附录

附录一　PL/0 语言及其实现

1.1 PL/0 语言简介

PL/0 语言是 Pascal 语言的一个子集，是 Pascal 语言的设计者 Niklaus Wirth 在其专著 Algorithms + Data Structures = Programs 一书（译著书名：算法＋数据结构＝程序）中给出的。

PL/0 是一个小巧的高级语言。虽然它只有整数类型，但它是相当完全的可嵌套的分程序（*block*）的程序结构，分程序中可以有常量定义、变量声明和无参过程声明，过程体又是分程序。PL/0 有赋值语句、条件语句、循环语句、过程调用语句、复合语句和空语句。

由于上面这些语言概念已为大家熟知，因此不再进行语义解释。下面用习题 3.7 所介绍的扩展方式来给出 PL/0 语言的文法。

Program → Block .
Block → [ConstDecl] [VarDecl][ProcDecl] Stmt
ConstDecl → **const** ConstDef {, ConstDef} ;
ConstDef → **ident** = **number**
VarDecl → **var ident** {, **ident**} ;
ProcDecl → **procedure ident** ; Block ; {**procedure ident** ; Block ;}
Stmt → **ident** := Exp | **call ident** | **begin** Stmt {; Stmt} **end** |
if Cond **then** Stmt | **while** Cond **do** Stmt | ε
Cond → **odd** Exp | Exp RelOp Exp
RelOp → = | <> | < | > | <= | >=
Exp → [+ | –] Term {+ Term | – Term}
Term → Factor {* Factor | / Factor}
Factor → **ident** | **number** | (Exp)

其中的标识符 **ident** 是字母开头的字母数字串，**number** 是无符号整数，**begin**、**call**、**const**、**do**、**end**、**if**、**odd**、**procedure**、**then**、**var**、**while** 是保留字。

用 PL/0 语言写的一个程序如下，它有 3 个过程，分别计算两个整数相乘、相除和求最大公约数。

```
const m=7, n=85;
var x,y,z,q,r;
procedure multiply;
    var a,b;
    begin
        a:=x; b:=y; z:=0;
        while b>0 do
            begin
                if odd b then z:=z+a;
                a:=2*a; b:=b/2;
            end
    end;

procedure divide;
```

```
    var w;
    begin
        r:=x; q:=0; w:=y;
        while w<=r do w:=2*w;
        while w>y do
            begin
                q:=2*q; w:=w/2;
                if w<=r then
                    begin
                        r:=r-w;
                        q:=q+1;
                    end
            end
    end;

procedure gcd;
    var f,g;
    begin
        f:=x;
        g:=y;
        while f<>g do
            begin
                if f<g then g:=g-f;
                if g<f then f:=f-g;
            end
    end;

begin
    x:=m; y:=n; call multiply;
    x:=25; y:=3; call divide;
    x:=34; y:=36; call gcd;
end.
```

1.2 PL/0 语言编译器

在《算法＋数据结构＝程序》一书中，Niklaus Wirth 设计的 PL/0 语言编译器分成两部分，把源语言翻译成中间语言的编译器和中间语言解释器，编译器用的是递归下降的预测分析方法，并且用 3.3.6 节介绍的紧急方式的错误恢复方法。

中间语言是一种栈机器代码，其指令集是根据 PL/0 语言的需要来设计的。它包括下列指令：

（1）lit：将常数装入栈顶的指令；

（2）lod：将变量的值装入栈顶的指令；

（3）sto：对应于赋值语句的存储指令；

（4）cal：对应于过程调用的指令；

（5）int：增加栈顶寄存器的值，完成对局部变量的存储分配的指令；

（6）jmp，jpc：对应条件语句和循环语句的无条件转移和条件转移的控制转移指令；

（7）opr：包括一组算术和关系运算的指令。

一条指令由三个域组成：

（1）操作码 f：上面已经列出了所有 8 种操作码。

（2）层次差 l：这里的层次差就是 5.3.2 节介绍嵌套深度时的 $n_p - n_a$。该域仅用于存取指令和调用指令。

（3）多用途 a：在运算指令中，a 的值用来区分不同的运算；在其他情况，a 或是一个数（lit，int），或是一个程序地址（jmp，jpc，cal），或是一个数据地址（lod，sto）。

编译器对 PL/0 源程序进行一遍扫描，并逐行输出源程序。在源程序无错的情况下，编译器每编译完一个分程序，就列出该分程序的代码，这由编译器的 listcode 过程完成。每个分程序的第一条指令是 jmp 指令，其作用是绕过该分程序声明部分产生的代码（即绕过内嵌过程的代码）。listcode 过程没有列出这条代码。

解释器是编译器中的一个过程，若源程序无错，则编译结束时调用解释过程 interpret。由于 PL/0 语言没有输出语句，解释器按执行次序，每遇到对变量赋值时就输出该值。

由于 PL/0 语言是过程嵌套语言，因此程序运行时，活动记录栈中每个活动记录需要包含控制链和访问链，并按 5.3.2 节所讲的方法来建立访问链。

活动记录栈的栈顶以外的存储空间作为代码执行过程中所需要的计算栈，无需另外设立计算栈。

PL/0 语言虽然简单，但它的编译器能够展示编译高级语言的许多基本概念，它提出的基本方法对设计复杂语言的编译器也是适用的。Niklaus Wirth 的这个编译器已经有 30 多年的历史，虽然 30 年来编译技术有了很大发展，但是对普通高校计算机专业的学生来说，通过阅读该编译器的源代码来融会贯通编译的理论和技术，并以此编译器为基础来进行课程实践，仍不失为一种好的选择。

1.3 PL/0 语言编译器的源代码

Niklaus Wirth 设计的编译器是用 Pascal 语言编写的，本书作者用 C 语言改写后的代码在下面给出。为尊重 Niklaus Wirth 的原意，也为了便于对照原来用 Pascal 语言写的代码，我们尽量不改动原来代码的流程和风格等，除非那些由于 Pascal 语言和 C 语言不共同支持的程序结构、数据类型和语句等引起的问题。

代码改写中碰到的问题整理如下，若有读者感兴趣同时了解两种不同语言写的版本，则需要关注下面这几点。

（1）Pascal 语言允许过程嵌套，导致有非全局且非局部的变量，而 C 语言的非局部变量则一定是全局的。

（2）Pascal 语言有集合类型，而 C 语言没有。这是改写中碰到的最大问题，我们用位串来表示集合，用位运算来实现集合运算。这样做使得改写的代码容易理解，但是由于字长的限制，给扩展语言的实现带来一点点麻烦。

（3）Pascal 有子界类型，而 C 语言没有。

（4）Pascal 语言用子界类型来规定数组下标的上下界，而 C 语言的下标从 0 开始。

（5）Pascal 语言用紧缩字符数组类型作为字符串类型，由于长度相同的字符串才可能属于同一个类型，因此字符串无需‘\0’作为结束标记，而 C 语言的字符串一定以‘\0’结尾。

（6）Pascal 语言的变体记录类型可以有标志域，它和 C 语言的共用体类型是有区别的。

（7）对于两种语言都有的整型，在 C 语言中统一使用 long 类型（统一使用 int 类型也是一样的）。

（8）两种语言的文件（包括标准输入和输出）操作有较大区别。

（9）两种语言的注释形式不同。

（10）由于 int 是 C 语言的关键字，因此中间语言指令的操作码 int 被改成 Int。

（11）改写版本中增加了制表符作为单词的分隔符，因为现在大家已经习惯用制表符。带来的问题是编译器报错的箭头所指位置可能不准确。

Niklaus Wirth 设计的编译器中有一些不影响大局的小问题(也许是 Niklaus Wirth 有意为之)，我们也没有修改。例如：

（1）没有检查一个分程序中的名字是否重复声明，导致出现重复声明的名字时，默认的是最后出现的那一个。

（2）按附录 1.1 节的文法，常量定义和变量声明用逗号分隔，但实际情况是用分号分隔也被认可。

（3）无符号整数超过一定位数时，以报告编号为 30 而不是 31 的错误更为准确。

下面是编译器两个文件 pl0.h 和 pl0.c 的代码。

（1）文件 pl0.h

```
#include <stdio.h>

#define norw        11          // no. of reserved words
#define txmax       100         // length of identifier table
#define nmax        14          // max. no. of digits in numbers
#define al          10          // length of identifiers
#define amax        2047        // maximum address
#define levmax      3           // maximum depth of block nesting
#define cxmax       2000        // size of code array

#define nul         0x1
#define ident       0x2
#define number      0x4
#define plus        0x8
#define minus       0x10
#define times       0x20
#define slash       0x40
#define oddsym      0x80
#define eql         0x100
#define neq         0x200
#define lss         0x400
#define leq         0x800
#define gtr         0x1000
#define geq         0x2000
#define lparen      0x4000
#define rparen      0x8000
#define comma       0x10000
#define semicolon   0x20000
#define period      0x40000
```

```
#define becomes    0x80000
#define beginsym   0x100000
#define endsym     0x200000
#define ifsym      0x400000
#define thensym    0x800000
#define whilesym   0x1000000
#define dosym      0x2000000
#define callsym    0x4000000
#define constsym   0x8000000
#define varsym     0x10000000
#define procsym    0x20000000

enum object {
    constant, variable, proc
};

enum fct {
    lit, opr, lod, sto, cal, Int, jmp, jpc         // functions
};

typedef struct{
    enum fct f;     // function code
    long l;         // level
    long a;         // displacement address
} instruction;
/*  lit 0, a : load constant a
    opr 0, a : execute operation a
    lod l, a : load variable l, a
    sto l, a : store variable l, a
    cal l, a : call procedure a at level l
    Int 0, a : increment t-register by a
    jmp 0, a : jump to a
    jpc 0, a : jump conditional to a       */

char ch;               // last character read
unsigned long sym;     // last symbol read
char id[al+1];         // last identifier read
long num;              // last number read
long cc;               // character count
long ll;               // line length
long kk, err;
long cx;               // code allocation index

char line[81];
```

```
char a[al+1];
instruction code[cxmax+1];
char word[norw][al+1];
unsigned long wsym[norw];
unsigned long ssym[256];

char mnemonic[8][3+1];
unsigned long declbegsys, statbegsys, facbegsys;

struct{
    char name[al+1];
    enum object kind;
    long val;
    long level;
    long addr;
}table[txmax+1];

char infilename[80];
FILE* infile;

// the following variables for block
long dx;        // data allocation index
long lev;       // current depth of block nesting
long tx;        // current table index

// the following array space for interpreter
#define stacksize 50000
long s[stacksize];  // datastore
```

（2）文件 pl0.c

```
// pl/0 compiler with code generation
#include <stdlib.h>
#include <string.h>
#include "pl0.h"

void error(long n){
    long i;

    printf(" ****");
    for (i=1; i<=cc-1; i++){
        printf(" ");
    }
    printf("^%2d\n",n);
    err++;
```

```
}

void getch() {
    if(cc==ll){
        if(feof(infile)){
            printf("************************************\n");
            printf("      program incomplete\n");
            printf("************************************\n");
            exit(1);
        }
        ll=0; cc=0;
        printf("%5d ", cx);
        while((!feof(infile))&&((ch=getc(infile))!='\n')){
            printf("%c",ch);
            ll=ll+1; line[ll]=ch;
        }
        printf("\n");
        ll=ll+1; line[ll]=' ';
    }
    cc=cc+1; ch=line[cc];
}

void getsym(){
    long i,j,k;

    while(ch==' '||ch=='\t'){
        getch();
    }
    if(isalpha(ch)){    // identified or reserved
        k=0;
        do{
            if(k<al){
            a[k]=ch; k=k+1;
            }
            getch();
        }while(isalpha(ch)||isdigit(ch));
        if(k>=kk){
            kk=k;
        }else{
            do{
                kk=kk-1; a[kk]=' ';
            }while(k<kk);
        }
        strcpy(id,a); i=0; j=norw-1;
```

```
    do{
        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;
    }
}else if(isdigit(ch)){ // number
    k=0; num=0; sym=number;
    do{
        num=num*10+(ch-'0');
        k=k+1; getch();
    }while(isdigit(ch));
    if(k>nmax){
        error(31);
    }
}else if(ch==':'){
    getch();
    if(ch=='='){
        sym=becomes; getch();
    }else{
        sym=nul;
    }
}else if(ch=='<'){
    getch();
    if(ch=='='){
        sym=leq; getch();
    }else if(ch=='>'){
        sym=neq; getch();
    }else{
        sym=lss;
    }
}else if(ch=='>'){
    getch();
    if(ch=='='){
        sym=geq; getch();
    }else{
```

```
            sym=gtr;
        }
    }else{
        sym=ssym[(unsigned char)ch]; getch();
    }
}

void gen(enum fct x, long y, long z){
    if(cx>cxmax){
        printf("program too long\n");
        exit(1);
    }
    code[cx].f=x; code[cx].l=y; code[cx].a=z;
    cx=cx+1;
}

void test(unsigned long s1, unsigned long s2, long n){
    if (!(sym & s1)){
        error(n);
        s1=s1|s2;
        while(!(sym & s1)){
            getsym();
        }
    }
}

void enter(enum object k){      // enter object into table
    tx=tx+1;
    strcpy(table[tx].name,id);
    table[tx].kind=k;
    switch(k){
        case constant:
            if(num>amax){
                error(31);
                num = 0;
            }
            table[tx].val=num;
            break;
        case variable:
            table[tx].level=lev; table[tx].addr=dx; dx=dx+1;
            break;
        case proc:
            table[tx].level=lev;
            break;
```

```
    }
}

long position(char* id){    // find identifier id in table
    long i;

    strcpy(table[0].name,id);
    i=tx;
    while(strcmp(table[i].name,id)!=0){
        i=i-1;
    }
    return i;
}

void constdeclaration(){
    if(sym==ident){
        getsym();
        if(sym==eql||sym==becomes){
            if(sym==becomes){
                error(1);
            }
            getsym();
            if(sym==number){
                enter(constant); getsym();
            }else{
                error(2);
            }
        }else{
            error(3);
        }
    }else{
        error(4);
    }
}

void vardeclaration(){
    if(sym==ident){
        enter(variable); getsym();
    }else{
        error(4);
    }
}

void listcode(long cx0){    // list code generated for this block
```

```
    long i;

    for(i=cx0; i<=cx-1; i++){
        printf("%10d%5s%3d%5d\n", i, mnemonic[code[i].f], code[i].l, code[i].a);
    }

}

void expression(unsigned long);
void factor(unsigned long fsys){
    long i;

    test(facbegsys,fsys,24);
    while(sym & facbegsys){
        if(sym==ident){
            i=position(id);
            if(i==0){
                error(11);
            }else{
                switch(table[i].kind){
                    case constant:
                        gen(lit,0,table[i].val);
                        break;
                    case variable:
                        gen(lod,lev-table[i].level,table[i].addr);
                        break;
                    case proc:
                        error(21);
                        break;
                }
            }
            getsym();
        }else if(sym==number){
            if(num>amax){
                error(31); num=0;
            }
            gen(lit,0,num);
            getsym();
        }else if(sym==lparen){
            getsym();
            expression(rparen|fsys);
            if(sym==rparen){
                getsym();
            }else{
```

```
                error(22);
            }
        }
        test(fsys,lparen,23);
    }
}

void term(unsigned long fsys){
    unsigned long mulop;

    factor(fsys|times|slash);
    while(sym==times||sym==slash){
        mulop=sym; getsym();
        factor(fsys|times|slash);
        if(mulop==times){
            gen(opr,0,4);
        }else{
            gen(opr,0,5);
        }
    }
}

void expression(unsigned long fsys){
    unsigned long addop;

    if(sym==plus||sym==minus){
        addop=sym; getsym();
        term(fsys|plus|minus);
        if(addop==minus){
            gen(opr,0,1);
        }
    }else{
        term(fsys|plus|minus);
    }
    while(sym==plus||sym==minus){
        addop=sym; getsym();
        term(fsys|plus|minus);
        if(addop==plus){
            gen(opr,0,2);
        }else{
            gen(opr,0,3);
        }
    }
}
```

```
void condition(unsigned long fsys){
    unsigned long relop;

    if(sym==oddsym){
        getsym(); expression(fsys);
        gen(opr,0,6);
    }else{
        expression(fsys|eql|neq|lss|gtr|leq|geq);
        if(!(sym&(eql|neq|lss|gtr|leq|geq))){
            error(20);
        }else{
            relop=sym; getsym();
            expression(fsys);
            switch(relop){
                case eql:
                    gen(opr,0,8);
                    break;
                case neq:
                    gen(opr,0,9);
                    break;
                case lss:
                    gen(opr,0,10);
                    break;
                case geq:
                    gen(opr,0,11);
                    break;
                case gtr:
                    gen(opr,0,12);
                    break;
                case leq:
                    gen(opr,0,13);
                    break;
            }
        }
    }
}

void statement(unsigned long fsys){
    long i,cx1,cx2;

    if(sym==ident){
        i=position(id);
        if(i==0){
```

```
            error(11);
        }else if(table[i].kind!=variable){  // assignment to non-variable
            error(12); i=0;
        }
        getsym();
        if(sym==becomes){
            getsym();
        }else{
            error(13);
        }
        expression(fsys);
        if(i!=0){
            gen(sto,lev-table[i].level,table[i].addr);
        }
    }else if(sym==callsym){
        getsym();
        if(sym!=ident){
            error(14);
        }else{
            i=position(id);
            if(i==0){
                error(11);
            }else if(table[i].kind==proc){
                gen(cal,lev-table[i].level,table[i].addr);
            }else{
                error(15);
            }
            getsym();
        }
    }else if(sym==ifsym){
        getsym(); condition(fsys|thensym|dosym);
        if(sym==thensym){
            getsym();
        }else{
            error(16);
        }
        cx1=cx; gen(jpc,0,0);
        statement(fsys);
        code[cx1].a=cx;
    }else if(sym==beginsym){
        getsym(); statement(fsys|semicolon|endsym);
        while(sym==semicolon||(sym&statbegsys)){
            if(sym==semicolon){
                getsym();
```

```
            }else{
                error(10);
            }
            statement(fsys|semicolon|endsym);
        }
        if(sym==endsym){
            getsym();
        }else{
            error(17);
        }
    }else if(sym==whilesym){
        cx1=cx; getsym();
        condition(fsys|dosym);
        cx2=cx; gen(jpc,0,0);
        if(sym==dosym){
            getsym();
        }else{
            error(18);
        }
        statement(fsys); gen(jmp,0,cx1);
        code[cx2].a=cx;
    }
    test(fsys,0,19);
}

void block(unsigned long fsys){
    long tx0;       // initial table index
    long cx0;       // initial code index
    long tx1;       // save current table index before processing nested procedures
    long dx1;       // save data allocation index

    dx=3; tx0=tx; table[tx].addr=cx; gen(jmp,0,0);
    if(lev>levmax){
        error(32);
    }
    do{
        if(sym==constsym){
            getsym();
            do{
                constdeclaration();
                while(sym==comma){
                    getsym(); constdeclaration();
                }
                if(sym==semicolon){
```

```
                getsym();
            }else{
                error(5);
            }
        }while(sym==ident);
    }
    if(sym==varsym){
        getsym();
        do{
            vardeclaration();
            while(sym==comma){
                getsym(); vardeclaration();
            }
            if(sym==semicolon) {
                getsym();
            }else{
                error(5);
            }
        }while(sym==ident);
    }
    while(sym==procsym){
        getsym();
        if(sym==ident){
            enter(proc); getsym();
        }else{
            error(4);
        }
        if(sym==semicolon){
            getsym();
        }else{
            error(5);
        }
        lev=lev+1; tx1=tx; dx1=dx;
        block(fsys|semicolon);
        lev=lev-1; tx=tx1; dx=dx1;
        if(sym==semicolon){
            getsym();
            test(statbegsys|ident|procsym,fsys,6);
        }else{
            error(5);
        }
    }
    test(statbegsys|ident,declbegsys,7);
}while(sym&declbegsys);
```

```
    code[table[tx0].addr].a=cx;
    table[tx0].addr=cx;     // start addr of code
    cx0=cx; gen(Int,0,dx);
    statement(fsys|semicolon|endsym);
    gen(opr,0,0); // return
    test(fsys,0,8);
    listcode(cx0);
}

long base(long b, long l){
    long b1;

    b1=b;
    while (l>0){    // find base l levels down
        b1=s[b1]; l=l-1;
    }
    return b1;
}

void interpret(){
    long p,b,t;     // program-, base-, topstack-registers
    instruction i;  // instruction register

    printf("start PL/0\n");
    t=0; b=1; p=0;
    s[1]=0; s[2]=0; s[3]=0;
    do{
        i=code[p]; p=p+1;
        switch(i.f){
            case lit:
                t=t+1; s[t]=i.a;
                break;
            case opr:
                switch(i.a){    // operator
                    case 0: // return
                        t=b-1; p=s[t+3]; b=s[t+2];
                        break;
                    case 1:
                        s[t]=-s[t];
                        break;
                    case 2:
                        t=t-1; s[t]=s[t]+s[t+1];
                        break;
                    case 3:
```

```
            t=t-1; s[t]=s[t]-s[t+1];
            break;
        case 4:
            t=t-1; s[t]=s[t]*s[t+1];
            break;
        case 5:
            t=t-1; s[t]=s[t]/s[t+1];
            break;
        case 6:
            s[t]=s[t]%2;
            break;
        case 8:
            t=t-1; s[t]=(s[t]==s[t+1]);
            break;
        case 9:
            t=t-1; s[t]=(s[t]!=s[t+1]);
            break;
        case 10:
            t=t-1; s[t]=(s[t]<s[t+1]);
            break;
        case 11:
            t=t-1; s[t]=(s[t]>=s[t+1]);
            break;
        case 12:
            t=t-1; s[t]=(s[t]>s[t+1]);
            break;
        case 13:
            t=t-1; s[t]=(s[t]<=s[t+1]);
    }
    break;
case lod:
    t=t+1; s[t]=s[base(b,i.l)+i.a];
    break;
case sto:
    s[base(b,i.l)+i.a]=s[t]; printf("%10d\n", s[t]); t=t-1;
    break;
case cal:       // generate new block mark
    s[t+1]=base(b,i.l); s[t+2]=b; s[t+3]=p;
    b=t+1; p=i.a;
    break;
case Int:
    t=t+i.a;
    break;
case jmp:
```

```
                p=i.a;
                break;
            case jpc:
                if(s[t]==0){
                    p=i.a;
                }
                t=t-1;
        }
    }while(p!=0);
    printf("end PL/0\n");
}

main(){
    long i;
    for(i=0; i<256; i++){
        ssym[i]=nul;
    }
    strcpy(word[0],  "begin     ");
    strcpy(word[1],  "call      ");
    strcpy(word[2],  "const     ");
    strcpy(word[3],  "do        ");
    strcpy(word[4],  "end       ");
    strcpy(word[5],  "if        ");
    strcpy(word[6],  "odd       ");
    strcpy(word[7],  "procedure ");
    strcpy(word[8],  "then      ");
    strcpy(word[9],  "var       ");
    strcpy(word[10], "while     ");
    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]=thensym;
    wsym[9]=varsym;
    wsym[10]=whilesym;
    ssym['+']=plus;
    ssym['-']=minus;
    ssym['*']=times;
    ssym['/']=slash;
    ssym['(']=lparen;
```

```
    ssym[')']=rparen;
    ssym['=']=eql;
    ssym[',']=comma;
    ssym['.']=period;
    ssym[';']=semicolon;
    strcpy(mnemonic[lit],"lit");
    strcpy(mnemonic[opr],"opr");
    strcpy(mnemonic[lod],"lod");
    strcpy(mnemonic[sto],"sto");
    strcpy(mnemonic[cal],"cal");
    strcpy(mnemonic[Int],"int");
    strcpy(mnemonic[jmp],"jmp");
    strcpy(mnemonic[jpc],"jpc");
    declbegsys=constsym|varsym|procsym;
    statbegsys=beginsym|callsym|ifsym|whilesym;
    facbegsys=ident|number|lparen;

    printf("please input source program file name: ");
    scanf("%s",infilename);
    printf("\n");
    if((infile=fopen(infilename,"r"))==NULL){
        printf("File %s can't be opened.\n", infilename);
        exit(1);
    }

    err=0;
    cc=0; cx=0; ll=0; ch=' '; kk=al; getsym();
    lev=0; tx=0;
    block(declbegsys|statbegsys|period);
    if(sym!=period){
        error(9);
    }
    if(err==0){
        interpret();
    }else{
        printf("errors in PL/0 program\n");
    }
    fclose(infile);
}
```

下面是编译器报告错误的错误编号及含义。

错误编号	含义
1	应为=而不是:=
2	=后应为数

3	标识符后应为=
4	const，var，procedure 后应为标识符
5	遗漏逗号或分号
6	过程声明后的记号不正确
7	应为语句
8	分程序内的语句部分后的记号不正确
9	应为句号
10	语句之间漏分号
11	标识符未声明
12	不可向常量或过程赋值
13	应为赋值运算符:=
14	call 后应为标识符
15	不可调用常量或变量
16	应为 then
17	应为分号或 end
18	应为 do
19	语句后的记号不正确
20	应为关系运算符
21	表达式内不可有过程标识符
22	遗漏右括号
23	因子后不可为此记号
24	表达式不能以此记号开始
30	这个数太大
31	这个常数或地址偏移太大
32	程序嵌套层次太多

下面是一个 PL/0 源程序编译和解释执行时的输出。

```
please input source program file name:
    0 const m=7, n=85;
    1 var x,y,z,q,r;
    1 procedure multiply;
    1     var a,b;
    2     begin
    3         a:=x; b:=y; z:=0;
    9         while b>0 do
   13             begin
   13                 if odd b then z:=z+a;
   20                 a:=2*a; b:=b/2;
   28             end
   28     end;
       2  int  0    5
       3  lod  1    3
       4  sto  0    3
       5  lod  1    4
```

```
         6  sto  0    4
         7  lit  0    0
         8  sto  1    5
         9  lod  0    4
        10  lit  0    0
        11  opr  0   12
        12  jpc  0   29
        13  lod  0    4
        14  opr  0    6
        15  jpc  0   20
        16  lod  1    5
        17  lod  0    3
        18  opr  0    2
        19  sto  1    5
        20  lit  0    2
        21  lod  0    3
        22  opr  0    4
        23  sto  0    3
        24  lod  0    4
        25  lit  0    2
        26  opr  0    5
        27  sto  0    4
        28  jmp  0    9
        29  opr  0    0
   30 begin
   31     x:=m; y:=n;
   35     call multiply;
   36 end.
        30  int  0    8
        31  lit  0    7
        32  sto  0    3
        33  lit  0   85
        34  sto  0    4
        35  cal  0    2
        36  opr  0    0
start PL/0
         7
        85
         7
        85
         0
         7
        14
        42
```

```
  28
  21
  35
  56
  10
 112
   5
 147
 224
   2
 448
   1
 595
 896
   0
end PL/0
```

附录二　基于 PL/0 语言的课程实践选题

基于附录一的 PL/0 语言及其编译器，可以提出很多课程实践的课题，下面列出一些比较简单的课题，可选择实现其中的一部分。

1、给 PL/0 语言增加像 C 语言那样的形式为/* …… */的注释。

2、给 PL/0 语言增加带 else 子句的条件语句和 exit 语句。

exit 语句作为 while 语句的非正常出口语句。若处于多层 while 语句中，则它只作为最内层 while 语句的非正常出口。若它没有处于任何 while 语句中，则是一个错误。

3、给 PL/0 语言增加输入输出语句。

4、给 PL/0 语言增加带参数的过程，参数传递按值调用方式。

5、给 PL/0 语言增加布尔类型，并且布尔类型的表达式按短路方式计算。

6、给 PL/0 语言增加数组类型。

7、给 PL/0 语言增加函数类型。

8、给 PL/0 语言增加实数类型。

9、用 Yacc 或类似的生成器来生成 PL/0 语言的编译器。

10、为 PL/0 实现效果更好的语法错误恢复机制。

11、分离解释器和编译器。

把解释器从现在的编译器中分离出来，变成一个独立的 C 语言程序。分离后，编译器和解释器的接口是二进制形式的中间代码文件。