PL0 编译程序注释

```
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;
   object = (constant, variable, procedure);
   symset = set of symbol;
   fct = (lit, opr, lod, sto, cal, int, jmp, jpc); {functions}
```

```
    instruction = packed record
                    f : fct;    {功能码}
                    l : 0..levmax; {相对层数}
                    a : 0..amax; {相对地址}
                  end;
    {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  }
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 : object of
                  constant : (val : integer);
                  variable, procedure : (level, adr : integer)
              end;
procedure error (n : integer);
begin
    writeln('****', ' ' : cc－1, '↑', n : 2);    err := err + 1
    {cc 为当前行已读的字符数, n 为错误号}{错误数 err 加 1}
end {error};
procedure getsym;
```

```
    var    i, j, k : integer;
    procedure    getch ; {取下一字符}
    begin
      if cc = ll then {如果 cc 指向行末}
      begin
        if eof(input) then {如果已到文件尾}
        begin
          write('PROGRAM INCOMPLETE'); goto 99
        end;
        {读新的一行}
        ll := 0; cc := 0; write(cx : 5, ' ');
                                    {cx : 5 位数}
        while  ¬eoln(input) do {如果不是行末}
        begin
          ll := ll + 1; read(ch); write(ch);
          line[ll] := ch    {一次读一行入 line}
        end;
        writeln; ll := ll + 1; read(line[ll])    {line[ll]中是行末符}
      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 ¬(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
    until kk = k;                    大长度, 后面补空白}
  id := a;  i := 1;  j := norw;
  {id 中存放当前标识符或保留字的字符串}
  repeat  k := (i+j) div 2; {用二分查找法在保留字表中找
    if id ≤ word[k] then j := k－1;  当前的标识符 id}
    if id ≥ word[k] then i := k + 1
  until i > j;
  if i－1 > j then sym := wsym[k] else sym := ident
  {如果找到, 当前记号 sym 为保留字, 否则 sym 为标识符}
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  ¬(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 {处理其它算符或标点符号}
    begin    sym := ssym[ch];    getch
    end
end {getsym};
procedure    gen(x : fct; y, z : integer);
begin
    if cx > cxmax then {如果当前指令序号>代码的最大长度}
```

```
    begin write('PROGRAM TOO LONG'); 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  ¬(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  ¬(sym in s1) do getsym
        {跳过一些记号, 直到当前记号属于 S1∪S2}
    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 : object);
```

```
begin {把 object 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
      constant : begin {当前标识符是常数名}
                    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;
      variable : begin {当前标识符是变量名}
                    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;
      procedure : 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(i, mnemonic[f] : 5, 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
            if sym = ident then {当前记号是标识符}
            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}
            procedure : 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 ¬(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) else
      {若在符号表中查不到 id, 则出错, 否则做以下工作}
      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 = procedure 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;
if sym = varsym then {当前记号是变量说明语句开始符号}
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;
```

```
while sym = procsym do {处理过程说明}
begin    getsym;
  if sym = ident then {如果当前记号是过程名}
  begin    enter(procedure);    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  ¬(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('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(s[t]);
        {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('END PL/0');
   end {interpret};
begin   {主程序}
   for ch := 'A' to ';' do    ssym[ch] := nul;
   {ASCII 码的顺序}
   word[1] := 'BEGIN         '; word[2] := 'CALL          ';
保 word[3] := 'CONST         '; word[4] := 'DO            ';
留 word[5] := 'END           '; word[6] := 'IF            ';
字 word[7] := 'ODD           '; word[8] := 'PROCEDURE ';
   word[9] := 'THEN          '; word[10] := 'VAR          ';
   word[11] := 'WHILE         ';
保 wsym[1] := beginsym;     wsym[2] := callsym;
留 wsym[3] := constsym;     wsym[4] := dosym;
字 wsym[5] := endsym;       wsym[6] := ifsym;
的 wsym[7] := oddsym;       wsym[8] := procsym;
记 wsym[9] := thensym;      wsym[10] := varsym;
号 wsym[11] := whilesym;
   ssym['+'] := plus;          ssym['－'] := minus;
   ssym['*'] := times;         ssym['/'] := slash;
```

ssym[‘(’] := lparen;　　ssym[‘)’] := rparen;

ssym[‘=’] := eql;　　ssym[‘, ’] := comma;

ssym[‘.’] := period;　　ssym[‘≠’] := neq;

ssym[‘<’] := lss;　　ssym[‘>’] := gtr;

ssym[‘≤’] := leq;　　ssym[‘≥’] := geq;

ssym[‘;’] := semicolon;

{算符和标点符号的记号}

mnemonic[lit] := ‘LIT’;　　mnemonic[opr] := ‘OPR’;

mnemonic[lod] := ‘LOD’;　　mnemonic[sto] := ‘STO’;

mnemonic[cal] := ‘CAL’;　　mnemonic[int] := ‘INT’;

mnemonic[jmp] := ‘JMP’;　　mnemonic[jpc] := ‘JPC’;

{中间代码指令的字符串}

declbegsys := [constsym, varsym, procsym];

{说明语句的开始符号}

statbegsys := [beginsym, callsym, ifsym, whilesym];

{语句的开始符号}

facbegsys := [ident, number, lparen];

{因子的开始符号}

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
end.
```