编译原理

课程实践报告书

学员姓名：徐玉伟

学　　号：201306043017

专　　业：信息安全

年　　级：2013 级

2015 年 12 月 29 日

1、课程实践目的

通过扩充 PL/0 语言，熟悉编译的各个过程在实际中的运用，掌握 LL(1)的递归下降分析法，锻炼编程能力和计算思维。

2、任务概述

本次实验实验对 PL.PAS 及 INTERPRETER.PAS 的源代码进行了分析，在原有代码基础上扩充了 for 循环、repeat 循环，以及 case 语句，此外还增加了识别注释，添加 break 及 continue 语句，以及函数调用的功能。

3、实现方法

一、代码分析

本次实验主要在 linux 环境下进行，使用 free pascal 编译器进行编译。前期使用 sublime text3 来阅读代码，并结合实验指导书对代码的结构，各种表的填写，指令的细节进行详细了解，大多数过程在实验指导书上均有涉及，此处不做展开。

二、功能拓展

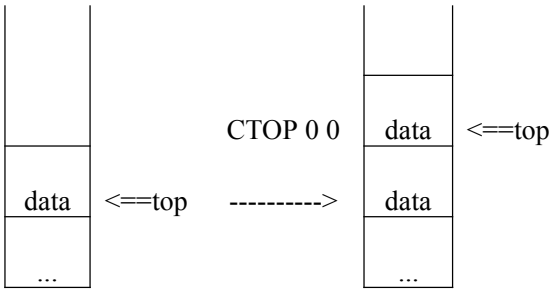
（一）添加 for 循环

1、文法

〈for 循环语句〉::= “for”〈变量〉“:=”〈表达式〉（“to”|“downto”）〈表达式〉do〈语句〉

2、汇编结构

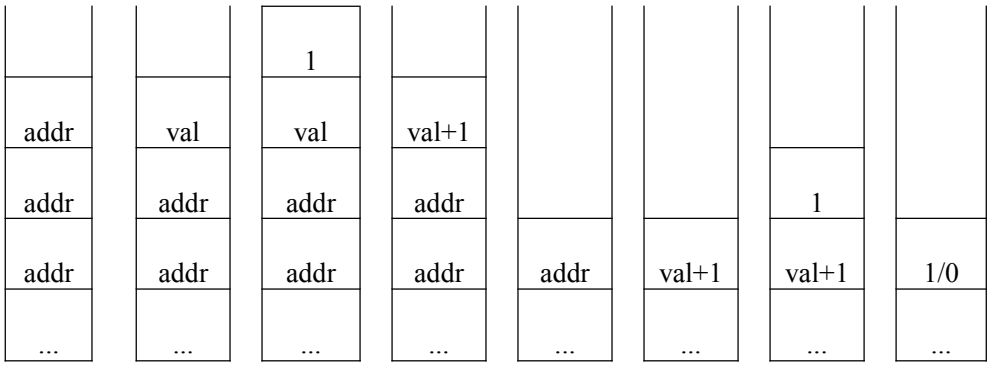
为了实现的方便，扩充了一条指令 CTOP，无参数，功能为将栈顶内容拷贝一份在加在栈顶。



```
        ;计算控制变量地址
        ;计算初始值
        STO    0  0
        ;计算控制变量地址
L1:     ;计算终止值
        LE     0  0   ;若是是 downto 这里是 GE
        JPC    0  L2
        ;循环体
        ;计算控制变量地址
        CTOP   0  0
        CTOP   0  0
        LODT   0  0
        LIT    0  1   ;若是是 downto 这里就是-1
        ADD    0  0
        STO    0  0
        LODT   0  0
        JMP    0  L1
L2:     ;循环语句之后的代码
```

在计算过程中总是先加载控制变量的地址，这是由于控制变量的地址可能是一个变量，例如“for a[i] := 1 to j * 2 do”，故而在实现时需保留计算变量地址的代码片段，需要时复制到相应位置。

在控制变量自加 1 前，程序使用 CTOP 使变量地址复制了两份，这是由于其加 1 后，立即又用其与终止值比较，采取这种方法可以少计算两次变量地址，此时栈变化状态如下图。



3、代码实现

代码实现中先是参考赋值语句，不同的是记录赋值号左侧的地址计算语句的起始地址和结束地址，以便后需要加载地址时可拷贝此段代码。循环的终止值调用 expression 过程分析，翻译循环体的部分则调用 statement 过程，在中间添加前面汇编结构中所提到的汇编指令。此外，注意回填 jpc 指令的跳转地址。详细的代码见所附文件 pl.pas 中的 forstatement 过程。

（二）添加 repeat 循环

1、文法

在正常的 pascal 语法下 repeat 语句的文法为：

〈repeat 循环〉 ::= “repeat”〈语句〉{“;”〈语句〉}“until”〈表达式〉

不过为了实现的方便，此处采取以下的定义：

〈repeat 循环〉 ::= “repeat”〈语句〉“until”〈表达式〉

二者的区别在于第一种定义下符合语句不用以 being、end 来标志开始和结束。

2、汇编结构

```
L1:   ;循环体
      ;计算表达式
      JPC   0  L1
      ;循环语句之后的代码
```

3、代码实现

repeatstatement 过程的实现比较简单，只需使用调用 statement 过程翻译循环体，调用 expression 过程翻译表达式，再添加 JPC 指令跳转到循环体开始的地址。过程注意 follow 集合的添加，以及表达式值类型的判断

```
procedure repeatstatement;
  var x:item;
begin  { repeatstatement }
```

```
    getsym;
    enterrep;
    labtab[lx]:=cx;lx:=lx+1;
    cx1:=cx;
    statement([untilsym]);

    if sym=untilsym then getsym else error(77);

    cx2 := cx;
    expression(fsys, x);
    if x.typ <> bool then error(34);
    gen(jpc,0,cx1);
    labtab[lx]:=cx;lx:=lx+1;
    outrep(cx, cx2);
  end;  { repeatstatement }
```

（三）添加 case 语句

1、文法

<case 语句>::=“case”<表达式>“of”{<表达式>“:”<语句>“;”}[“else”<语句>]“end”

2、汇编结构

```
        ;计算表达式
        CTOP   0  0
        ;计算条件表达式 1
        EQ     0  0
        JPC    0  L2
        ;语句 1
        JMP    0  Out
  L2:   CTOP   0  0
        计算条件表达式 2
        EQ     0  0
        JPC    0  L3
        ;语句 2
        JMP    0  Out
  L3:   CTOP   0  0
        ;计算条件表达式 3
```

```
        ;...
    Ln: ;若有 else 的话，此处有语句的代码
    Out: JPC   0   cx+1   ;调用 gen(jpc,0,cx+1)获得,
                          ;为了弹出栈顶多余的元素
        ;case 结束后的代码
```

3、实现方法

先使用 expression 过程翻译 case 后的表达式，此后运行时对应的栈顶为表达式的结果，对于每一个分支，先产生自定义的 CTOP 指令拷贝栈顶，然后还是调用 expression 翻译条件，进行比较，再产生 JPC 指令，指向下一处分支开始的位置（待回填），使用 statement 翻译分支所执行的语句，产生 JMP 指令，指向 case 结束的地方，此处是形成 JMP 链，待回填，填写前面 JPC 的地址为 cx，循环进行每个分支的翻译。

对于待回填同一目标地址的 JMP 可以先连接成链，保留链首，每次加入新的 JMP 指令时，先将新 JMP 指向原链首，再将链首指向 cx-1。回填时沿链首赋值即可。在实验中使用下面的函数实现回填 JMP 链。

```
procedure jmpbacktrace(head, target:integer);
  var pre:integer;
begin
  while head <> 0 do
    begin
      pre := code[head].a;
      code[head].a := target;
      head := pre
    end
end;
```

详细的 casestatement 过程见源代码文件。

先前的三个功能都是对语句进行扩充，也即 statement 的内容，故而需要增加 statbegsym 集合，同时这三个过程也是定义在过程 statement 中的。

（四）添加 break 和 continue 语句

break 语句和 continue 语句都是 statement 的一个子集，且对应到代码中只是一条 JMP 指令，不同的是 break 跳出最近一层循环，而 continue 是跳到最近一层循环的起始位置，且若是在 for 中 continue 是会对控制变量自增或是自减。换一种角度说，对于每一层循环中的 break 语句都是指向同一点，continue 语句指向另为一个点，可以用前面所提的 JMP 链回填的方法，在循环结束后回填对应的 break 点和 continue 点。

此外，由于 break 和 continue 可以出现在 if 语句中，且循环可以嵌套。为了找到每一层循环的 break 链和 continue 的链首，需将这些链首压入栈中，初值为 0。进入循环后，压入新链首，每次遇到 break 和 continue 后，至于利用相应栈顶的链首指针进行加入链中的操作，最后在出循环时，指定 break 点和 continue 点回填，弹栈。

链首形成的栈定义如下

```
brkt: array[0..100] of integer; {break head stack}
cont: array[0..100] of integer; {continue head stack}
rx:integer; {top of stack}
```

进入循环时，出循环分别调用 enterrep 和 outrep 过程，且出循环时指出跳转点。

```
procedure enterrep;
begin
  if rx = rxmax then
    begin
      error(88);
      writeln('too many repetition');
      close(sfile);
      close(listfile);
      exit
```

```
    end
  else begin
      rx := rx + 1; brkt[rx] := 0;
    cont[rx] := 0;
  end
end;

procedure outrep(bpoint, cpoint:integer);
begin
  if rx > 0 then
    begin
      jmpbacktrace(brkt[rx], bpoint);
      jmpbacktrace(cont[rx], cpoint);
      rx := rx - 1;
    end
end;
```

（五）添加注释识别

注释识别是相对比较好实现的功能，可以在 getch 中就屏蔽掉注释，这样注释对上层的词法分析和翻译都是透明的。

在 getch 中若当前字符为‘{’，则进入在注释状态，循环 getch，直到当前字符为‘}’，在 getch 一次，出注释状态，下面的代码添加到 getch 最后，且 inzs 是定义在 getsym 中的 boolean 量，初值为 false。

```
 if (ch = '{') and not inzs then
   begin
     inzs := true;
     while ch <> '}' do getch;
     getch;
     inzs := false
   end;
```

（六）添加函数调用

1、语法

〈函数说明〉::=“function”〈标识符〉[〈参数表〉]“:”〈类型名标识符〉“;”

〈程序体〉“;”

〈函数调用语句〉:=〈函数标识符〉“（”[〈表达式〉{“,”〈表达式〉}]“)”

在这里为了保持栈结构的完整，只允许函数调用出现于表达式中，且为了在表达中区分函数调用及函数返回值的引用，函数调用时，不管是否有参数，必须加括号。

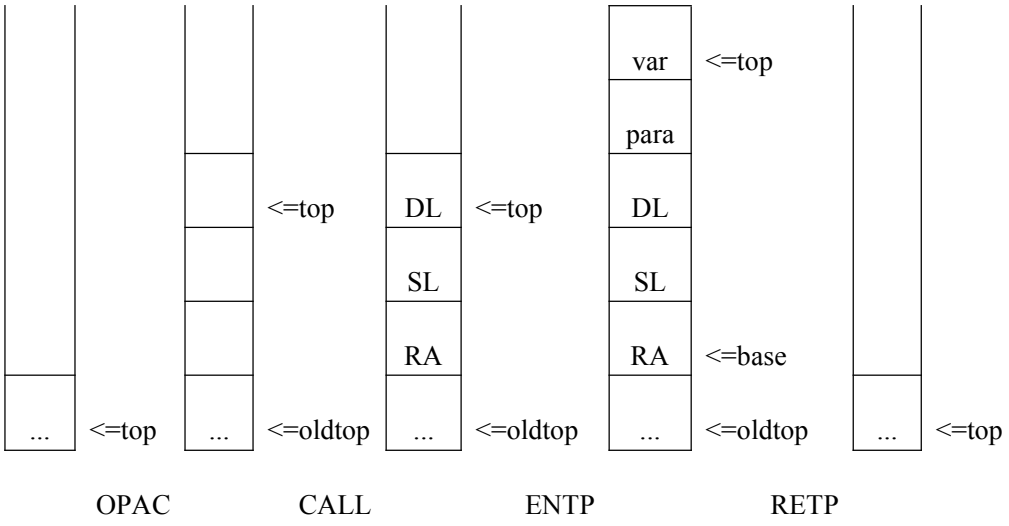
2、汇编细节

函数调用的功能实现参考了过程调用，可以说是在过程调用的基础上的一种扩充，下原先过程调用的过程如下

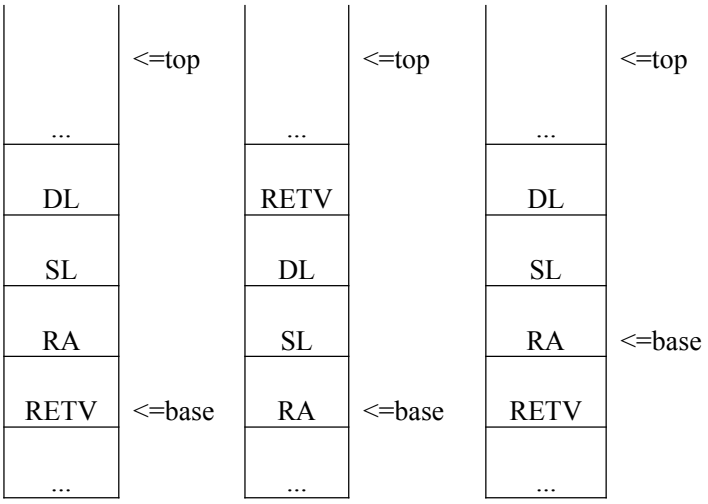
```
        OPAC   0  0
        ;计算参数列表
        CALL   l  Fn
Ra:     UDIS   0  0 ;若调用过程的 lev 大于等于 level,
                    ;则此处为调用后的其他代码
```

```
Fn:     ENTP   l  size
        ;程序体
        RETP   0  0
```

过程调用的栈变化如下：



因此考虑在活动记录中加入返回值有如下几种方法：



第一种是实验指导书上所参考的方法，这种填法是很难实现的，因为其 RA，SL 和 DL 的偏移量都变了，首先是 entp、outp 这些指令都得改，其次在运行中如果遇到 UDIS 指令，在栈中的某一项记录里没有额外信息是不知道其是过程还是函数，也就不知道对应的 DL、SL 的位置。

第二种是一种可行的方法，只需要增加与 RETP 对应的从函数返回指令即可，返回时，将返回值移到栈底，逻辑上也比较合理。不过可能修改的代码比较多。

第三种是一种强行进行修改的方法，也是将返回值放在原活动记录的顶部，不过 base 指向了返回值之上，访问时使用-retsize（返回值的 size）来当偏移量，这样对于汇编程序来说，形式上活动记录并没有发生变化，CALL、ENTP、RETP 和 UDIS 这些涉及修改活动记录的指令都不用进行修改，函数调用和过程调用也是有 OPAC 这条指令上的区别，对于翻译中 block 过程，可以完全不用考虑活动记录结构的事，唯一的不足是必须引入偏移地址为负，同时对当前活动记录以外的地方进行访存也是一种越界的行为，会使返回地址受到攻击。

综合考虑，作为一个实验性的功能选择第三种方案。

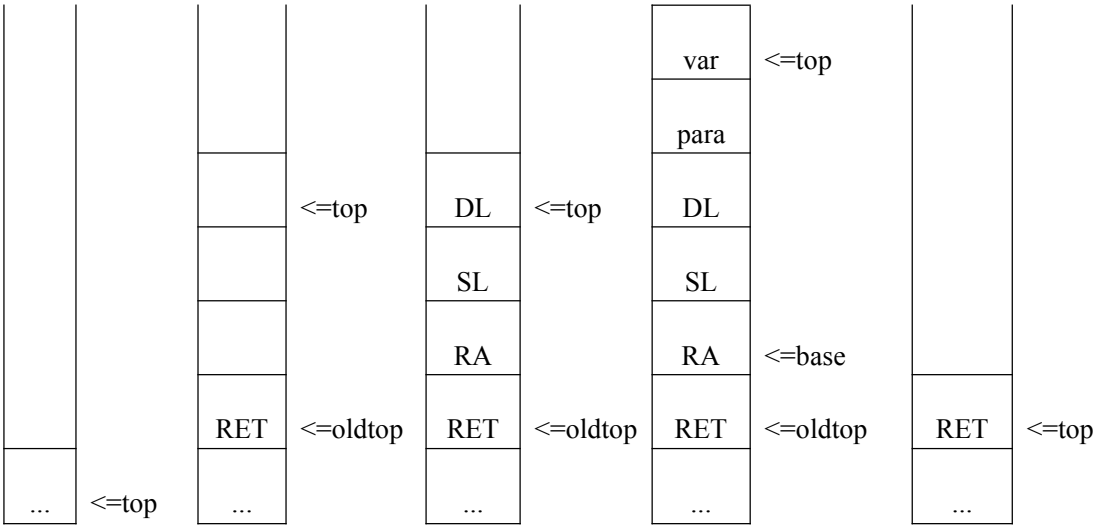
此时函数调用的汇编代码如下：

```
        OPF    0  retsize
```

```
        ;计算参数列表
        CALL   l   Fn
   Ra:  UDIS   0   0 ;若调用过程的 lev 大于等于 level,
                     ;则此处为调用后的其他代码
```

```
   Fn:  ENTP   l   size
        ;程序体
        RETP   0   0
```

OPF 是扩充的指令，用以打开函数的活动记录，a 参数是返回值的 size，栈变化如下：



3、实现方法

实现上主要进行了下面几项工作

（1）增加 nametab 的含义，function 也以 prosedure 为其 kind，不同是其 typ 为返回值的 type

（2）为 btab 增加成员，fsz:integer，当 block 为 function 时，为其返回值的 size，否则无定义；为 btab 表增加 ref:integer，指向此记录对应的 procedure 在 nametab 中的位置，用于在函数调用中检测函数值是否合法。

（3）增加过程 funcdeclaration，用于识别函数定义，代码与 procdeclaration 基本相同。

（4）修改 block 部分，在参数列表后增加识别“:”的逻辑。识别到冒号后调用 typ 过程识别后面的类型标识符，填入 nametab 中。

（5）修改 assignment 过程，若赋值号前的标识符是一个 procedure 类型，且有 typ，且标识符与当前 block 过程同名，则此为函数返回值，调用 gen(loda,level,-fsz);注意函数名在 nametab 中的 lev 会比 level 少 1 层。此处按前面的逻辑应该用 level。

（6）修改 factor 过程，若当前标识符为 procedure 类型，且有 typ，后无括号时，且位置合法，调用 gen(lod,level,-fsz)，后有括号时，进行函数调用。

（7）增加函数调用过程，代码与 call 中识别完左括号后的代码相似，需注意参数个数问题。

（8）修改 interpreter 文件，扩充相关指令。

具体代码细节可以看工程文件。若要查看此处所修改的部分，可以利用目录下的.git，查看提交了修改函数功能前后的不同。

4、实验的例程

样例一：for 循环，控制变量为数组中元素

```
program main;
  var i, j:integer;
      a:array[1..5] of integer;

begin
  j := -10;
  for a[4] := 1 downto 2 * j do
  begin
    call write(a[4])
  end
end.
```

样例二：嵌套 case

```
program main;
var
  a, b: integer;
begin
  a := 100;
  b := 200;
  case (a) of
    1: call write(1);
    2: call write(2);
    3: call write(3);
    100: begin
           call write(100);
           case (b) of
              200: call write(200);
           end;
           case a of
              200: call write(101);
           else
              call write(111);
           end
         end;
    100: call write('f');
    100: call write('f');
    100: call write('f');
  end;
  call write(-1)
end.
```

样例三：递归计算阶乘

```
program prime;

const pmax=50;
var cnt:integer;
    nop: array[1..pmax] of boolean;

function getp:integer;
  var i,j:integer;
begin
  for i := 2 to pmax do
```

```
    begin
      if nop[i] then continue;
      call write(i);
      getp := getp + 1;
      j := i * 2;
      while j <= pmax do
        begin
          nop[j] := true;
          j := j + i
        end
    end
end;

begin
  cnt := getp();
  call write(cnt)
end.
```

样例四：递归实现的拓展欧几里德算法

```
program main;
var
  i,j,k,l:integer;

function exgcd(a,b:integer; var x,y:integer):integer;
  var t:integer;
begin
  if b = 0 then
  begin
    x := 1;
    y := 0;
    exgcd := a
  end
  else begin
    exgcd := exgcd(b, a mod b, x, y);
    t := x;
    x := y;
    y := t - a / b * y
  end
end;
```

```
begin
  while true do
  begin
    call read(i);
    call read(j);
    i := exgcd(i,j,k,l);
    call write(i);
    call write(k);
    call write(l)
  end
end.
```

5、总结

本次实验可以说是一次充满了乐趣的实验，通过自己扩充编译器来实现功能可以获得极大的满足感，同时在这个过程中锻炼了自己读代码，写代码的能力，同时通过这次实验也学会了 pascal 这门语言，感觉收获很多。实验前后共花费了约一周时间，在前四天基本都是在阅读代码，添加基础功能用了一天，添加函数用了一天半，剩下的时间用了做一些额外的测试。