# 编译原理大作业

学号：16340173

姓名：欧穗新

目录

[编译原理大作业 1](#_Toc534227358)

[PL0编译程序代码-第一阶段 2](#_Toc534227359)

[PL0源程序代码-第一阶段 21](#_Toc534227360)

[中间代码-第一阶段 22](#_Toc534227361)

[栈中数据-第一阶段 25](#_Toc534227362)

[PL0编译程序代码-第二阶段 27](#_Toc534227363)

[PL0源程序代码-第二阶段 48](#_Toc534227364)

[中间代码-第二阶段 50](#_Toc534227365)

[输入输出-第二阶段 53](#_Toc534227366)

## PL0编译程序代码-第一阶段

program PL0 (input, output);

{带有代码生成的PL0编译程序}

const

norw = 11; {保留字的个数}

txmax = 100; {标识符表长度}

nmax = 14; {数字的最大位数}

al = 10; {标识符的长度}

amax = 2047; {最大地址}

levmax = 3; {程序体嵌套的最大深度}

cxmax = 200; {代码数组的大小}

type

symbol = (nul, ident, number, plus, minus, times, slash, oddsym,

eql, neq, lss, leq, gtr, geq, lparen, rparen, comma, semicolon,

period, becomes, beginsym, endsym, ifsym, thensym,

whilesym, dosym, callsym, constsym, varsym, procsym);

alfa = packed array [1..al] of char;

myObject = (constant, variable, myProcedure);

symset = set of symbol;

fct = (lit, opr, lod, sto, cal, int, jmp, jpc); {functions}

instruction = packed record

f : fct; {功能码}

l : 0..levmax; {相对层数}

a : 0..amax; {相对地址}

end;

{LIT 0,a : 取常数a

OPR 0,a : 执行运算a

LOD l,a : 取层差为l的层﹑相对地址为a的变量

STO l,a : 存到层差为l的层﹑相对地址为a的变量

CAL l,a : 调用层差为l的过程

INT 0,a : t寄存器增加a

JMP 0,a : 转移到指令地址a处

JPC 0,a : 条件转移到指令地址a处 }

var

ch : char; {最近读到的字符}

sym : symbol; {最近读到的符号}

id : alfa; {最近读到的标识符}

num : integer; {最近读到的数}

cc : integer; {当前行的字符计数}

ll : integer; {当前行的长度}

kk, err : integer;

cx : integer; {代码数组的当前下标}

line : array [1..81] of char; {当前行,存储的当前行的字符串}

a : alfa; {当前标识符的字符串}

code : array [0..cxmax] of instruction; {中间代码数组}

word : array [1..norw] of alfa; {存放保留字的字符串}

wsym : array [1..norw] of symbol; {存放保留字的记号}

ssym : array [char] of symbol; {存放算符和标点符号的记号}

mnemonic : array [fct] of packed array [1..5] of char;

{中间代码算符的字符串}

declbegsys, statbegsys, facbegsys : symset;

table : array [0..txmax] of record {符号表}

name : alfa;

case kind : myObject of

constant : (val : integer);

variable, myProcedure : (level, adr : integer);

end; {这里有改动,不一定能对}

fin, fout : text;

file1, file2 : string;

{----------------------------------------------------------------}

procedure error (n : integer);

begin

writeln('\*\*\*\*', ' ' : cc-1, '^', n : 2); {cc为当前行已读的字符数, n为错误号}

writeln(fout, '\*\*\*\*', ' ' : cc-1, '^', n : 2);

err := err + 1; {错误数err加1}

end; {error}

{----------------------------------------------------------------}

procedure getsym; {读取一个符号（标识符）}

var i, j, k : integer;

procedure getch; {读取一个字符}

begin

if cc = ll then {如果读取到行尾,就要先将下一行所有内容填到line数组中,并设置cc为0}

begin

if eof(fin) then {如果读到了文件尾}

begin

write('PROGRAM INCOMPLETE');

write(fout, 'PROGRAM INCOMPLETE');

writeln; writeln(fout);

close(fin); close(fout);

exit();

end;

{上一行结束,重新初始化,读取新行：}

ll := 0;

cc := 0;

write(cx : 5, ' ');

write(fout, cx : 5, ' ');

while not eoln(fin) do

begin

ll := ll + 1;

read(fin, ch);

write(ch);

write(fout, ch);

line[ll] := ch

end;

writeln;

writeln(fout);

ll := ll + 1;

readln(fin, line[ll]); {这里将行末符存到 line[ll] 中,即#13}

end;

{从line取出下一个字符到ch,无论之前是否读到行尾：}

cc := cc + 1;

ch := line[cc];

end {getch};

begin {getsym}

while (ch = ' ') or (ch = #13) do {跳过空格或者换行符}

getch;

if ch in ['a'..'z'] then

begin {标识符或保留字(以字母开头)}

k := 0;

repeat

if k < al then

begin

k:= k + 1;

a[k] := ch;

end;

getch;

until not(ch in ['a'..'z', '0'..'9']);

if k >= kk then

kk := k

else

repeat

a[kk] := ' ';

kk := kk-1;

until kk = k;

id := a;

i := 1;

j := norw;

repeat {二分查找当前标识符在保留字表中的对应的记号}

k := (i+j) div 2;

if id <= word[k] then j := k-1;

if id >= word[k] then i := k + 1;

until i > j;

if i-1 > j then

sym := wsym[k] {如果找到了,这时候因为前面repeat里面两个if都满足,所以i=k+1,j=k-1,i-1=k>j,然后在这里被置为wsym[k]（一个保留字）}

else

sym := ident; {如果没有找到,sym被设置为是一个标识符}

end

else if ch in ['0'..'9'] then

begin {数字}

k := 0;

num := 0;

sym := number;

repeat

num := 10\*num + (ord(ch)-ord('0'));

k := k + 1;

getch;

until not (ch in ['0'..'9']);

if k > nmax then

error(30); {出现越界的常数,出错}

end

else if ch = ':' then

begin

getch;

if ch = '=' then

begin

sym := becomes;

getch;

end

else

sym := nul;

end

else

begin

sym := ssym[ch];

getch;

end

end; {getsym}

{----------------------------------------------------------------}

procedure gen(x : fct; y, z : integer); {使用传入的参数x,y,z生成一条中间代码}

begin

if cx > cxmax then {代码过长,导致指令序列长度超出代码数组容量,报错并退出程序}

begin

write('PROGRAM TOO LONG');

write(fout, 'PROGRAM TOO LONG');

writeln; writeln(fout);

close(fin); close(fout);

exit();

end;

with code[cx] do {生成的中间代码的功能码为x,相对层数为y,相对地址为z}

begin

f := x;

l := y;

a := z;

end;

cx := cx + 1;

end; {gen}

{----------------------------------------------------------------}

procedure test(s1, s2 : symset; n : integer);

begin

if not (sym in s1) then

begin

error(n); {标识符不在s1+s2集合中,出错}

s1 := s1 + s2;

while not (sym in s1) do {跳过不在s1+s2集合中的那些记号}

getsym;

end

end; {test}

{----------------------------------------------------------------}

procedure block(lev, tx : integer; fsys : symset);

var

dx : integer; {本过程数据空间分配下标}

tx0 : integer; {本过程标识表起始下标}

cx0 : integer; {本过程代码起始下标}

procedure enter(k : myObject);

begin {把自定义的 myObject 填入符号表中}

tx := tx +1;

with table[tx] do

begin

name := id; {自定义标识符名字,是一个字符串}

kind := k; {自定义标识符种类,是myObject类型}

case k of

constant : begin {如果该标识符为常数}

if num > amax then {如果常数值超出界限,报错}

begin

error(30); {常量填入符号表时常量数值越界,出错}

num := 0;

end;

val := num; {将该标识符的值赋值为num}

end;

variable : begin {如果该标识符是变量}

level := lev; {该变量的过程嵌套层数即该过程的层数lev}

adr := dx; {该变量的地址为当前过程数据空间栈顶}

dx := dx +1; {栈顶指针加一,下一个变量将继续向上移动,过程数据空间向上增长}

end;

myProcedure :

level := lev {如果该标识符是一个过程名,则该过程的嵌套层数为lev}

end

end

end; {enter}

function position(id : alfa) : integer; {返回该id在符号表table的入口}

var i : integer;

begin {在标识符表中查标识符id}

table[0].name := id; {在符号表栈的最下方预填标识符id}

i := tx; {符号表栈顶指针}

while table[i].name <> id do

i := i-1;

position := i; {如果给出的id在符号表中,那么最后position就是该符号在符号表的入口,否则position就是0}

end; {position}

procedure constdeclaration; {常量声明}

begin

if sym = ident then {当前记号是一个标识符而不是保留字}

begin

getsym;

if sym in [eql, becomes] then {读取的标识符下一个记号是等于号或者赋值号}

begin

if sym = becomes then {如果是赋值号,则出错,因为常量声明使用=而不是:=}

error(1); {常量声明使用:=出错}

getsym; {读取第三个位置的数字,如果不是数字说明出错,如果正确那么就将声明的常数添加到符号表,常数的数值就是=后面跟随的数字}

if sym = number then

begin

enter(constant); {将声明的常数添加到符号表}

getsym;

end

else

error(2); {常量声明,不是数字说明出错}

end

else

error(3); {标识符后面既不是等号也不是赋值号出错}

end

else

error(4); {常量声明时不给出常量标识符,出错}

end; {constdeclaration}

procedure vardeclaration;

begin

if sym = ident then {当前记号是一个标识符而不是保留字}

begin

enter(variable); {将该变量加入到符号表}

getsym;

end

else

error(4); {变量声明时不给出变量标识符,出错}

end {vardeclaration};

procedure listcode;

var i : integer;

begin {列出本程序体生成的代码}

for i := cx0 to cx-1 do {cx0是这个过程体的第一条代码序号,cx-1是最后一条代码序号}

with code[i] do {打印第i条代码到文件/屏幕,其中mnemonic[f]是功能码对应字符串,l是相对层数,a是相对地址/运算符号码}

begin

writeln(i : 3, mnemonic[f] : 5, l : 3, a : 5);

writeln(fout, i : 3, mnemonic[f] : 5, l : 3, a : 5);

end

end; {listcode}

procedure statement(fsys : symset);

var i, cx1, cx2 : integer;

procedure expression(fsys : symset);

var addop : symbol;

procedure term(fsys : symset);

var mulop : symbol;

procedure factor(fsys : symset);

var i : integer;

begin

test(facbegsys, fsys, 24); {测试当前sym是否是是在facbegsys中, 不在就报错,并跳过不在fsys+facbegsys中的记号}

while sym in facbegsys do

begin

if sym = ident then {处理标识符}

begin

i := position(id);

if i = 0 then

error(11) {出现未声明标识符,报错}

else

with table[i] do

case kind of

constant : gen(lit, 0, val); {若id是常数, 生成指令,将常数val取到栈顶}

variable : gen(lod, lev-level, adr);

{若id是变量, 生成指令,将该变量取到栈顶; lev: 当前语句所在过程的层号; level: 定义该变量的过程层号; adr: 变量在其过程的数据空间的相对地址}

myProcedure : error(21) {若id是过程名, 则出错}

end;

getsym;

end

else if sym = number then {处理数字}

begin

if num > amax then

begin

error(30); {数值越界报错}

num := 0;

end;

gen(lit, 0, num); {将常数num取到栈顶}

getsym;

end

else if sym = lparen then {处理左括号}

begin

getsym;

expression([rparen]+fsys); {处理表达式}

if sym = rparen then

getsym

else

error(22); {左括号和表达式处理完了需要处理右括号,如果下一个符号不是右括号说明出错,括号不匹配错误}

end;

test(fsys, [lparen], 23); {测试当前记号是否同步,如果不是就出错,并跳过一些记号}

end

end; {factor}

begin {term}

factor(fsys+[times, slash]); {处理项中第一个因子}

while sym in [times, slash] do {如果当前的sym是乘号或者除号}

begin

mulop := sym; {乘号存入mul operate}

getsym;

factor(fsys+[times, slash]); {因为仍然是乘号或\,所以一样的处理}

if mulop = times then

gen(opr, 0, 4)

else

gen(opr, 0, 5);

end

end; {term}

begin {expression}

if sym in [plus, minus] then {如果当前的sym是加号或者减号}

begin

addop := sym; {加号或者减号存入add operator}

getsym;

term(fsys+[plus, minus]); {处理加减法之外的东西}

if addop = minus then

gen(opr, 0, 1); {下一个符号如果是minus就表示这是一个负号}

end

else

term(fsys+[plus, minus]); {处理加减法之外的东西}

{一直做和上面一样的事情,处理加减乘除运算}

while sym in [plus, minus] do

begin

addop := sym;

getsym;

term(fsys+[plus, minus]);

if addop = plus then

gen(opr, 0, 2)

else

gen(opr, 0, 3);

end

end; {expression}

procedure condition(fsys : symset);

var relop : symbol;

begin

if sym = oddsym then {如果当前记号是odd}

begin

getsym;

expression(fsys); {处理算术表达式}

gen(opr, 0, 6); {生成指令,判断表达式的值是否为奇数,如果是奇数就返回真,否则就返回假}

end

else

begin

expression([eql, neq, lss, gtr, leq, geq] + fsys); {处理完逻辑表达式之后,处理算术表达式}

if not (sym in [eql, neq, lss, leq, gtr, geq]) then

error(20) {算术表达式处理完之后,如果紧随着的不是逻辑符号（大于小于之类）,说明出错}

else

begin

relop := sym; {关系符号存入relation operator}

getsym;

expression(fsys); {处理关系符号右侧的算术表达式}

case relop of

eql : gen(opr, 0, 8); {生成指令,判断两个表达式的值是否相等}

neq : gen(opr, 0, 9); {生成指令,判断两个表达式的值是否不相等}

lss : gen(opr, 0, 10); {生成指令,判断前一个表达式的值是否小于后一个表达式的值}

geq : gen(opr, 0, 11); {生成指令,判断前一个表达式的值是否大于等于后一个表达式的值}

gtr : gen(opr, 0, 12); {生成指令,判断前一个表达式的值是否大于后一个表达式的值}

leq : gen(opr, 0, 13); {生成指令,判断前一个表达式的值是否小于等于后一个表达式的值}

end

end

end

end; {condition}

begin {statement}

if sym = ident then {赋值语句处理}

begin

i := position(id); {查找id在符号表的入口}

if i = 0 then

error(11) {出现未声明标识符,报错}

else if table[i].kind <> variable then

begin {对非变量赋值,报错}

error(12);

i := 0;

end;

getsym;

if sym = becomes then

getsym

else

error(13); {赋值语句中,变量后面未马上跟随赋值号：=,出错}

expression(fsys); {处理赋值号右边的表达式}

if i <> 0 then {赋值号左侧的标识符有定义,就执行如下操作：}

with table[i] do

gen(sto, lev-level, adr); {产生一条存数指令,将栈顶表达式的值存入变量id中,功能码sto表示这是存数指令、lev-level是相对层数,adr是变量在过程中的数据空间相对地址}

{其中lev是当前语句所在过程的层号,level是定义该赋值语句变量id的过程层号}

end {赋值语句处理完毕}

else if sym = callsym then {处理过程调用语句}

begin

getsym;

if sym <> ident then

error(14) {过程调用记号后面不是标识符,出错}

else

begin

i := position(id);

if i = 0 then

error(11) {出现未声明标识符（反正在符号表查不到就是未声明）,报错}

else

with table[i] do

if kind = myProcedure then

gen(cal, lev-level, adr) {产生一条过程调用语句,传入功能码cal表示这时过程调用、传入相对层数,过程id的代码中第一条指令地址}

{其中lev是当前语句所在过程的层号,level是定义该过程变量名字的层数}

else

error(15); {过程调用记号后面不是过程标识符,出错}

getsym;

end

end {处理过程调用语句处理完毕}

else if sym = ifsym then {处理条件语句}

begin

getsym;

condition([thensym, dosym]+fsys); {取出if后面的条件表达式进行分析}

if sym = thensym then

getsym

else

error(16); {if后面没跟then,出错}

cx1 := cx; {下一条代码地址}

gen(jpc, 0, 0); {生成跳转指令,表达式为假则转到（待填）地址,否则顺序执行}

statement(fsys); {正常处理if表达式里面的语句}

code[cx1].a := cx; {将下一个指令的地址回填到前面jpc待填地址中}

end {处理条件语句完毕}

else if sym = beginsym then {处理语句序列}

begin

getsym;

statement([semicolon, endsym]+fsys); {处理第一个语句}

{循环处理后续语句,判断是语句开始的方法：sym分号或者是语句的开头}

while sym in ([semicolon]+statbegsys) do

begin

if sym = semicolon then

getsym

else

error(10); {一条语句结束了不加分号,出错}

statement([semicolon, endsym]+fsys); {上一个语句结束,处理下一个语句}

end;

if sym = endsym then

getsym

else

error(17); {语句序列处理完毕,需要用end结束,但是没检测到end,出错}

end {语句序列处理完毕}

else if sym = whilesym then {处理循环语句}

begin

cx1 := cx; {记录第一条指令地址,即条件表达式语句的第一条指令地址}

getsym;

condition([dosym]+fsys); {处理条件表达式}

cx2 := cx; {记录下一条指令的地址}

gen(jpc, 0, 0); {生成跳转指令,表达式为假则转到（待填）地址,否则顺序执行}

if sym = dosym then

getsym

else

error(18); {while + 表达式 + do,没有检测到do,出错}

statement(fsys); {正常处理do后的语句}

gen(jmp, 0, cx1); {产生一条无条件转移指令,转移到while循环后面的条件表达式的第一条指令}

code[cx2].a := cx; {把这时候的下一条指令地址回填到前面生成的jpc指令的待填地址处}

end; {处理循环语句完毕}

test(fsys, [ ], 19); {测试记号是否正常,否则报错,并跳过不正常的记号}

end; {statement}

begin {block}

dx := 3; {本过程数据空间栈顶指针}

tx0 := tx; {标识符表的长度(当前表头指针)}

table[tx].adr := cx; {本过程名的地址, 即紧跟的指令的序号}

gen(jmp, 0, 0); {生成一条转移指令，开始当前过程代码的执行（用来跳过声明，不再执行声明部分）}

if lev > levmax then

error(32); {当前过程层号超出限制，报错}

repeat

if sym = constsym then {处理常数声明语句}

begin

getsym;

repeat {循环处理所有的常数声明语句}

constdeclaration; {处理第一个常数声明}

while sym = comma do {遇到逗号继续处理下一个声明}

begin

getsym;

constdeclaration;

end;

if sym = semicolon then

getsym

else

error(5); {上一个常数声明语句处理完，应该以分号结束，但是没检测到分号，报错}

until sym <> ident; {下一个sym非标识符，结束}

end; {处理常数声明语句完毕}

if sym = varsym then {处理变量声明语句}

begin

getsym;

repeat {循环处理分号分隔的声明语句}

vardeclaration;

{循环处理逗号分隔的变量标识符}

while sym = comma do

begin

getsym;

vardeclaration;

end;

if sym = semicolon then

getsym

else

error(5); {上一个变量声明语句处理完，应该以分号结束，但是没检测到分号，报错}

until sym <> ident; {下一个sym非标识符，结束}

end; {处理变量声明语句完毕}

while sym = procsym do {处理过程声明语句}

begin

getsym;

if sym = ident then {如果是一个正常的标识符，该标识符表示过程名，填入符号表}

begin

enter(myProcedure);

getsym;

end

else

error(4); {缺少过程名，报错}

if sym = semicolon then

getsym

else

error(5); {过程名后面不加分号，报错}

block(lev+1, tx, [semicolon]+fsys); {处理过程体，过程嵌套层数加一，新过程符号表起始位置为当前符号表栈顶指针tx，[semicolon]+fsys过程体开始和末尾符号集}

if sym = semicolon then

begin

getsym;

test(statbegsys+[ident, procsym], fsys, 6); {测试当前记号是否是某些语句的开始符号或者声明的开始符号，如果不是就报错6，并跳过错误符号}

end

else

error(5); {过程声明完毕，分号结束，分号检测不到，报错}

end; {处理过程声明语句完毕}

test(statbegsys+[ident], declbegsys, 7); {测试当前记号是否是某些语句的开始符号或者声明的开始符号，如果不是就报错6，并跳过错误符号}

until not (sym in declbegsys); {直到新的sym不再是声明语句的开头，循环处理声明语句完毕}

code[table[tx0].adr].a := cx; {table[tx0].adr是本过程名的第1条代码(jmp, 0, 0)的地址,本语句即是将下一代码(当前过程语句的第1条代码)的地址回填到该jmp指令中,得(jmp, 0, cx)，方便跳过声明语句}

with table[tx0] do

begin

adr := cx; {代码开始地址}

end;

cx0 := cx; {cx0记录起始代码地址}

gen(int, 0, dx); {生成一条指令, 在栈顶为本过程留出数据空间}

statement([semicolon, endsym]+fsys); {处理一个语句}

gen(opr, 0, 0); {生成返回指令}

test(fsys, [ ], 8); {测试过程体语句后的符号是否正常,否则报错并跳过错误符号}

listcode; {打印本过程的中间代码序列}

end; {block}

procedure interpret;

const stacksize = 500;

var p, b, t : integer; {程序地址寄存器, 基地址寄存器,栈顶地址寄存器}

i : instruction; {指令寄存器}

s : array [1..stacksize] of integer; {数据存储栈}

function base(l : integer) : integer;

var b1 : integer;

begin

b1 := b; {顺静态链求层差为l的层的基地址}

while l > 0 do

begin

b1 := s[b1];

l := l-1;

end;

base := b1;

end; {base}

begin

writeln('START PL/0');

writeln(fout, 'START PL/0');

t := 0;

b := 1;

p := 0;

s[1] := 0;

s[2] := 0;

s[3] := 0;

repeat

i := code[p]; {当前指令}

p := p+1; {地址加一}

with i do

case f of

lit : begin {取数指令，栈顶指针加1, 把常数a取到栈顶}

t := t+1;

s[t] := a;

end;

opr : case a of {运算指令}

0 : begin {返回调用过程指令}

t := b-1;

p := s[t+3];

b := s[t+2];

end;

1 : s[t] := -s[t]; {负数运算}

2 : begin {加法}

t := t-1;

s[t] := s[t] + s[t+1];

end;

3 : begin {剑法}

t := t-1;

s[t] := s[t]-s[t+1];

end;

4 : begin {乘法}

t := t-1;

s[t] := s[t] \* s[t+1];

end;

5 : begin {除法}

t := t-1;

s[t] := s[t] div s[t+1];

end;

6 : s[t] := ord(odd(s[t])); {奇数判断}

8 : begin {相等判断}

t := t-1;

s[t] := ord(s[t] = s[t+1]);

end;

9: begin {不等判断}

t := t-1;

s[t] := ord(s[t] <> s[t+1]);

end;

10 : begin {判断小于}

t := t-1;

s[t] := ord(s[t] < s[t+1]);

end;

11: begin {判读大于等于}

t := t-1;

s[t] := ord(s[t] >= s[t+1]);

end;

12 : begin {判断大于}

t := t-1;

s[t] := ord(s[t] > s[t+1]);

end;

13 : begin {判断小于等于}

t := t-1;

s[t] := ord(s[t] <= s[t+1]);

end;

end; {case a of opr}

lod : begin {取数（变量）指令}

t := t + 1;

s[t] := s[base(l) + a];

end;

sto : begin {存数（变量）指令}

s[base(l) + a] := s[t];

writeln(s[t]);

writeln(fout, s[t]);

t := t-1;

end;

cal : begin {过程调用指令，为被调用过程数据空间建立连接数据}

s[t+1] := base( l );

s[t+2] := b;

s[t+3] := p;

b := t+1;

p := a;

end;

int : t := t + a; {为数据空间栈顶预留a大小的空间}

jmp : p := a; {无条件跳转指令}

jpc : begin {条件跳转指令，如果条件不成立跳转到地址a，否则顺序执行}

if s[t] = 0 then

p := a;

t := t-1;

end

end {with, case, end of case f of}

until p = 0; {读取到最外层程序的返回地址，执行结束}

write('END PL/0');

write(fout, 'END PL/0');

end {interpret};

begin {主程序}

write('please input the file you want to compile: '); writeln;

readln(file1);

assign(fin, file1);

reset(fin);

write('please input the file you want to store the result: '); writeln;

readln(file2);

assign(fout, file2);

rewrite(fout);

{除了个别的算符、标点符号记号有意义,其实其它的符号记号都无意义,因此这里设置为nul（讲道理不止这几个的）}

for ch := 'a' to ';' do ssym[ch] := nul;

{按照大小进行编排,以便能够使用二分法对其进行查找}

word[1] := 'begin '; word[2] := 'call ';

word[3] := 'const '; word[4] := 'do ';

word[5] := 'end '; word[6] := 'if ';

word[7] := 'odd '; word[8] := 'procedure ';

word[9] := 'then '; word[10]:= 'var ';

word[11]:= 'while ';

{保留字记号数组,我们通过查找word保留字对应字符串得到保留字}

wsym[1] := beginsym; wsym[2] := callsym;

wsym[3] := constsym; wsym[4] := dosym;

wsym[5] := endsym; wsym[6] := ifsym;

wsym[7] := oddsym; wsym[8] := procsym;

wsym[9] := thensym; wsym[10] := varsym;

wsym[11] := whilesym;

{算符和标点符号对应记号}

ssym['+'] := plus; ssym['-'] := minus;

ssym['\*'] := times; ssym['/'] := slash;

ssym['('] := lparen; ssym[')'] := rparen;

ssym['='] := eql; ssym[','] := comma;

ssym['.'] := period; ssym['!'] := neq; {不等于用！代替（没有≠这个符号）}

ssym['<'] := lss; ssym['>'] := gtr;

ssym['@'] := leq; ssym['#'] := geq; {小于等于用@代替（没有≤这个符号）, 大于等于用#代替（没有≥这个符号）}

ssym[';'] := semicolon;

{中间代码指令的字符串}

mnemonic[lit] := ' LIT '; mnemonic[opr] := ' OPR ';

mnemonic[lod] := ' LOD '; mnemonic[sto] := ' STO ';

mnemonic[cal] := ' CAL '; mnemonic[int] := ' INT ';

mnemonic[jmp] := ' JMP '; mnemonic[jpc] := ' JPC ';

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

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

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

{page(fout);} {为找到id page标识符}

err := 0;

cc := 0;

cx := 0;

ll := 0;

ch := ' ';

kk := al;

getsym;

block(0, 0, [period]+declbegsys+statbegsys); {处理程序体}

if sym <> period then error(9); {原来的≠是错的,不等号在Pascal里面是<>, 如果当前记号不是句号, 则出错}

if err = 0 then interpret {如果编译无错误, 则解释执行中间代码}

else

begin

write('ERRORS IN PL/0 PROGRAM');

write(fout, 'ERRORS IN PL/0 PROGRAM');

end;

writeln; writeln(fout);

close(fin); close(fout);

end.

## PL0源程序代码-第一阶段

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.

## 中间代码-第一阶段

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

## 栈中数据-第一阶段

START PL/0

7

85

7

85

0

7

14

42

28

21

35

56

10

112

5

147

224

2

448

1

595

896

0

25

3

25

0

3

6

12

24

48

0

24

1

1

2

12

4

6

8

3

84

36

84

36

48

12

24

12

12

END PL/0

## PL0编译程序代码-第二阶段

program PL0 (input, output);

{带有代码生成的PL0编译程序}

const

norw = 13; {保留字的个数}

txmax = 100; {标识符表长度}

nmax = 14; {数字的最大位数}

al = 10; {标识符的长度}

amax = 2047; {最大地址}

levmax = 3; {程序体嵌套的最大深度}

cxmax = 200; {代码数组的大小}

type

symbol = (nul, ident, number, plus, minus, times, slash, oddsym,

eql, neq, lss, leq, gtr, geq, lparen, rparen, comma, semicolon,

period, becomes, beginsym, endsym, ifsym, thensym,

whilesym, dosym, callsym, constsym, varsym, procsym, readsym, writesym);

alfa = packed array [1..al] of char;

myObject = (constant, variable, myProcedure);

symset = set of symbol;

fct = (lit, opr, lod, sto, cal, int, jmp, jpc, red, wrt); {functions}

instruction = packed record

f : fct; {功能码}

l : 0..levmax; {相对层数}

a : 0..amax; {相对地址}

end;

{LIT 0,a : 取常数a

OPR 0,a : 执行运算a

LOD l,a : 取层差为l的层﹑相对地址为a的变量

STO l,a : 存到层差为l的层﹑相对地址为a的变量

CAL l,a : 调用层差为l的过程

INT 0,a : t寄存器增加a

JMP 0,a : 转移到指令地址a处

JPC 0,a : 条件转移到指令地址a处

READ 0,a : 从输入流读取输入

WRITE 0,a : 向输出流打印输出}

var

ch : char; {最近读到的字符}

sym : symbol; {最近读到的符号}

id : alfa; {最近读到的标识符}

num : integer; {最近读到的数}

cc : integer; {当前行的字符计数}

ll : integer; {当前行的长度}

kk, err : integer;

cx : integer; {代码数组的当前下标}

line : array [1..81] of char; {当前行,存储的当前行的字符串}

a : alfa; {当前标识符的字符串}

code : array [0..cxmax] of instruction; {中间代码数组}

word : array [1..norw] of alfa; {存放保留字的字符串}

wsym : array [1..norw] of symbol; {存放保留字的记号}

ssym : array [char] of symbol; {存放算符和标点符号的记号}

mnemonic : array [fct] of packed array [1..5] of char;

{中间代码算符的字符串}

declbegsys, statbegsys, facbegsys : symset;

table : array [0..txmax] of record {符号表}

name : alfa;

case kind : myObject of

constant : (val : integer);

variable, myProcedure : (level, adr : integer);

end; {这里有改动,不一定能对}

fin, fout : text;

file1, file2 : string;

{----------------------------------------------------------------}

procedure error (n : integer);

begin

writeln('\*\*\*\*', ' ' : cc-1, '^', n : 2); {cc为当前行已读的字符数, n为错误号}

writeln(fout, '\*\*\*\*', ' ' : cc-1, '^', n : 2);

err := err + 1; {错误数err加1}

end; {error}

{----------------------------------------------------------------}

procedure getsym; {读取一个符号（标识符）}

var i, j, k : integer;

procedure getch; {读取一个字符}

begin

if cc = ll then {如果读取到行尾,就要先将下一行所有内容填到line数组中,并设置cc为0}

begin

if eof(fin) then {如果读到了文件尾}

begin

write('PROGRAM INCOMPLETE');

write(fout, 'PROGRAM INCOMPLETE');

writeln; writeln(fout);

close(fin); close(fout);

exit();

end;

{上一行结束,重新初始化,读取新行：}

ll := 0;

cc := 0;

write(cx : 5, ' ');

write(fout, cx : 5, ' ');

while not eoln(fin) do

begin

ll := ll + 1;

read(fin, ch);

write(ch);

write(fout, ch);

line[ll] := ch

end;

writeln;

writeln(fout);

ll := ll + 1;

readln(fin, line[ll]); {这里将行末符存到 line[ll] 中,即#13}

end;

{从line取出下一个字符到ch,无论之前是否读到行尾：}

cc := cc + 1;

ch := line[cc];

end {getch};

begin {getsym}

while (ch = ' ') or (ch = #13) do {跳过空格或者换行符}

getch;

if ch in ['a'..'z'] then

begin {标识符或保留字(以字母开头)}

k := 0;

repeat

if k < al then

begin

k:= k + 1;

a[k] := ch;

end;

getch;

until not(ch in ['a'..'z', '0'..'9']);

if k >= kk then

kk := k

else

repeat

a[kk] := ' ';

kk := kk-1;

until kk = k;

id := a;

i := 1;

j := norw;

repeat {二分查找当前标识符在保留字表中的对应的记号}

k := (i+j) div 2;

if id <= word[k] then j := k-1;

if id >= word[k] then i := k + 1;

until i > j;

if i-1 > j then

sym := wsym[k] {如果找到了,这时候因为前面repeat里面两个if都满足,所以i=k+1,j=k-1,i-1=k>j,然后在这里被置为wsym[k]（一个保留字）}

else

sym := ident; {如果没有找到,sym被设置为是一个标识符}

end

else if ch in ['0'..'9'] then

begin {数字}

k := 0;

num := 0;

sym := number;

repeat

num := 10\*num + (ord(ch)-ord('0'));

k := k + 1;

getch;

until not (ch in ['0'..'9']);

if k > nmax then

error(30); {出现越界的常数,出错}

end

else if ch = ':' then

begin

getch;

if ch = '=' then

begin

sym := becomes;

getch;

end

else

sym := nul;

end

else

begin

sym := ssym[ch];

getch;

end

end; {getsym}

{----------------------------------------------------------------}

procedure gen(x : fct; y, z : integer); {使用传入的参数x,y,z生成一条中间代码}

begin

if cx > cxmax then {代码过长,导致指令序列长度超出代码数组容量,报错并退出程序}

begin

write('PROGRAM TOO LONG');

write(fout, 'PROGRAM TOO LONG');

writeln; writeln(fout);

close(fin); close(fout);

exit();

end;

with code[cx] do {生成的中间代码的功能码为x,相对层数为y,相对地址为z}

begin

f := x;

l := y;

a := z;

end;

cx := cx + 1;

end; {gen}

{----------------------------------------------------------------}

procedure test(s1, s2 : symset; n : integer);

begin

if not (sym in s1) then

begin

error(n); {标识符不在s1+s2集合中,出错}

s1 := s1 + s2;

while not (sym in s1) do {跳过不在s1+s2集合中的那些记号}

getsym;

end

end; {test}

{----------------------------------------------------------------}

procedure block(lev, tx : integer; fsys : symset);

var

dx : integer; {本过程数据空间分配下标}

tx0 : integer; {本过程标识表起始下标}

cx0 : integer; {本过程代码起始下标}

procedure enter(k : myObject);

begin {把自定义的 myObject 填入符号表中}

tx := tx +1;

with table[tx] do

begin

name := id; {自定义标识符名字,是一个字符串}

kind := k; {自定义标识符种类,是myObject类型}

case k of

constant : begin {如果该标识符为常数}

if num > amax then {如果常数值超出界限,报错}

begin

error(30); {常量填入符号表时常量数值越界,出错}

num := 0;

end;

val := num; {将该标识符的值赋值为num}

end;

variable : begin {如果该标识符是变量}

level := lev; {该变量的过程嵌套层数即该过程的层数lev}

adr := dx; {该变量的地址为当前过程数据空间栈顶}

dx := dx +1; {栈顶指针加一,下一个变量将继续向上移动,过程数据空间向上增长}

end;

myProcedure :

level := lev {如果该标识符是一个过程名,则该过程的嵌套层数为lev}

end

end

end; {enter}

function position(id : alfa) : integer; {返回该id在符号表table的入口}

var i : integer;

begin {在标识符表中查标识符id}

table[0].name := id; {在符号表栈的最下方预填标识符id}

i := tx; {符号表栈顶指针}

while table[i].name <> id do

i := i-1;

position := i; {如果给出的id在符号表中,那么最后position就是该符号在符号表的入口,否则position就是0}

end; {position}

procedure constdeclaration; {常量声明}

begin

if sym = ident then {当前记号是一个标识符而不是保留字}

begin

getsym;

if sym in [eql, becomes] then {读取的标识符下一个记号是等于号或者赋值号}

begin

if sym = becomes then {如果是赋值号,则出错,因为常量声明使用=而不是:=}

error(1); {常量声明使用:=出错}

getsym; {读取第三个位置的数字,如果不是数字说明出错,如果正确那么就将声明的常数添加到符号表,常数的数值就是=后面跟随的数字}

if sym = number then

begin

enter(constant); {将声明的常数添加到符号表}

getsym;

end

else

error(2); {常量声明,不是数字说明出错}

end

else

error(3); {标识符后面既不是等号也不是赋值号出错}

end

else

error(4); {常量声明时不给出常量标识符,出错}

end; {constdeclaration}

procedure vardeclaration;

begin

if sym = ident then {当前记号是一个标识符而不是保留字}

begin

enter(variable); {将该变量加入到符号表}

getsym;

end

else

error(4); {变量声明时不给出变量标识符,出错}

end {vardeclaration};

procedure listcode;

var i : integer;

begin {列出本程序体生成的代码}

for i := cx0 to cx-1 do {cx0是这个过程体的第一条代码序号,cx-1是最后一条代码序号}

with code[i] do {打印第i条代码到文件/屏幕,其中mnemonic[f]是功能码对应字符串,l是相对层数,a是相对地址/运算符号码}

begin

writeln(i : 3, mnemonic[f] : 5, l : 3, a : 5);

writeln(fout, i : 3, mnemonic[f] : 5, l : 3, a : 5);

end

end; {listcode}

procedure statement(fsys : symset);

var i, cx1, cx2 : integer;

procedure expression(fsys : symset);

var addop : symbol;

procedure term(fsys : symset);

var mulop : symbol;

procedure factor(fsys : symset);

var i : integer;

begin

test(facbegsys, fsys, 24); {测试当前sym是否是是在facbegsys中, 不在就报错,并跳过不在fsys+facbegsys中的记号}

while sym in facbegsys do

begin

if sym = ident then {处理标识符}

begin

i := position(id);

if i = 0 then

error(11) {出现未声明标识符,报错}

else

with table[i] do

case kind of

constant : gen(lit, 0, val); {若id是常数, 生成指令,将常数val取到栈顶}

variable : gen(lod, lev-level, adr);

{若id是变量, 生成指令,将该变量取到栈顶; lev: 当前语句所在过程的层号; level: 定义该变量的过程层号; adr: 变量在其过程的数据空间的相对地址}

myProcedure : error(21) {若id是过程名, 则出错}

end;

getsym;

end

else if sym = number then {处理数字}

begin

if num > amax then

begin

error(30); {数值越界报错}

num := 0;

end;

gen(lit, 0, num); {将常数num取到栈顶}

getsym;

end

else if sym = lparen then {处理左括号}

begin

getsym;

expression([rparen]+fsys); {处理表达式}

if sym = rparen then

getsym

else

error(22); {左括号和表达式处理完了需要处理右括号,如果下一个符号不是右括号说明出错,括号不匹配错误}

end;

test(fsys, [lparen], 23); {测试当前记号是否同步,如果不是就出错,并跳过一些记号}

end

end; {factor}

begin {term}

factor(fsys+[times, slash]); {处理项中第一个因子}

while sym in [times, slash] do {如果当前的sym是乘号或者除号}

begin

mulop := sym; {乘号存入mul operate}

getsym;

factor(fsys+[times, slash]); {因为仍然是乘号或\,所以一样的处理}

if mulop = times then

gen(opr, 0, 4)

else

gen(opr, 0, 5);

end

end; {term}

begin {expression}

if sym in [plus, minus] then {如果当前的sym是加号或者减号}

begin

addop := sym; {加号或者减号存入add operator}

getsym;

term(fsys+[plus, minus]); {处理加减法之外的东西}

if addop = minus then

gen(opr, 0, 1); {下一个符号如果是minus就表示这是一个负号}

end

else

term(fsys+[plus, minus]); {处理加减法之外的东西}

{一直做和上面一样的事情,处理加减乘除运算}

while sym in [plus, minus] do

begin

addop := sym;

getsym;

term(fsys+[plus, minus]);

if addop = plus then

gen(opr, 0, 2)

else

gen(opr, 0, 3);

end

end; {expression}

procedure condition(fsys : symset);

var relop : symbol;

begin

if sym = oddsym then {如果当前记号是odd}

begin

getsym;

expression(fsys); {处理算术表达式}

gen(opr, 0, 6); {生成指令,判断表达式的值是否为奇数,如果是奇数就返回真,否则就返回假}

end

else

begin

expression([eql, neq, lss, gtr, leq, geq] + fsys); {处理完逻辑表达式之后,处理算术表达式}

if not (sym in [eql, neq, lss, leq, gtr, geq]) then

error(20) {算术表达式处理完之后,如果紧随着的不是逻辑符号（大于小于之类）,说明出错}

else

begin

relop := sym; {关系符号存入relation operator}

getsym;

expression(fsys); {处理关系符号右侧的算术表达式}

case relop of

eql : gen(opr, 0, 8); {生成指令,判断两个表达式的值是否相等}

neq : gen(opr, 0, 9); {生成指令,判断两个表达式的值是否不相等}

lss : gen(opr, 0, 10); {生成指令,判断前一个表达式的值是否小于后一个表达式的值}

geq : gen(opr, 0, 11); {生成指令,判断前一个表达式的值是否大于等于后一个表达式的值}

gtr : gen(opr, 0, 12); {生成指令,判断前一个表达式的值是否大于后一个表达式的值}

leq : gen(opr, 0, 13); {生成指令,判断前一个表达式的值是否小于等于后一个表达式的值}

end

end

end

end; {condition}

begin {statement}

if sym = ident then {赋值语句处理}

begin

i := position(id); {查找id在符号表的入口}

if i = 0 then

error(11) {出现未声明标识符,报错}

else if table[i].kind <> variable then

begin

error(12); {对非变量赋值,报错}

i := 0;

end;

getsym;

if sym = becomes then

getsym

else

error(13); {赋值语句中,变量后面未马上跟随赋值号：=,出错}

expression(fsys); {处理赋值号右边的表达式}

if i <> 0 then {赋值号左侧的标识符有定义,就执行如下操作：}

with table[i] do

gen(sto, lev-level, adr); {产生一条存数指令,将栈顶表达式的值存入变量id中,功能码sto表示这是存数指令、lev-level是相对层数,adr是变量在过程中的数据空间相对地址}

{其中lev是当前语句所在过程的层号,level是定义该赋值语句变量id的过程层号}

end {赋值语句处理完毕}

else if sym = callsym then {处理过程调用语句}

begin

getsym;

if sym <> ident then

error(14) {过程调用记号后面不是标识符,出错}

else

begin

i := position(id);

if i = 0 then

error(11) {出现未声明标识符（反正在符号表查不到就是未声明）,报错}

else

with table[i] do

if kind = myProcedure then

gen(cal, lev-level, adr) {产生一条过程调用语句,传入功能码cal表示这时过程调用、传入相对层数,过程id的代码中第一条指令地址}

{其中lev是当前语句所在过程的层号,level是定义该过程变量名字的层数}

else

error(15); {过程调用记号后面不是过程标识符,出错}

getsym;

end

end {处理过程调用语句处理完毕}

else if sym = ifsym then {处理条件语句}

begin

getsym;

condition([thensym, dosym]+fsys); {取出if后面的条件表达式进行分析}

if sym = thensym then

getsym

else

error(16); {if后面没跟then,出错}

cx1 := cx; {下一条代码地址}

gen(jpc, 0, 0); {生成跳转指令,表达式为假则转到（待填）地址,否则顺序执行}

statement(fsys); {正常处理if表达式里面的语句}

code[cx1].a := cx; {将下一个指令的地址回填到前面jpc待填地址中}

end {处理条件语句完毕}

else if sym = beginsym then {处理语句序列}

begin

getsym;

statement([semicolon, endsym]+fsys); {处理第一个语句}

{循环处理后续语句,判断是语句开始的方法：sym分号或者是语句的开头}

while sym in ([semicolon]+statbegsys) do

begin

if sym = semicolon then

getsym

else

error(10); {一条语句结束了不加分号,出错}

statement([semicolon, endsym]+fsys); {上一个语句结束,处理下一个语句}

end;

if sym = endsym then

getsym

else

error(17); {语句序列处理完毕,需要用end结束,但是没检测到end,出错}

end {语句序列处理完毕}

else if sym = whilesym then {处理循环语句}

begin

cx1 := cx; {记录第一条指令地址,即条件表达式语句的第一条指令地址}

getsym;

condition([dosym]+fsys); {处理条件表达式}

cx2 := cx; {记录下一条指令的地址}

gen(jpc, 0, 0); {生成跳转指令,表达式为假则转到（待填）地址,否则顺序执行}

if sym = dosym then

getsym

else

error(18); {while + 表达式 + do,没有检测到do,出错}

statement(fsys); {正常处理do后的语句}

gen(jmp, 0, cx1); {产生一条无条件转移指令,转移到while循环后面的条件表达式的第一条指令}

code[cx2].a := cx; {把这时候的下一条指令地址回填到前面生成的jpc指令的待填地址处}

end {处理循环语句完毕}

else if sym = readsym then {处理读指令，第二阶段添加部分}

begin

getsym;

if sym = ident then

Repeat

i := position(id);

if i = 0 then

error(11) {出现未定义标识符，报错}

else if table[i].kind <> variable then

begin

error(12); {尝试向非变量标识符写数据，报错}

i := 0

end

else

With table[i] do

gen(red, lev-level, adr);

getsym;

if sym = comma then

getsym

else if sym <> semicolon then

error(5); {语句不以分号结束}

Until sym <> ident

else

error(25); {读指令后未跟随标识符}

end {处理读指令完毕}

else if sym = writesym then {处理写指令，第二阶段添加部分}

begin

getsym;

if sym = ident then

Repeat

expression([comma, semicolon]+fsys);

gen(wrt, 0, 0);

if sym = comma then

getsym

else if sym <> semicolon then

error(5) {语句不以分号结束}

Until sym <> ident

else

error(26); {写指令后未跟随标识符}

end; {处理写指令完毕}

test(fsys, [ ], 19); {测试记号是否正常,否则报错,并跳过不正常的记号}

end; {statement}

begin {block}

dx := 3; {本过程数据空间栈顶指针}

tx0 := tx; {标识符表的长度(当前表头指针)}

table[tx].adr := cx; {本过程名的地址, 即紧跟的指令的序号}

gen(jmp, 0, 0); {生成一条转移指令，开始当前过程代码的执行（用来跳过声明，不再执行声明部分）}

if lev > levmax then

error(32); {当前过程层号超出限制，报错}

repeat

if sym = constsym then {处理常数声明语句}

begin

getsym;

repeat {循环处理所有的常数声明语句}

constdeclaration; {处理第一个常数声明}

while sym = comma do {遇到逗号继续处理下一个声明}

begin

getsym;

constdeclaration;

end;

if sym = semicolon then

getsym

else

error(5); {上一个常数声明语句处理完，应该以分号结束，但是没检测到分号，报错}

until sym <> ident; {下一个sym非标识符，结束}

end; {处理常数声明语句完毕}

if sym = varsym then {处理变量声明语句}

begin

getsym;

repeat {循环处理分号分隔的声明语句}

vardeclaration;

{循环处理逗号分隔的变量标识符}

while sym = comma do

begin

getsym;

vardeclaration;

end;

if sym = semicolon then

getsym

else

error(5); {上一个变量声明语句处理完，应该以分号结束，但是没检测到分号，报错}

until sym <> ident; {下一个sym非标识符，结束}

end; {处理变量声明语句完毕}

while sym = procsym do {处理过程声明语句}

begin

getsym;

if sym = ident then {如果是一个正常的标识符，该标识符表示过程名，填入符号表}

begin

enter(myProcedure);

getsym;

end

else

error(4); {缺少过程名，报错}

if sym = semicolon then

getsym

else

error(5); {过程名后面不加分号，报错}

block(lev+1, tx, [semicolon]+fsys); {处理过程体，过程嵌套层数加一，新过程符号表起始位置为当前符号表栈顶指针tx，[semicolon]+fsys过程体开始和末尾符号集}

if sym = semicolon then

begin

getsym;

test(statbegsys+[ident, procsym], fsys, 6); {测试当前记号是否是某些语句的开始符号或者声明的开始符号，如果不是就报错6，并跳过错误符号}

end

else

error(5); {过程声明完毕，分号结束，分号检测不到，报错}

end; {处理过程声明语句完毕}

test(statbegsys+[ident], declbegsys, 7); {测试当前记号是否是某些语句的开始符号或者声明的开始符号，如果不是就报错6，并跳过错误符号}

until not (sym in declbegsys); {直到新的sym不再是声明语句的开头，循环处理声明语句完毕}

code[table[tx0].adr].a := cx; {table[tx0].adr是本过程名的第1条代码(jmp, 0, 0)的地址,本语句即是将下一代码(当前过程语句的第1条代码)的地址回填到该jmp指令中,得(jmp, 0, cx)，方便跳过声明语句}

with table[tx0] do

begin

adr := cx; {代码开始地址}

end;

cx0 := cx; {cx0记录起始代码地址}

gen(int, 0, dx); {生成一条指令, 在栈顶为本过程留出数据空间}

statement([semicolon, endsym]+fsys); {处理一个语句}

gen(opr, 0, 0); {生成返回指令}

test(fsys, [ ], 8); {测试过程体语句后的符号是否正常,否则报错并跳过错误符号}

listcode; {打印本过程的中间代码序列}

end; {block}

procedure interpret;

const stacksize = 500;

var p, b, t : integer; {程序地址寄存器, 基地址寄存器,栈顶地址寄存器}

i : instruction; {指令寄存器}

s : array [1..stacksize] of integer; {数据存储栈}

function base(l : integer) : integer;

var b1 : integer;

begin

b1 := b; {顺静态链求层差为l的层的基地址}

while l > 0 do

begin

b1 := s[b1];

l := l-1;

end;

base := b1;

end; {base}

begin

writeln('START PL/0');

writeln(fout, 'START PL/0');

t := 0;

b := 1;

p := 0;

s[1] := 0;

s[2] := 0;

s[3] := 0;

repeat

i := code[p]; {当前指令}

p := p+1; {地址加一}

with i do

case f of

lit : begin {取数指令，栈顶指针加1, 把常数a取到栈顶}

t := t+1;

s[t] := a;

end;

opr : case a of {运算指令}

0 : begin {返回调用过程指令}

t := b-1;

p := s[t+3];

b := s[t+2];

end;

1 : s[t] := -s[t]; {负数运算}

2 : begin {加法}

t := t-1;

s[t] := s[t] + s[t+1];

end;

3 : begin {剑法}

t := t-1;

s[t] := s[t]-s[t+1];

end;

4 : begin {乘法}

t := t-1;

s[t] := s[t] \* s[t+1];

end;

5 : begin {除法}

t := t-1;

s[t] := s[t] div s[t+1];

end;

6 : s[t] := ord(odd(s[t])); {奇数判断}

8 : begin {相等判断}

t := t-1;

s[t] := ord(s[t] = s[t+1]);

end;

9: begin {不等判断}

t := t-1;

s[t] := ord(s[t] <> s[t+1]);

end;

10 : begin {判断小于}

t := t-1;

s[t] := ord(s[t] < s[t+1]);

end;

11: begin {判读大于等于}

t := t-1;

s[t] := ord(s[t] >= s[t+1]);

end;

12 : begin {判断大于}

t := t-1;

s[t] := ord(s[t] > s[t+1]);

end;

13 : begin {判断小于等于}

t := t-1;

s[t] := ord(s[t] <= s[t+1]);

end;

end; {case a of opr}

lod : begin {取数（变量）指令}

t := t + 1;

s[t] := s[base(l) + a];

end;

sto : begin {存数（变量）指令}

s[base(l) + a] := s[t];

// writeln(s[t]);

// writeln(fout, s[t]);

// 阶段二不需要栈中数据

t := t-1;

end;

cal : begin {过程调用指令，为被调用过程数据空间建立连接数据}

s[t+1] := base( l );

s[t+2] := b;

s[t+3] := p;

b := t+1;

p := a;

end;

int : t := t + a; {为数据空间栈顶预留a大小的空间}

jmp : p := a; {无条件跳转指令}

jpc : begin {条件跳转指令，如果条件不成立跳转到地址a，否则顺序执行}

if s[t] = 0 then

p := a;

t := t-1;

end;

red : begin {读取输入指令，第二部分添加}

writeln('Input a number: ');

writeln(fout, 'Input a number: ');

readln(s[base(l) + a]); {将从键盘读取的数据写入变量}

writeln(fout, s[base(l) + a]);

end;

wrt : begin {写指令，第二部分添加}

writeln('Output a number: ');

writeln(fout, 'Output a number: ');

writeln(s[t]); {将栈顶数据（计算出来的write语句后面的表达式的值）输出到屏幕}

writeln(fout, s[t]);

t := t-1; {将栈顶数据丢弃，因为这个数是临时的表达式计算值，没有意义}

end

end {with, case, end of case f of}

until p = 0; {读取到最外层程序的返回地址，执行结束}

write('END PL/0');

write(fout, 'END PL/0');

end {interpret};

begin {主程序}

write('please input the file you want to compile: '); writeln;

readln(file1);

assign(fin, file1);

reset(fin);

write('please input the file you want to store the result: '); writeln;

readln(file2);

assign(fout, file2);

rewrite(fout);

{除了个别的算符、标点符号记号有意义,其实其它的符号记号都无意义,因此这里设置为nul（讲道理不止这几个的）}

for ch := 'a' to ';' do ssym[ch] := nul;

{按照大小进行编排,以便能够使用二分法对其进行查找}

word[1] := 'begin '; word[2] := 'call ';

word[3] := 'const '; word[4] := 'do ';

word[5] := 'end '; word[6] := 'if ';

word[7] := 'odd '; word[8] := 'procedure ';

word[9] := 'read '; word[10]:= 'then ';

word[11]:= 'var '; word[12]:= 'while ';

word[13]:= 'write ';

{保留字记号数组,我们通过查找word保留字对应字符串得到保留字}

wsym[1] := beginsym; wsym[2] := callsym;

wsym[3] := constsym; wsym[4] := dosym;

wsym[5] := endsym; wsym[6] := ifsym;

wsym[7] := oddsym; wsym[8] := procsym;

wsym[9] := readsym; wsym[10]:= thensym;

wsym[11]:= varsym; wsym[12]:= whilesym;

wsym[13]:= writesym;

{算符和标点符号对应记号}

ssym['+'] := plus; ssym['-'] := minus;

ssym['\*'] := times; ssym['/'] := slash;

ssym['('] := lparen; ssym[')'] := rparen;

ssym['='] := eql; ssym[','] := comma;

ssym['.'] := period; ssym['!'] := neq; {不等于用！代替（没有≠这个符号）}

ssym['<'] := lss; ssym['>'] := gtr;

ssym['@'] := leq; ssym['#'] := geq; {小于等于用@代替（没有≤这个符号）, 大于等于用#代替（没有≥这个符号）}

ssym[';'] := semicolon;

{中间代码指令的字符串}

mnemonic[lit] := ' LIT '; mnemonic[opr] := ' OPR ';

mnemonic[lod] := ' LOD '; mnemonic[sto] := ' STO ';

mnemonic[cal] := ' CAL '; mnemonic[int] := ' INT ';

mnemonic[jmp] := ' JMP '; mnemonic[jpc] := ' JPC ';

mnemonic[red] := ' RED '; mnemonic[wrt] := ' WRT ';

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

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

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

{page(fout);} {为找到id page标识符}

err := 0;

cc := 0;

cx := 0;

ll := 0;

ch := ' ';

kk := al;

getsym;

block(0, 0, [period]+declbegsys+statbegsys); {处理程序体}

if sym <> period then error(9); {原来的≠是错的,不等号在Pascal里面是<>, 如果当前记号不是句号, 则出错}

if err = 0 then interpret {如果编译无错误, 则解释执行中间代码}

else

begin

write('ERRORS IN PL/0 PROGRAM');

write(fout, 'ERRORS IN PL/0 PROGRAM');

end;

writeln; writeln(fout);

close(fin); close(fout);

end.

## PL0源程序代码-第二阶段

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;

write z;

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;

write q;

write r;

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;

write z;

end;

begin

x := m; y := n; call multiply;

read x, y; call divide;

read x; read y; call gcd;

end.

## 中间代码-第二阶段

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 LOD 1 5

30 WRT 0 0

31 OPR 0 0

33 INT 0 4

34 LOD 1 3

35 STO 1 7

36 LIT 0 0

37 STO 1 6

38 LOD 1 4

39 STO 0 3

40 LOD 0 3

41 LOD 1 7

42 OPR 0 13

43 JPC 0 49

44 LIT 0 2

45 LOD 0 3

46 OPR 0 4

47 STO 0 3

48 JMP 0 40

49 LOD 0 3

50 LOD 1 4

51 OPR 0 12

52 JPC 0 74

53 LIT 0 2

54 LOD 1 6

55 OPR 0 4

56 STO 1 6

57 LOD 0 3

58 LIT 0 2

59 OPR 0 5

60 STO 0 3

61 LOD 0 3

62 LOD 1 7

63 OPR 0 13

64 JPC 0 73

65 LOD 1 7

66 LOD 0 3

67 OPR 0 3

68 STO 1 7

69 LOD 1 6

70 LIT 0 1

71 OPR 0 2

72 STO 1 6

73 JMP 0 49

74 LOD 1 6

75 WRT 0 0

76 LOD 1 7

77 WRT 0 0

78 OPR 0 0

80 INT 0 5

81 LOD 1 3

82 STO 0 3

83 LOD 1 4

84 STO 0 4

85 LOD 0 3

86 LOD 0 4

87 OPR 0 9

88 JPC 0 106

89 LOD 0 3

90 LOD 0 4

91 OPR 0 10

92 JPC 0 97

93 LOD 0 4

94 LOD 0 3

95 OPR 0 3

96 STO 0 4

97 LOD 0 4

98 LOD 0 3

99 OPR 0 10

100 JPC 0 105

101 LOD 0 3

102 LOD 0 4

103 OPR 0 3

104 STO 0 3

105 JMP 0 85

106 LOD 0 3

107 STO 1 5

108 LOD 1 5

109 WRT 0 0

110 OPR 0 0

111 INT 0 8

112 LIT 0 7

113 STO 0 3

114 LIT 0 85

115 STO 0 4

116 CAL 0 2

117 RED 0 3

118 RED 0 4

119 CAL 0 33

120 RED 0 3

121 RED 0 4

122 CAL 0 80

123 OPR 0 0

## 输入输出-第二阶段

START PL/0

Output a number:

595

Input a number:

40

Input a number:

9

Output a number:

4

Output a number:

4

Input a number:

35

Input a number:

12

Output a number:

1

END PL/0

下面解释一下上面的输入输出：

因为我阶段二的PL0源程序代码是在第一阶段的代码基础上修改的：

（0）仍然调用三个函数multiple、divide、gcd

（1）只是divide函数和gcd函数采用键盘输入的值进行计算（而不是给定值）

（2）并且将三个函数的结果输出到屏幕：multiple函数的乘积计算结果、divide计算出的余数和商、gcd函数计算出的最大公因数

所以屏幕上看到的内容如下：

