# PL0 编译器实验

| 学号 | 姓名 | 专业（方向） | 任课老师 |
| --- | --- | --- | --- |
| 16340030 | 陈斯敏 | 计算机应用 | 娄定俊 |

## 第一部分

### 1. 编译程序源代码

```
program  PL0 ( input, output);
{带有代码生成的PL0编译程序}

//label  99;

const
  norw = 11; {保留字的个数}
  txmax = 100; {标识符表长度}
  nmax = 14; {数字的最大位数}
  al = 10; {标识符的长度}
  amax = 2047; {最大地址}
  levmax = 3; {程序体嵌套的最大深度}
  cxmax = 200; {代码数组的大小}

type
  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, dosym, callsym, constsym, varsym, procsym );
  alfa = packed array [1..al] of char;
  objecttype = (constant, variable, proceduretype);
  symset = set of symbol;
  fct = (lit, opr, lod, sto, cal, int, jmp, jpc); {functions}
  {
    LIT 0,a : 取常数a
    OPR 0,a : 执行运算a
    LOD l,a : 取层差为l的层、相对地址为a的变量
    STO l,a : 存到层差为l的层、相对地址为a的变量
    CAL l,a : 调用层差为l的过程
    INT 0,a : t寄存器增加a
    JMP 0,a : 转移到指令地址a处
    JPC 0,a : 条件转移到指令地址a处
  }
  instruction = packed record
        f : fct;  {功能码}
        l : 0..levmax; {相对层数}
        a : 0..amax; {相对地址}
  end;

var
```

```
  ch : char; {最近读到的字符}
  sym : symbol; {最近读到的符号}
  id : alfa; {最近读到的标识符}
  num : integer; {最近读到的数}
  cc : integer; {当前行的字符计数}
  ll : integer; {当前行的长度}
  kk, err : integer;
  cx : integer; {代码数组的当前下标}
  line : array [1..81] of char; {当前行}
  a : alfa; {当前标识符的字符串}
  code : array [0..cxmax] of instruction; {中间代码数组}
  word : array [1..norw] of alfa; {存放保留字的字符串}
  wsym : array [1..norw] of symbol; {存放保留字的记号}
  ssym : array [char] of symbol; {存放算符和标点符号的记号}
  mnemonic : array [fct] of packed array [1..5] of char;
  {中间代码算符的字符串}
  declbegsys, statbegsys, facbegsys : symset;
  table : array [0..txmax] of {符号表}
          record
            name : alfa;
            case kind : objecttype of
             constant : (val : integer);
             variable, proceduretype : (level, adr : integer)
          end;
  fin : text;       {源代码文件}
  fout : text;      {输出文件}

procedure error (n : integer);
begin
  writeln(fout, '****', ' ':cc-1, '^', n:2);  {cc为当前行已读的字符数, n为错误号}
  err := err + 1{错误数err加1}
end {error};

procedure getsym;
var  i, j, k : integer;

procedure  getch ;
begin
    if cc = ll then {如果cc指向行末}
    begin
        if eof(fin) then {如果已到文件尾}
        begin
            writeln(fout, 'PROGRAM INCOMPLETE');
            close(fin);
            close(fout);
            exit;
            //goto 99
        end;
        {读新的一行}
        ll := 0;
        cc := 0;
        write(fout, cx : 5, ' ');{cx : 5位数}
        while not eoln(fin) do {如果不是行末}
        begin
```

```
            ll := ll + 1; {将行缓冲区的长度+1}
            read(fin, ch); {从源文件中读取一个字符到ch中}
            write(fout, ch);{输出ch到输出文件中}
            line[ll] := ch  {把这个字符放到当前行末尾}
        end;
        writeln(fout); {换行}
        readln(fin);{从源文件下一行开始读取}
        ll := ll + 1; {将行缓冲区的长度+1}
        line[ll] := ' ' { process end-line }    {行数组最后一个元素为空格}
    end;
    cc := cc + 1;
    ch := line[cc]  {ch取line中下一个字符}
end {getch};


begin {getsym}
    while ch = ' ' do getch; {跳过无用空白}

    if ch in ['a'..'z'] then
    begin {标识符或保留字}
        k := 0;
        repeat {处理字母开头的字母、数字串}
            if k < al then
                begin
                    k:= k + 1;
                    a[k] := ch
                end;
            getch
        until  not (ch in ['a'..'z', '0'..'9']);
        if k >= kk  then kk := k
        else
            repeat
                a[kk] := ' ';
                kk := kk-1 {如果标识符长度不是最大长度，后面补空白}
            until kk = k;
        {id中存放当前标识符或保留字的字符串}
        id := a;  i := 1;  j := norw;

        repeat
            k := (i+j) div 2;
            {用二分查找法在保留字表中找当前的标识符id}
            if id <= word[k] then j := k-1;
            if id >= word[k] then i := k + 1
        until i > j;

        {如果找到，当前记号sym为保留字，否则sym为标识符}
        if i-1 > j then
            sym := wsym[k]
        else
            sym := ident
    end

    else if ch in ['0'..'9'] then
    begin {数字}
```

```
        k := 0;  num := 0;  sym := number; {当前记号sym为数字}
        repeat {计算数字串的值}
            num := 10*num + (ord(ch)-ord('0'));
            {ord(ch)和ord(0)是ch和0在ASCII码中的序号}
            k := k + 1;
            getch;
        until not(ch in ['0'..'9']);
        if k > nmax then error(30)
        {当前数字串的长度超过上界,则报告错误}
    end

    else if ch = ':' then {处理赋值号}
    begin
        getch;
        if ch = '=' then
            begin
                sym := becomes;
                getch
            end
        else
            sym := nul;
    end

    else if ch = '<' then
    begin {处理'<'}
        getch;
        if ch = '=' then {'<='}
        begin
            sym := leq; {表示小于等于}
            getch
        end
        else if ch = '>' then {'<>'}
        begin
            sym := neq; {表示不等于}
            getch
        end
        else sym := lss {表示小于}
    end

    else if ch = '>' then
    begin {处理'>'}
        getch;
        if ch = '=' then {'>='}
        begin
            sym := geq; {表示大于等于}
            getch
        end
        else sym := gtr {表示大于}
    end

    {处理其它算符或标点符号}
    else
    begin
        sym := ssym[ch];
```

```
        getch
    end

end {getsym};


procedure  gen(x : fct; y, z : integer);
begin
    {如果当前指令序号>代码的最大长度}
    if cx > cxmax then
        begin
            write(fout, 'PROGRAM TOO LONG');
            close(fin);
            close(fout);
            exit
            //goto 99
        end;
        with code[cx] do {在代码数组cx位置生成一条新代码}
        begin
            f := x; {功能码}
            l := y; {层号}
            a := z {地址}
        end;
    cx := cx + 1 {指令序号加1}
end {gen};


procedure  test(s1, s2 : symset; n : integer);
begin
    if not (sym in s1) then
    {如果当前记号不属于集合S1,则报告错误n}
    begin
        error(n);
        s1 := s1 + s2;
        while not (sym in s1) do
            getsym
        {跳过一些记号, 直到当前记号属于S1US2}
    end
end {test};


procedure  block(lev, tx : integer; fsys : symset);
var
    dx : integer; {本过程数据空间分配下标}
    tx0 : integer; {本过程标识表起始下标}
    cx0 : integer; {本过程代码起始下标}

procedure  enter(k : objecttype);
begin {把objecttype填入符号表中}
    tx := tx + 1; {符号表指针加1}
    with table[tx] do {在符号表中增加新的一个条目}
    begin
        name := id; {当前标识符的名字}
        kind := k; {当前标识符的种类}
```

```
        case k of
            {当前标识符是常数名}
            constant :
                begin
                    if num > amax then {当前常数值大于上界,则出错}
                    begin
                        error(30);
                        num := 0
                    end;
                    val := num
                end;
            {当前标识符是变量名}
            variable :
                begin
                    level := lev; {定义该变量的过程的嵌套层数}
                    adr := dx; {变量地址为当前过程数据空间栈顶}
                    dx := dx +1; {栈顶指针加1}
                end;
            proceduretype :
                level := lev; {本过程的嵌套层数}
        end
    end
end {enter};


function  position(id : alfa) : integer; {返回id在符号表的入口}
var  i : integer;
begin {在标识符表中查标识符id}
    table[0].name := id; {在符号表栈的最下方预填标识符id}
    i := tx; {符号表栈顶指针}
    while table[i].name <> id do i := i-1;
    {从符号表栈顶往下查标识符id}
    position := i {若查到,i为id的入口,否则i=0 }
end {position};


procedure constdeclaration;
begin
    if sym = ident then {当前记号是常数名}
    begin
        getsym;
        if sym in [eql, becomes] then {当前记号是等号或赋值号}
        begin
            if sym = becomes then error(1);
            {如果当前记号是赋值号,则出错}
            getsym;
            if sym = number then {等号后面是常数}
            begin
                enter(constant); {将常数名加入符号表}
                getsym
            end
            else error(2) {等号后面不是常数出错}
        end
        else error(3) {标识符后不是等号或赋值号出错}
```

```
    end
    else error(4) {常数说明中没有常数名标识符}
end {constdeclaration};


procedure  vardeclaration;
begin
    if sym = ident then {如果当前记号是标识符}
    begin
        enter(variable); {将该变量名加入符号表的下一条目}
        getsym
    end
    else error(4) {如果变量说明未出现标识符,则出错}
end {vardeclaration};


procedure  listcode;
var  i : integer;
begin  {列出本程序体生成的代码}
    for i := cx0 to cx-1 do {cx0: 本过程第一个代码的序号, cx-1: 本过程最后一个代码的序
号}
    with code[i] do {打印第i条代码}
        writeln(fout, i:5, mnemonic[f]:7, l:3, a:5)
        {
            i: 代码序号;
            mnemonic[f]: 功能码的字符串;
            l: 相对层号(层差);
            a: 相对地址或运算号码
        }
end {listcode};


procedure  statement(fsys : symset);
var i, cx1, cx2 : integer;


procedure  expression(fsys : symset);
var  addop : symbol;


procedure  term(fsys : symset);
var  mulop : symbol;


procedure  factor(fsys : symset);
var i : integer;
begin
    test(facbegsys, fsys, 24);
    {测试当前的记号是否因子的开始符号, 否则出错, 跳过一些记号}
    while sym in facbegsys do
    {如果当前的记号是否因子的开始符号}
    begin
        {当前记号是标识符}
        if sym = ident then
```

```
        begin
            i := position(id); {查符号表,返回id的入口}
            if i = 0 then
                error(11)
            else
                {若在符号表中查不到id, 则出错, 否则,做以下工作}
                with table[i] do
                case kind of
                    constant : gen(lit, 0, val); {若id是常数, 生成指令,将常数val取到
栈顶}
                    variable : gen(lod, lev-level, adr);
                    {
                        若id是变量, 生成指令,将该变量取到栈顶;
                        lev: 当前语句所在过程的层号;
                        level: 定义该变量的过程层号;
                        adr: 变量在其过程的数据空间的相对地址
                    }
                    proceduretype : error(21)
                    {若id是过程名, 则出错}
                end;
                getsym {取下一记号}
        end
        else if sym = number then {当前记号是数字}
        begin
            if num > amax then {若数值越界,则出错}
            begin
                error(30);
                num := 0
            end;
            gen(lit, 0, num);
            {生成一条指令, 将常数num取到栈顶}
            getsym {取下一记号}
        end
        else if sym = lparen then {如果当前记号是左括号}
        begin
            getsym; {取下一记号}
            expression([rparen]+fsys); {处理表达式}
            if sym = rparen then
                getsym
                {如果当前记号是右括号, 则取下一记号,否则出错}
            else error(22)
        end;
        test(fsys, [lparen], 23)
        {测试当前记号是否同步, 否则出错, 跳过一些记号}
    end {while}
end {factor};


begin {term}
    factor(fsys+[times, slash]); {处理项中第一个因子}
    while sym in [times, slash] do
    {当前记号是“乘”或“除”号}
    begin
        mulop := sym; {运算符存入mulop}
```

```
        getsym; {取下一记号}
        factor(fsys+[times, slash]); {处理一个因子}
        if mulop = times then gen(opr, 0, 4)
            {若mulop是“乘”号,生成一条乘法指令}
        else gen(opr, 0, 5)
            {否则, mulop是除号, 生成一条除法指令}
    end
end {term};


begin {expression}
    if sym in [plus, minus] then {若第一个记号是加号或减号}
    begin
        addop := sym;  {“+”或“-”存入addop}
        getsym;
        term(fsys+[plus, minus]); {处理一个项}
        if addop = minus then gen(opr, 0, 1)
        {若第一个项前是负号, 生成一条“负运算”指令}
    end
    else term(fsys+[plus, minus]);
    {第一个记号不是加号或减号, 则处理一个项}
    while sym in [plus, minus] do {若当前记号是加号或减号}
    begin
        addop := sym; {当前算符存入addop}
        getsym; {取下一记号}
        term(fsys+[plus, minus]); {处理一个项}
        if addop = plus then gen(opr, 0, 2)
            {若addop是加号, 生成一条加法指令}
        else gen(opr, 0, 3)
            {否则, addop是减号, 生成一条减法指令}
    end
end {expression};


procedure  condition(fsys : symset);
var  relop : symbol;
begin
    if sym = oddsym then {如果当前记号是“odd”}
    begin
        getsym;  {取下一记号}
        expression(fsys); {处理算术表达式}
        gen(opr, 0, 6) {生成指令,判定表达式的值是否为奇数,是,则取“真”;不是, 则取“假”}
    end
    else {如果当前记号不是“odd”}
    begin
        expression([eql, neq, lss, gtr, leq, geq] + fsys);
        {处理算术表达式}
        if not (sym in [eql, neq, lss, leq, gtr, geq]) then
        {如果当前记号不是关系符, 则出错; 否则,做以下工作}
            error(20)
        else
        begin
            relop := sym; {关系符存入relop}
            getsym; {取下一记号}
```

```
            expression(fsys); {处理关系符右边的算术表达式}
            case relop of
                eql : gen(opr, 0, 8);  {生成指令, 判定两个表达式的值是否相等}
                neq : gen(opr, 0, 9);  {生成指令, 判定两个表达式的值是否不等}
                lss : gen(opr, 0, 10); {生成指令,判定前一表达式是否小于后一表达式}
                geq : gen(opr, 0, 11); {生成指令,判定前一表达式是否大于等于后一表达式}
                gtr : gen(opr, 0, 12); {生成指令,判定前一表达式是否大于后一表达式}
                leq : gen(opr, 0, 13); {生成指令,判定前一表达式是否小于等于后一表达式}
            end
        end
    end
end {condition};

begin {statement}
    if sym = ident then {处理赋值语句}
    begin
        i := position(id);
        {在符号表中查id, 返回id在符号表中的入口}
        if i = 0 then error(11)
        {若在符号表中查不到id, 则出错, 否则做以下工作}
        else if table[i].kind <> variable then
        {若标识符id不是变量, 则出错}
        begin {对非变量赋值}
            error(12);
            i := 0;
        end;
        getsym; {取下一记号}
        if sym = becomes then getsym else error(13);
        {若当前是赋值号, 取下一记号, 否则出错}
        expression(fsys); {处理表达式}
        if i <> 0 then {若赋值号左边的变量id有定义}
            with table[i] do
                gen(sto, lev-level, adr)
                {
                    生成一条存数指令, 将栈顶(表达式)的值存入变量id中;
                    lev: 当前语句所在过程的层号;
                    level: 定义变量id的过程的层号;
                    adr: 变量id在其过程的数据空间的相对地址
                }
    end
    else if sym = callsym then {处理过程调用语句}
    begin
        getsym; {取下一记号}
        if sym <> ident then error(14) else
        {如果下一记号不是标识符(过程名),则出错, 否则做以下工作}
        begin
            i := position(id); {查符号表,返回id在表中的位置}
            if i = 0 then error(11) else
            {如果在符号表中查不到, 则出错; 否则,做以下工作}
                with table[i] do
                    if kind = proceduretype then
                        {如果在符号表中id是过程名}
                        gen(cal, lev-level, adr)
                        {
```

```
                                生成一条过程调用指令;
                                lev: 当前语句所在过程的层号
                                level: 定义过程名id的层号;
                                adr: 过程id的代码中第一条指令的地址
                            }
                        else error(15); {若id不是过程名,则出错}
                        getsym {取下一记号}
            end
        end
        else if sym = ifsym then {处理条件语句}
        begin
            getsym; {取下一记号}
            condition([thensym, dosym]+fsys); {处理条件表达式}
            if sym = thensym then getsym else error(16);
            {如果当前记号是"then",则取下一记号; 否则出错}
            cx1 := cx; {cx1记录下一代码的地址}
            gen(jpc, 0, 0); {生成指令,表达式为"假"转到某地址(待填),否则顺序执行}
            statement(fsys); {处理一个语句}
            code[cx1].a := cx {将下一个指令的地址回填到上面的jpc指令地址栏}
        end
        else if sym = beginsym then {处理语句序列}
        begin
            getsym;
            statement([semicolon, endsym]+fsys);
            {取下一记号, 处理第一个语句}
            while sym in [semicolon]+statbegsys do
            {如果当前记号是分号或语句的开始符号,则做以下工作}
            begin
                if sym = semicolon then getsym else error(10);
                {如果当前记号是分号,则取下一记号, 否则出错}
                statement([semicolon, endsym]+fsys) {处理下一个语句}
            end;
            if sym = endsym then getsym else error(17)
            {如果当前记号是"end",则取下一记号,否则出错}
        end
        else if sym = whilesym then {处理循环语句}
        begin
            cx1 := cx; {cx1记录下一指令地址,即条件表达式的第一条代码的地址}
            getsym; {取下一记号}
            condition([dosym]+fsys); {处理条件表达式}
            cx2 := cx; {记录下一指令的地址}
            gen(jpc, 0, 0); {生成一条指令,表达式为"假"转到某地址(待回填), 否则顺序执行}
            if sym = dosym then getsym else error(18);
            {如果当前记号是"do",则取下一记号, 否则出错}
            statement(fsys); {处理"do"后面的语句}
            gen(jmp, 0, cx1); {生成无条件转移指令, 转移到"while"后的条件表达式的代码的第一
条指令处}
            code[cx2].a := cx
            {把下一指令地址回填到前面生成的jpc指令的地址栏}
        end;
        test(fsys, [ ], 19)
        {测试下一记号是否正常, 否则出错, 跳过一些记号}
    end {statement};
```

```
begin {block}
    dx := 3; {本过程数据空间栈顶指针}
    tx0 := tx; {标识符表的长度(当前指针)}
    table[tx].adr := cx; {本过程名的地址，即下一条指令的序号}
    gen(jmp, 0, 0); {生成一条转移指令}
    if lev > levmax then error(32);
    {如果当前过程层号>最大层数，则出错}
    repeat
        if sym = constsym then {处理常数说明语句}
        begin
            getsym;
            repeat
                constdeclaration; {处理一个常数说明}
                while sym = comma do {如果当前记号是逗号}
                    begin
                        getsym;
                        constdeclaration
                    end;
                    {处理下一个常数说明}
                if sym = semicolon then getsym else error(5)
                {如果当前记号是分号,则常数说明已处理完，否则出错}
            until sym <> ident
        {跳过一些记号，直到当前记号不是标识符(出错时才用到)}
        end;
        {当前记号是变量说明语句开始符号}
        if sym = varsym then
        begin  getsym;
            repeat
                vardeclaration; {处理一个变量说明}
                while sym = comma do {如果当前记号是逗号}
                    begin
                        getsym;
                        vardeclaration
                    end;
                    {处理下一个变量说明}
                if sym = semicolon then getsym else error(5)
                {如果当前记号是分号,则变量说明已处理完，否则出错}
            until sym <> ident;
            {跳过一些记号，直到当前记号不是标识符(出错时才用到)}
        end;
        {处理过程说明}
        while sym = procsym do
        begin
            getsym;
            if sym = ident then {如果当前记号是过程名}
            begin
                enter(proceduretype);
                getsym
            end
            {把过程名填入符号表}
            else error(4); {否则，缺少过程名出错}
            if sym = semicolon then getsym else error(5);
            {当前记号是分号，则取下一记号,否则,过程名后漏掉分号出错}
            block(lev+1, tx, [semicolon]+fsys); {处理过程体}
```

```
            {lev+1: 过程嵌套层数加1; tx: 符号表当前栈顶指针,也是新过程符号表起始位置;
[semicolon]+fsys: 过程体开始和末尾符号集}
            if sym = semicolon then {如果当前记号是分号}
                begin
                    getsym; {取下一记号}
                    test(statbegsys+[ident, procsym], fsys, 6)
                    {测试当前记号是否语句开始符号或过程说明开始符号,否则报告错误6, 并跳
过一些记号}
                end
            else error(5) {如果当前记号不是分号,则出错}
        end; {while}
        test(statbegsys+[ident], declbegsys, 7)
        {检测当前记号是否语句开始符号, 否则出错, 并跳过一些记号}
    until not (sym in declbegsys);
    {回到说明语句的处理(出错时才用),直到当前记号不是说明语句的开始符号}
    code[table[tx0].adr].a := cx;
    {
        table[tx0].adr是本过程名的第1条代码(jmp, 0, 0)的地址,
        本语句即是将下一代码(本过程语句的第1条代码)的地址回填到该jmp指令中,得(jmp, 0,
cx)
    }
    with table[tx0] do {本过程名的第1条代码的地址改为下一指令地址cx}
    begin
        adr := cx; {代码开始地址}
    end;
    cx0 := cx; {cx0记录起始代码地址}
    gen(int, 0, dx); {生成一条指令, 在栈顶为本过程留出数据空间}
    statement([semicolon, endsym]+fsys); {处理一个语句}
    gen(opr, 0, 0); {生成返回指令}
    test(fsys, [ ], 8); {测试过程体语句后的符号是否正常,否则出错}
    listcode; {打印本过程的中间代码序列}
end  {block};


procedure  interpret;
const  stacksize = 500; {运行时数据空间(栈)的上界}
var  p, b, t : integer; {程序地址寄存器, 基地址寄存器,栈顶地址寄存器}
     i : instruction; {指令寄存器}
     s : array [1..stacksize] of integer; {数据存储栈}


function  base(l : integer) : integer;
var  b1 : integer;
begin
  b1 := b; {顺静态链求层差为l的外层的基地址}
  while l > 0 do
  begin
    b1 := s[b1];
    l := l-1
  end;
  base := b1
end {base};

begin
```

```
    writeln(fout, 'START PL/0');
    t := 0; {栈顶地址寄存器}
    b := 1; {基地址寄存器}
    p := 0; {程序地址寄存器}
    s[1] := 0; s[2] := 0;  s[3] := 0;
    {最外层主程序数据空间栈最下面预留三个单元}
    {
        每个过程运行时的数据空间的前三个单元是:SL, DL, RA;
        SL: 指向本过程静态直接外层过程的SL单元;
        DL: 指向调用本过程的过程的最新数据空间的第一个单元;
        RA: 返回地址
    }
    repeat
        i := code[p]; {i取程序地址寄存器p指示的当前指令}
        p := p+1; {程序地址寄存器p加1,指向下一条指令}
        with i do
            case f of
                lit :
                    begin {当前指令是取常数指令(lit, 0, a)}
                        t := t+1;
                        s[t] := a
                    end;
                {栈顶指针加1, 把常数a取到栈顶}
                opr :
                    case a of {当前指令是运算指令(opr, 0, a)}
                        0 :
                            begin {a=0时,是返回调用过程指令}
                                t := b-1; {恢复调用过程栈顶}
                                p := s[t+3]; {程序地址寄存器p取返回地址}
                                b := s[t+2];
                                {基地址寄存器b指向调用过程的基地址}
                            end;
                        1 : s[t] := -s[t]; {一元负运算, 栈顶元素的值反号}
                        2 :
                            begin {加法}
                            t := t-1;
                            s[t] := s[t] + s[t+1]
                            end;
                        3 :
                            begin {减法}
                                t := t-1;
                                s[t] := s[t]-s[t+1]
                            end;
                        4 :
                            begin {乘法}
                                t := t-1;
                                s[t] := s[t] * s[t+1]
                            end;
                        5 :
                            begin {整数除法}
                                t := t-1;
                                s[t] := s[t] div s[t+1]
                            end;
                        6 :
```

```
                        s[t] := ord(odd(s[t]));
                        {算s[t]是否奇数, 是则s[t]=1, 否则s[t]=0}
                    8 :
                        begin
                        t := t-1;
                        s[t] := ord(s[t] = s[t+1])
                        end;
                        {判两个表达式的值是否相等,是则s[t]=1, 否则s[t]=0}
                    9:
                        begin
                            t := t-1;
                            s[t] := ord(s[t] <> s[t+1])
                        end; {判两个表达式的值是否不等,是则s[t]=1, 否则s[t]=0}
                    10 :
                        begin
                            t := t-1;
                            s[t] := ord(s[t] < s[t+1])
                        end; {判前一表达式是否小于后一表达式,是则s[t]=1, 否则
s[t]=0}
                    11:
                        begin
                            t := t-1;
                            s[t] := ord(s[t] >= s[t+1])
                        end; {判前一表达式是否大于或等于后一表达式, 是则s[t]=1,
否则s[t]=0}
                    12 :
                        begin
                            t := t-1;
                            s[t] := ord(s[t] > s[t+1])
                        end; {判前一表达式是否大于后一表达式, 是则s[t]=1, 否则
s[t]=0}
                    13 :
                        begin
                            t := t-1;
                            s[t] := ord(s[t] <= s[t+1])
                        end; {判前一表达式是否小于或等于后一表达式, 是则s[t]=1,
否则s[t]=0}
                end;
            lod :
                begin {当前指令是取变量指令(lod, l, a)}
                    t := t + 1;
                    s[t] := s[base(l) + a]
                    {栈顶指针加1, 根据静态链SL,将层差为l,相对地址为a的变量值取到栈
顶}
                end;
            sto :
                begin {当前指令是保存变量值(sto, l, a)指令}
                    s[base(l) + a] := s[t];
                    writeln(fout, s[t] : 4);
                    {根据静态链SL,将栈顶的值存入层差为l,相对地址为a的变量中}
                    t := t-1 {栈顶指针减1}
                end;
            cal :
                begin {当前指令是(cal, l, a)}
```

```
                        {为被调用过程数据空间建立连接数据}
                        s[t+1] := base( l );
                        {根据层差l找到本过程的静态直接外层过程的数据空间的SL单元,
                        将其地址存入本过程新的数据空间的SL单元}
                        s[t+2] := b;
                        {调用过程的数据空间的起始地址存入本过程DL单元}
                        s[t+3] := p;
                        {调用过程cal指令的下一条的地址存入本过程RA单元}
                        b := t+1; {b指向被调用过程新的数据空间起始地址}
                        p := a {指令地址寄存储器指向被调用过程的地址a}
                    end;
                int : t := t + a;
                {若当前指令是(int, 0, a), 则数据空间栈顶留出a大小的空间}
                jmp : p := a;
                {若当前指令是(jmp, 0, a), 则程序转到地址a执行}
                jpc :
                    begin {当前指令是(jpc, 0, a)}
                        if s[t] = 0 then p := a;
                        {如果当前运算结果为"假"(0), 程序转到地址a执行, 否则顺序执行}
                        t := t-1 {数据栈顶指针减1}
                    end
            end {with, case}
    until p = 0;
    {程序一直执行到p取最外层主程序的返回地址0时为止}
    write(fout, 'END PL/0');
end {interpret};

begin  {主程序}
    assign(fin,paramstr(1));
    assign(fout,paramstr(2));   {将命令行参数str变量赋值给文件变量}
    reset(fin);
    rewrite(fout);      {打开输入输出文件}

    {ASCII码的顺序}
    For ch := 'A' To ';' Do ssym[ch] := nul;

    {保留字}
    word[1] := 'begin        ';
    word[2] := 'call         ';
    word[3] := 'const        ';
    word[4] := 'do           ';
    word[5] := 'end          ';
    word[6] := 'if           ';
    word[7] := 'odd          ';
    word[8] := 'procedure    ';
    word[9] := 'then         ';
    word[10] := 'var          ';
    word[11] := 'while        ';

    {保留字的记号}
    wsym[1] := beginsym;
    wsym[2] := callsym;
    wsym[3] := constsym;
    wsym[4] := dosym;
```

```
    wsym[5] := endsym;
    wsym[6] := ifsym;
    wsym[7] := oddsym;
    wsym[8] := procsym;
    wsym[9] := thensym;
    wsym[10] := varsym;
    wsym[11] := whilesym;

    {算符和标点符号的记号}
    ssym['+'] := plus;
    ssym['-'] := minus;
    ssym['*'] := times;
    ssym['/'] := slash;
    ssym['('] := lparen;
    ssym[')'] := rparen;
    ssym['='] := eql;
    ssym[','] := comma;
    ssym['.'] := period;
    ssym['<'] := lss;
    ssym['>'] := gtr;
    ssym[';'] := semicolon;

    {中间代码指令的字符串}
    mnemonic[lit] := 'LIT  ';
    mnemonic[opr] := 'OPR  ';
    mnemonic[lod] := 'LOD  ';
    mnemonic[sto] := 'STO  ';
    mnemonic[cal] := 'CAL  ';
    mnemonic[int] := 'INT  ';
    mnemonic[jmp] := 'JMP  ';
    mnemonic[jpc] := 'JPC  ';

    {说明语句的开始符号}
    declbegsys := [constsym, varsym, procsym];

    {语句的开始符号}
    statbegsys := [beginsym, callsym, ifsym, whilesym];

    {因子的开始符号}
    facbegsys := [ident, number, lparen];

    //page(output);

    err := 0; {发现错误的个数}
    cc := 0; {当前行中输入字符的指针}
    cx := 0; {代码数组的当前指针}
    ll := 0; {输入当前行的长度}
    ch := ' '; {当前输入的字符}
    kk := al; {标识符的长度}

    getsym; {取下一个记号}
    block(0, 0, [period]+declbegsys+statbegsys); {处理程序体}

    if sym <> period then error(9);
```

```
    {如果当前记号不是句号，则出错}

    if err = 0 then interpret
    {如果编译无错误，则解释执行中间代码}
    else write('ERRORS IN PL/0 PROGRAM');

    //99 : writeln
    writeln;
    close(fin);
    close(fout);
end.
```

## 2. PL0源程序代码

```
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;
    z := f
end;

begin
    x := m;
    y := n;
    call multiply;
    x := 25;
    y:= 3;
    call divide;
    x := 84;
    y := 36;
    call gcd;
end.
```

---

## 3.中间代码

```
    0 const m = 7, n = 85;
    1
    1 var x, y, z, q, r;
    1
    1 procedure  multiply;
    1 var  a, b;
    2 begin
    3     a := x;
    5     b := y;
    7     z := 0;
    9     while b > 0 do
   13     begin
   13         if odd b then
   15         z := z + a;
   20         a := 2*a ;
   24         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
30 procedure  divide;
30 var  w;
31 begin
32     r := x;
34     q := 0;
36     w := y;
38     while w <= r do w := 2*w ;
47     while w > y do
51     begin
51         q := 2*q;
55         w := w/2;
59         if w <= r then
62         begin
63             r := r-w;
67             q := q+1
69         end
71     end
71 end;
31  INT    0    4
32  LOD    1    3
33  STO    1    7
34  LIT    0    0
35  STO    1    6
36  LOD    1    4
37  STO    0    3
38  LOD    0    3
39  LOD    1    7
40  OPR    0   13
41  JPC    0   47
```

```
 42  LIT    0    2
 43  LOD    0    3
 44  OPR    0    4
 45  STO    0    3
 46  JMP    0   38
 47  LOD    0    3
 48  LOD    1    4
 49  OPR    0   12
 50  JPC    0   72
 51  LIT    0    2
 52  LOD    1    6
 53  OPR    0    4
 54  STO    1    6
 55  LOD    0    3
 56  LIT    0    2
 57  OPR    0    5
 58  STO    0    3
 59  LOD    0    3
 60  LOD    1    7
 61  OPR    0   13
 62  JPC    0   71
 63  LOD    1    7
 64  LOD    0    3
 65  OPR    0    3
 66  STO    1    7
 67  LOD    1    6
 68  LIT    0    1
 69  OPR    0    2
 70  STO    1    6
 71  JMP    0   47
 72  OPR    0    0
 73
 73 procedure  gcd;
 73 var  f, g ;
 74 begin
 75     f := x;
 77     g := y;
 79     while f <> g do
 83     begin
 83         if f < g then g := g-f;
 91         if g < f then f := f-g;
 99     end;
100     z := f
101 end;
 74  INT    0    5
 75  LOD    1    3
 76  STO    0    3
 77  LOD    1    4
 78  STO    0    4
 79  LOD    0    3
 80  LOD    0    4
 81  OPR    0    9
 82  JPC    0  100
 83  LOD    0    3
```

```
 84  LOD    0    4
 85  OPR    0   10
 86  JPC    0   91
 87  LOD    0    4
 88  LOD    0    3
 89  OPR    0    3
 90  STO    0    4
 91  LOD    0    4
 92  LOD    0    3
 93  OPR    0   10
 94  JPC    0   99
 95  LOD    0    3
 96  LOD    0    4
 97  OPR    0    3
 98  STO    0    3
 99  JMP    0   79
100  LOD    0    3
101  STO    1    5
102  OPR    0    0
103
103 begin
104     x := m;
106     y := n;
108     call multiply;
109     x := 25;
111     y:= 3;
113     call divide;
114     x := 84;
116     y := 36;
118     call gcd;
119 end.
103  INT    0    8
104  LIT    0    7
105  STO    0    3
106  LIT    0   85
107  STO    0    4
108  CAL    0    2
109  LIT    0   25
110  STO    0    3
111  LIT    0    3
112  STO    0    4
113  CAL    0   31
114  LIT    0   84
115  STO    0    3
116  LIT    0   36
117  STO    0    4
118  CAL    0   74
119  OPR    0    0
```

---

## 4. 栈中的数据

```
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
  25
   3
  25
   0
   3
   6
  12
  24
  48
   0
  24
   1
   1
   2
  12
   4
   6
   8
   3
  84
  36
  84
  36
  48
  12
  24
  12
  12
END PL/0
```

# 第二部分

##1. 编译程序源代码

```
program  PL0 ( input, output);
{带有代码生成的PL0编译程序}

//label  99;

const
  norw = 13; {保留字的个数}
  txmax = 100; {标识符表长度}
  nmax = 14; {数字的最大位数}
  al = 10; {标识符的长度}
  amax = 2047; {最大地址}
  levmax = 3; {程序体嵌套的最大深度}
  cxmax = 200; {代码数组的大小}

type
  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, dosym, callsym, constsym, varsym, procsym, readsym,
writesym);
  alfa = packed array [1..al] of char;
  objecttype = (constant, variable, proceduretype);
  symset = set of symbol;
  fct = (lit, opr, lod, sto, cal, int, jmp, jpc, red, wrt); {functions}
  {
    LIT 0,a : 取常数a
    OPR 0,a : 执行运算a
    LOD l,a : 取层差为l的层、相对地址为a的变量
    STO l,a : 存到层差为l的层、相对地址为a的变量
    CAL l,a : 调用层差为l的过程
    INT 0,a : t寄存器增加a
    JMP 0,a : 转移到指令地址a处
    JPC 0,a : 条件转移到指令地址a处
    RED
    WRT
  }
  instruction = packed record
        f : fct;  {功能码}
        l : 0..levmax; {相对层数}
        a : 0..amax; {相对地址}
  end;

var
  ch : char; {最近读到的字符}
  sym : symbol; {最近读到的符号}
  id : alfa; {最近读到的标识符}
```

```
  num : integer; {最近读到的数}
  cc : integer; {当前行的字符计数}
  ll : integer; {当前行的长度}
  kk, err : integer;
  cx : integer; {代码数组的当前下标}
  line : array [1..81] of char; {当前行}
  a : alfa; {当前标识符的字符串}
  code : array [0..cxmax] of instruction; {中间代码数组}
  word : array [1..norw] of alfa; {存放保留字的字符串}
  wsym : array [1..norw] of symbol; {存放保留字的记号}
  ssym : array [char] of symbol; {存放算符和标点符号的记号}
  mnemonic : array [fct] of packed array [1..5] of char;
  {中间代码算符的字符串}
  declbegsys, statbegsys, facbegsys : symset;
  table : array [0..txmax] of {符号表}
         record
           name : alfa;
           case kind : objecttype of
            constant : (val : integer);
            variable, proceduretype : (level, adr : integer)
         end;
  fin : text;       {源代码文件}
  fout : text;      {输出文件}

procedure error (n : integer);
begin
  writeln(fout, '****', ' ':cc-1, '^', n:2);  {cc为当前行已读的字符数, n为错误号}
  err := err + 1{错误数err加1}
end {error};

procedure getsym;
var  i, j, k : integer;

procedure  getch ;
begin
    if cc = ll then {如果cc指向行末}
    begin
        if eof(fin) then {如果已到文件尾}
        begin
            writeln(fout, 'PROGRAM INCOMPLETE');
            close(fin);
            close(fout);
            exit;
            //goto 99
        end;
        {读新的一行}
        ll := 0;
        cc := 0;
        write(fout, cx : 5, ' ');{cx : 5位数}
        while not eoln(fin) do {如果不是行末}
        begin
            ll := ll + 1; {将行缓冲区的长度+1}
            read(fin, ch); {从源文件中读取一个字符到ch中}
            write(fout, ch);{输出ch到输出文件中}
```

```
            line[ll] := ch  {把这个字符放到当前行末尾}
        end;
        writeln(fout); {换行}
        readln(fin);{从源文件下一行开始读取}
        ll := ll + 1; {将行缓冲区的长度+1}
        line[ll] := ' ' { process end-line }    {行数组最后一个元素为空格}
    end;
    cc := cc + 1;
    ch := line[cc]  {ch取line中下一个字符}
end {getch};


begin {getsym}
    while ch = ' ' do getch; {跳过无用空白}

    if ch in ['a'..'z'] then
    begin {标识符或保留字}
        k := 0;
        repeat {处理字母开头的字母、数字串}
            if k < al then
                begin
                    k:= k + 1;
                    a[k] := ch
                end;
            getch
        until  not (ch in ['a'..'z', '0'..'9']);
        if k >= kk  then kk := k
        else
            repeat
                a[kk] := ' ';
                kk := kk-1 {如果标识符长度不是最大长度，后面补空白}
            until kk = k;
        {id中存放当前标识符或保留字的字符串}
        id := a;  i := 1;  j := norw;

        repeat
            k := (i+j) div 2;
            {用二分查找法在保留字表中找当前的标识符id}
            if id <= word[k] then j := k-1;
            if id >= word[k] then i := k + 1
        until i > j;

        {如果找到，当前记号sym为保留字，否则sym为标识符}
        if i-1 > j then
            sym := wsym[k]
        else
            sym := ident
    end

    else if ch in ['0'..'9'] then
    begin {数字}
        k := 0;  num := 0;  sym := number; {当前记号sym为数字}
        repeat {计算数字串的值}
            num := 10*num + (ord(ch)-ord('0'));
```

```
            {ord(ch)和ord(0)是ch和0在ASCII码中的序号}
            k := k + 1;
            getch;
        until not(ch in ['0'..'9']);
        if k > nmax then error(30)
        {当前数字串的长度超过上界,则报告错误}
    end

    else if ch = ':' then {处理赋值号}
    begin
        getch;
        if ch = '=' then
            begin
                sym := becomes;
                getch
            end
        else
            sym := nul;
    end

    else if ch = '<' then
    begin {处理'<'}
        getch;
        if ch = '=' then {'<='}
        begin
            sym := leq; {表示小于等于}
            getch
        end
        else if ch = '>' then {'<>'}
        begin
            sym := neq; {表示不等于}
            getch
        end
        else sym := lss {表示小于}
    end

    else if ch = '>' then
    begin {处理'>'}
        getch;
        if ch = '=' then {'>='}
        begin
            sym := geq; {表示大于等于}
            getch
        end
        else sym := gtr {表示大于}
    end

    {处理其它算符或标点符号}
    else
    begin
        sym := ssym[ch];
        getch
    end
```

```
end {getsym};


procedure  gen(x : fct; y, z : integer);
begin
    {如果当前指令序号>代码的最大长度}
    if cx > cxmax then
        begin
            write(fout, 'PROGRAM TOO LONG');
            close(fin);
            close(fout);
            exit
            //goto 99
        end;
        with code[cx] do {在代码数组cx位置生成一条新代码}
        begin
            f := x; {功能码}
            l := y; {层号}
            a := z {地址}
        end;
    cx := cx + 1 {指令序号加1}
end {gen};


procedure  test(s1, s2 : symset; n : integer);
begin
    if not (sym in s1) then
    {如果当前记号不属于集合S1,则报告错误n}
    begin
        error(n);
        s1 := s1 + s2;
        while not (sym in s1) do
            getsym
        {跳过一些记号, 直到当前记号属于S1US2}
    end
end {test};


procedure  block(lev, tx : integer; fsys : symset);
var
    dx : integer; {本过程数据空间分配下标}
    tx0 : integer; {本过程标识表起始下标}
    cx0 : integer; {本过程代码起始下标}

procedure  enter(k : objecttype);
begin {把objecttype填入符号表中}
    tx := tx + 1; {符号表指针加1}
    with table[tx] do {在符号表中增加新的一个条目}
    begin
        name := id; {当前标识符的名字}
        kind := k; {当前标识符的种类}
        case k of
            {当前标识符是常数名}
            constant :
```

```
                begin
                    if num > amax then {当前常数值大于上界,则出错}
                    begin
                        error(30);
                        num := 0
                    end;
                    val := num
                end;
            {当前标识符是变量名}
            variable :
                begin
                    level := lev; {定义该变量的过程的嵌套层数}
                    adr := dx; {变量地址为当前过程数据空间栈顶}
                    dx := dx +1; {栈顶指针加1}
                end;
            proceduretype :
                level := lev; {本过程的嵌套层数}
        end
    end
end {enter};


function  position(id : alfa) : integer; {返回id在符号表的入口}
var  i : integer;
begin {在标识符表中查标识符id}
    table[0].name := id; {在符号表栈的最下方预填标识符id}
    i := tx; {符号表栈顶指针}
    while table[i].name <> id do i := i-1;
    {从符号表栈顶往下查标识符id}
    position := i {若查到,i为id的入口,否则i=0 }
end {position};


procedure constdeclaration;
begin
    if sym = ident then {当前记号是常数名}
    begin
        getsym;
        if sym in [eql, becomes] then {当前记号是等号或赋值号}
        begin
            if sym = becomes then error(1);
            {如果当前记号是赋值号,则出错}
            getsym;
            if sym = number then {等号后面是常数}
            begin
                enter(constant); {将常数名加入符号表}
                getsym
            end
            else error(2) {等号后面不是常数出错}
        end
        else error(3) {标识符后不是等号或赋值号出错}
    end
    else error(4) {常数说明中没有常数名标识符}
end {constdeclaration};
```

```
procedure  vardeclaration;
begin
    if sym = ident then {如果当前记号是标识符}
    begin
        enter(variable); {将该变量名加入符号表的下一条目}
        getsym
    end
    else error(4) {如果变量说明未出现标识符,则出错}
end {vardeclaration};


procedure  listcode;
var  i : integer;
begin  {列出本程序体生成的代码}
    for i := cx0 to cx-1 do {cx0: 本过程第一个代码的序号, cx-1: 本过程最后一个代码的序
号}
    with code[i] do {打印第i条代码}
        writeln(fout, i:4, mnemonic[f]:7, l:3, a:5)
        {
            i: 代码序号;
            mnemonic[f]: 功能码的字符串;
            l: 相对层号(层差);
            a: 相对地址或运算号码
        }
end {listcode};


procedure  statement(fsys : symset);
var i, cx1, cx2 : integer;


procedure  expression(fsys : symset);
var  addop : symbol;


procedure  term(fsys : symset);
var  mulop : symbol;


procedure  factor(fsys : symset);
var i : integer;
begin
    test(facbegsys, fsys, 24);
    {测试当前的记号是否因子的开始符号, 否则出错, 跳过一些记号}
    while sym in facbegsys do
    {如果当前的记号是否因子的开始符号}
    begin
        {当前记号是标识符}
        if sym = ident then
        begin
            i := position(id); {查符号表,返回id的入口}
            if i = 0 then
```

```
                error(11)
            else
                {若在符号表中查不到id, 则出错, 否则,做以下工作}
                with table[i] do
                case kind of
                    constant : gen(lit, 0, val); {若id是常数, 生成指令,将常数val取到
栈顶}
                    variable : gen(lod, lev-level, adr);
                    {
                        若id是变量, 生成指令,将该变量取到栈顶;
                        lev: 当前语句所在过程的层号;
                        level: 定义该变量的过程层号;
                        adr: 变量在其过程的数据空间的相对地址
                    }
                    proceduretype : error(21)
                    {若id是过程名, 则出错}
                end;
                getsym {取下一记号}
        end
        else if sym = number then {当前记号是数字}
        begin
            if num > amax then {若数值越界,则出错}
            begin
                error(30);
                num := 0
            end;
            gen(lit, 0, num);
            {生成一条指令, 将常数num取到栈顶}
            getsym {取下一记号}
        end
        else if sym = lparen then {如果当前记号是左括号}
        begin
            getsym; {取下一记号}
            expression([rparen]+fsys); {处理表达式}
            if sym = rparen then
                getsym
                {如果当前记号是右括号, 则取下一记号,否则出错}
            else error(22)
        end;
        test(fsys, [lparen], 23)
        {测试当前记号是否同步, 否则出错, 跳过一些记号}
    end {while}
end {factor};


begin {term}
    factor(fsys+[times, slash]); {处理项中第一个因子}
    while sym in [times, slash] do
    {当前记号是“乘”或“除”号}
    begin
        mulop := sym; {运算符存入mulop}
        getsym; {取下一记号}
        factor(fsys+[times, slash]); {处理一个因子}
        if mulop = times then gen(opr, 0, 4)
```

```
        {若mulop是“乘”号,生成一条乘法指令}
        else gen(opr, 0, 5)
        {否则, mulop是除号, 生成一条除法指令}
    end
end {term};


begin {expression}
    if sym in [plus, minus] then {若第一个记号是加号或减号}
    begin
        addop := sym;  {“+”或“-”存入addop}
        getsym;
        term(fsys+[plus, minus]); {处理一个项}
        if addop = minus then gen(opr, 0, 1)
        {若第一个项前是负号, 生成一条“负运算”指令}
    end
    else term(fsys+[plus, minus]);
    {第一个记号不是加号或减号, 则处理一个项}
    while sym in [plus, minus] do {若当前记号是加号或减号}
    begin
        addop := sym; {当前算符存入addop}
        getsym; {取下一记号}
        term(fsys+[plus, minus]); {处理一个项}
        if addop = plus then gen(opr, 0, 2)
        {若addop是加号, 生成一条加法指令}
        else gen(opr, 0, 3)
        {否则, addop是减号, 生成一条减法指令}
    end
end {expression};


procedure  condition(fsys : symset);
var  relop : symbol;
begin
    if sym = oddsym then {如果当前记号是“odd”}
    begin
        getsym;  {取下一记号}
        expression(fsys); {处理算术表达式}
        gen(opr, 0, 6) {生成指令,判定表达式的值是否为奇数,是,则取“真”;不是, 则取“假”}
    end
    else {如果当前记号不是“odd”}
    begin
        expression([eql, neq, lss, gtr, leq, geq] + fsys);
        {处理算术表达式}
        if not (sym in [eql, neq, lss, leq, gtr, geq]) then
        {如果当前记号不是关系符, 则出错; 否则,做以下工作}
            error(20)
        else
        begin
            relop := sym; {关系符存入relop}
            getsym; {取下一记号}
            expression(fsys); {处理关系符右边的算术表达式}
            case relop of
                eql : gen(opr, 0, 8);  {生成指令, 判定两个表达式的值是否相等}
```

```
                neq : gen(opr, 0, 9);  {生成指令, 判定两个表达式的值是否不等}
                lss : gen(opr, 0, 10); {生成指令,判定前一表达式是否小于后一表达式}
                geq : gen(opr, 0, 11); {生成指令,判定前一表达式是否大于等于后一表达式}
                gtr : gen(opr, 0, 12); {生成指令,判定前一表达式是否大于后一表达式}
                leq : gen(opr, 0, 13); {生成指令,判定前一表达式是否小于等于后一表达式}
            end
        end
    end
end {condition};

begin {statement}
    if sym = ident then {处理赋值语句}
    begin
        i := position(id);
        {在符号表中查id, 返回id在符号表中的入口}
        if i = 0 then error(11)
        {若在符号表中查不到id, 则出错, 否则做以下工作}
        else if table[i].kind <> variable then
        {若标识符id不是变量, 则出错}
        begin {对非变量赋值}
            error(12);
            i := 0;
        end;
        getsym; {取下一记号}
        if sym = becomes then getsym else error(13);
        {若当前是赋值号, 取下一记号, 否则出错}
        expression(fsys); {处理表达式}
        if i <> 0 then {若赋值号左边的变量id有定义}
            with table[i] do
                gen(sto, lev-level, adr)
                {
                    生成一条存数指令, 将栈顶(表达式)的值存入变量id中;
                    lev: 当前语句所在过程的层号;
                    level: 定义变量id的过程的层号;
                    adr: 变量id在其过程的数据空间的相对地址
                }
    end
    else if sym = callsym then {处理过程调用语句}
    begin
        getsym; {取下一记号}
        if sym <> ident then error(14) else
        {如果下一记号不是标识符(过程名),则出错, 否则做以下工作}
        begin
            i := position(id); {查符号表,返回id在表中的位置}
            if i = 0 then error(11) else
            {如果在符号表中查不到, 则出错; 否则,做以下工作}
                with table[i] do
                    if kind = proceduretype then
                        {如果在符号表中id是过程名}
                        gen(cal, lev-level, adr)
                        {
                            生成一条过程调用指令;
                            lev: 当前语句所在过程的层号
                            level: 定义过程名id的层号;
```

```
                                adr: 过程id的代码中第一条指令的地址
                            }
                        else error(15); {若id不是过程名,则出错}
                        getsym {取下一记号}
            end
        end
        else if sym = ifsym then {处理条件语句}
        begin
            getsym; {取下一记号}
            condition([thensym, dosym]+fsys); {处理条件表达式}
            if sym = thensym then getsym else error(16);
            {如果当前记号是“then”,则取下一记号; 否则出错}
            cx1 := cx; {cx1记录下一代码的地址}
            gen(jpc, 0, 0); {生成指令,表达式为“假”转到某地址(待填),否则顺序执行}
            statement(fsys); {处理一个语句}
            code[cx1].a := cx {将下一个指令的地址回填到上面的jpc指令地址栏}
        end
        else if sym = beginsym then {处理语句序列}
        begin
            getsym;
            statement([semicolon, endsym]+fsys);
            {取下一记号, 处理第一个语句}
            while sym in [semicolon]+statbegsys do
            {如果当前记号是分号或语句的开始符号,则做以下工作}
            begin
                if sym = semicolon then getsym else error(10);
                {如果当前记号是分号,则取下一记号, 否则出错}
                statement([semicolon, endsym]+fsys) {处理下一个语句}
            end;
            if sym = endsym then getsym else error(17)
            {如果当前记号是“end”,则取下一记号,否则出错}
        end
        else if sym = whilesym then {处理循环语句}
        begin
            cx1 := cx; {cx1记录下一指令地址,即条件表达式的第一条代码的地址}
            getsym; {取下一记号}
            condition([dosym]+fsys); {处理条件表达式}
            cx2 := cx; {记录下一指令的地址}
            gen(jpc, 0, 0); {生成一条指令,表达式为“假”转到某地址(待回填), 否则顺序执行}
            if sym = dosym then getsym else error(18);
            {如果当前记号是“do”,则取下一记号, 否则出错}
            statement(fsys); {处理“do”后面的语句}
            gen(jmp, 0, cx1); {生成无条件转移指令, 转移到“while”后的条件表达式的代码的第一
条指令处}
            code[cx2].a := cx
            {把下一指令地址回填到前面生成的jpc指令的地址栏}
        end
        else if sym = readsym then {处理read关键字}
        begin
            getsym;
            if sym = lparen then
            begin
                repeat
                    getsym;
```

```
                if sym = ident then {如果sym是标识符}
                begin
                    i := position(id); {记录当前符号在符号表中的位置}
                    if i = 0 then error(11) {如果i为0,说明符号表中没有找到id对应的符
号}
                    else if table[i].kind <> variable then
                    begin
                        error(12);
                        i := 0
                    end
                    {如果是变量类型，生成一条red指令，读取数据}
                    else with table[i] do
                        gen(red, lev-level, adr)
                end
                else error(4);
                getsym;
            until sym <> comma
        end
        else error(40);
        if sym <> rparen then error(22);
        getsym
    end

    else if sym = writesym then {处理write关键字}
    begin
        getsym;
        if sym = lparen then
        begin
            repeat
                getsym;
                expression([rparen, comma]+fsys);
                gen(wrt, 0, 0);
            until sym <> comma;
            if sym <> rparen then error(22);
            getsym
        end
        else error(40)
    end;
    test(fsys, [ ], 19)
    {测试下一记号是否正常，否则出错，跳过一些记号}
end {statement};

begin {block}
    dx := 3; {本过程数据空间栈顶指针}
    tx0 := tx; {标识符表的长度(当前指针)}
    table[tx].adr := cx; {本过程名的地址，即下一条指令的序号}
    gen(jmp, 0, 0); {生成一条转移指令}
    if lev > levmax then error(32);
    {如果当前过程层号>最大层数，则出错}
    repeat
        if sym = constsym then {处理常数说明语句}
        begin
            getsym;
            repeat
```

```
                constdeclaration; {处理一个常数说明}
                while sym = comma do {如果当前记号是逗号}
                    begin
                        getsym;
                        constdeclaration
                    end;
                    {处理下一个常数说明}
                if sym = semicolon then getsym else error(5)
                {如果当前记号是分号,则常数说明已处理完, 否则出错}
            until sym <> ident
        {跳过一些记号, 直到当前记号不是标识符(出错时才用到)}
        end;
        {当前记号是变量说明语句开始符号}
        if sym = varsym then
        begin  getsym;
            repeat
                vardeclaration; {处理一个变量说明}
                while sym = comma do {如果当前记号是逗号}
                    begin
                        getsym;
                        vardeclaration
                    end;
                    {处理下一个变量说明}
                if sym = semicolon then getsym else error(5)
                {如果当前记号是分号,则变量说明已处理完, 否则出错}
            until sym <> ident;
            {跳过一些记号, 直到当前记号不是标识符(出错时才用到)}
        end;
        {处理过程说明}
        while sym = procsym do
        begin
            getsym;
            if sym = ident then {如果当前记号是过程名}
            begin
                enter(proceduretype);
                getsym
            end
            {把过程名填入符号表}
            else error(4); {否则, 缺少过程名出错}
            if sym = semicolon then getsym else error(5);
            {当前记号是分号, 则取下一记号,否则,过程名后漏掉分号出错}
            block(lev+1, tx, [semicolon]+fsys); {处理过程体}
            {lev+1: 过程嵌套层数加1; tx: 符号表当前栈顶指针,也是新过程符号表起始位置;
[semicolon]+fsys: 过程体开始和末尾符号集}
            if sym = semicolon then {如果当前记号是分号}
            begin
                getsym; {取下一记号}
                test(statbegsys+[ident, procsym], fsys, 6)
                {测试当前记号是否语句开始符号或过程说明开始符号,否则报告错误6, 并跳过一
些记号}
            end
            else error(5) {如果当前记号不是分号,则出错}
        end; {while}
        test(statbegsys+[ident], declbegsys, 7)
```

```
        {检测当前记号是否语句开始符号，否则出错，并跳过一些记号}
    until not (sym in declbegsys);
    {回到说明语句的处理(出错时才用),直到当前记号不是说明语句的开始符号}
    code[table[tx0].adr].a := cx;
    {
        table[tx0].adr是本过程名的第1条代码(jmp，0，0)的地址，
        本语句即是将下一代码(本过程语句的第1条代码)的地址回填到该jmp指令中,得(jmp，0，
cx)
    }
    with table[tx0] do {本过程名的第1条代码的地址改为下一指令地址cx}
    begin
        adr := cx; {代码开始地址}
    end;
    cx0 := cx; {cx0记录起始代码地址}
    gen(int，0，dx); {生成一条指令，在栈顶为本过程留出数据空间}
    statement([semicolon，endsym]+fsys); {处理一个语句}
    gen(opr，0，0); {生成返回指令}
    test(fsys，[ ]，8); {测试过程体语句后的符号是否正常,否则出错}
    listcode; {打印本过程的中间代码序列}
end  {block};


procedure  interpret;
const  stacksize = 500; {运行时数据空间(栈)的上界}
var  p，b，t : integer; {程序地址寄存器，基地址寄存器,栈顶地址寄存器}
     i : instruction; {指令寄存器}
     s : array [1..stacksize] of integer; {数据存储栈}


function  base(l : integer) : integer;
var  b1 : integer;
begin
  b1 := b; {顺静态链求层差为l的外层的基地址}
  while l > 0 do
  begin
    b1 := s[b1];
    l := l-1
  end;
  base := b1
end {base};

begin
    writeln(fout，'START PL/0');
    t := 0; {栈顶地址寄存器}
    b := 1; {基地址寄存器}
    p := 0; {程序地址寄存器}
    s[1] := 0; s[2] := 0;  s[3] := 0;
    {最外层主程序数据空间栈最下面预留三个单元}
    {
        每个过程运行时的数据空间的前三个单元是:SL，DL，RA;
        SL: 指向本过程静态直接外层过程的SL单元;
        DL: 指向调用本过程的过程的最新数据空间的第一个单元;
        RA: 返回地址
    }
```

```
    repeat
        i := code[p]; {i取程序地址寄存器p指示的当前指令}
        p := p+1; {程序地址寄存器p加1,指向下一条指令}
        with i do
            case f of
                lit :
                    begin {当前指令是取常数指令(lit, 0, a)}
                        t := t+1;
                        s[t] := a
                    end;
                {栈顶指针加1, 把常数a取到栈顶}
                opr :
                    case a of {当前指令是运算指令(opr, 0, a)}
                        0 :
                            begin {a=0时,是返回调用过程指令}
                                t := b-1; {恢复调用过程栈顶}
                                p := s[t+3]; {程序地址寄存器p取返回地址}
                                b := s[t+2];
                                {基地址寄存器b指向调用过程的基地址}
                            end;
                        1 : s[t] := -s[t]; {一元负运算, 栈顶元素的值反号}
                        2 :
                            begin {加法}
                            t := t-1;
                            s[t] := s[t] + s[t+1]
                            end;
                        3 :
                            begin {减法}
                                t := t-1;
                                s[t] := s[t]-s[t+1]
                            end;
                        4 :
                            begin {乘法}
                                t := t-1;
                                s[t] := s[t] * s[t+1]
                            end;
                        5 :
                            begin {整数除法}
                                t := t-1;
                                s[t] := s[t] div s[t+1]
                            end;
                        6 :
                            s[t] := ord(odd(s[t]));
                            {算s[t]是否奇数, 是则s[t]=1, 否则s[t]=0}
                        8 :
                            begin
                            t := t-1;
                            s[t] := ord(s[t] = s[t+1])
                            end;
                            {判两个表达式的值是否相等,是则s[t]=1, 否则s[t]=0}
                        9:
                            begin
                                t := t-1;
                                s[t] := ord(s[t] <> s[t+1])
```

```
                        end; {判两个表达式的值是否不等,是则s[t]=1, 否则s[t]=0}
                    10 :
                        begin
                            t := t-1;
                            s[t] := ord(s[t] < s[t+1])
                        end; {判前一表达式是否小于后一表达式,是则s[t]=1, 否则
s[t]=0}
                    11:
                        begin
                            t := t-1;
                            s[t] := ord(s[t] >= s[t+1])
                        end; {判前一表达式是否大于或等于后一表达式, 是则s[t]=1,
否则s[t]=0}
                    12 :
                        begin
                            t := t-1;
                            s[t] := ord(s[t] > s[t+1])
                        end; {判前一表达式是否大于后一表达式, 是则s[t]=1, 否则
s[t]=0}
                    13 :
                        begin
                            t := t-1;
                            s[t] := ord(s[t] <= s[t+1])
                        end; {判前一表达式是否小于或等于后一表达式, 是则s[t]=1,
否则s[t]=0}
                end;
            lod :
                begin {当前指令是取变量指令(lod, l, a)}
                    t := t + 1;
                    s[t] := s[base(l) + a]
                    {栈顶指针加1, 根据静态链SL,将层差为l,相对地址为a的变量值取到栈
顶}
                end;
            sto :
                begin {当前指令是保存变量值(sto, l, a)指令}
                    s[base(l) + a] := s[t];
                    // writeln(fout, s[t] : 4);
                    {根据静态链SL,将栈顶的值存入层差为l,相对地址为a的变量中}
                    t := t-1 {栈顶指针减1}
                end;
            cal :
                begin {当前指令是(cal, l, a)}
                {为被调用过程数据空间建立连接数据}
                    s[t+1] := base( l );
                    {根据层差l找到本过程的静态直接外层过程的数据空间的SL单元,
                    将其地址存入本过程新的数据空间的SL单元}
                    s[t+2] := b;
                    {调用过程的数据空间的起始地址存入本过程DL单元}
                    s[t+3] := p;
                    {调用过程cal指令的下一条的地址存入本过程RA单元}
                    b := t+1; {b指向被调用过程新的数据空间起始地址}
                    p := a {指令地址寄存储器指向被调用过程的地址a}
                end;
            int : t := t + a;
```

```
                {若当前指令是(int, 0, a), 则数据空间栈顶留出a大小的空间}
                jmp : p := a;
                {若当前指令是(jmp, 0, a), 则程序转到地址a执行}
                jpc :
                    begin {当前指令是(jpc, 0, a)}
                        if s[t] = 0 then p := a;
                        {如果当前运算结果为“假”(0), 程序转到地址a执行, 否则顺序执行}
                        t := t-1 {数据栈顶指针减1}
                    end;
                red :
                    begin
                        writeln('please input a number:');
                        readln(s[base(l)+a]);{读一行数据,读入到相差l层,层内偏移为a的
数据栈中的数据的信息}
                    end;
                wrt :
                    begin
                        writeln(fout, s[t]); {输出栈顶的信息}
                        t := t + 1 {栈顶上移}
                    end
            end {with, case}
    until p = 0;
    {程序一直执行到p取最外层主程序的返回地址0时为止}
    write(fout, 'END PL/0');
end {interpret};

begin  {主程序}
    assign(fin,paramstr(1));
    assign(fout,paramstr(2));   {将命令行参数str变量赋值给文件变量}
    reset(fin);
    rewrite(fout);      {打开输入输出文件}

    {ASCII码的顺序}
    For ch := 'A' To ';' Do ssym[ch] := nul;

    {保留字}
    word[1] := 'begin        ';
    word[2] := 'call         ';
    word[3] := 'const        ';
    word[4] := 'do           ';
    word[5] := 'end          ';
    word[6] := 'if           ';
    word[7] := 'odd          ';
    word[8] := 'procedure    ';
    word[9] := 'read         ';
    word[10] := 'then         ';
    word[11] := 'var          ';
    word[12] := 'while        ';
    word[13] := 'write        ';

    {保留字的记号}
    wsym[1] := beginsym;
    wsym[2] := callsym;
    wsym[3] := constsym;
```

```
    wsym[4] := dosym;
    wsym[5] := endsym;
    wsym[6] := ifsym;
    wsym[7] := oddsym;
    wsym[8] := procsym;
    wsym[9] :=  readsym;
    wsym[10] := thensym;
    wsym[11] := varsym;
    wsym[12] := whilesym;
    wsym[13] := writesym;

    {算符和标点符号的记号}
    ssym['+'] := plus;
    ssym['-'] := minus;
    ssym['*'] := times;
    ssym['/'] := slash;
    ssym['('] := lparen;
    ssym[')'] := rparen;
    ssym['='] := eql;
    ssym[','] := comma;
    ssym['.'] := period;
    ssym['<'] := lss;
    ssym['>'] := gtr;
    ssym[';'] := semicolon;

    {中间代码指令的字符串}
    mnemonic[lit] := 'LIT  ';
    mnemonic[opr] := 'OPR  ';
    mnemonic[lod] := 'LOD  ';
    mnemonic[sto] := 'STO  ';
    mnemonic[cal] := 'CAL  ';
    mnemonic[int] := 'INT  ';
    mnemonic[jmp] := 'JMP  ';
    mnemonic[jpc] := 'JPC  ';
    mnemonic[red] := 'RED  ';
    mnemonic[wrt] := 'WRT  ';

    {说明语句的开始符号}
    declbegsys := [constsym, varsym, procsym];

    {语句的开始符号}
    statbegsys := [beginsym, callsym, ifsym, whilesym];

    {因子的开始符号}
    facbegsys := [ident, number, lparen];

    //page(output);

    err := 0; {发现错误的个数}
    cc := 0; {当前行中输入字符的指针}
    cx := 0; {代码数组的当前指针}
    ll := 0; {输入当前行的长度}
    ch := ' '; {当前输入的字符}
    kk := al; {标识符的长度}
```

```
    getsym; {取下一个记号}
    block(0, 0, [period]+declbegsys+statbegsys); {处理程序体}

    if sym <> period then error(9);
    {如果当前记号不是句号，则出错}

    if err = 0 then interpret
    {如果编译无错误，则解释执行中间代码}
    else write('ERRORS IN PL/0 PROGRAM');

    //99 : writeln
    writeln;
    close(fin);
    close(fout);
end.
```

---

## 2. PL0 源程序代码

```
const m = 7, n = 85;

var x, y, z, q, r, z1;

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;
    z1 := f
end;

begin
    read(x,y);  call multiply;
    read(x,y);  call divide;
    read(x,y);  call gcd;
    write(z,q,r,z1);
end.
```

---

## 3. 中间代码

```
    0 const m = 7, n = 85;
    1
    1 var x, y, z, q, r, z1;
    1
    1 procedure  multiply;
    1 var  a, b;
    2 begin
    3     a := x;
    5     b := y;
    7     z := 0;
    9     while b > 0 do
   13     begin
   13         if odd b then
   15         z := z + a;
   20         a := 2*a ;
   24         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
 30 procedure  divide;
 30 var  w;
 31 begin
 32     r := x;
 34     q := 0;
 36     w := y;
 38     while w <= r do w := 2*w ;
 47     while w > y do
 51     begin
 51         q := 2*q;
 55         w := w/2;
 59         if w <= r then
 62         begin
 63             r := r-w;
 67             q := q+1
 69         end
 71     end
 71 end;
31  INT    0    4
32  LOD    1    3
33  STO    1    7
34  LIT    0    0
35  STO    1    6
36  LOD    1    4
37  STO    0    3
38  LOD    0    3
39  LOD    1    7
40  OPR    0   13
```

```
 41  JPC    0   47
 42  LIT    0    2
 43  LOD    0    3
 44  OPR    0    4
 45  STO    0    3
 46  JMP    0   38
 47  LOD    0    3
 48  LOD    1    4
 49  OPR    0   12
 50  JPC    0   72
 51  LIT    0    2
 52  LOD    1    6
 53  OPR    0    4
 54  STO    1    6
 55  LOD    0    3
 56  LIT    0    2
 57  OPR    0    5
 58  STO    0    3
 59  LOD    0    3
 60  LOD    1    7
 61  OPR    0   13
 62  JPC    0   71
 63  LOD    1    7
 64  LOD    0    3
 65  OPR    0    3
 66  STO    1    7
 67  LOD    1    6
 68  LIT    0    1
 69  OPR    0    2
 70  STO    1    6
 71  JMP    0   47
 72  OPR    0    0
  73
  73 procedure  gcd;
  73 var  f, g ;
  74 begin
  75     f := x;
  77     g := y;
  79     while f <> g do
  83     begin
  83         if f < g then g := g-f;
  91         if g < f then f := f-g;
  99     end;
 100     z1 := f
 101 end;
 74  INT    0    5
 75  LOD    1    3
 76  STO    0    3
 77  LOD    1    4
 78  STO    0    4
 79  LOD    0    3
 80  LOD    0    4
 81  OPR    0    9
 82  JPC    0  100
```

```
 83  LOD    0    3
 84  LOD    0    4
 85  OPR    0   10
 86  JPC    0   91
 87  LOD    0    4
 88  LOD    0    3
 89  OPR    0    3
 90  STO    0    4
 91  LOD    0    4
 92  LOD    0    3
 93  OPR    0   10
 94  JPC    0   99
 95  LOD    0    3
 96  LOD    0    4
 97  OPR    0    3
 98  STO    0    3
 99  JMP    0   79
100  LOD    0    3
101  STO    1    8
102  OPR    0    0
 103
 103 begin
 104     read(x,y);  call multiply;
 107     read(x,y);  call divide;
 110     read(x,y);  call gcd;
 113     write(z,q,r,z1);
 121 end.
103  INT    0    9
104  RED    0    3
105  RED    0    4
106  CAL    0    2
107  RED    0    3
108  RED    0    4
109  CAL    0   31
110  RED    0    3
111  RED    0    4
112  CAL    0   74
113  LOD    0    5
114  WRT    0    0
115  LOD    0    6
116  WRT    0    0
117  LOD    0    7
118  WRT    0    0
119  LOD    0    8
120  WRT    0    0
121  OPR    0    0
```

---

输入数据：

```
10
5
18
4
12
18
```

## 输出数据：

```
START PL/0
50
4
2
6
END PL/0
```

## 输入输出数据分析

- 输入相乘的两个数10，5；输入相除的两个数18，4；输入求最大公约数的两个数12和18
- 输出10*5的结果50；输出18/4的结果4余数2；输出12和18的最大公约数6