
{
    int i;
    for (i=0; i<n; i++)
    {
        sr[i] = s1[i]&&s2[i];
    }
    return 0;
}
```

5.编写出错处理函数

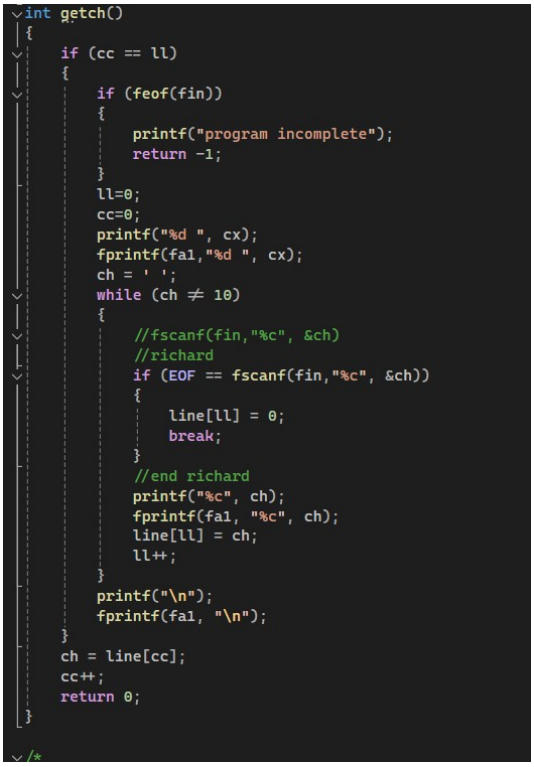
```c
void error(int n)
{
    char space[81];
    memset(space,32,81);

    space[cc-1]=0; //出错时当前符号已经读完，所以cc-1

    printf("****%s!%d\n", space, n);
    fprintf(fa1,"****%s!%d\n", space, n);

    err++;
}
```

6.编写字符读取函数，无视空格读取一个字符



```
int getch()
{
    if (cc == ll)
    {
        if (feof(fin))
        {
            printf("program incomplete");
            return -1;
        }
        ll=0;
        cc=0;
        printf("%d ", cx);
        fprintf(fa1,"%d ", cx);
        ch = ' ';
        while (ch ≠ 10)
        {
            //fscanf(fin,"%c", &ch)
            //richard
            if (EOF == fscanf(fin,"%c", &ch))
            {
                line[ll] = 0;
                break;
            }
            //end richard
            printf("%c", ch);
            fprintf(fa1, "%c", ch);
            line[ll] = ch;
            ll++;
        }
        printf("\n");
        fprintf(fa1, "\n");
    }
    ch = line[cc];
    cc++;
    return 0;
}

/*
```

7.编写 token 获取函数

```
int getsym()
{
    int i,j,k;

    /* the original version lacks "\r", thanks to foolevery */
    while (ch==' ' || ch==10 || ch==13 || ch==9)  /* 忽略空格、换行、回车和TAB */
    {
        getchdo;
    }
    if (ch≥'a' && ch≤'z')
    {               /* 名字或保留字以a..z开头 */
        k = 0;
        do {
            if(k<al)
            {
                a[k] = ch;
                k++;
            }
            getchdo;
        } while (ch≥'a' && ch≤'z' || ch≥'0' && ch≤'9');
        a[k] = 0;
        strcpy(id, a);
        i = 0;
        j = norw-1;
        do {    /* 搜索当前符号是否为保留字 */
            k = (i+j)/2;
            if (strcmp(id,word[k]) ≤ 0)
            {
                j = k - 1;
            }
            if (strcmp(id,word[k]) ≥ 0)
            {
                i = k + 1;
            }
        } while (i ≤ j);
        if (i-1 > j)
        {
            sym = wsym[k];
        }
        else
        {
            sym = ident; /* 搜索失败则，是名字或数字 */
        }
    }
    else
    {
        if (ch≥'0' && ch≤'9')
        {           /* 检测是否为数字：以0..9开头 */
            k = 0;
            num = 0;
            sym = number;
            do {
                num = 10*num + ch - '0';
                k++;
                getchdo;
            } while (ch≥'0' && ch≤'9'); /* 获取数字的值 */
            k--;
            if (k > nmax)
            {
                error(30);
            }
        }
        else
        {
            if (ch == ':')      /* 检测赋值符号 */
            {
                getchdo;
                if (ch == '=')
                {
                    sym = becomes;
                    getchdo;
                }
                else
                {
                    sym = nul;  /* 不能识别的符号 */
                }
            }
            else
            {
                if (ch == '<')      /* 检测小于或小于等于符号 */
                {
                    getchdo;
                    if (ch == '=')
                    {
                        sym = leq;
                        getchdo;
                    }
                    else
                    {
                        sym = lss;
                    }
                }
                else
                {
                    if (ch=='>')        /* 检测大于或大于等于符号 */
                    {
                        getchdo;
                        if (ch == '=')
                        {
                            sym = geq;
                            getchdo;
                        }
                        else
                        {
                            sym = gtr;
                        }
                    }
                    else
                    {
                        sym = ssym[ch];     /* 当符号不满足上述条件时，全部按照单字符符号处
                        //getchdo;
                        //richard
                        if (sym ≠ period)
                        {
                            getchdo;
                        }
                        //end richard
                    }
                }
            }
        }
    }
    return 0;
}
```

8.编写代码生成函数

```
int gen(enum fct x, int y, int z )
{
    if (cx ≥ cxmax)
    {
        printf("Program too long"); /* 程序过长 */
        return -1;
    }
    code[cx].f = x;
    code[cx].l = y;
    code[cx].a = z;
    cx++;
    return 0;
}
```

9.编写测试函数，测试当前符号是否合法

```
int test(bool* s1, bool* s2, int n)
{
    if (!inset(sym, s1))
    {
        error(n);
        /* 当检测不通过时，不停获取符号，直到它属于需要的集合或补救的集合 */
        while ((!inset(sym,s1)) && (!inset(sym,s2)))
        {
            getsymdo;
        }
    }
    return 0;
}
```

10.编写编译主程序 bolck()

```
int block(int lev, int tx, bool* fsys)
{
    int i;

    int dx;                 /* 名字分配到的相对地址 */
    int tx0;                /* 保留初始tx */
    int cx0;                /* 保留初始cx */
    bool nxtlev[symnum];    /* 在下级函数的参数中，符号集合均为值参，但由于使用数组实现，
                               传递进来的是指针，为防止下级函数改变上级函数的集合，开辟新的空?
                               传递给下级函数*/

    dx = 3;
    tx0 = tx;               /* 记录本层名字的初始位置 */
    table[tx].adr = cx;

    gendo(jmp, 0, 0);

    if (lev > levmax)
    {
        error(32);
    }
```

```
do {
    if (sym == constsym)    /* 收到常量声明符号，开始处理常量声明 */
    {
        getsymdo;

        /* the original do ... while(sym == ident) is problematic, thanks to calculous */
        /* do { */
        constdeclarationdo(&tx, lev, &dx);  /* dx的值会被constdeclaration改变，使用指针 */
        while (sym == comma)
        {
            getsymdo;
            constdeclarationdo(&tx, lev, &dx);
        }
        if (sym == semicolon)
        {
            getsymdo;
        }
        else
        {
            error(5);   /*漏掉了逗号或者分号*/
        }
        /* } while (sym == ident); */
    }
```

```
if (sym == varsym)      /* 收到变量声明符号，开始处理变量声明 */
{
    getsymdo;

    /* the original do ... while(sym == ident) is problematic, thanks to calculous */
    /* do {  */
    vardeclarationdo(&tx, lev, &dx);
    while (sym == comma)
    {
        getsymdo;
        vardeclarationdo(&tx, lev, &dx);
    }
    if (sym == semicolon)
    {
        getsymdo;
    }
    else
    {
        error(5);
    }
    /* } while (sym == ident);  */
}
```

```
while (sym == procsym) /* 收到过程声明符号，开始处理过程声明 */
{
    getsymdo;

    if (sym == ident)
    {
        enter(procedur, &tx, lev, &dx); /* 记录过程名字 */
        getsymdo;
    }
    else
    {
        error(4);   /* procedure后应为标识符 */
    }

    if (sym == semicolon)
    {
        getsymdo;
    }
    else
    {
        error(5);   /* 漏掉了分号 */
    }

    memcpy(nxtlev, fsys, sizeof(bool)*symnum);
    nxtlev[semicolon] = true;
    if (-1 == block(lev+1, tx, nxtlev))
    {
        return -1;  /* 递归调用 */
    }

    if(sym == semicolon)
    {
        getsymdo;
        memcpy(nxtlev, statbegsys, sizeof(bool)*symnum);
        nxtlev[ident] = true;
        nxtlev[procsym] = true;
        testdo(nxtlev, fsys, 6);
    }
    else
    {
        error(5);   /* 漏掉了分号 */
    }
}
memcpy(nxtlev, statbegsys, sizeof(bool)*symnum);
```

```
    memcpy(nxtlev, statbegsys, sizeof(bool)*symnum);
    nxtlev[ident] = true;
    testdo(nxtlev, declbegsys, 7);
} while (inset(sym, declbegsys));   /* 直到没有声明符号 */

code[table[tx0].adr].a = cx;    /* 开始生成当前过程代码 */
table[tx0].adr = cx;            /* 当前过程代码地址 */
table[tx0].size = dx;           /* 声明部分中每增加一条声明都会给dx增加1，声明部分已经结束，dx就是当前过程数据的size */
cx0 = cx;
gendo(inte, 0, dx);             /* 生成分配内存代码 */
```

```
    if (tableswitch)        /* 输出名字表 */
    {
        printf("TABLE:\n");
        if (tx0+1 > tx)
        {
            printf("    NULL\n");
        }
        for (i=tx0+1; i≤tx; i++)
        {
            switch (table[i].kind)
            {
            case constant:
                printf("    %d const %s ", i, table[i].name);
                printf("val=%d\n", table[i].val);
                fprintf(fas, "    %d const %s ", i, table[i].name);
                fprintf(fas, "val=%d\n", table[i].val);
                break;
            case variable:
                printf("    %d var   %s ", i, table[i].name);
                printf("lev=%d addr=%d\n", table[i].level, table[i].adr);
                fprintf(fas, "    %d var   %s ", i, table[i].name);
                fprintf(fas, "lev=%d addr=%d\n", table[i].level, table[i].adr);
                break;
            case procedur:
                printf("    %d proc  %s ", i, table[i].name);
                printf("lev=%d addr=%d size=%d\n", table[i].level, table[i].adr, table[i].size);
                fprintf(fas,"    %d proc  %s ", i, table[i].name);
                fprintf(fas,"lev=%d addr=%d size=%d\n", table[i].level, table[i].adr, table[i].size);
                break;
            }
        }
        printf("\n");
    }

    /* 语句后跟符号为分号或end */
    memcpy(nxtlev, fsys, sizeof(bool)*symnum);  /* 每个后跟符号集和都包含上层后跟符号集和，以便补救 */
    nxtlev[semicolon] = true;
    nxtlev[endsym] = true;
    statementdo(nxtlev, &tx, lev);
    gendo(opr, 0, 0);                           /* 每个过程出口都要使用的释放数据段指令 */
    memset(nxtlev, 0, sizeof(bool)*symnum); /*分程序没有补救集合 */
    testdo(fsys, nxtlev, 8);                    /* 检测后跟符号正确性 */
    listcode(0);                               /* 输出代码 */
    return 0;
}

/*
```

11.编写 enter()函数，在名字表中加入一项

```
void enter(enum object k, int* ptx, int lev, int* pdx)
{
    (*ptx)++;
    strcpy(table[(*ptx)].name, id); /* 全局变量id中已存有当前名字的名字 */
    table[(*ptx)].kind = k;
    switch (k)
    {
    case constant:  /* 常量名字 */
        if (num > amax)
        {
            error(31);  /* 数越界 */
            num = 0;
        }
        table[(*ptx)].val = num;
        break;
    case variable:  /* 变量名字 */
        table[(*ptx)].level = lev;
        table[(*ptx)].adr = (*pdx);
        (*pdx)++;
        break;
    case procedur:  /*  过程名字  */
        table[(*ptx)].level = lev;
        break;
    }
}
```

12.编写查找函数，查找在名字表中的位置

```
int position(char* idt, int tx)
{
    int i;
    strcpy(table[0].name, idt);
    i = tx;
    while (strcmp(table[i].name, idt) ≠ 0)
    {
        i--;
    }
    return i;
}
```

13.编写常量声明处理函数

```
int constdeclaration(int* ptx, int lev, int* pdx)
{
    if (sym == ident)
    {
        getsymdo;
        if (sym==eql || sym==becomes)
        {
            if (sym == becomes)
            {
                error(1);   /* 把=写成了:= */
            }
            getsymdo;
            if (sym == number)
            {
                enter(constant, ptx, lev, pdx);
                getsymdo;
            }
            else
            {
                error(2);   /* 常量说明=后应是数字 */
            }
        }
        else
        {
            error(3);   /* 常量说明标识后应是= */
        }
    }
    else
    {
        error(4);   /* const后应是标识 */
    }
    return 0;
}
```

14.编写变量声明处理函数

```
int vardeclaration(int* ptx,int lev,int* pdx)
{
    if (sym == ident)
    {
        enter(variable, ptx, lev, pdx); // 填写名字表
        getsymdo;
    }
    else
    {
        error(4);   /* var后应是标识 */
    }
    return 0;
}
```

15.编写目标代码输出程序

```
void listcode(int cx0)
{
    int i;
    if (listswitch)
    {
        for (i=cx0; i<cx; i++)
        {
            printf("%d %s %d %d\n", i, mnemonic[code[i].f], code[i].l, code[i].a);
            fprintf(fa,"%d %s %d %d\n", i, mnemonic[code[i].f], code[i].l, code[i].a);
        }
    }
}
```

16.编写语句处理函数

```
int statement(bool* fsys, int* ptx, int lev)
{
    int i, cx1, cx2;
    bool nxtlev[symnum];

    if (sym == ident)   /* 准备按照赋值语句处理 */
    {
        i = position(id, *ptx);
        if (i == 0)
        {
            error(11);  /* 变量未找到 */
        }
        else
        {
            if(table[i].kind ≠ variable)
            {
                error(12);  /* 赋值语句格式错误 */
                i = 0;
            }
            else
            {
                getsymdo;
                if(sym == becomes)
                {
                    getsymdo;
                }
                else
                {
                    error(13);  /* 没有检测到赋值符号 */
                }
                memcpy(nxtlev, fsys, sizeof(bool)*symnum);
                expressiondo(nxtlev, ptx, lev); /* 处理赋值符号右侧表达式 */
                if(i ≠ 0)
                {
                    /* expression将执行一系列指令，但最终结果将会保存在栈顶，执行sto命令
                    gendo(sto, lev-table[i].level, table[i].adr);
                }
            }
        }//if (i == 0)
    }
    else
```

```
else
{
    if (sym == readsym) /* 准备按照read语句处理 */
    {
        getsymdo;
        if (sym ≠ lparen)
        {
            error(34);  /* 格式错误，应是左括号 */
        }
        else
        {
            do {
                getsymdo;
                if (sym == ident)
                {
                    i = position(id, *ptx); /* 查找要读的变量 */
                }
                else
                {
                    i=0;
                }

                if (i == 0)
                {
                    error(35);  /* read()中应是声明过的变量名 */
                }
                else if (table[i].kind ≠ variable)
                {
                    error(32);  /* read()参数表的标识符不是变量，thanks to amd */
                }
                else
                {
                    gendo(opr, 0, 16);  /* 生成输入指令，读取值到栈顶 */
                    gendo(sto, lev-table[i].level, table[i].adr);   /* 储存到变量 */
                }
                getsymdo;

            } while (sym == comma); /* 一条read语句可读多个变量 */
```

```
    else
    {
        if (sym == writesym)    /* 准备按照write语句处理，与read类似 */
        {
            getsymdo;
            if (sym == lparen)
            {
                do {
                    getsymdo;
                    memcpy(nxtlev, fsys, sizeof(bool)*symnum);
                    nxtlev[rparen] = true;
                    nxtlev[comma] = true;       /* write的后跟符号为) or , */
                    expressiondo(nxtlev, ptx, lev); /* 调用表达式处理，此处与read不同，read为给变量赋值 */
                    gendo(opr, 0, 14);  /* 生成输出指令，输出栈顶的值 */
                } while (sym == comma);
                if (sym ≠ rparen)
                {
                    error(33);  /* write()中应为完整表达式 */
                }
                else
                {
                    getsymdo;
                }
            }
            gendo(opr, 0, 15);  /* 输出换行 */
        }
        else
```

```
if (sym == writesym)    /* 准备按照write语句处理，与read类似 */
{
    getsymdo;
    if (sym == lparen)
    {
        do {
            getsymdo;
            memcpy(nxtlev, fsys, sizeof(bool)*symnum);
            nxtlev[rparen] = true;
            nxtlev[comma] = true;       /* write的后跟符号为) or , */
            expressiondo(nxtlev, ptx, lev); /* 调用表达式处理，此处与read不同，read为给变量赋值 */
            gendo(opr, 0, 14);  /* 生成输出指令，输出栈顶的值 */
        } while (sym == comma);
        if (sym ≠ rparen)
        {
            error(33);  /* write()中应为完整表达式 */
        }
        else
        {
            getsymdo;
        }
    }
    gendo(opr, 0, 15);  /* 输出换行 */
}
```

```
else
{
    if (sym == callsym) /* 准备按照call语句处理 */
    {
        getsymdo;
        if (sym ≠ ident)
        {
            error(14);  /* call后应为标识符 */
        }
        else
        {
            i = position(id, *ptx);
            if (i == 0)
            {
                error(11);  /* 过程未找到 */
            }
            else
            {
                if (table[i].kind == procedur)
                {
                    gendo(cal, lev-table[i].level, table[i].adr);   /* 生成call指令 */
                }
                else
                {
                    error(15);  /* call后标识符应为过程 */
                }
            }
            getsymdo;
        }
    }
    else
    {
        if (sym == ifsym)   /* 准备按照if语句处理 */
        {
            getsymdo;
            memcpy(nxtlev, fsys, sizeof(bool)*symnum);
            nxtlev[thensym] = true;
            nxtlev[dosym] = true;   /* 后跟符号为then或do */
            conditiondo(nxtlev, ptx, lev); /* 调用条件处理（逻辑运算）函数 */
            if (sym == thensym)
            {
                getsymdo;
            }
            else
            {
                error(16);  /* 缺少then */
            }
            cx1 = cx;   /* 保存当前指令地址 */
            gendo(jpc, 0, 0);   /* 生成条件跳转指令，跳转地址未知，暂时写0 */
            statementdo(fsys, ptx, lev);    /* 处理then后的语句 */
            code[cx1].a = cx;   /* 经statement处理后，cx为then后语句执行完的位置，它正是前面未定的跳
        }
        else
        {
            if (sym == beginsym)    /* 准备按照复合语句处理 */
            {
                getsymdo;
                memcpy(nxtlev, fsys, sizeof(bool)*symnum);
                nxtlev[semicolon] = true;
                nxtlev[endsym] = true;  /* 后跟符号为分号或end */
                /* 循环调用语句处理函数，直到下一个符号不是语句开始符号或收到end */
                statementdo(nxtlev, ptx, lev);

                while (inset(sym, statbegsys) || sym==semicolon)
                {
                    if (sym == semicolon)
                    {
                        getsymdo;
                    }
                    else
                    {
                        error(10);  /* 缺少分号 */
                    }
                    statementdo(nxtlev, ptx, lev);
                }
                if(sym == endsym)
                {
                    getsymdo;
                }
                else
                {
                    error(17);  /* 缺少end或分号 */
                }
            }
            else
            {
                if (sym == whilesym)    /* 准备按照while语句处理 */
                {
                    cx1 = cx;   /* 保存判断条件操作的位置 */
                    getsymdo;
                    memcpy(nxtlev, fsys, sizeof(bool)*symnum);
                    nxtlev[dosym] = true;   /* 后跟符号为do */
                    conditiondo(nxtlev, ptx, lev);  /* 调用条件处理 */
                    cx2 = cx;   /* 保存循环体的结束的下一个位置 */
                    gendo(jpc, 0, 0);   /* 生成条件跳转，但跳出循环的地址未知 */
                    if (sym == dosym)
                    {
                        getsymdo;
                    }
                    else
                    {
                        error(18);  /* 缺少do */
                    }
                    statementdo(fsys, ptx, lev);    /* 循环体 */
                    gendo(jmp, 0, cx1); /* 回头重新判断条件 */
                    code[cx2].a = cx;   /* 反填跳出循环的地址，与if类似 */
                }
                else
                {
                    memset(nxtlev, 0, sizeof(bool)*symnum); /* 语句结束无补救集合 */
                    testdo(fsys, nxtlev, 19);   /* 检测语句结束的正确性 */
                }
            }
        }
    }
}
```

17.编写表达式处理函数

```
int expression(bool* fsys, int* ptx, int lev)
{
    enum symbol addop;  /* 用于保存正负号 */
    bool nxtlev[symnum];

    if(sym==plus || sym==minus) /* 开头的正负号，此时当前表达式被看作一个正的或负的项 */
    {
        addop = sym;    /* 保存开头的正负号 */
        getsymdo;
        memcpy(nxtlev, fsys, sizeof(bool)*symnum);
        nxtlev[plus] = true;
        nxtlev[minus] = true;
        termdo(nxtlev, ptx, lev);   /* 处理项 */
        if (addop == minus)
        {
            gendo(opr,0,1); /* 如果开头为负号生成取负指令 */
        }
    }
    else    /* 此时表达式被看作项的加减 */
    {
        memcpy(nxtlev, fsys, sizeof(bool)*symnum);
        nxtlev[plus] = true;
        nxtlev[minus] = true;
        termdo(nxtlev, ptx, lev);   /* 处理项 */
    }
    while (sym==plus || sym==minus)
    {
        addop = sym;
        getsymdo;
        memcpy(nxtlev, fsys, sizeof(bool)*symnum);
        nxtlev[plus] = true;
        nxtlev[minus] = true;
        termdo(nxtlev, ptx, lev);   /* 处理项 */
        if (addop == plus)
        {
            gendo(opr, 0, 2);   /* 生成加法指令 */
        }
        else
        {
            gendo(opr, 0, 3);   /* 生成减法指令 */
        }
    }
    return 0;
}

/*
```

18.编写项处理函数

```
int term(bool* fsys, int* ptx, int lev)
{
    enum symbol mulop;  /* 用于保存乘除法符号 */
    bool nxtlev[symnum];

    memcpy(nxtlev, fsys, sizeof(bool)*symnum);
    nxtlev[times] = true;
    nxtlev[slash] = true;
    factordo(nxtlev, ptx, lev); /* 处理因子 */
    while(sym==times || sym==slash)
    {
        mulop = sym;
        getsymdo;
        factordo(nxtlev, ptx, lev);
        if(mulop == times)
        {
            gendo(opr, 0, 4);   /* 生成乘法指令 */
        }
        else
        {
            gendo(opr, 0, 5);   /* 生成除法指令 */
        }
    }
    return 0;
}
```

19.编写因子处理函数

```
int factor(bool* fsys, int* ptx, int lev)
{
    int i;
    bool nxtlev[symnum];
    testdo(facbegsys, fsys, 24);    /* 检测因子的开始符号 */
    /* while(inset(sym, facbegsys)) */  /* 循环直到不是因子开始符号 */
    if(inset(sym,facbegsys))    /* BUG: 原来的方法var1(var2+var3)会被错误识别为因子 */
    {
        if(sym == ident)    /* 因子为常量或变量 */
        {
            i = position(id, *ptx); /* 查找名字 */
            if (i == 0)
            {
                error(11);  /* 标识符未声明 */
            }
            else
            {
                switch (table[i].kind)
                {
                case constant:  /* 名字为常量 */
                    gendo(lit, 0, table[i].val);    /* 直接把常量的值入栈 */
                    break;
                case variable:  /* 名字为变量 */
                    gendo(lod, lev-table[i].level, table[i].adr);   /* 找到变量地址并将其值入栈 */
                    break;
                case procedur:  /* 名字为过程 */
                    error(21);  /* 不能为过程 */
                    break;
                }
            }
            getsymdo;
        }
        else
        {
            if(sym == number)   /* 因子为数 */
            {
                if (num > amax)
                {
                    error(31);
                    num = 0;
                }
                gendo(lit, 0, num);
                getsymdo;
            }
            else
            {
                if (sym == lparen)  /* 因子为表达式 */
                {
                    getsymdo;
                    memcpy(nxtlev, fsys, sizeof(bool)*symnum);
                    nxtlev[rparen] = true;
                    expressiondo(nxtlev, ptx, lev);
                    if (sym == rparen)
                    {
                        getsymdo;
                    }
                    else
                    {
                        error(22);  /* 缺少右括号 */
                    }
                }
                testdo(fsys, facbegsys, 23);    /* 因子后有非法符号 */
            }
        }
    }
    return 0;
}
```

20.编写条件处理函数

```
int condition(bool* fsys, int* ptx, int lev)
{
    enum symbol relop;
    bool nxtlev[symnum];

    if(sym == oddsym)   /* 准备按照odd运算处理 */
    {
        getsymdo;
        expressiondo(fsys, ptx, lev);
        gendo(opr, 0, 6);   /* 生成odd指令 */
    }
    else
    {
        /* 逻辑表达式处理 */
        memcpy(nxtlev, fsys, sizeof(bool)*symnum);
        nxtlev[eql] = true;
        nxtlev[neq] = true;
        nxtlev[lss] = true;
        nxtlev[leq] = true;
        nxtlev[gtr] = true;
        nxtlev[geq] = true;
        expressiondo(nxtlev, ptx, lev);
        if (sym≠eql && sym≠neq && sym≠lss && sym≠leq && sym≠gtr && sym≠geq)
        {
            error(20);
        }
        else
        {
            relop = sym;
            getsymdo;
            expressiondo(fsys, ptx, lev);
            switch (relop)
            {
            case eql:
                gendo(opr, 0, 8);
                break;
            case neq:
                gendo(opr, 0, 9);
                break;
            case lss:
                gendo(opr, 0, 10);
                break;
            case geq:
                gendo(opr, 0, 11);
                break;
            case gtr:
                gendo(opr, 0, 12);
                break;
            case leq:
                gendo(opr, 0, 13);
                break;
            }
        }
    }
    return 0;
}

/*
```

21.编写解释程序

```
void interpret()
{
    int p, b, t;    /* 指令指针，指令基址，栈顶指针 */
    struct instruction i;   /* 存放当前指令 */
    int s[stacksize];   /* 栈 */

    printf("start pl0\n");
    t = 0;
    b = 0;
    p = 0;
    s[0] = s[1] = s[2] = 0;
    do {
        i = code[p];    /* 读当前指令 */
        p++;
        switch (i.f)
        {
        case lit:   /* 将a的值取到栈顶 */
            s[t] = i.a;
            t++;
            break;
        case opr:   /* 数学、逻辑运算 */
            switch (i.a)
            {
            case 0:
                t = b;
                p = s[t+2];
                b = s[t+1];
                break;
            case 1:
                s[t-1] = -s[t-1];
                break;
            case 2:
                t--;
                s[t-1] = s[t-1]+s[t];
                break;
            case 3:
                t--;
                s[t-1] = s[t-1]-s[t];
                break;
            case 4:
                t--;
                s[t-1] = s[t-1]*s[t];
                break;
            case 5:
                t--;
                s[t-1] = s[t-1]/s[t];
                break;
            case 6:
                s[t-1] = s[t-1]%2;
                break;
            case 8:
                t--;
                s[t-1] = (s[t-1] == s[t]);
                break;
            case 9:
                t--;
                s[t-1] = (s[t-1] ≠ s[t]);
                break;
            case 10:
                t--;
                s[t-1] = (s[t-1] < s[t]);
                break;
            case 11:
                t--;
                s[t-1] = (s[t-1] ≥ s[t]);
                break;
            case 12:
                t--;
                s[t-1] = (s[t-1] > s[t]);
                break;
            case 13:
                t--;
                s[t-1] = (s[t-1] ≤ s[t]);
                break;
            case 14:
                printf("%d", s[t-1]);
                fprintf(fa2, "%d", s[t-1]);
                t--;
                break;
            case 15:
                printf("\n");
                fprintf(fa2,"\n");
                break;
            case 16:
                printf("?");
                fprintf(fa2, "?");
                scanf("%d", &(s[t]));
                fprintf(fa2, "%d\n", s[t]);
                t++;
                break;
            }
            break;
        case lod:   /* 取相对当前过程的数据基地址为a的内存的值到栈顶 */
            s[t] = s[base(i.l,s,b)+i.a];
            t++;
            break;
        case sto:   /* 栈顶的值存到相对当前过程的数据基地址为a的内存 */
            t--;
            s[base(i.l, s, b) + i.a] = s[t];
            break;
        case cal:   /* 调用子过程 */
            s[t] = base(i.l, s, b); /* 将父过程基地址入栈 */
            s[t+1] = b; /* 将本过程基地址入栈，此两项用于base函数 */
            s[t+2] = p; /* 将当前指令指针入栈 */
            b = t;  /* 改变基地址指针值为新过程的基地址 */
            p = i.a;    /* 跳转 */
            break;
        case inte:  /* 分配内存 */
            t += i.a;
            break;
        case jmp:   /* 直接跳转 */
            p = i.a;
            break;
        case jpc:   /* 条件跳转 */
            t--;
            if (s[t] == 0)
            {
                p = i.a;
            }
            break;
        }
    } while (p ≠ 0);
}
```

22.编写过程基址求上层基址函数

```
int base(int l, int* s, int b)
{
    int b1;
    b1 = b;
    while (l > 0)
    {
        b1 = s[b1];
        l--;
    }
    return b1;
}
```

# 3 实验结果

输入文件：

```
const a=10;
var b,c;
procedure p;
    begin
        c:=b+a;
    end;
begin
    read(b);
    while b#0 do
    begin
        call p;
        write(2*c);
        read(b);
    end
end.

```

输出虚拟机代码：

```
0 jmp 0 0
1 jmp 0 2
2 int 0 3
3 lod 1 3
4 lit 0 10
5 opr 0 2
6 sto 1 4
7 opr 0 0
0 jmp 0 8
1 jmp 0 2
2 int 0 3
3 lod 1 3
4 lit 0 10
5 opr 0 2
6 sto 1 4
7 opr 0 0
8 int 0 5
9 opr 0 16
10 sto 0 3
11 lod 0 3
12 lit 0 0
13 opr 0 9
14 jpc 0 24
15 cal 0 2
16 lit 0 2
17 lod 0 4
18 opr 0 4
19 opr 0 14
20 opr 0 15
21 opr 0 16
22 sto 0 3
23 jmp 0 11
24 opr 0 0
```

# 4 我的体会

在设计和实现一个 PL/0 编译器的过程中，我获得了许多宝贵的经验和体会。

1、深入理解编程语言原理：通过编写 PL/0 编译器，我深入理解了编程语言的语法结构、语义和编译原理。这包括了词法分析、语法分析、语义分析以及代码生成等方面的知识。

2、掌握了编译器设计与实现的基本流程：在编写编译器的过程中，我逐步学习了编译器的基本流程，包括词法分析器的设计、语法分析器的设计、语义分析器的设计以及代码生成器的设计等。

3、提高了代码设计和优化能力：编写编译器需要考虑到代码的效率和可维护性，我学会了如何设计清晰、高效的代码结构，并对代码进行优化以提高编译器的性能。

4、增强了问题解决能力：在编写编译器的过程中，我遇到了许多问题和挑战，如语法歧义、性能优化等。通过解决这些问题，我提高了自己的问题解决能力和技巧。

5、加深了对计算机体系结构的理解：编写编译器需要深入了解计算机体系结构和指令集，尤其是对于 P-code 语言的生成，我学会了如何将高级语言转换成底层的指令序列。

总的来说，设计和实现一个 PL/0 编译器是一次极具挑战性和收获的经历，它不仅提升了我的编程能力和理论水平，也让我更深入地理解了计算机科学的核心原理。