第 9 章　符号表

第 1 题：

根据你所了解的某个 FORTRAN 语言的实现版本，该语言的名字作用域有哪几种?

答案：

FORTRAN 中，名字作用域有四种：

<1>在 BLOCK DATA 块中定义的标识符，其作用域是整个程序。

<2>在 COMMON 块中定义的标识符,其作用域是声明了该 COMMON 块的所有例程(包括函数和过程)。

<3>在例程中定义的标识符（包括哑变量），其作用域是声明该标识符的例程。

<4>在例程中用 SAVE 定义的标识符，其作用域是声明该标识符的例程，且在退出该例程时，该标识符的值仍保留（即内部静态量）。

第 2 题：

C 语言中规定变量标识符的定义可分为 extern,extern static,auto,local static 和 register 五种存储类：

(1) 对五种存储类所定义的每种变量，分别说明其作用域。

(2) 试给出适合上述存储类变量的内存分配方式。

(3) 符号表中登录的存储类属性，在编译过程中支持什么样的语义检查。

答案：

(1) extern 定义的变量，其作用域是整个 C 语言程序。

extern static 定义的变量，其作用域是该定义所在的 C 程序文件。

auto 定义的变量，其作用域是该定义所在的例程。

local static 定义的变量，其作用域是该定义所在的例程。且在退出该例程时，该变量的值仍保留。

register 定义的变量，其作用域与 auto 定义的变量一样。这种变量的值，在寄存器有条件时，可存放在寄存器中，以提高运行效率。

(2) 对 extern 变量，设置一个全局的静态公共区进行分配。

对 extern static 变量，为每个 C 程序文件，分别设置一个局部静态公共区进行分配。

对 auto 和 register 变量，设定它们在该例程的动态区中的相对区头的位移量。而例程动态区在运行时再做动态分配。

对 local static 变量，为每个具有这类定义的例程，分别设置一个内部静态区进行分配。

(3) 实施标识符变量重复定义合法性检查，及引用变量的作用域范围的合法性检查。

第 3 题：

对分程序结构的语言，为其符号表建立重名动态下推链的目的是什么?概述编译过程中重名动态下推链的工作原理。

答案：

重名动态下推链的目的是，保证在合法重名定义情况下，提供完整确切的符号表项，从而保证引用变量的上下文合法性检查和非法重名定义检查。

其工作原理是：当发生合法重名定义时，将上层重名表项下推到链中，而在符号标中原重名表项处登录当前重名定义的符号属性；在退出本层时，将最近一次下推的表项，回推登录到符号表中原重名表项处。

第 4 题：

某 BASIC 语言的变量名字表示为字母开头的字母或数字两个字节的标识符，该语言的符号表拟采用杂凑法组织，请为其设计实现一个有效散列的杂凑算法，并为解决散列冲突，设计实现一个再散列算法。

答案：

可采用下列散列杂凑算法（设表长为 N） 散列地址=MOD（(<第一字节值>+<第二字节值>)，N)。

发生冲突时再散列的方法：在该冲突处开始，向下寻找第一个空表项，若找到则该表项地址为再散列地址；若找不到空表项，则循环到表头开始，向下寻找第一个空表项，一直到发生冲突处为止，若找到空表项则该表项地址为再散列地址，否则表示符号表已满，编译系统给出符号表溢出信息，终止编译。

附加题

问题 1：

利用 Pascal 的作用域规则，试确定在下面的 Pascal 程序中的名字 a 和 b 的每一次出现所应用的说明。

```
program m ( input, output ) ;
  procedure n ( u, v, x, y : integer ) ;
    var m : record m, n : interger end ;
        n : record n, m : interger end ;
    begin
      with m do begin m := u ; n:= v end ;
      with n do begin m := x ; n := y end ;
      writeln ( m.m, m.n, n.m, n.n )
    end ;
  begin
    m ( 1, 2, 3, 4 )
  end.
```

答案：

图中用蓝色数字下标相应标明。

```
program m1 ( input, output ) ;
  procedure n1( u, v, x, y : integer ) ;
    var m2 : record m3, n2 : interger end ;
        n3 : record n4, m4 : interger end ;
    begin
      with m2 do begin m3 := u ; n2 := v end ;
      with n3 do begin m4 := x ; n4 := y end ;
      writeln ( m2.m3, m2.n2, n3.m4, n3.n4 )
    end ;
  begin
    m1 ( 1, 2, 3, 4 )
  end.
```

问题 2：

当一个过程作为参数被传递时，我们假定有以下三种与此相联系的环境可以考虑，下面的 Pascal 程序是用来说明这一问题的。

一种是词法环境（lexical environment），如此这样的一个过程的环境是由这一过程定义之处的各标识符的联编所构成；

一种是传递环境（passing environment），是由这一过程作为参数被传递之处的各标识

符的联编所构成；

另一种是活动环境（activation environment），是由这一过程活动之处的各标识符的联编所构成。

试考虑在第（11）行上的作为一个参数被传递的函数 f。利用对于 f 的词法环境、传递环境和活动环境，在第（8）行上的非局部量 m 将分别处在第（6）行、（10）行和（3）行上的 m 的说明的作用域中。

（a）图示出每个过程的活动记录。

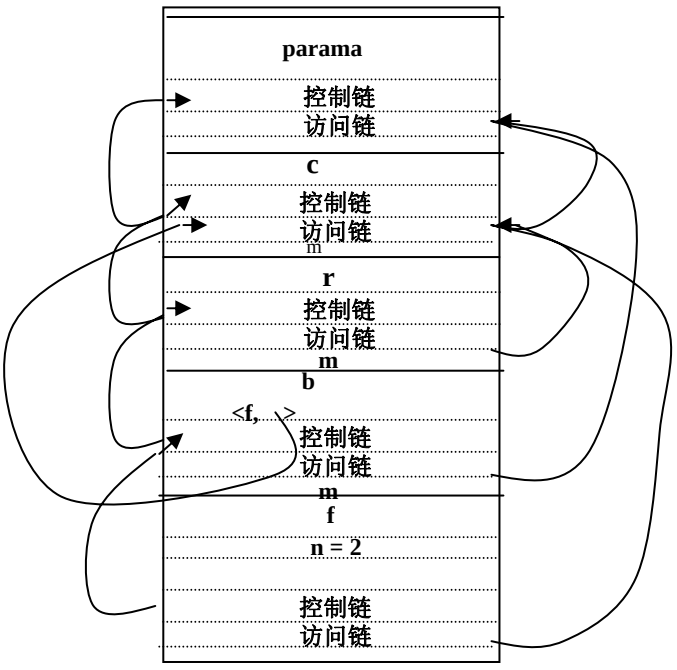
（b）试为此程序画出活动树。

（c）试给出程序的输出。

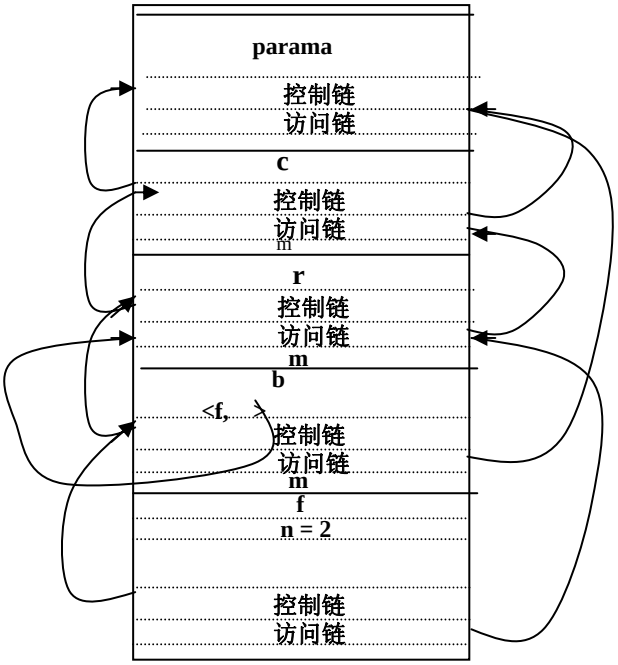
```
(1) program param ( inout, output ) ;
(2)    procedure b ( function h ( n : integer ) : integer ) ;
(3)        var m : integer ;
(4)        begin m := 3 ; writeln ( h ( 2 ) ) end { b } ;
(5)    procedure c ;
(6)        var m : integer ;
(7)        function f ( n : integer ) : integer ;
(8)           begin f := m + n end { f } ;
(9)        procedure r ;
(10)           var m : integer ;
(11)           begin m := 7 ; b ( f ) end    { r } ;
(12)       begin m := 0 ; r end { c } ;
(13)   begin
(14)       c
(15)   end .
```

答案：

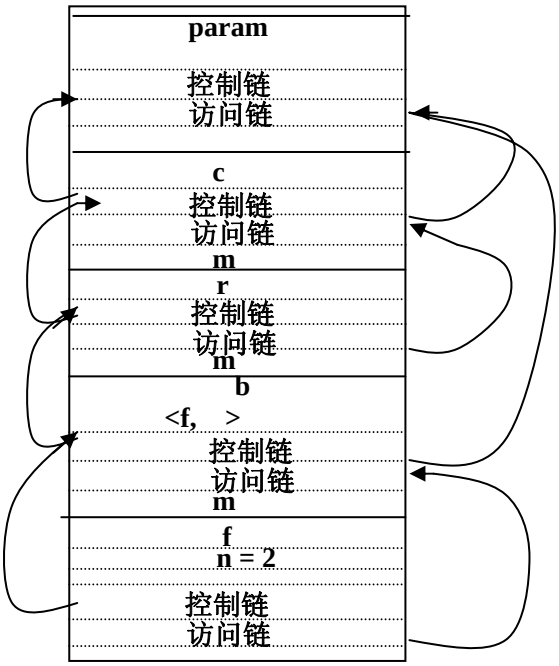
（a）词法环境



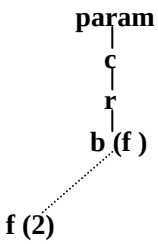
传递环境


parama
控制链
访问链
c
控制链
访问链
m
r
控制链
访问链
m
b
<f, >
控制链
访问链
m
f
n = 2
控制链
访问链

活动环境


param
控制链
访问链
c
控制链
访问链
m
r
控制链
访问链
m
b
<f, >
控制链
访问链
m
f
n = 2
控制链
访问链

（b）活动树



（c）词法环境：2；传递环境：9；活动环境：5。

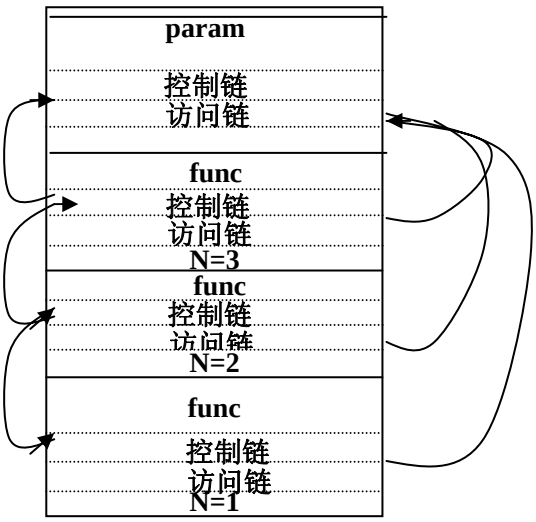
问题 3：

己知程序段：

```
BEGIN
  integer i;
  read(i);
  write("value=",func(i));
  ...
END
integer procedure func(N)
  integer N;
  BEGIN
    IF N==0 THEN func=1;
    ELSE IF N==1,THEN func=1;
    ELSE func=N*func(N-1)
  END;
```

求当输入 i=3 时，本程序执行期间对运行栈的存储分配图。

答案：

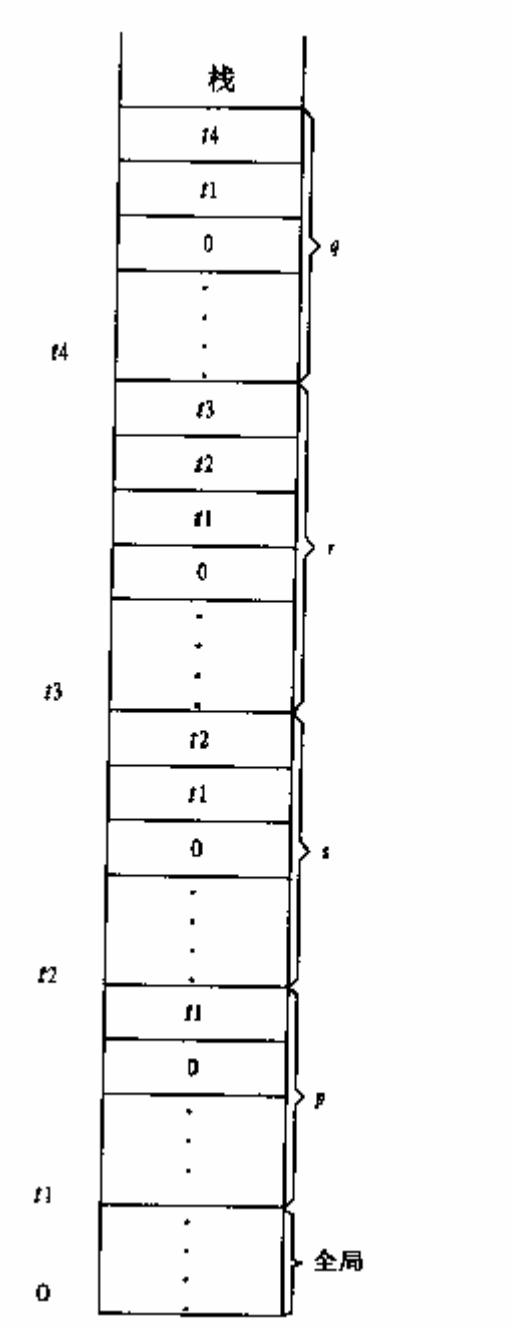


问题 4：

某语言允许过程嵌套定义和递归调用(如 Pascal 语言)，若在栈式动态存储分配中采用嵌套层次显示表 display 解决对非局部变量的引用问题，试给出下列程序执行到语句“b：＝10；”时运行栈及 display 表的示意图。

```
var x，y；
PROCEDURE P；
var a;
PROCEDURE q；
    var b：
    BEGIN{q}
     b：＝10；
    END{q};
    PROCEDURE s;
        var c，d；
       PROCEDURE r;
        var e,f;
        BEGIN{r}
           call q；
        END {r}；
       BEGIN{s}
        call r；
       END{s}；
    BEGIN {p}
        call s;
    END{p}；
    BEGIN{main}
      call p；
    END{main}.
```

答案：



问题 5：

试问下面的程序将有怎样的输出？分别假定：
（a）传值调用（call-by-value）；
（b）引用调用（call-by-reference）；
（c）复制恢复（copy-restore）；
（d）传名调用（call-by-name）。

```
program main ( input, output ) ;
  procedure p ( x, y, z ) ;
    begin
      y := y + 1 ;
      z := z + x ;
    end ;
  begin
    a := 2 ;
    b := 3 ;
    p ( a + b , a, a ) ;
    print a
  end.
```

答案：

1). 传地址：所谓传地址是指把实在参数的地址传递给相应的形式参数。在过程段中每个形式参数都有一对应的单元，称为形式单元。形式单元将用来存放相应的实在参数的地址。当调用一个过程时，调用段必须领先把实在参数的地址传递到一个为被调用段可以拿得到的地方。当程序控制转入被调用段之后，被调用段首先把实参地址捎进自己相应的形式单元中，过程体对形式参数的任何引用 1 或赋值都被处理成对形式单元的间接访问。当调用段工作完毕返回时，形式单元(它们都是指示器)所指的实在参数单元就持有所指望的值。

2). 传结果：和“传地址”相似(但不等价)的另一种参数传递力法是所谓“传结果”。这种方法的实质是，每个形式参数对应有两个申元，第 1 个单元存放实参的地址，第 2 个单元存放实参的值。在过程体中对形参的任何引用或赋值都看成是对它的第 2 个单元的直接访问。但在过程工作完毕返回前必须把第 2 个单元的内容行放到第 1 个单元所指的那个实参单元之中。

3). 传值：所谓传值，是一种简单的参数传递方法。调用段把实在参数的值计算出来并存放在一个被调用段可以拿得到的地方。被调用段开始丁作时，首先把这些值抄入形式中元中，然后就好像使用局部名一样使用这些形式单元。如果实在参数不为指示器，那末，在被调用段中无法改变实参的值。

4). 传名：所谓传名，是一种特殊的形——实参数结合方式。解释“传名”参数的意义：过程调用的作用相当于把被调用段的过程体抄到调用出现的地方，但把其中任一出现的形式参数都替换成相应的实在参数(文字替换)。它与采用“传地址”或“传值”的方式所产生的结果均不相同。

(a)2；　　(b)8；　　(c)7；　　(d)9。

问题 6：

下面程序的结果是 120，但是如果把第 11 行的 abs(1)改成 1 的话，则程序结果是 1，分析为什么会有这样不同的结果。

```
int fact()
{
 static int i=5;
 if(i==0)
  {
    return(i);
  }
 else
  {
    i=i-1;
    return((i+abs(1))*fact());/*第 11 行*/
  }
}
main()
{
 printf("factor of 5=%d\n",fact());
}
```

答案：

(1)i 是静态变量，所有地方对 i 的操作都是对同一地址空间的操作，所以每次递归进入 fact 函数后，上一层对 i 的赋值仍然有效。值得注意的是，每次递归时，(i+abs(1)*fact()中(i+abs(1))的值都先于 fact 算出。所以，第 1 次递归时，所求值为 5*fact,第 2 次递归时，所求值为 4*fact，第 3 次递归时，所求值为 3*fact，如此类推，第 5 次递归时所求值为 1*fact,而此时 fact 值为 1。这样一来，实质上是求一个阶乘函数 5*4*3*2*1，所以结果为 120。

(2)之所以改动 abs(1)为 1 后会产生变化，这主要和编译时的代码生成策略有关。对于表达式(i+abs(1))*fact()，因两个子表达式都有函数调用，因此编译器先产生左子表达式代码，后产生右子表达式代码。当 abs(1)改成 1 后，那么左子表达式就没有函数调用了，于是编译器会先产生右子表达式代码，这样一来，每次递归时，(i+abs(1))*fact()如 c4)中(i+abs(1))的值都后于 fact 算出。第 1 次递归时，所求值为(i+1)*fact，第 2 次递归时，所求值为(i+1)*fact，第 3 次递归时，所求值为(i+1)*fact，如此类推，第 5 次递归时所求值为(i+1)*fact，而此时 fact 为 1，i 为 0。这样每次递归时所求的实际都是 1*fact，最后输出还是 1。因此有不问的结果。

问题 7：

下面是一个 c 语言程序及其运行结果。从运行结果看，函数 FUNC 中 4 个局部变量 i1,j1，f1，e1 的地址间隔和它们类型的大小是一致的。而 4 个形式参数 i，j，f，e 的地址间隔和它们类型的大小不一致，试分析不一致的原因。

```
#include＜stdio.h>
FUNC(i,j,f,e)
short i,j;
float f,e;
{
short i1,j1;
float f1,e1;
i1=i;j1=j;f1=f;e1=e;
printf("Addrsses of i,j,f,e＝%ld %ld %ld %ld\n",&i,&j,&f,&e);
printf("Addrsses of i1,j1,f1,e1＝%ld %ld %ld %ld\n",
    &i1,&j1,&f1,&e1);
printf("size of short,int,long,float,double＝%ld %ld %ld %ld\n",
      sizeof(short),sizeof(int),sizeof(long),sizeof(float),sizeof(double));
}
main()
{
    short i,j;
    float f,e;
    i=j=1;f=e=1.0;
    FUNC(i,j,f,e);
}
```

运行结果为：

```
Addrsses of i,j,f,e＝-268438178,-268438174,-268438172,-2684364
Addrsses of i1,j1,f1,e1＝-268438250,-268438252,-268438256,-268438260
size of short,int,long,float,double＝2,4,4,4,8
```

答案：

C 编译是不作调用时形参和实参一致性检查的。注意在 C 语言中，整数有 short，int 和 long 共 3 种类型，实数有 float 和 double 两种类型。C 语言的参数调用是按传值方式进行的，一个整型表达式的值究竟是按 short，int 还是 long 方式传递呢?

显然，这儿约定为应该按取参数的最大方式来读取参数，即对于整数用 long 方式，实数用 double 方式。虽然参数传递是按最大方式进行，但是被调用函数中形式参数是按自己类型定义来取值的。这样一来，参数传递和取值是不一致的，就要考虑边界对齐问题。因 float 类型需要 4 字节，而 double 类型为 8 字节，故当从 double 类型变量取 float 类型的值时，应从第 l~4 个字节分配 float 类型数。short 型的形参是取 long 型值 4 字节中的后两字节内容，float 型的形参是取 double 值 8 字节的前 4 字节。这样才出现这 4 个形参地址的结果。