编译原理实验

徐达烽
16340260
数字媒体技术

思路

1. 使用规范名称，提高代码可读性。
2. 将大于等于，小于等于和不等于号修改成>=, <=, <>, 并修改这些符号的处理逻辑。
3. 修改源程序中存在的bug，使其成功运行。
4. 添加RED, WRT命令，从而实现read, write函数。

第一部分

1. 编译程序源代码

```
program  PL0;
{带有代码生成的PL0编译程序}
const
    kReservedWords = 11; {保留字的个数}
    kIdentsMax = 100; {标识符表长度}
    kNumLengthMax = 14; {数字的最大位数}
    kIdentLengthMax = 10; {标识符的长度}
    kAddrMax = 2047; {最大地址}
    kNestingLayersMax = 3; {程序体嵌套的最大深度}
    kInstructionsMax = 200; {代码数组的大小}
    kDebugMessageOn = 1;
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);
    Identifier = packed array [1..kIdentLengthMax] of char;
    ObjectType = (kConstant, kVariable, kProcedure);
    SymbolSet = set of Symbol;
    FunctionCode = (LIT, OPR, LOD, STO, CAL, INT, JMP, JPC);
```

```
{functions}
    Instruction = packed record
        func: FunctionCode;  {功能码}
        level : 0..kNestingLayersMax; {相对层数}
        adr : 0..kAddrMax; {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
    intermediate: text;
    stack_data: text;

    curr_char: char; {最近读到的字符}
    curr_symbol : Symbol; {最近读到的符号}
    id : Identifier; {最近读到的标识符}
    curr_ident : Identifier; {当前标识符的字符串}

    num : integer; {最近读到的数}
    char_count : integer; {当前行的字符计数}
    line_length : integer; {当前行的长度}
    error_count : integer;
    code_count : integer; {代码数组的当前下标}
    line : array [1..81] of char; {当前行}


    code : array [0..kInstructionsMax] of Instruction; {中间代码数组}
    words : array [1..kReservedWords] of Identifier; {存放保留字的字符串}
    words_symbol : array [1..kReservedWords] 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 [FunctionCode] of string;
    {中间代码算符的字符串}
    declare_symbols, stat_begin_symbols, factor_begin_symbols : SymbolS
et;
    table : array [0..kIdentsMax] of {符号表}
            record
                name : Identifier;
                case kind : ObjectType of
                kConstant : (val : integer);
```

```
                    kVariable, kProcedure : (level, adr : integer)
                end;

procedure ExitWithError(message: string);
begin
    writeln('Fatal Error: ', message);
    halt;
end;


procedure error (n : integer);
begin
    writeln('****', ' ' : char_count - 1, '^', n : 2);
    {当前行已读的字符数}
    error_count := error_count + 1;
    {错误数err加1}
    //halt;
end {error};



procedure GetSymbol; {Lexical Analyzer}
var  i, j, k : integer;

procedure  GetChar; {取下一字符}
begin

    if char_count = line_length then {如果cc指向行末}
    begin
        {如果已到文件尾}
        if eof(input) then ExitWithError('PROGRAM INCOMPLETE');
        {读新的一行}

        line_length := 0;
        char_count := 0;
        //writeln('char_count reset');
        write(code_count : 5, ' ');    {code_count : 5位数}

        while not eoln(input) do {如果不是行末}
        begin
            line_length := line_length + 1;
            read(curr_char);
            write(curr_char);
            line[line_length] := curr_char; {一次读一行入line}
        end;
```

```
            writeln;

            line_length := line_length + 1;
            //writeln('line length: ',line_length);
            read(line[line_length]) ; {line[line_length]中是行末符}
        end;
        char_count := char_count + 1;
        curr_char:= line[char_count];  {取line中下一个字符}
        //writeln('Getchar: ', ord(curr_char));
    end {GetChar};

    begin {GetSymbol}
        while curr_char in [' ', #13, #9, #10] do GetChar; {跳过无用空白}
        if curr_char 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 < kIdentLengthMax then
                begin
                    k := k + 1;
                    curr_ident[k] := curr_char;
                    //write(curr_char);
                end;
                GetChar;
            until  not(curr_char in ['a'..'z', '0'..'9']);
            //writeln;

            {id中存放当前标识符或保留字的字符串}
            id := curr_ident;
            curr_ident := '';

            i := 0;
            j := kReservedWords + 1;
            {用二分查找法在保留字表中找当前的标识符id}
            repeat
                k := (i + j) div 2;
                if words[k] >= id then j := k
                else i := k
            until i + 1 >= j;
            {如果找到，当前记号sym为保留字，否则sym为标识符}
            if (j = kReservedWords + 1) or (words[j] <> id) then
            begin
                curr_symbol := IDENT;
                //writeln('find indent: ', id);
            end
```

```
154.            else
155.            begin
156.                curr_symbol := words_symbol[j] ;
157.                //writeln('find reserved: ', id);
158.            end
159.        end
160.
161.        else if curr_char in ['0'..'9'] then
162.        begin {数字}
163.            k := 0;
164.            num := 0;
165.            curr_symbol := NUMBER; {当前记号sym为数字}
166.
167.            repeat {计算数字串的值}
168.                num := 10*num + (ord(curr_char)-ord('0'));
169.                k := k + 1;
170.                GetChar;
171.            until  not(curr_char in ['0'..'9']);
172.
173.            {当前数字串的长度超过上界,则报告错误}
174.            if k > kNumLengthMax then  error(30);
175.            //writeln('find number: ', num);{debug}
176.        end
177.        else if curr_char= ':' then {处理赋值号}
178.        begin
179.            GetChar;
180.            if curr_char= '=' then
181.            begin
182.                curr_symbol := BECOMES;
183.                GetChar
184.            end
185.            else
186.                curr_symbol := NUL;
187.        end
188.        else if curr_char = '<' then
189.        begin
190.            GetChar;
191.            if curr_char = '>' then {处理不等号}
192.            begin
193.                curr_symbol := NEQ;
194.                GetChar;
195.            end
196.            else if curr_char = '=' then {处理小于等于号}
197.            begin
198.                curr_symbol := LEQ;
```

```
            GetChar;
        end
        else  curr_symbol := LSS;
    end

    else if curr_char = '>' then
    begin
        Getchar;
        if curr_char = '=' then
        begin
            curr_symbol := GEQ;
            GetChar;
        end
        else  curr_symbol := GTR;
    end

    else {处理其它算符或标点符号}
    begin
        //writeln('curr_symbol curr_char: ', ord(curr_char));
        curr_symbol := ssym[curr_char];
        GetChar;
    end;
end {GetSymbol};


procedure  GenerateCode(next_func : FunctionCode; next_level,
next_addr : integer);
begin
    {如果当前指令序号>代码的最大长度}
    if code_count > kInstructionsMax then ExitWithError('PROGRAM TOO
LONG');

    with code[code_count] do {生成一条新代码}
    begin
        func := next_func; {功能码}
        level := next_level; {层号}
        adr := next_addr {地址}
    end;
    code_count := code_count + 1 {指令序号加1}
end {GenerateCode};


procedure Test(s1, s2 : SymbolSet; n : integer);
    {如果当前记号不属于集合S1,则报告错误n,跳过一些记号, 直到当前记号属于S1US2}
begin
```

```
242.     if  not (curr_symbol in s1) then
243.     begin
244.         error(n);
245.         s1 := s1 + s2;
246.         while not (curr_symbol in s1) do GetSymbol
247.     end
248. end {Test};
249.
250.
251. procedure  Block(lev, table_top : integer; symbol_set : SymbolSet); {程
     序体}
252. var
253.     data_top : integer; {本过程数据空间分配下标} {栈顶指针}
254.     symbol_start : integer; {本过程标识表起始下标}
255.     code_start : integer; {本过程代码起始下标}
256.
257. procedure  Enter(k : ObjectType);
258. begin {把obj填入符号表中}
259.     table_top := table_top + 1; {符号表指针加1}
260.
261.     with table[table_top] do{在符号表中增加新的一个条目}
262.     begin
263.         name := id; {当前标识符的名字}
264.         kind := k; {当前标识符的种类}
265.         case k of
266.             kConstant :
267.                 begin {当前标识符是常数名}
268.                     if num > kAddrMax then {当前常数值大于上界,则出错}
269.                     begin
270.                         error(30);
271.                         num := 0
272.                     end;
273.
274.                     val := num
275.                 end;
276.
277.             kVariable :
278.                 begin {当前标识符是变量名}
279.                     level := lev; {定义该变量的过程的嵌套层数}
280.                     adr := data_top; {变量地址为当前过程数据空间栈顶}
281.                     data_top := data_top +1; {栈顶指针加1}
282.                 end;
283.
284.             kProcedure :
285.                 level := lev {本过程的嵌套层数}
```

```
        end
    end
end {Enter};


function  position(id : Identifier) : 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 := table_top; {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 curr_symbol = IDENT then {当前记号是常数名}
    begin
        GetSymbol;
        if curr_symbol in [EQL, BECOMES] then {当前记号是等号或赋值号}
        begin
            if curr_symbol = BECOMES then error(1);
            {如果当前记号是赋值号,则出错}
            GetSymbol;

            if curr_symbol = NUMBER then {等号后面是常数}
            begin
                Enter(kConstant); {将常数名加入符号表}
                GetSymbol
            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 curr_symbol = IDENT then {如果当前记号是标识符}
    begin
        Enter(kVariable); {将该变量名加入符号表的下一条目}
        GetSymbol
    end
    else error(4) {如果变量说明未出现标识符,则出错}
end {VarDeclaration};


procedure  ListCode;
{列出本程序体生成的代码}
var  i : integer;
begin
    {code_start: 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}
    for i := code_start to code_count - 1 do
        with code[i] do {打印第i条代码}
            writeln(intermediate, i:3, mnemonic[func]:5, level : 3, adr
: 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(symbol_set : SymbolSet);
var  i, next_node, next_node_2 : integer;

procedure  Expression(symbol_set : SymbolSet);
var  addop : Symbol;

procedure  Term(symbol_set : SymbolSet);
var  mulop : Symbol;

procedure  Factor(symbol_set : SymbolSet);
var i : integer;
begin
    Test(factor_begin_symbols, symbol_set, 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 curr_symbol in factor_begin_symbols do
       {如果当前的记号是否因子的开始符号}
    begin
        if curr_symbol = 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)
                    {若在符号表中查不到id, 则出错, 否则,做以下工作}
                else
                    with table[i] do
                    case kind of
                        kConstant : GenerateCode(LIT, 0, val);
                            {若id是常数, 生成指令,将常数val取到栈顶}
                        kVariable : GenerateCode(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}
                        kProcedure : error(21)
                            {若id是过程名, 则出错}
                    end;

                GetSymbol {取下一记号}
            end
            else if curr_symbol = NUMBER then {当前记号是数字}
            begin
                if num > kAddrMax then {若数值越界,则出错}
                begin
                    error(30);
                    num := 0
                end;
                GenerateCode(LIT, 0, num); {生成一条指令, 将常数num取到栈顶}
                GetSymbol {取下一记号}
            end
            else if curr_symbol = LPAREN then {如果当前记号是左括号}
            begin
                GetSymbol; {取下一记号}
                Expression([RPAREN]+symbol_set); {处理表达式}
                if curr_symbol = RPAREN then GetSymbol
                {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(symbol_set, [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};
```

```
420.  begin {Term}
421.      Factor(symbol_set+[TIMES, SLASH]); {处理项中第一个因子}
422.      while curr_symbol in [TIMES, SLASH] do
423.          {当前记号是"乘"或"除"号}
424.      begin
425.          mulop := curr_symbol; {运算符存入mulop}
426.          GetSymbol; {取下一记号}
427.          Factor(symbol_set+[TIMES, SLASH]); {处理一个因子}
428.          if mulop = TIMES then GenerateCode(OPR, 0, 4)
429.          {若mulop是"乘"号,生成一条乘法指令}
430.                           else GenerateCode(OPR, 0, 5)
431.          {否则, mulop是除号, 生成一条除法指令}
432.      end
433.  end {Term};
434.
435.
436.  begin {Expression}
437.      if curr_symbol in [PLUS, MINUS] then {若第一个记号是加号或减号}
438.      begin
439.          addop := curr_symbol;  {"+"或"-"存入addop}
440.          GetSymbol;
441.          Term(symbol_set+[PLUS, MINUS]); {处理一个项}
442.          if addop = MINUS then GenerateCode(OPR, 0, 1)
443.          {若第一个项前是负号, 生成一条"负运算"指令}
444.      end
445.      else Term(symbol_set+[PLUS, MINUS]);
446.          {第一个记号不是加号或减号, 则处理一个项}
447.
448.      while curr_symbol in [PLUS, MINUS] do {若当前记号是加号或减号}
449.      begin
450.          addop := curr_symbol; {当前算符存入addop}
451.          GetSymbol; {取下一记号}
452.          Term(symbol_set+[PLUS, MINUS]); {处理一个项}
453.          if addop = PLUS then GenerateCode(OPR, 0, 2)
454.          {若addop是加号, 生成一条加法指令}
455.                       else GenerateCode(OPR, 0, 3)
456.          {否则, addop是减号, 生成一条减法指令}
457.      end
458.  end {Expression};
459.
460.
461.  procedure  Condition(symbol_set : SymbolSet);
462.  var  relop : Symbol;
463.  begin {Condition}
464.      if curr_symbol = ODDSYM then {如果当前记号是"odd"}
```

```
465.        begin
466.            GetSymbol;  {取下一记号}
467.            Expression(symbol_set); {处理算术表达式}
468.            GenerateCode(OPR, 0, 6) {生成指令,判定表达式的值是否为奇数,
469.            是,则取"真";不是, 则取"假"}
470.        end
471.        else {如果当前记号不是"odd"}
472.        begin
473.            Expression([EQL, NEQ, LSS, GTR, LEQ, GEQ] + symbol_set);
474.            {处理算术表达式}
475.            if  not (curr_symbol in [EQL, NEQ, LSS, LEQ, GTR, GEQ]) then
476.            {如果当前记号不是关系符, 则出错; 否则,做以下工作}
477.                error(20)
478.            else
479.            begin
480.                relop := curr_symbol; {关系符存入relop}
481.                GetSymbol; {取下一记号}
482.                Expression(symbol_set); {处理关系符右边的算术表达式}
483.                case relop of
484.                    EQL : GenerateCode(OPR, 0, 8);
485.                        {生成指令, 判定两个表达式的值是否相等}
486.                    NEQ : GenerateCode(OPR, 0, 9);
487.                        {生成指令, 判定两个表达式的值是否不等}
488.                    LSS : GenerateCode(OPR, 0, 10);
489.                        {生成指令,判定前一表达式是否小于后一表达式}
490.                    GEQ : GenerateCode(OPR, 0, 11);
491.                        {生成指令,判定前一表达式是否大于等于后一表达式}
492.                    GTR : GenerateCode(OPR, 0, 12);
493.                        {生成指令,判定前一表达式是否大于后一表达式}
494.                    LEQ : GenerateCode(OPR, 0, 13);
495.                        {生成指令,判定前一表达式是否小于等于后一表达式}
496.                end
497.            end
498.        end
499.    end {Condition};
500.
501.
502.    begin {Statement}
503.        if curr_symbol = IDENT then {处理赋值语句}
504.        begin
505.            i := position(id);  {在符号表中查id, 返回id在符号表中的入口}
506.            if i = 0 then error(11) {若在符号表中查不到id, 则出错}
507.            else if table[i].kind <> kVariable then {对非变量赋值, 则出错}
508.            begin
509.                error(12);
```

```
510.                i := 0;
511.            end;
512.
513.            GetSymbol; {取下一记号}
514.            if curr_symbol = BECOMES then GetSymbol else error(13);
515.            {若当前是赋值号，取下一记号，否则出错}
516.            Expression(symbol_set); {处理表达式}
517.            if i <> 0 then {若赋值号左边的变量id有定义}
518.                with table[i] do GenerateCode(STO, lev-level, adr)
519.
520.        end
521.        else if curr_symbol = CALLSYM then {处理过程调用语句}
522.        begin
523.            GetSymbol; {取下一记号}
524.            if curr_symbol <> IDENT then error(14) {下一记号不是标识符(过程名),
    出错}
525.            else
526.            begin
527.                i := position(id); {查符号表,返回id在表中的位置}
528.                if i = 0 then error(11) {在符号表中查不到，出错}
529.                else
530.                    with table[i] do
531.                        if kind = kProcedure then GenerateCode(CAL, lev-lev
    el, adr)
532.                        {如果在符号表中id是过程名}
533.                        else error(15); {若id不是过程名,则出错}
534.
535.                GetSymbol {取下一记号}
536.            end
537.        end
538.        else if curr_symbol = IFSYM then {处理条件语句}
539.        begin
540.            GetSymbol; {取下一记号}
541.            Condition([THENSYM, DOSYM]+symbol_set); {处理条件表达式}
542.            if curr_symbol = THENSYM then GetSymbol else error(16);
543.            {如果当前记号是"then",则取下一记号；否则出错}
544.            next_node := code_count; {next_node记录下一代码的地址}
545.            GenerateCode(JPC, 0, 0); {生成指令,表达式为"假"转到某地址(待填),
546.            否则顺序执行}
547.            Statement(symbol_set); {处理一个语句}
548.            code[next_node].adr := code_count
549.            {将下一个指令的地址回填到上面的jpc指令地址栏}
550.        end
551.        else if curr_symbol = BEGINSYM then {处理语句序列}
552.        begin
```

```
            GetSymbol;
            Statement([SEMICOLON, ENDSYM]+symbol_set);
                {取下一记号, 处理第一个语句}
            while curr_symbol in [SEMICOLON]+stat_begin_symbols 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 curr_symbol = SEMICOLON then GetSymbol 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]+symbol_set) {处理下一个语句}
            end;
            if curr_symbol = ENDSYM then GetSymbol else error(17)
                {如果当前记号是"end",则取下一记号,否则出错}
        end
        else if curr_symbol = WHILESYM then {处理循环语句}
        begin
            next_node := code_count; {next_node记录下一指令地址,即条件表达式的
            第一条代码的地址}
            GetSymbol; {取下一记号}
            Condition([DOSYM]+symbol_set); {处理条件表达式}
            next_node_2 := code_count; {记录下一指令的地址}
            GenerateCode(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 curr_symbol = DOSYM then GetSymbol else error(18);
            {如果当前记号是"do",则取下一记号, 否则出错}
            Statement(symbol_set); {处理"do"后面的语句}
            GenerateCode(JMP, 0, next_node); {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[next_node_2].adr := code_count
            {把下一指令地址回填到前面生成的jpc指令的地址栏}
        end;

        Test(symbol_set, [ ], 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}
        data_top := 3; {本过程数据空间栈顶指针}
        symbol_start := table_top; {标识符表的长度(当前指针)}
        table[table_top].adr := code_count; {本过程名的地址, 即下一条指令的序号}
        GenerateCode(JMP, 0, 0); {生成一条转移指令}
        if lev > kNestingLayersMax then error(32);
            {如果当前过程层号>最大层数, 则出错}
        repeat
```

```
597.            if curr_symbol = CONSTSYM then {处理常数说明语句}
598.            begin
599.                GetSymbol;
600.                repeat
601.                    ConstDeclaration; {处理一个常数说明}
602.                    while curr_symbol = COMMA do {如果当前记号是逗号}
603.                    begin
604.                        GetSymbol;
605.                        ConstDeclaration
606.                    end; {处理下一个常数说明}
607.                    if curr_symbol = SEMICOLON then GetSymbol else error(5)
608.                    {如果当前记号是分号,则常数说明已处理完, 否则出错}
609.                until curr_symbol <> IDENT
610.                {跳过一些记号, 直到当前记号不是标识符(出错时才用到)}
611.            end;
612.
613.            if curr_symbol = VARSYM then {当前记号是变量说明语句开始符号}
614.            begin
615.                GetSymbol;
616.                repeat
617.                    VarDeclaration; {处理一个变量说明}
618.                    while curr_symbol = COMMA do {如果当前记号是逗号}
619.                    begin
620.                        GetSymbol;
621.                        VarDeclaration
622.                    end;
623.                        {处理下一个变量说明}
624.                    if curr_symbol = SEMICOLON then GetSymbol else error(5)
625.                        {如果当前记号是分号,则变量说明已处理完, 否则出错}
626.                until curr_symbol <> IDENT;
627.                    {跳过一些记号, 直到当前记号不是标识符(出错时才用到)}
628.            end;
629.
630.            while curr_symbol = PROCSYM do {处理过程说明}
631.            begin
632.                GetSymbol;
633.                if curr_symbol = IDENT then {如果当前记号是过程名}
634.                begin
635.                    Enter(kProcedure);
636.                    GetSymbol
637.                end {把过程名填入符号表}
638.                else error(4); {否则, 缺少过程名出错}
639.
640.                if curr_symbol = SEMICOLON then GetSymbol else error(5);
641.                    {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}
```

```
642.
643.             Block(lev+1, table_top, [SEMICOLON]+symbol_set); {处理过程体}
644.                 {lev+1: 过程嵌套层数加1; table_top: 符号表当前栈顶指针,也是新
    过程符号表起始位置; [SEMICOLON]+symbol_set: 过程体开始和末尾符号集}
645.
646.             if curr_symbol = SEMICOLON then {如果当前记号是分号}
647.             begin
648.                 GetSymbol; {取下一记号}
649.                 Test(stat_begin_symbols+[IDENT, PROCSYM], symbol_set, 6
    )
650.                     {测试当前记号是否语句开始符号或过程说明开始符号,
651.                     否则报告错误6, 并跳过一些记号}
652.             end
653.             else error(5) {如果当前记号不是分号,则出错}
654.         end;
655.         //writeln('Ha??');
656.         Test(stat_begin_symbols+[IDENT], declare_symbols, 7)
657.             {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}
658.
659.     until  not (curr_symbol in declare_symbols);
660.     {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
661.     的开始符号}
662.     code[table[symbol_start].adr].adr := code_count;  {table[symbol_sta
    rt].addr是本过程名的第1条
663.         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
664.         1条代码)的地址回填到该jmp指令中,得(JMP, 0, code_count)}
665.
666.     with table[symbol_start] do {本过程名的第1条代码的地址改为下一指令地址cx}
667.     begin
668.         adr := code_count; {代码开始地址}
669.     end;
670.     code_start := code_count; {code_start记录起始代码地址}
671.     GenerateCode(INT, 0, data_top); {生成一条指令, 在栈顶为本过程留出数据空间
    }
672.     Statement([SEMICOLON, ENDSYM]+symbol_set); {处理一个语句}
673.     GenerateCode(OPR, 0, 0); {生成返回指令}
674.     Test(symbol_set, [ ], 8); {测试过程体语句后的符号是否正常,否则出错}
675.     ListCode; {打印本过程的中间代码序列}
676. end  {Block};
677.
678.
679.
680. procedure  Interpret;
681. const  kStackSize = 500; {运行时数据空间(栈)的上界}
682. var  pc, base, top : integer; {程序地址寄存器, 基地址寄存器,栈顶地址寄存器}
```

```
683.        i : Instruction; {指令寄存器}
684.        stack : array [1..kStackSize] of integer; {数据存储栈}
685.
686.    function  BaseOf(lev : integer) : integer;
687.    var  b1 : integer;
688.    begin {BaseOf}
689.        b1 := base; {顺静态链求层差为lev的外层的基地址}
690.        while lev > 0 do
691.        begin
692.            b1 := stack[b1];
693.            lev := lev - 1
694.        end;
695.        BaseOf := b1
696.    end; {BaseOf}
697.
698.    begin  {Interpret}
699.        writeln('START PL/0');
700.        top := 0; {栈顶地址寄存器}
701.        base := 1; {基地址寄存器}
702.        pc := 0; {程序地址寄存器}
703.        stack[1] := 0;
704.        stack[2] := 0;
705.        stack[3] := 0;
706.            {最外层主程序数据空间栈最下面预留三个单元}
707.            {每个过程运行时的数据空间的前三个单元是:SL, DL, RA;
708.            SL: 指向本过程静态直接外层过程的SL单元;
709.            DL: 指向调用本过程的过程的最新数据空间的第一个单元;
710.            RA: 返回地址 }
711.        repeat
712.            i := code[pc]; {i取程序地址寄存器p指示的当前指令}
713.            pc := pc+1; {程序地址寄存器p加1,指向下一条指令}
714.            with i do
715.                case func of
716.                    LIT :
717.                        begin {当前指令是取常数指令(LIT, 0, a)}
718.                            top := top+1;
719.                            stack[top] := adr
720.                        end; {栈顶指针加1, 把常数a取到栈顶}
721.
722.                    OPR :
723.                        case adr of {当前指令是运算指令(OPR, 0, a)}
724.                            0 : begin {a=0时,是返回调用过程指令}
725.                                    top := base-1; {恢复调用过程栈顶}
726.                                    pc := stack[top+3]; {程序地址寄存器p取返回地
    址}
```

```
                                    base := stack[top+2];
                                        {基地址寄存器b指向调用过程的基地址}
                                end;
                            1 : stack[top] := -stack[top]; {一元负运算, 栈顶元素的值反号}
                            2 : begin {加法}
                                    top := top-1;
                                    stack[top] := stack[top] + stack[top+1]
                                end;
                            3 : begin {减法}
                                    top := top-1;
                                    stack[top] := stack[top]-stack[top+1]
                                end;
                            4 : begin {乘法}
                                    top := top-1;
                                    stack[top] := stack[top] * stack[top+1]
                                end;
                            5 : begin {整数除法}
                                    top := top-1;
                                    stack[top] := stack[top] div stack[top+1]
                                end;
                            6 : stack[top] := ord(odd(stack[top])); {算s[top]是否奇数, 是则s[top]=1, 否则s[top]=0}

                            8 : begin
                                    top := top-1;
                                    stack[top] := ord(stack[top] = stack[top+1])
                                end; {判两个表达式的值是否相等,
                                    是则s[top]=1, 否则s[top]=0}

                            9:  begin
                                    top := top-1;
                                    stack[top] := ord(stack[top] <> stack[top+1])
                                end; {判两个表达式的值是否不等,
                                    是则s[top]=1, 否则s[top]=0}
                            10: begin
                                    top := top-1;
                                    stack[top] := ord(stack[top] < stack[top+1])
                                end; {判前一表达式是否小于后一表达式,
                                    是则s[top]=1, 否则s[top]=0}

```

```
                            11: begin
                                    top := top-1;
                                    stack[top] := ord(stack[top] >= stack[to
p+1])
                                end; {判前一表达式是否大于或等于后一表达式,
                                    是则s[top]=1, 否则s[top]=0}

                            12: begin
                                    top := top-1;
                                    stack[top] := ord(stack[top] > stack[top
+1])
                                end; {判前一表达式是否大于后一表达式,
                                    是则s[top]=1, 否则s[top]=0}
                            13: begin
                                    top := top-1;
                                    stack[top] := ord(stack[top] <= stack[to
p+1])
                                end; {判前一表达式是否小于或等于后一表达式,
                                    是则s[top]=1, 否则s[top]=0}
                        end;

                    LOD :
                        begin {当前指令是取变量指令(LOD, l, a)}
                            top := top + 1;
                            stack[top] := stack[BaseOf(level) + adr]
                            {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}
                            stack[BaseOf(level) + adr] := stack[top];
                            writeln(stack_data, stack[top]);
                            {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}
                            top := top-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}
                            stack[top+1] := BaseOf(level);
                                {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}
                            stack[top+2] := base;
                            {调用过程的数据空间的起始地址存入本过程DL单元}
```

```
                        stack[top+3] := pc;
                        {调用过程cal指令的下一条的地址存入本过程RA单元}
                        base := top+1; {b指向被调用过程新的数据空间起始地址}
                        pc := adr {指令地址寄存储器指向被调用过程的地址a}
                    end;
                INT : top := top + adr;
                    {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 : pc := adr;
                    {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 stack[top] = 0 then pc := adr;
                        {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}
                        top := top-1 {数据栈顶指针减1}
                    end
            end {with, case}
    until pc = 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}
    writeln('END PL/0');
end; {Interpret}

begin  {主程序}
    assign(input, 'pl0_src.pas');
    reset(input);

    assign(intermediate, 'intermediate_code.txt');
    rewrite(intermediate);

    assign(stack_data, 'stack_data.txt');
    rewrite(stack_data);

    for curr_char:= 'a' to ';' do  ssym[curr_char] := NUL;
    {ASCII码的顺序}
    words[1] := 'begin';
    words[2] := 'call';
    words[3] := 'const';
    words[4] := 'do';
    words[5] := 'end';
    words[6] := 'if';
    words[7] := 'odd';
    words[8] := 'procedure';
    words[9] := 'then';
    words[10] := 'var';
    words[11] := 'while';
```

```
852.      words_symbol[1] := BEGINSYM;    words_symbol[2] := CALLSYM;
853.      words_symbol[3] := CONSTSYM;    words_symbol[4] := DOSYM;
854.      words_symbol[5] := ENDSYM;     words_symbol[6] := IFSYM;
855.      words_symbol[7] := ODDSYM;     words_symbol[8] := PROCSYM;
856.      words_symbol[9] := THENSYM;     words_symbol[10] := VARSYM;
857.      words_symbol[11] := WHILESYM;
858.      ssym['+'] := PLUS;       ssym['-'] := MINUS;
859.      ssym['*'] := TIMES;      ssym['/'] := SLASH;
860.      ssym['('] := LPAREN;      ssym[')'] := RPAREN;
861.      ssym['='] := EQL;        ssym[','] := COMMA;
862.      ssym['.'] := PERIOD;
863.      ssym['<'] := LSS;        ssym['>'] := GTR;
864.
865.      ssym[';'] := SEMICOLON;
866.      {算符和标点符号的记号}
867.      mnemonic[LIT] := 'LIT';      mnemonic[OPR] := 'OPR';
868.      mnemonic[LOD] := 'LOD';     mnemonic[STO] := 'STO';
869.      mnemonic[CAL] := 'CAL';     mnemonic[INT] := 'INT';
870.      mnemonic[JMP] := 'JMP';     mnemonic[JPC] := 'JPC';
871.      {中间代码指令的字符串}
872.      declare_symbols := [CONSTSYM, VARSYM, PROCSYM];
873.      {说明语句的开始符号}
874.      stat_begin_symbols := [BEGINSYM, CALLSYM, IFSYM, WHILESYM];
875.      {语句的开始符号}
876.      factor_begin_symbols := [IDENT, NUMBER, LPAREN];
877.      {因子的开始符号}
878.
879.
880.      error_count := 0; {发现错误的个数}
881.      char_count := 0; {当前行中输入字符的指针}
882.      code_count := 0; {代码数组的当前指针}
883.      line_length := 0; {输入当前行的长度}
884.      curr_char:= ' '; {当前输入的字符}
885.      GetSymbol; {取下一个记号}
886.
887.      Block(0, 0, [PERIOD] + declare_symbols + stat_begin_symbols); {处理
      程序体}
888.
889.      if curr_symbol <> PERIOD then error(9);
890.      {如果当前记号不是句号, 则出错}
891.
892.
893.      if error_count = 0 then Interpret
894.      {如果编译无错误, 则解释执行中间代码}
895.      else writeln(error_count, ' ERROR(S) IN PL/0 PROGRAM');
```

```
896.
897.        close(intermediate);
898.        close(stack_data);
899.    end.
```

2. PL0源程序代码

```
1.  const  m = 7, n = 85;
2.  var  x, y, z, q, r;
3.
4.  procedure  multiply;
5.  var  a, b;
6.  begin
7.      a := x;
8.      b := y;
9.      z := 0;
10.     while b > 0 do
11.     begin
12.         if odd b then z := z + a;
13.         a := 2*a ;
14.         b := b/2 ;
15.     end
16. end;
17.
18. procedure  divide;
19. var  w;
20. begin
21.     r := x;
22.     q := 0;
23.     w := y;
24.     while w <= r do  w := 2*w ;
25.     while w > y do
26.     begin
27.         q := 2*q;
28.         w := w/2;
29.         if w <= r then
30.         begin
31.             r := r-w;
32.             q := q+1
33.         end
34.     end
35. end;
36.
```

```
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

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
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
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 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

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

第二部分

1. 编译程序源代码

```
program  PL0;
{支持read, write函数的PL0编译程序}
```

```
const
    kReservedWords = 13; {保留字的个数}
    kIdentsMax = 100; {标识符表长度}
    kNumLengthMax = 14; {数字的最大位数}
    kIdentLengthMax = 10; {标识符的长度}
    kAddrMax = 2047; {最大地址}
    kNestingLayersMax = 3; {程序体嵌套的最大深度}
    kInstructionsMax = 200; {代码数组的大小}
    kDebugMessageOn = 1;
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, W
RITESYM);
    Identifier = packed array [1..kIdentLengthMax] of char;
    ObjectType = (kConstant, kVariable, kProcedure);
    SymbolSet = set of Symbol;
    FunctionCode = (LIT, OPR, LOD, STO, CAL, INT, JMP, JPC, RED, WRT);
{functions}
    Instruction = packed record
        func: FunctionCode;  {功能码}
        level : 0..kNestingLayersMax; {相对层数}
        adr : 0..kAddrMax; {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
    intermediate: text;
    stack_data: text;
    pl0_input: text;
    curr_char: char; {最近读到的字符}
    curr_symbol : Symbol; {最近读到的符号}
    id : Identifier; {最近读到的标识符}
    curr_ident : Identifier; {当前标识符的字符串}

    num : integer; {最近读到的数}
    char_count : integer; {当前行的字符计数}
    line_length : integer; {当前行的长度}
```

```
46.     error_count : integer;
47.     code_count : integer; {代码数组的当前下标}
48.     line : array [1..81] of char; {当前行}
49.
50.
51.     code : array [0..kInstructionsMax] of Instruction; {中间代码数组}
52.     words : array [1..kReservedWords] of Identifier; {存放保留字的字符串}
53.     word_symbol : array [1..kReservedWords] of Symbol; {存放保留字的记号}
54.
55.     ssym : array [char] of Symbol; {存放算符和标点符号的记号}
56.     code_str : array [FunctionCode] of string;
57.     {中间代码算符的字符串}
58.     declare_symbols, stat_begin_symbols, factor_begin_symbols : SymbolS
    et;
59.     table : array [0..kIdentsMax] of {符号表}
60.             record
61.                 name : Identifier;
62.                 case kind : ObjectType of
63.                 kConstant : (val : integer);
64.                 kVariable, kProcedure : (level, adr : integer)
65.             end;
66.
67. procedure ExitWithError(message: string);
68. begin
69.     writeln('Fatal Error: ', message);
70.     halt;
71. end;
72.
73.
74. procedure error (n : integer);
75. begin
76.     writeln('****', ' ' : char_count - 1, '^', n : 2);
77.     {当前行已读的字符数}
78.     error_count := error_count + 1;
79.     {错误数err加1}
80.     //halt;
81. end {error};
82.
83.
84. procedure GetSymbol; {Lexical Analyzer}
85. var  i, j, k : integer;
86.
87. procedure  GetChar; {取下一字符}
88. begin
89.
```

```
90.     if char_count = line_length then {如果cc指向行末}
91.     begin
92.         {如果已到文件尾}
93.         if eof(input) then ExitWithError('PROGRAM INCOMPLETE');
94.         {读新的一行}
95.
96.         line_length := 0;
97.         char_count := 0;
98.         //writeln('char_count reset');
99.         write(code_count : 5, ' ');   {code_count : 5位数}
100.
101.        while not eoln(input) do {如果不是行末}
102.        begin
103.            line_length := line_length + 1;
104.            read(curr_char);
105.            write(curr_char);
106.            line[line_length] := curr_char; {一次读一行入line}
107.        end;
108.        writeln;
109.
110.        line_length := line_length + 1;
111.        //writeln('line length: ',line_length);
112.        read(line[line_length]) ; {line[line_length]中是行末符}
113.    end;
114.    char_count := char_count + 1;
115.    curr_char:= line[char_count];  {取line中下一个字符}
116.    //writeln('Getchar: ', ord(curr_char));
117. end {GetChar};
118.
119. begin {GetSymbol}
120.    while curr_char in [' ', #13, #9, #10] do GetChar; {跳过无用空白}
121.    if curr_char in ['a'..'z'] then
122.    begin {标识符或保留字}
123.        k := 0;
124.        repeat {处理字母开头的字母、数字串}
125.            if k < kIdentLengthMax then
126.            begin
127.                k := k + 1;
128.                curr_ident[k] := curr_char;
129.            end;
130.            GetChar;
131.        until  not(curr_char in ['a'..'z', '0'..'9']);
132.
133.        {id中存放当前标识符或保留字的字符串}
134.        id := curr_ident;
```

```
135.            curr_ident := '';
136.
137.            i := 0;
138.            j := kReservedWords + 1;
139.            {用二分查找法在保留字表中找当前的标识符id}
140.            repeat
141.                k := (i + j) div 2;
142.                if words[k] >= id then j := k
143.                else i := k
144.            until i + 1 >= j;
145.            {如果找到，当前记号sym为保留字，否则sym为标识符}
146.            if (j = kReservedWords + 1) or (words[j] <> id) then
147.            begin
148.                curr_symbol := IDENT;
149.                //writeln('find indent: ', id);
150.            end
151.            else
152.            begin
153.                curr_symbol := word_symbol[j] ;
154.                //writeln('find reserved: ', id);
155.            end
156.
157.
158.        end
159.
160.        else if curr_char in ['0'..'9'] then
161.        begin {数字}
162.            k := 0;
163.            num := 0;
164.            curr_symbol := NUMBER; {当前记号sym为数字}
165.
166.            repeat {计算数字串的值}
167.                num := 10*num + (ord(curr_char)-ord('0'));
168.                k := k + 1;
169.                GetChar;
170.            until  not(curr_char in ['0'..'9']);
171.
172.            {当前数字串的长度超过上界,则报告错误}
173.            if k > kNumLengthMax then  error(30);
174.            //writeln('find number: ', num);{debug}
175.        end
176.        else if curr_char= ':' then {处理赋值号}
177.        begin
178.            GetChar;
179.            if curr_char= '=' then
```

```
180.            begin
181.                curr_symbol := BECOMES;
182.                GetChar
183.            end
184.            else
185.                curr_symbol := NUL;
186.        end
187.        else if curr_char = '<' then
188.        begin
189.            GetChar;
190.            if curr_char = '>' then {处理不等号}
191.            begin
192.                curr_symbol := NEQ;
193.                GetChar;
194.            end
195.            else if curr_char = '=' then {处理小于等于号}
196.            begin
197.                curr_symbol := LEQ;
198.                GetChar;
199.            end
200.            else  curr_symbol := LSS;
201.        end
202.
203.        else if curr_char = '>' then
204.        begin
205.            Getchar;
206.            if curr_char = '=' then
207.            begin
208.                curr_symbol := GEQ;
209.                GetChar;
210.            end
211.            else  curr_symbol := GTR;
212.        end
213.
214.        else {处理其它算符或标点符号}
215.        begin
216.            //writeln('curr_symbol curr_char: ', ord(curr_char));
217.            curr_symbol := ssym[curr_char];
218.            GetChar;
219.        end;
220.    end {GetSymbol};
221.
222.
223.    procedure  GenerateCode(next_func : FunctionCode; next_level,
        next_addr : integer);
```

```
224.  begin
225.      {如果当前指令序号>代码的最大长度}
226.      if code_count > kInstructionsMax then ExitWithError('PROGRAM TOO
      LONG');
227.
228.      with code[code_count] do {生成一条新代码}
229.      begin
230.          func := next_func; {功能码}
231.          level := next_level; {层号}
232.          adr := next_addr {地址}
233.      end;
234.      code_count := code_count + 1 {指令序号加1}
235.  end {GenerateCode};
236.
237.
238.  procedure Test(s1, s2 : SymbolSet; n : integer);
239.      {如果当前记号不属于集合S1,则报告错误n,跳过一些记号, 直到当前记号属于S1US2}
240.  begin
241.      if  not (curr_symbol in s1) then
242.      begin
243.          error(n);
244.          s1 := s1 + s2;
245.          while not (curr_symbol in s1) do GetSymbol
246.      end
247.  end {Test};
248.
249.
250.  procedure  Block(lev, table_top : integer; symbol_set : SymbolSet); {程
      序体}
251.  var
252.      data_top : integer; {本过程数据空间分配下标} {栈顶指针}
253.      symbol_start : integer; {本过程标识表起始下标}
254.      code_start : integer; {本过程代码起始下标}
255.
256.  procedure  Enter(k : ObjectType);
257.  begin {把obj填入符号表中}
258.      table_top := table_top + 1; {符号表指针加1}
259.
260.      with table[table_top] do{在符号表中增加新的一个条目}
261.      begin
262.          name := id; {当前标识符的名字}
263.          kind := k; {当前标识符的种类}
264.          case k of
265.              kConstant :
266.                  begin {当前标识符是常数名}
```

```
                    if num > kAddrMax then {当前常数值大于上界,则出错}
                    begin
                        error(30);
                        num := 0
                    end;

                    val := num
                end;

            kVariable :
                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 := data_top; {变量地址为当前过程数据空间栈顶}
                    data_top := data_top +1; {栈顶指针加1}
                end;

            kProcedure :
                level := lev {本过程的嵌套层数}
        end
    end
end {Enter};


function  position(id : Identifier) : 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 := table_top; {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 curr_symbol = IDENT then {当前记号是常数名}
    begin
        GetSymbol;
        if curr_symbol in [EQL, BECOMES] then {当前记号是等号或赋值号}
        begin
```

```
312.                if curr_symbol = BECOMES then error(1);
313.                {如果当前记号是赋值号,则出错}
314.                GetSymbol;
315.
316.                if curr_symbol = NUMBER then {等号后面是常数}
317.                begin
318.                    Enter(kConstant); {将常数名加入符号表}
319.                    GetSymbol
320.                end
321.                else error(2) {等号后面不是常数出错}
322.            end
323.            else error(3) {标识符后不是等号或赋值号出错}
324.        end
325.        else error(4) {常数说明中没有常数名标识符}
326.    end {ConstDeclaration};
327.
328.
329.    procedure  VarDeclaration;
330.    begin
331.        if curr_symbol = IDENT then {如果当前记号是标识符}
332.        begin
333.            Enter(kVariable); {将该变量名加入符号表的下一条目}
334.            GetSymbol
335.        end
336.        else error(4) {如果变量说明未出现标识符,则出错}
337.    end {VarDeclaration};
338.
339.
340.    procedure  ListCode;
341.    {列出本程序体生成的代码}
342.    var  i : integer;
343.    begin
344.        {code_start: 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}
345.        for i := code_start to code_count - 1 do
346.            with code[i] do {打印第i条代码}
347.                writeln(intermediate, i:3, code_str[func]:5, level : 3, adr
    : 5)//
348.        {i: 代码序号;
349.         code_str[f]: 功能码的字符串;
350.         l: 相对层号(层差);
351.         a: 相对地址或运算号码}
352.    end {ListCode};
353.
354.
355.    procedure  Statement(symbol_set : SymbolSet);
```

```
356.  var  i, next_node, next_node_2 : integer;
357.
358.  procedure  Expression(symbol_set : SymbolSet);
359.  var  addop : Symbol;
360.
361.  procedure  Term(symbol_set : SymbolSet);
362.  var  mulop : Symbol;
363.
364.  procedure  Factor(symbol_set : SymbolSet);
365.  var i : integer;
366.  begin
367.      Test(factor_begin_symbols, symbol_set, 24);
368.      {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}
369.      while curr_symbol in factor_begin_symbols do
370.         {如果当前的记号是否因子的开始符号}
371.      begin
372.          if curr_symbol = IDENT then {当前记号是标识符}
373.          begin
374.              i := position(id); {查符号表,返回id的入口}
375.              if i = 0 then
376.                  error(11)
377.                  {若在符号表中查不到id, 则出错, 否则,做以下工作}
378.              else
379.                  with table[i] do
380.                  case kind of
381.                      kConstant : GenerateCode(LIT, 0, val);
382.                          {若id是常数, 生成指令,将常数val取到栈顶}
383.                      kVariable : GenerateCode(LOD, lev-level, adr);
384.                          {若id是变量, 生成指令,将该变量取到栈顶;
385.                              lev: 当前语句所在过程的层号;
386.                              level: 定义该变量的过程层号;
387.                              adr: 变量在其过程的数据空间的相对地址}
388.                      kProcedure : error(21)
389.                          {若id是过程名, 则出错}
390.                  end;
391.
392.              GetSymbol {取下一记号}
393.          end
394.          else if curr_symbol = NUMBER then {当前记号是数字}
395.          begin
396.              if num > kAddrMax then {若数值越界,则出错}
397.              begin
398.                  error(30);
399.                  num := 0
400.              end;
```

```
401.                GenerateCode(LIT, 0, num); {生成一条指令, 将常数num取到栈顶}
402.                GetSymbol {取下一记号}
403.            end
404.            else if curr_symbol = LPAREN then {如果当前记号是左括号}
405.            begin
406.                GetSymbol; {取下一记号}
407.                Expression([RPAREN]+symbol_set); {处理表达式}
408.                if curr_symbol = RPAREN then GetSymbol
409.                {如果当前记号是右括号, 则取下一记号,否则出错}
410.                else error(22)
411.            end;
412.
413.            Test(symbol_set, [LPAREN], 23)
414.            {测试当前记号是否同步, 否则出错, 跳过一些记号}
415.        end {while}
416.    end {Factor};
417.
418.
419.    begin {Term}
420.        Factor(symbol_set+[TIMES, SLASH]); {处理项中第一个因子}
421.        while curr_symbol in [TIMES, SLASH] do
422.            {当前记号是"乘"或"除"号}
423.        begin
424.            mulop := curr_symbol; {运算符存入mulop}
425.            GetSymbol; {取下一记号}
426.            Factor(symbol_set+[TIMES, SLASH]); {处理一个因子}
427.            if mulop = TIMES then GenerateCode(OPR, 0, 4)
428.            {若mulop是"乘"号,生成一条乘法指令}
429.                                else GenerateCode(OPR, 0, 5)
430.            {否则, mulop是除号, 生成一条除法指令}
431.        end
432.    end {Term};
433.
434.
435.    begin {Expression}
436.        if curr_symbol in [PLUS, MINUS] then {若第一个记号是加号或减号}
437.        begin
438.            addop := curr_symbol;  {"+"或"-"存入addop}
439.            GetSymbol;
440.            Term(symbol_set+[PLUS, MINUS]); {处理一个项}
441.            if addop = MINUS then GenerateCode(OPR, 0, 1)
442.            {若第一个项前是负号, 生成一条"负运算"指令}
443.        end
444.        else Term(symbol_set+[PLUS, MINUS]);
445.            {第一个记号不是加号或减号, 则处理一个项}
```

```
446.
447.      while curr_symbol in [PLUS, MINUS] do {若当前记号是加号或减号}
448.      begin
449.          addop := curr_symbol; {当前算符存入addop}
450.          GetSymbol; {取下一记号}
451.          Term(symbol_set+[PLUS, MINUS]); {处理一个项}
452.          if addop = PLUS then GenerateCode(OPR, 0, 2)
453.          {若addop是加号, 生成一条加法指令}
454.                      else GenerateCode(OPR, 0, 3)
455.          {否则, addop是减号, 生成一条减法指令}
456.      end
457.  end {Expression};
458.
459.
460.  procedure  Condition(symbol_set : SymbolSet);
461.  var  relop : Symbol;
462.  begin {Condition}
463.      if curr_symbol = ODDSYM then {如果当前记号是"odd"}
464.      begin
465.          GetSymbol;  {取下一记号}
466.          Expression(symbol_set); {处理算术表达式}
467.          GenerateCode(OPR, 0, 6) {生成指令,判定表达式的值是否为奇数,
468.          是,则取"真";不是, 则取"假"}
469.      end
470.      else {如果当前记号不是"odd"}
471.      begin
472.          Expression([EQL, NEQ, LSS, GTR, LEQ, GEQ] + symbol_set);
473.          {处理算术表达式}
474.          if  not (curr_symbol in [EQL, NEQ, LSS, LEQ, GTR, GEQ]) then
475.          {如果当前记号不是关系符, 则出错; 否则,做以下工作}
476.              error(20)
477.          else
478.          begin
479.              relop := curr_symbol; {关系符存入relop}
480.              GetSymbol; {取下一记号}
481.              Expression(symbol_set); {处理关系符右边的算术表达式}
482.              case relop of
483.                  EQL : GenerateCode(OPR, 0, 8);
484.                      {生成指令, 判定两个表达式的值是否相等}
485.                  NEQ : GenerateCode(OPR, 0, 9);
486.                      {生成指令, 判定两个表达式的值是否不等}
487.                  LSS : GenerateCode(OPR, 0, 10);
488.                      {生成指令,判定前一表达式是否小于后一表达式}
489.                  GEQ : GenerateCode(OPR, 0, 11);
490.                      {生成指令,判定前一表达式是否大于等于后一表达式}
```

```
                    GTR : GenerateCode(OPR, 0, 12);
                        {生成指令,判定前一表达式是否大于后一表达式}
                    LEQ : GenerateCode(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 curr_symbol = 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}
            else if table[i].kind <> kVariable then {对非变量赋值, 则出错}
            begin
                error(12);
                i := 0;
            end;

            GetSymbol; {取下一记号}
            if curr_symbol = BECOMES then GetSymbol else error(13);
            {若当前是赋值号, 取下一记号, 否则出错}
            Expression(symbol_set); {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 GenerateCode(STO, lev-level, adr)

        end
        else if curr_symbol = CALLSYM then {处理过程调用语句}
        begin
            GetSymbol; {取下一记号}
            if curr_symbol <> IDENT then error(14) {下一记号不是标识符(过程名),
    出错}
            else
            begin
                i := position(id); {查符号表,返回id在表中的位置}
                if i = 0 then error(11) {在符号表中查不到, 出错}
                else
                    with table[i] do
                        if kind = kProcedure then GenerateCode(CAL, lev-lev
    el, adr)
                        {如果在符号表中id是过程名}
                        else error(15); {若id不是过程名,则出错}

```

```
534.                GetSymbol {取下一记号}
535.            end
536.        end
537.        else if curr_symbol = IFSYM then {处理条件语句}
538.        begin
539.            GetSymbol; {取下一记号}
540.            Condition([THENSYM, DOSYM]+symbol_set); {处理条件表达式}
541.            if curr_symbol = THENSYM then GetSymbol else error(16);
542.            {如果当前记号是"then",则取下一记号; 否则出错}
543.            next_node := code_count; {next_node记录下一代码的地址}
544.            GenerateCode(JPC, 0, 0); {生成指令,表达式为"假"转到某地址(待填),
545.            否则顺序执行}
546.            Statement(symbol_set); {处理一个语句}
547.            code[next_node].adr := code_count
548.            {将下一个指令的地址回填到上面的jpc指令地址栏}
549.        end
550.        else if curr_symbol = BEGINSYM then {处理语句序列}
551.        begin
552.            GetSymbol;
553.            Statement([SEMICOLON, ENDSYM]+symbol_set);
554.                {取下一记号, 处理第一个语句}
555.            while curr_symbol in [SEMICOLON]+stat_begin_symbols do
556.                {如果当前记号是分号或语句的开始符号,则做以下工作}
557.            begin
558.                if curr_symbol = SEMICOLON then GetSymbol else error(10);
559.                    {如果当前记号是分号,则取下一记号, 否则出错}
560.                Statement([SEMICOLON, ENDSYM]+symbol_set) {处理下一个语句}
561.            end;
562.            if curr_symbol = ENDSYM then GetSymbol else error(17)
563.                {如果当前记号是"end",则取下一记号,否则出错}
564.        end
565.        else if curr_symbol = WHILESYM then {处理循环语句}
566.        begin
567.            next_node := code_count; {next_node记录下一指令地址,即条件表达式的
568.            第一条代码的地址}
569.            GetSymbol; {取下一记号}
570.            Condition([DOSYM]+symbol_set); {处理条件表达式}
571.            next_node_2 := code_count; {记录下一指令的地址}
572.            GenerateCode(JPC, 0, 0); {生成一条指令,表达式为"假"转到某地
573.            址(待回填), 否则顺序执行}
574.            if curr_symbol = DOSYM then GetSymbol else error(18);
575.            {如果当前记号是"do",则取下一记号, 否则出错}
576.            Statement(symbol_set); {处理"do"后面的语句}
577.            GenerateCode(JMP, 0, next_node); {生成无条件转移指令, 转移到"while"
    后的
```

```
578.            条件表达式的代码的第一条指令处}
579.            code[next_node_2].adr := code_count
580.            {把下一指令地址回填到前面生成的jpc指令的地址栏}
581.        end
582.        else if curr_symbol = READSYM then {处理读入语句}
583.        begin
584.            GetSymbol;
585.            if curr_symbol <> LPAREN then error(10)
586.            else
587.            begin
588.                repeat
589.                    GetSymbol;
590.                    if curr_symbol <> IDENT then error(4)
591.                    else
592.                    begin
593.                        i := position(id);  {在符号表中查id, 返回id在符号表中的
    入口}
594.                        if i = 0 then error(11) {若在符号表中查不到id, 则出错}
595.                        else if table[i].kind <> kVariable then error(12) {
    对非变量赋值, 则出错}
596.                        else
597.                            with table[i] do GenerateCode(RED, lev-level, ad
    r);
598.                    end;
599.                    GetSymbol;
600.                until curr_symbol <> COMMA;
601.                if curr_symbol <> RPAREN then error(22);
602.                GetSymbol;
603.            end
604.        end
605.        else if curr_symbol = WRITESYM then {处理输出语句}
606.        begin
607.            GetSymbol;
608.            if curr_symbol <> LPAREN then error(10)
609.            else
610.            begin
611.                repeat
612.                    GetSymbol;
613.                    Expression([RPAREN,COMMA] + symbol_set);
614.                    GenerateCode(WRT,0,0);
615.                until curr_symbol <> COMMA;
616.                if curr_symbol <> RPAREN then error(22);
617.                GetSymbol;
618.            end
619.        end;
```

```
620.
621.      Test(symbol_set, [ ], 19)
622.          {测试下一记号是否正常, 否则出错, 跳过一些记号}
623.  end {Statement};
624.
625.
626.  begin {Block}
627.      data_top := 3; {本过程数据空间栈顶指针}
628.      symbol_start := table_top; {标识符表的长度(当前指针)}
629.      table[table_top].adr := code_count; {本过程名的地址, 即下一条指令的序号}
630.      GenerateCode(JMP, 0, 0); {生成一条转移指令}
631.      if lev > kNestingLayersMax then error(32);
632.          {如果当前过程层号>最大层数, 则出错}
633.      repeat
634.          if curr_symbol = CONSTSYM then {处理常数说明语句}
635.          begin
636.              GetSymbol;
637.              repeat
638.                  ConstDeclaration; {处理一个常数说明}
639.                  while curr_symbol = COMMA do {如果当前记号是逗号}
640.                  begin
641.                      GetSymbol;
642.                      ConstDeclaration
643.                  end; {处理下一个常数说明}
644.                  if curr_symbol = SEMICOLON then GetSymbol else error(5)
645.                  {如果当前记号是分号,则常数说明已处理完, 否则出错}
646.              until curr_symbol <> IDENT
647.              {跳过一些记号, 直到当前记号不是标识符(出错时才用到)}
648.          end;
649.
650.          if curr_symbol = VARSYM then {当前记号是变量说明语句开始符号}
651.          begin
652.              GetSymbol;
653.              repeat
654.                  VarDeclaration; {处理一个变量说明}
655.                  while curr_symbol = COMMA do {如果当前记号是逗号}
656.                  begin
657.                      GetSymbol;
658.                      VarDeclaration
659.                  end;
660.                      {处理下一个变量说明}
661.                  if curr_symbol = SEMICOLON then GetSymbol else error(5)
662.                      {如果当前记号是分号,则变量说明已处理完, 否则出错}
663.              until curr_symbol <> IDENT;
664.                  {跳过一些记号, 直到当前记号不是标识符(出错时才用到)}
```

```
665.            end;
666.
667.            while curr_symbol = PROCSYM do {处理过程说明}
668.            begin
669.                GetSymbol;
670.                if curr_symbol = IDENT then {如果当前记号是过程名}
671.                begin
672.                    Enter(kProcedure);
673.                    GetSymbol
674.                end {把过程名填入符号表}
675.                else error(4); {否则, 缺少过程名出错}
676.
677.                if curr_symbol = SEMICOLON then GetSymbol else error(5);
678.                    {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}
679.
680.                Block(lev+1, table_top, [SEMICOLON]+symbol_set); {处理过程体}
681.                    {lev+1: 过程嵌套层数加1; table_top: 符号表当前栈顶指针,也是新
    过程符号表起始位置; [SEMICOLON]+symbol_set: 过程体开始和末尾符号集}
682.
683.                if curr_symbol = SEMICOLON then {如果当前记号是分号}
684.                begin
685.                    GetSymbol; {取下一记号}
686.                    Test(stat_begin_symbols+[IDENT, PROCSYM], symbol_set, 6
    )
687.                        {测试当前记号是否语句开始符号或过程说明开始符号,
688.                        否则报告错误6, 并跳过一些记号}
689.                end
690.                else error(5) {如果当前记号不是分号,则出错}
691.            end;
692.            //writeln('Ha??');
693.            Test(stat_begin_symbols+[IDENT], declare_symbols, 7)
694.                {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}
695.
696.        until  not (curr_symbol in declare_symbols);
697.        {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
698.        的开始符号}
699.        code[table[symbol_start].adr].adr := code_count;  {table[symbol_sta
    rt].addr是本过程名的第1条
700.            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
701.            1条代码)的地址回填到该jmp指令中,得(JMP, 0, code_count)}
702.
703.        with table[symbol_start] do {本过程名的第1条代码的地址改为下一指令地址cx}
704.        begin
705.            adr := code_count; {代码开始地址}
706.        end;
```

```
707.        code_start := code_count; {code_start记录起始代码地址}
708.        GenerateCode(INT, 0, data_top); {生成一条指令, 在栈顶为本过程留出数据空间
    }
709.        Statement([SEMICOLON, ENDSYM]+symbol_set); {处理一个语句}
710.        GenerateCode(OPR, 0, 0); {生成返回指令}
711.        Test(symbol_set, [ ], 8); {测试过程体语句后的符号是否正常,否则出错}
712.        ListCode; {打印本过程的中间代码序列}
713.    end  {Block};
714.
715.
716.
717.    procedure  Interpret;
718.    const  kStackSize = 500; {运行时数据空间(栈)的上界}
719.    var  pc, base, top : integer; {程序地址寄存器, 基地址寄存器,栈顶地址寄存器}
720.         i : Instruction; {指令寄存器}
721.         stack : array [1..kStackSize] of integer; {数据存储栈}
722.
723.    function  BaseOf(lev : integer) : integer;
724.    var  b1 : integer;
725.    begin {BaseOf}
726.        b1 := base; {顺静态链求层差为lev的外层的基地址}
727.        while lev > 0 do
728.        begin
729.            b1 := stack[b1];
730.            lev := lev - 1
731.        end;
732.        BaseOf := b1
733.    end; {BaseOf}
734.
735.    begin  {Interpret}
736.        writeln('START PL/0');
737.        top := 0; {栈顶地址寄存器}
738.        base := 1; {基地址寄存器}
739.        pc := 0; {程序地址寄存器}
740.        stack[1] := 0;
741.        stack[2] := 0;
742.        stack[3] := 0;
743.            {最外层主程序数据空间栈最下面预留三个单元}
744.            {每个过程运行时的数据空间的前三个单元是:SL, DL, RA;
745.            SL: 指向本过程静态直接外层过程的SL单元;
746.            DL: 指向调用本过程的过程的最新数据空间的第一个单元;
747.            RA: 返回地址 }
748.        repeat
749.            i := code[pc]; {i取程序地址寄存器p指示的当前指令}
750.            pc := pc+1; {程序地址寄存器p加1,指向下一条指令}
```

```
751.          with i do
752.              case func of
753.                  LIT :
754.                      begin {当前指令是取常数指令(LIT, 0, a)}
755.                          top := top+1;
756.                          stack[top] := adr
757.                      end; {栈顶指针加1, 把常数a取到栈顶}
758.
759.                  OPR :
760.                      case adr of {当前指令是运算指令(OPR, 0, a)}
761.                          0 : begin {a=0时,是返回调用过程指令}
762.                                  top := base-1; {恢复调用过程栈顶}
763.                                  pc := stack[top+3]; {程序地址寄存器p取返回地
    址}
764.                                  base := stack[top+2];
765.                                      {基地址寄存器b指向调用过程的基地址}
766.                              end;
767.                          1 : stack[top] := -stack[top]; {一元负运算, 栈顶元
    素的值反号}
768.                          2 : begin {加法}
769.                                  top := top-1;
770.                                  stack[top] := stack[top] + stack[top+1]
771.                              end;
772.                          3 : begin {减法}
773.                                  top := top-1;
774.                                  stack[top] := stack[top]-stack[top+1]
775.                              end;
776.                          4 : begin {乘法}
777.                                  top := top-1;
778.                                  stack[top] := stack[top] * stack[top+1]
779.                              end;
780.                          5 : begin {整数除法}
781.                                  top := top-1;
782.                                  stack[top] := stack[top] div stack[top+1
    ]
783.                              end;
784.                          6 : stack[top] := ord(odd(stack[top])); {算s[top
    ]是否奇数, 是则s[top]=1, 否则s[top]=0}
785.
786.                          8 : begin
787.                                  top := top-1;
788.                                  stack[top] := ord(stack[top] = stack[top
    +1])
789.                              end; {判两个表达式的值是否相等,
790.                                  是则s[top]=1, 否则s[top]=0}
```

```
791.
792.                         9:  begin
793.                                 top := top-1;
794.                                 stack[top] := ord(stack[top] <> stack[to
    p+1])
795.                             end; {判两个表达式的值是否不等,
796.                                 是则s[top]=1, 否则s[top]=0}
797.                         10: begin
798.                                 top := top-1;
799.                                 stack[top] := ord(stack[top] < stack[top
    +1])
800.                             end; {判前一表达式是否小于后一表达式,
801.                                 是则s[top]=1, 否则s[top]=0}
802.
803.                         11: begin
804.                                 top := top-1;
805.                                 stack[top] := ord(stack[top] >= stack[to
    p+1])
806.                             end; {判前一表达式是否大于或等于后一表达式,
807.                                 是则s[top]=1, 否则s[top]=0}
808.
809.                         12: begin
810.                                 top := top-1;
811.                                 stack[top] := ord(stack[top] > stack[top
    +1])
812.                             end; {判前一表达式是否大于后一表达式,
813.                                 是则s[top]=1, 否则s[top]=0}
814.                         13: begin
815.                                 top := top-1;
816.                                 stack[top] := ord(stack[top] <= stack[to
    p+1])
817.                             end; {判前一表达式是否小于或等于后一表达式,
818.                                 是则s[top]=1, 否则s[top]=0}
819.                     end;
820.
821.                 LOD :
822.                     begin {当前指令是取变量指令(LOD, l, a)}
823.                         top := top + 1;
824.                         stack[top] := stack[BaseOf(level) + adr]
825.                         {栈顶指针加1, 根据静态链SL,将层差为l,相对地址
826.                         为a的变量值取到栈顶}
827.                     end;
828.                 STO :
829.                     begin {当前指令是保存变量值(STO, l, a)指令}
830.                         stack[BaseOf(level) + adr] := stack[top];
```

```
                            writeln(stack_data, stack[top]);
                            {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}
                            top := top - 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}
                            stack[top+1] := BaseOf(level);
                                {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}
                            stack[top+2] := base;
                            {调用过程的数据空间的起始地址存入本过程DL单元}
                            stack[top+3] := pc;
                            {调用过程cal指令的下一条的地址存入本过程RA单元}
                            base := top+1; {b指向被调用过程新的数据空间起始地址}
                            pc := adr {指令地址寄存储器指向被调用过程的地址a}
                        end;
                    INT : top := top + adr;
                        {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 : pc := adr;
                        {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 stack[top] = 0 then pc := adr;
                            {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}
                            top := top-1 {数据栈顶指针减1}
                        end;
                    RED :
                        begin
                            read(pl0_input, stack[BaseOf(level)+adr]);
                        end;
                    WRT :
                        begin
                            writeln(stack[top]);
                            top := top - 1
                        end
                end {with, case}
        until pc = 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}
        writeln('END PL/0');
    end; {Interpret}
```

```
875.  begin  {主程序}
876.      assign(input, 'pl0_src.pas');
877.      reset(input);
878.
879.      assign(intermediate, 'intermediate_code.txt');
880.      rewrite(intermediate);
881.
882.      assign(stack_data, 'stack_data.txt');
883.      rewrite(stack_data);
884.
885.      assign(pl0_input, 'input.txt');
886.      reset(pl0_input);
887.
888.      for curr_char:= 'a' to ';' do  ssym[curr_char] := NUL;
889.      {ASCII码的顺序}
890.      words[1] := 'begin';
891.      words[2] := 'call';
892.      words[3] := 'const';
893.      words[4] := 'do';
894.      words[5] := 'end';
895.      words[6] := 'if';
896.      words[7] := 'odd';
897.      words[8] := 'procedure';
898.      words[9] := 'read';
899.      words[10] := 'then';
900.      words[11] := 'var';
901.      words[12] := 'while';
902.      words[13] := 'write';
903.      word_symbol[1] := BEGINSYM;
904.      word_symbol[2] := CALLSYM;
905.      word_symbol[3] := CONSTSYM;
906.      word_symbol[4] := DOSYM;
907.      word_symbol[5] := ENDSYM;
908.      word_symbol[6] := IFSYM;
909.      word_symbol[7] := ODDSYM;
910.      word_symbol[8] := PROCSYM;
911.      word_symbol[9] := READSYM;
912.      word_symbol[10] := THENSYM;
913.      word_symbol[11] := VARSYM;
914.      word_symbol[12] := WHILESYM;
915.      word_symbol[13] := WRITESYM;
916.      ssym['+'] := PLUS;      ssym['-'] := MINUS;
917.      ssym['*'] := TIMES;     ssym['/'] := SLASH;
918.      ssym['('] := LPAREN;     ssym[')'] := RPAREN;
919.      ssym['='] := EQL;       ssym[','] := COMMA;
```

```
    ssym['.'] := PERIOD;
    ssym['<'] := LSS;       ssym['>'] := GTR;

    ssym[';'] := SEMICOLON;
    {算符和标点符号的记号}
    code_str[LIT] := 'LIT';     code_str[OPR] := 'OPR';
    code_str[LOD] := 'LOD';    code_str[STO] := 'STO';
    code_str[CAL] := 'CAL';    code_str[INT] := 'INT';
    code_str[JMP] := 'JMP';    code_str[JPC] := 'JPC';
    code_str[RED] := 'RED';    code_str[WRT] := 'WRT';

    {中间代码指令的字符串}
    declare_symbols := [CONSTSYM, VARSYM, PROCSYM];
    {说明语句的开始符号}
    stat_begin_symbols := [BEGINSYM, CALLSYM, IFSYM, WHILESYM, READSYM,
WRITESYM];
    {语句的开始符号}
    factor_begin_symbols := [IDENT, NUMBER, LPAREN];
    {因子的开始符号}


    error_count := 0; {发现错误的个数}
    char_count := 0; {当前行中输入字符的指针}
    code_count := 0; {代码数组的当前指针}
    line_length := 0; {输入当前行的长度}
    curr_char:= ' '; {当前输入的字符}
    GetSymbol; {取下一个记号}

    Block(0, 0, [PERIOD] + declare_symbols + stat_begin_symbols); {处理
程序体}

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

    if error_count = 0 then Interpret
    {如果编译无错误, 则解释执行中间代码}
    else writeln(error_count, ' ERROR(S) IN PL/0 PROGRAM');

    close(intermediate);
    close(stack_data);
end.
```

2. PL0源程序代码

这是一个输入x和y，输出x除以y的商和余数的程序。

```
var  x, y, q, r;
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;
begin
  read(x); read(y);
  call divide;
  write(q);
  write(r);
end.
```

3. 中间代码

2 INT 0 4
3 LOD 1 3
4 STO 1 6
5 LIT 0 0
6 STO 1 5
7 LOD 1 4
8 STO 0 3
9 LOD 0 3
10 LOD 1 6
11 OPR 0 13

12 JPC 0 18
13 LIT 0 2
14 LOD 0 3
15 OPR 0 4
16 STO 0 3
17 JMP 0 9
18 LOD 0 3
19 LOD 1 4
20 OPR 0 12
21 JPC 0 43
22 LIT 0 2
23 LOD 1 5
24 OPR 0 4
25 STO 1 5
26 LOD 0 3
27 LIT 0 2
28 OPR 0 5
29 STO 0 3
30 LOD 0 3
31 LOD 1 6
32 OPR 0 13
33 JPC 0 42
34 LOD 1 6
35 LOD 0 3
36 OPR 0 3
37 STO 1 6
38 LOD 1 5
39 LIT 0 1
40 OPR 0 2
41 STO 1 5
42 JMP 0 18
43 OPR 0 0
44 INT 0 7

45 RED 0 3
46 RED 0 4
47 CAL 0 2
48 LOD 0 5
49 WRT 0 0
50 LOD 0 6
51 WRT 0 0
52 OPR 0 0

4. 输入输出的数据

输入：29 8
输出：3 5