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 : SymbolSet;
    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
        else
        begin
            curr_symbol := words_symbol[j] ;
            //writeln('find reserved: ', id);
        end
    end

    else if curr_char in ['0'..'9'] then
    begin {数字}
        k := 0;
        num := 0;
        curr_symbol := NUMBER; {当前记号sym为数字}

        repeat {计算数字串的值}
            num := 10*num + (ord(curr_char)-ord('0'));
            k := k + 1;
            GetChar;
        until  not(curr_char in ['0'..'9']);

        {当前数字串的长度超过上界,则报告错误}
        if k > kNumLengthMax then  error(30);
        //writeln('find number: ', num);{debug}
    end
    else if curr_char= ':' then {处理赋值号}
    begin
        GetChar;
        if curr_char= '=' then
        begin
            curr_symbol := BECOMES;
            GetChar
        end
        else
            curr_symbol := NUL;
    end
    else if curr_char = '<' then
    begin
        GetChar;
        if curr_char = '>' then {处理不等号}
        begin
            curr_symbol := NEQ;
            GetChar;
        end
        else if curr_char = '=' then {处理小于等于号}
        begin
            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
```

```
    if  not (curr_symbol in s1) then
    begin
        error(n);
        s1 := s1 + s2;
        while not (curr_symbol in s1) do GetSymbol
    end
end {Test};


procedure  Block(lev, table_top : integer; symbol_set : SymbolSet); {程序体}
var
    data_top : integer; {本过程数据空间分配下标} {栈顶指针}
    symbol_start : integer; {本过程标识表起始下标}
    code_start : integer; {本过程代码起始下标}

procedure  Enter(k : ObjectType);
begin {把obj填入符号表中}
    table_top := table_top + 1; {符号表指针加1}

    with table[table_top] 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
            kConstant :
                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
            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};


```

```
begin {Term}
    Factor(symbol_set+[TIMES, SLASH]); {处理项中第一个因子}
    while curr_symbol in [TIMES, SLASH] do
        {当前记号是“乘”或“除”号}
    begin
        mulop := curr_symbol; {运算符存入mulop}
        GetSymbol; {取下一记号}
        Factor(symbol_set+[TIMES, SLASH]); {处理一个因子}
        if mulop = TIMES then GenerateCode(OPR, 0, 4)
        {若mulop是“乘”号,生成一条乘法指令}
                         else GenerateCode(OPR, 0, 5)
        {否则， mulop是除号，生成一条除法指令}
    end
end {Term};


begin {Expression}
    if curr_symbol in [PLUS, MINUS] then {若第一个记号是加号或减号}
    begin
        addop := curr_symbol;  {“+”或“-”存入addop}
        GetSymbol;
        Term(symbol_set+[PLUS, MINUS]); {处理一个项}
        if addop = MINUS then GenerateCode(OPR, 0, 1)
        {若第一个项前是负号，生成一条“负运算”指令}
    end
    else Term(symbol_set+[PLUS, MINUS]);
        {第一个记号不是加号或减号，则处理一个项}

    while curr_symbol in [PLUS, MINUS] do {若当前记号是加号或减号}
    begin
        addop := curr_symbol; {当前算符存入addop}
        GetSymbol; {取下一记号}
        Term(symbol_set+[PLUS, MINUS]); {处理一个项}
        if addop = PLUS then GenerateCode(OPR, 0, 2)
        {若addop是加号，生成一条加法指令}
                    else GenerateCode(OPR, 0, 3)
        {否则， addop是减号，生成一条减法指令}
    end
end {Expression};


procedure  Condition(symbol_set : SymbolSet);
var  relop : Symbol;
begin {Condition}
    if curr_symbol = ODDSYM then {如果当前记号是“odd”}
    begin
        GetSymbol;  {取下一记号}
        Expression(symbol_set); {处理算术表达式}
        GenerateCode(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] + symbol_set);
        {处理算术表达式}
        if  not (curr_symbol 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 := curr_symbol; {关系符存入relop}
            GetSymbol; {取下一记号}
            Expression(symbol_set); {处理关系符右边的算术表达式}
            case relop of
                EQL : GenerateCode(OPR, 0, 8);
                    {生成指令，判定两个表达式的值是否相等}
                NEQ : GenerateCode(OPR, 0, 9);
                    {生成指令，判定两个表达式的值是否不等}
                LSS : GenerateCode(OPR, 0, 10);
                    {生成指令,判定前一表达式是否小于后一表达式}
                GEQ : GenerateCode(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 : 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-level, 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}

            GetSymbol {取下一记号}
        end
    end
    else if curr_symbol = IFSYM then {处理条件语句}
    begin
        GetSymbol; {取下一记号}
        Condition([THENSYM, DOSYM]+symbol_set); {处理条件表达式}
        if curr_symbol = THENSYM then GetSymbol else error(16);
        {如果当前记号是“then”,则取下一记号；否则出错}
        next_node := code_count; {next_node记录下一代码的地址}
        GenerateCode(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(symbol_set); {处理一个语句}
        code[next_node].adr := code_count
        {将下一个指令的地址回填到上面的jpc指令地址栏}
    end
    else if curr_symbol = BEGINSYM then {处理语句序列}
    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
```

```
        if curr_symbol = CONSTSYM then {处理常数说明语句}
        begin
            GetSymbol;
            repeat
                ConstDeclaration; {处理一个常数说明}
                while curr_symbol = COMMA do {如果当前记号是逗号}
                begin
                    GetSymbol;
                    ConstDeclaration
                end; {处理下一个常数说明}
                if curr_symbol = SEMICOLON then GetSymbol 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 curr_symbol <> 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 curr_symbol = VARSYM then {当前记号是变量说明语句开始符号}
        begin
            GetSymbol;
            repeat
                VarDeclaration; {处理一个变量说明}
                while curr_symbol = COMMA do {如果当前记号是逗号}
                begin
                    GetSymbol;
                    VarDeclaration
                end;
                    {处理下一个变量说明}
                if curr_symbol = SEMICOLON then GetSymbol 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 curr_symbol <> 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 curr_symbol = PROCSYM do {处理过程说明}
        begin
            GetSymbol;
            if curr_symbol = IDENT then {如果当前记号是过程名}
            begin
                Enter(kProcedure);
                GetSymbol
            end {把过程名填入符号表}
            else error(4); {否则， 缺少过程名出错}

            if curr_symbol = SEMICOLON then GetSymbol 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, table_top, [SEMICOLON]+symbol_set); {处理过程体}
                {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}

            if curr_symbol = SEMICOLON then {如果当前记号是分号}
            begin
                GetSymbol; {取下一记号}
                Test(stat_begin_symbols+[IDENT, PROCSYM], symbol_set, 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;
        //writeln('Ha??');
        Test(stat_begin_symbols+[IDENT], declare_symbols, 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 (curr_symbol in declare_symbols);
    {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[symbol_start].adr].adr := code_count;  {table[symbol_start].addr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, code_count)}

    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}
    begin
        adr := code_count; {代码开始地址}
    end;
    code_start := code_count; {code_start记录起始代码地址}
    GenerateCode(INT, 0, data_top); {生成一条指令, 在栈顶为本过程留出数据空间}
    Statement([SEMICOLON, ENDSYM]+symbol_set); {处理一个语句}
    GenerateCode(OPR, 0, 0); {生成返回指令}
    Test(symbol_set, [ ], 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  kStackSize = 500; {运行时数据空间(栈)的上界}
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}
```

```
     i : Instruction; {指令寄存器}
     stack : array [1..kStackSize] of integer; {数据存储栈}

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

begin  {Interpret}
    writeln('START PL/0');
    top := 0; {栈顶地址寄存器}
    base := 1; {基地址寄存器}
    pc := 0; {程序地址寄存器}
    stack[1] := 0;
    stack[2] := 0;
    stack[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[pc]; {i取程序地址寄存器p指示的当前指令}
        pc := pc+1; {程序地址寄存器p加1,指向下一条指令}
        with i do
            case func of
                LIT :
                    begin {当前指令是取常数指令(LIT, 0, a)}
                        top := top+1;
                        stack[top] := adr
                    end; {栈顶指针加1, 把常数a取到栈顶}

                OPR :
                    case adr 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}
                                top := base-1; {恢复调用过程栈顶}
                                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[top+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[top+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';
    words_symbol[1] := BEGINSYM;   words_symbol[2] := CALLSYM;
    words_symbol[3] := CONSTSYM;   words_symbol[4] := DOSYM;
    words_symbol[5] := ENDSYM;    words_symbol[6] := IFSYM;
    words_symbol[7] := ODDSYM;    words_symbol[8] := PROCSYM;
    words_symbol[9] := THENSYM;    words_symbol[10] := VARSYM;
    words_symbol[11] := WHILESYM;
    ssym['+'] := PLUS;      ssym['-'] := MINUS;
    ssym['*'] := TIMES;     ssym['/'] := SLASH;
    ssym['('] := LPAREN;     ssym[')'] := RPAREN;
    ssym['='] := EQL;       ssym[','] := COMMA;
    ssym['.'] := PERIOD;
    ssym['<'] := LSS;       ssym['>'] := GTR;

    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}
    declare_symbols := [CONSTSYM, VARSYM, PROCSYM];
    {说明语句的开始符号}
    stat_begin_symbols := [BEGINSYM, CALLSYM, IFSYM, WHILESYM];
    {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

```
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

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, WRITESYM);
    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}
    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}
    word_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}
    code_str : array [FunctionCode] of string;
    {中间代码算符的字符串}
    declare_symbols, stat_begin_symbols, factor_begin_symbols : SymbolSet;
    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;
                end;
                GetChar;
            until  not(curr_char in ['a'..'z', '0'..'9']);

            {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
            else
            begin
                curr_symbol := word_symbol[j] ;
                //writeln('find reserved: ', id);
            end


        end

        else if curr_char in ['0'..'9'] then
        begin {数字}
            k := 0;
            num := 0;
            curr_symbol := NUMBER; {当前记号sym为数字}

            repeat {计算数字串的值}
                num := 10*num + (ord(curr_char)-ord('0'));
                k := k + 1;
                GetChar;
            until  not(curr_char in ['0'..'9']);

            {当前数字串的长度超过上界,则报告错误}
            if k > kNumLengthMax then  error(30);
            //writeln('find number: ', num);{debug}
        end
        else if curr_char= ':' then {处理赋值号}
        begin
            GetChar;
            if curr_char= '=' then
            begin
                curr_symbol := BECOMES;
                GetChar
            end
            else
                curr_symbol := NUL;
        end
        else if curr_char = '<' then
        begin
            GetChar;
            if curr_char = '>' then {处理不等号}
            begin
                curr_symbol := NEQ;
                GetChar;
            end
            else if curr_char = '=' then {处理小于等于号}
            begin
                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
    if  not (curr_symbol in s1) then
    begin
        error(n);
        s1 := s1 + s2;
        while not (curr_symbol in s1) do GetSymbol
    end
end {Test};


procedure  Block(lev, table_top : integer; symbol_set : SymbolSet); {程序体}
var
    data_top : integer; {本过程数据空间分配下标} {栈顶指针}
    symbol_start : integer; {本过程标识表起始下标}
    code_start : integer; {本过程代码起始下标}

procedure  Enter(k : ObjectType);
begin {把obj填入符号表中}
    table_top := table_top + 1; {符号表指针加1}

    with table[table_top] 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
            kConstant :
                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
```

```
            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, code_str[func]:5, level : 3, adr : 5)//
    {i: 代码序号;
     code_str[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};


begin {Term}
    Factor(symbol_set+[TIMES, SLASH]); {处理项中第一个因子}
    while curr_symbol in [TIMES, SLASH] do
        {当前记号是“乘”或“除”号}
    begin
        mulop := curr_symbol; {运算符存入mulop}
        GetSymbol; {取下一记号}
        Factor(symbol_set+[TIMES, SLASH]); {处理一个因子}
        if mulop = TIMES then GenerateCode(OPR, 0, 4)
        {若mulop是“乘”号,生成一条乘法指令}
                        else GenerateCode(OPR, 0, 5)
        {否则，mulop是除号，生成一条除法指令}
    end
end {Term};


begin {Expression}
    if curr_symbol in [PLUS, MINUS] then {若第一个记号是加号或减号}
    begin
        addop := curr_symbol;  {“+”或“-”存入addop}
        GetSymbol;
        Term(symbol_set+[PLUS, MINUS]); {处理一个项}
        if addop = MINUS then GenerateCode(OPR, 0, 1)
        {若第一个项前是负号，生成一条“负运算”指令}
    end
    else Term(symbol_set+[PLUS, MINUS]);
        {第一个记号不是加号或减号，则处理一个项}

    while curr_symbol in [PLUS, MINUS] do {若当前记号是加号或减号}
    begin
        addop := curr_symbol; {当前算符存入addop}
        GetSymbol; {取下一记号}
        Term(symbol_set+[PLUS, MINUS]); {处理一个项}
        if addop = PLUS then GenerateCode(OPR, 0, 2)
        {若addop是加号, 生成一条加法指令}
                    else GenerateCode(OPR, 0, 3)
        {否则, addop是减号, 生成一条减法指令}
    end
end {Expression};


procedure  Condition(symbol_set : SymbolSet);
var  relop : Symbol;
begin {Condition}
    if curr_symbol = ODDSYM then {如果当前记号是"odd"}
    begin
        GetSymbol;  {取下一记号}
        Expression(symbol_set); {处理算术表达式}
        GenerateCode(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] + symbol_set);
        {处理算术表达式}
        if  not (curr_symbol 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 := curr_symbol; {关系符存入relop}
            GetSymbol; {取下一记号}
            Expression(symbol_set); {处理关系符右边的算术表达式}
            case relop of
                EQL : GenerateCode(OPR, 0, 8);
                    {生成指令, 判定两个表达式的值是否相等}
                NEQ : GenerateCode(OPR, 0, 9);
                    {生成指令, 判定两个表达式的值是否不等}
                LSS : GenerateCode(OPR, 0, 10);
                    {生成指令,判定前一表达式是否小于后一表达式}
                GEQ : GenerateCode(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 : 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-level, 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}

            GetSymbol {取下一记号}
        end
    end
    else if curr_symbol = IFSYM then {处理条件语句}
    begin
        GetSymbol; {取下一记号}
        Condition([THENSYM, DOSYM]+symbol_set); {处理条件表达式}
        if curr_symbol = THENSYM then GetSymbol else error(16);
        {如果当前记号是"then",则取下一记号; 否则出错}
        next_node := code_count; {next_node记录下一代码的地址}
        GenerateCode(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(symbol_set); {处理一个语句}
        code[next_node].adr := code_count
        {将下一个指令的地址回填到上面的jpc指令地址栏}
    end
    else if curr_symbol = BEGINSYM then {处理语句序列}
    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
    else if curr_symbol = READSYM then {处理读入语句}
    begin
        GetSymbol;
        if curr_symbol <> LPAREN then error(10)
        else
        begin
            repeat
                GetSymbol;
                if curr_symbol <> IDENT then error(4)
                else
                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 error(12) {对非变量赋值，则出错}
                    else
                        with table[i] do GenerateCode(RED, lev-level, adr);
                end;
                GetSymbol;
            until curr_symbol <> COMMA;
            if curr_symbol <> RPAREN then error(22);
            GetSymbol;
        end
    end
    else if curr_symbol = WRITESYM then {处理输出语句}
    begin
        GetSymbol;
        if curr_symbol <> LPAREN then error(10)
        else
        begin
            repeat
                GetSymbol;
                Expression([RPAREN,COMMA] + symbol_set);
                GenerateCode(WRT,0,0);
            until curr_symbol <> COMMA;
            if curr_symbol <> RPAREN then error(22);
            GetSymbol;
        end
    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
        if curr_symbol = CONSTSYM then {处理常数说明语句}
        begin
            GetSymbol;
            repeat
                ConstDeclaration; {处理一个常数说明}
                while curr_symbol = COMMA do {如果当前记号是逗号}
                begin
                    GetSymbol;
                    ConstDeclaration
                end; {处理下一个常数说明}
                if curr_symbol = SEMICOLON then GetSymbol 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 curr_symbol <> 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 curr_symbol = VARSYM then {当前记号是变量说明语句开始符号}
        begin
            GetSymbol;
            repeat
                VarDeclaration; {处理一个变量说明}
                while curr_symbol = COMMA do {如果当前记号是逗号}
                begin
                    GetSymbol;
                    VarDeclaration
                end;
                    {处理下一个变量说明}
                if curr_symbol = SEMICOLON then GetSymbol 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 curr_symbol <> 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 curr_symbol = PROCSYM do {处理过程说明}
        begin
            GetSymbol;
            if curr_symbol = IDENT then {如果当前记号是过程名}
            begin
                Enter(kProcedure);
                GetSymbol
            end {把过程名填入符号表}
            else error(4); {否则, 缺少过程名出错}

            if curr_symbol = SEMICOLON then GetSymbol 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, table_top, [SEMICOLON]+symbol_set); {处理过程体}
                {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}

            if curr_symbol = SEMICOLON then {如果当前记号是分号}
            begin
                GetSymbol; {取下一记号}
                Test(stat_begin_symbols+[IDENT, PROCSYM], symbol_set, 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;
        //writeln('Ha??');
        Test(stat_begin_symbols+[IDENT], declare_symbols, 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 (curr_symbol in declare_symbols);
    {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[symbol_start].adr].adr := code_count;  {table[symbol_start].addr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, code_count)}

    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}
    begin
        adr := code_count; {代码开始地址}
    end;
    code_start := code_count; {code_start记录起始代码地址}
    GenerateCode(INT, 0, data_top); {生成一条指令， 在栈顶为本过程留出数据空间}
    Statement([SEMICOLON, ENDSYM]+symbol_set); {处理一个语句}
    GenerateCode(OPR, 0, 0); {生成返回指令}
    Test(symbol_set, [ ], 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  kStackSize = 500; {运行时数据空间(栈)的上界}
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}
     i : Instruction; {指令寄存器}
     stack : array [1..kStackSize] of integer; {数据存储栈}

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

begin  {Interpret}
    writeln('START PL/0');
    top := 0; {栈顶地址寄存器}
    base := 1; {基地址寄存器}
    pc := 0; {程序地址寄存器}
    stack[1] := 0;
    stack[2] := 0;
    stack[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[pc]; {i取程序地址寄存器p指示的当前指令}
        pc := pc+1; {程序地址寄存器p加1,指向下一条指令}
        with i do
            case func of
                LIT :
                    begin {当前指令是取常数指令(LIT, 0, a)}
                        top := top+1;
                        stack[top] := adr
                    end; {栈顶指针加1，把常数a取到栈顶}

                OPR :
                    case adr 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}
                                top := base-1; {恢复调用过程栈顶}
                                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[top+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[top+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;
                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}
```

```pascal
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);

    assign(pl0_input, 'input.txt');
    reset(pl0_input);

    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] := 'read';
    words[10] := 'then';
    words[11] := 'var';
    words[12] := 'while';
    words[13] := 'write';
    word_symbol[1] := BEGINSYM;
    word_symbol[2] := CALLSYM;
    word_symbol[3] := CONSTSYM;
    word_symbol[4] := DOSYM;
    word_symbol[5] := ENDSYM;
    word_symbol[6] := IFSYM;
    word_symbol[7] := ODDSYM;
    word_symbol[8] := PROCSYM;
    word_symbol[9] := READSYM;
    word_symbol[10] := THENSYM;
    word_symbol[11] := VARSYM;
    word_symbol[12] := WHILESYM;
    word_symbol[13] := WRITESYM;
    ssym['+'] := PLUS;      ssym['-'] := MINUS;
    ssym['*'] := TIMES;     ssym['/'] := SLASH;
    ssym['('] := LPAREN;     ssym[')'] := RPAREN;
    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