第十一章 运行时环境2024

语义分析相关里程碑

声明的语义分析：

变量声明D→Td

数组声明D→Td[Ǐ]

函数声明D→Td(Ǎ){ĎŠ}

形参变量声明A→Td

数组原型声明A→Td[]

函数原型声明A→Td()

函数符号表：

表头信息

登记项及其信息

语句的语义分析：

变量引用：右值E→d 左值S→d=E

数组元素引用：右值E→d[Ě] 左值S→d[Ě]=E

函数调用：语句S→d(Ě) 右值E→d(Ě)

形参变量↔实参E

形参数组原型↔实参d[]

形参函数原型↔实参d()

赋值语句及算术表达式

布尔表达式及顺序、分支、循环语句

标号与转移语句

函数符号表：

code域

中间语言代码；

运行时环境

代码优化与寄存器分配：

中间代码优化与目标代码优化

基于中间代码的寄存器分配

基于目标代码的寄存器分配

目标代码生成：

目标代码生成；

MIPS代码

目标代码优化；

指令模板与目标代码生成例

[t1=1	add t1, R0, 1
t2=j+t1	add t2, j, t1
t3=1	add t3, R0, 1
t4=i-t3	sub t4, i, t3
t5=t4*4	mul t5, t4, 4
t6=t5-1400	//t6=t5-1400
t7=M[t6]	lw t7,-1400(t5)
t8=t7*20	mul t8, t7, 20
t9=t8+k	add t9, t8, k
t10=t9*4	mul t10, t9, 4
t11=t10-1000	//t11=t10-1000
t12=M[t11]	lw t12, -1000(t10)
t13=k*t12	mul t13, k, t12
t14=i*20	mul t14, i, 20
t15=t14+t2	add t15, t14, t2
t16=t15*4	mul t16, t15, 4
t17=t16-1000	//t17=t16-1000
M[t17]=t13]	sw t13, -1000(t16)

[rd = k]
add rd, R0, k

[rd = rs + rt]
add rd, rs, rt

[rd = rs - rt]
sub rd, rs, rt

[rd = rs * k]
mul rd, rs, k

[t = rs - k; rt = M[t]]
lw rt, -k(rs)

[rd = rs + k]
add rd, rs, k

[rd = rs * rt]
mul rd, rs, rt

指令模板与目标代码生成示例

```
t1=123;
IF t1!=0 THEN l1 ELSE l2;
LABEL l2;
t2=1;
IF t2<x THEN l3 ELSE l4;
LABEL l3;  //紧跟上条
LABEL l1;
IF y!=0 THEN l5 ELSE l6]
```

```
add t1, R0, 123
bne t1, R0, l1
l2:
add t2, R0, 1
slt rd, t2, x
beq rd, R0, l4
l3:
l1:
bne y, R0, l5
j l6
```

```
[IF rs!=0 THEN l1 ELSE l2;LABEL l2]
bne rs, R0, l1
l2:
[IF rs!=0 THEN l1 ELSE l2]
bne rs, R0, l1
j l2:
```

```
[IF rs<rt THEN l1 ELSE l2;LABEL l1]
slt rd, rs, rt
beq rd, R0, l2
l1:
```

一些分支指令模板举例

[GOTO l1]
j l1

[LABEL l1]
l1:

[IF rs=rt THEN l1 ELSE l2;LABEL l2]
beq rs, rt, l1
l2:

[IF rs=rt THEN l1 ELSE l2;LABEL l1]
bne rs, rt, l2
l1:

[IF rs=rt THEN l1 ELSE l2]
beq rs, rt, l1
j l2

[IF rs<rt THEN l1 ELSE l2;LABEL l2]
slt rd, rs, rt
bne rd, R0, l1
l2:

[IF rs<rt THEN l1 ELSE l2;LABEL l1]
slt rd, rs, rt
beq rd, R0, l2
l1:

[IF rs<rt THEN l1 ELSE l2]
slt rd, rs, rt
bne rd, R0, l1
j l2

运算指令模板举例

[t = rs + k; rt = M[t]]//t死亡
lw rt, k(rs) //t若是临变则可

[rt = M[rs]]
lw rt, 0(rs)

[rt = M[k]]
lw rt, k(R0)//k为16位有符号立即数

[t = rs + k; M[t] = rt]//t死亡
sw rt, k(rs)

[M[rs] = rt]
sw rt, 0(rs)

[M[k] = rt]
sw rt, k(R0)

[rd = rs + rt]
add rd, rs, rt

[rd = rt]
add rd, R0, rt

[rd = rs + k]
add rd, rs, k

[rd = k]
add rd, R0, k

基于指令模板将中间语言程序映射为（拆分-匹配-转换为）目标语言程序。

观察

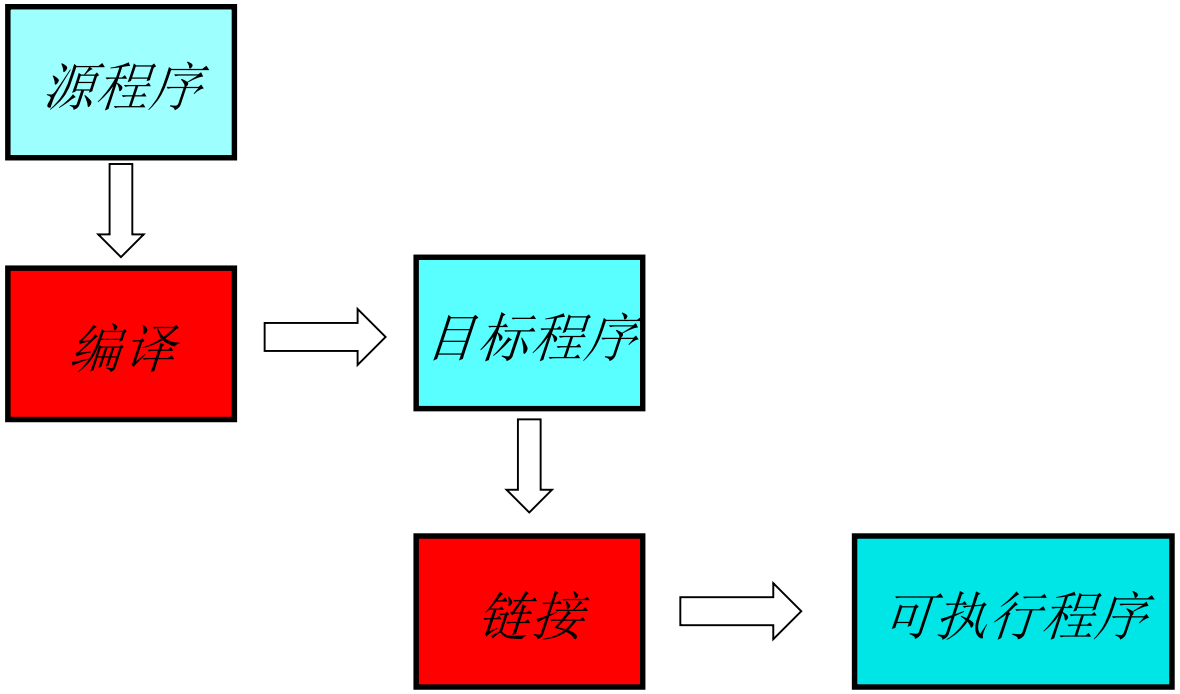
- 三地址指令受到MIPS指令模板制约。
- 寄存器分配必要性在于能使用寄存器操作指令和指令效率。
  - 针对中间代码的寄存器分配
  - 针对目标代码的寄存器分配
  - 所以指令模板中不区分变量还是寄存器
- 指令模板的设计有灵活性：
  - [t = rs + k; rt = M[t]]//t死亡　　lw rt, k(rs)
  - [rt = M[rs+k]]　　lw rt, k(rs)
  - 后者可减少中间代码生成开销，将在本章中得到使用。

关于函数可执行代码的构建

- 问题：关于函数调用的中间代码并不能正确执行
  - PAR r_1; ...; PAR r_n
  - v=CALL f, n
  - RETURN t
  - 已知f@code
- 需要添加运行时系统相关代码，形成如下结果：
  - 函数调用代码段　　//对应PAR-CALL代码段
  - 函数返回代码段　　//对应RETURN指令
  - f@label==　　//扩展函数代码为可执行代码
    f@prologue++　　//函数序言
    转换后的f@code++　　//转换名引用、PAR-CALL、RETURN
    f@epilogue　　//尾声
- 让程序中的所有名字引用都在各自正确的环境下进行。称为运行时存储分配。完成以上处理函数就能正常执行了。

从可执行程序说起

- 编译的终极目标是得到可执行程序

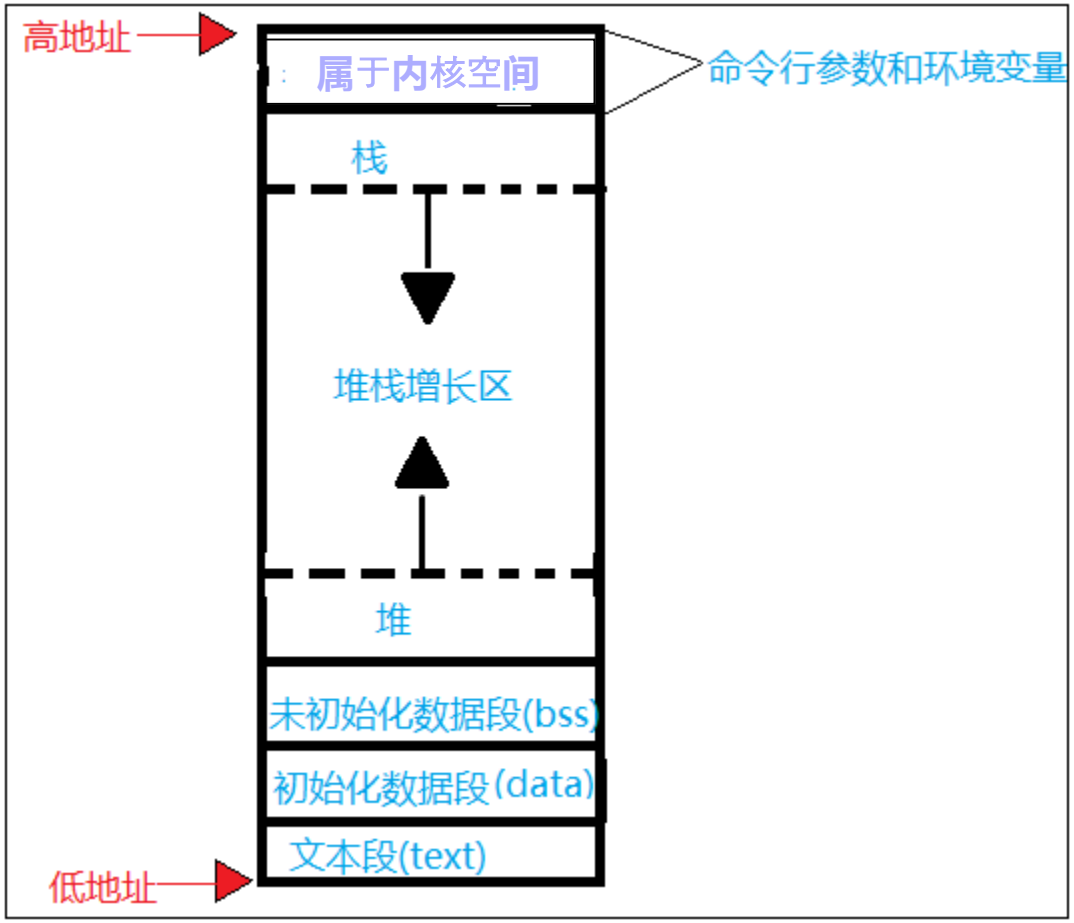


可执行程序：磁盘文件与内存映像

- 可执行程序有两种存在方式：
  - 编译过后，它以文件形式被保存在外存，被称为可执行程序文件，简称可执行文件。
  - 在运行时，可执行程序必须在内存里，此时它作为内存映像而存在，称为内存映像。
- 内存映像，指的是内核在内存中如何存放可执行程序文件。
- 可执行程序文件和内存映像的区别：
  - 可执行程序是位于硬盘上的，而内存映像位于内存中；
  - 可执行程序没有堆栈，因为只有当程序被加载到内存的时候才会分配相应的堆栈；
  - 可执行程序是静态的，因为它还没运行，但是内存映像是动态的，数据是随着运行过程改变的。
  - 内存映像、内存快照、程序断点与现场、瞬时描述？

内存映像示意

- elf格式（可执行可链接格式，为Linux、Android等采用）



内存映像一般布局

- Linux下的内存映像布局一般有如下几个段（从低地址到高地址）：
  1. 代码段： 即二进制机器代码，代码段是只读的，可以被多个进程共享；
  2. 数据段： 存储已初始化的变量，包括全局变量和初始化了的静态变量；
  3. 未初始化数据段： 存储未被初始化的静态变量，也就是BSS段；
  4. 堆： 用于存放动态分配的变量；
  5. 栈： 用于函数调用，保存函数返回值，参数等等。

bss区（未初始化区）

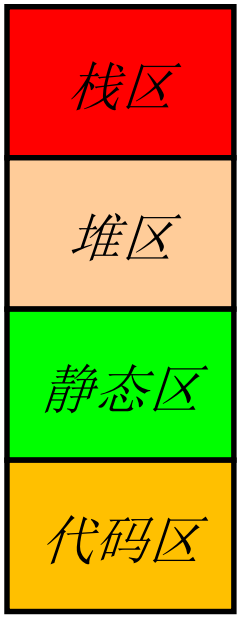
- C未初始化的全局变量和静态变量，初值为0或NULL

```
1 #include <stdio.h>
2 static int j;
3 int i;
4 int main(){
5         static int k;
6         printf("%d %d %d\n", i, j, k);
7         return 0;
8 }
```

- 为简略起见，课程内忽略未初始化区。

得到运行时存储空间划分

- 代码区：只读，
  - 存储方向为地址增大方向
- 静态区：全局变量，静态变量
  - 存储方向为地址增大方向、可读写
- 堆区：动态创建对象、可读写
  - 存储方向为地址增大方向
- 栈区：局部环境、可读写
  - 栈增长方向为地址减小方向



- 存储目标程序、分配存储单元给程序中的名字、存储运行时产生的中间结果、实现过程调用与返回时相关存储管理、实现动态申请与释放存储块的管理、正确访问名的值等。《运行时存储空间组织》

为什么这样能行？

- 事实依据（已有多种可执行程序格式）
- 必然性与合理性分析

从运行时环境说起

- 名与值是计算中最基本元素
  - 源程序中的名（变量名、数组名、过程名、标号）
  - 产生中间语言代码时生成的变量、标号
  - 基本类型的变量、数组元素取值（运行时计算得出）
  - 过程的代码块（编译时知道）
  - 标号对应的指令地址（运行前可确定）
- 程序运行就是不断对名的值进行计算的过程（求值eval）
  - 直接执行适合的代码完成名的值计算。
  - 在一个名的值计算过程中可能完成了另外一些名的值计算。
  - 在一个名的值计算过程中可能访问了一些名的值。

求值环境

- “在程序运行时，只看到了计算名的值，直到停机。”
- 所以运行时环境也叫求值环境
- 求值环境：名值对的列表。
- 名的值计算是在求值环境中进行：计算名的值，查找名的值，名值对加入环境（绑定）等。
- 一些术语：环境类与环境实例
  - 求值环境以过程为单位，每个过程都有一个环境类。
  - 在每一个过程的每一次被调用时，过程的环境类被实例化为一个环境实列，作为这次过程调用的求值环境。
  - 比如最外层过程只执行一次，所需环境实列就一个。
  - 如果允许过程声明嵌套的话，它们的环境类也有嵌套关系，可以认为是内层与外层的名值对列表连接在一起。

求值环境

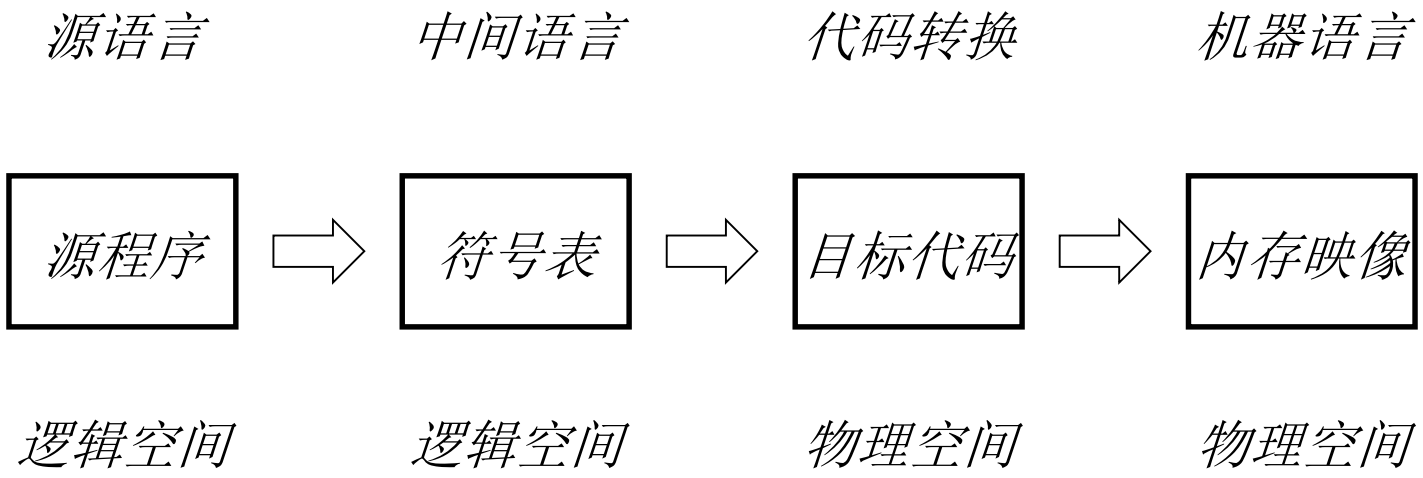
- 在一个程序的运行过程中，
  - 只有一个全局环境实例，它的生命期是永远
  - 可有一到多个局部环境实例
  - 一般地，局部环境实例是动态产生与消亡的，即有个有限生命期
  - 每次函数被调用时它的环境实例被创建、返回时释放

求值环境实现要点

- 给名分配存储单元存放它的值
- 在引用名时找到正确的存储单元

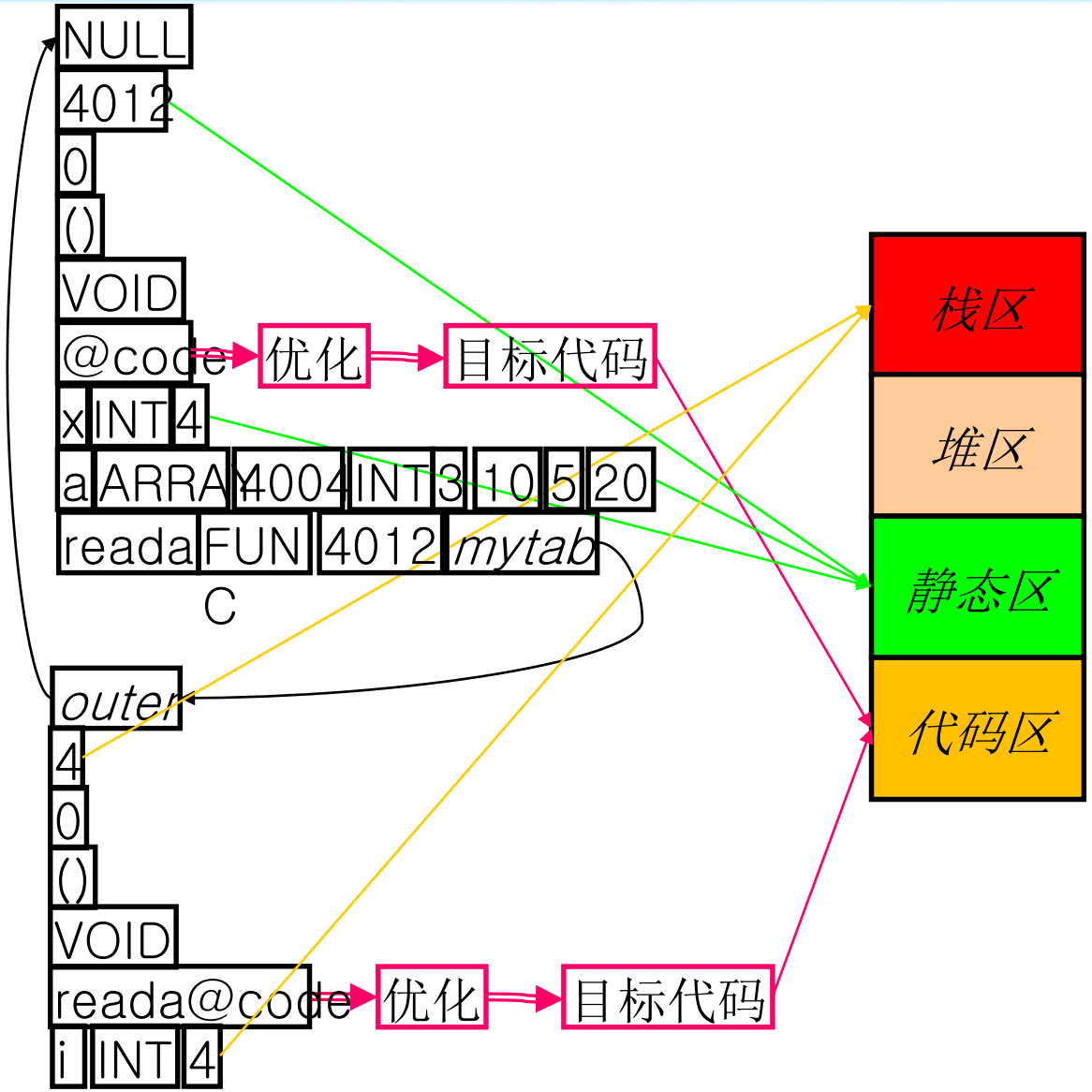
运行时环境（求值环境）

- 当源程序被翻译为中间语言代码后一切都在符号表中。
- 把符号表映射为内存映像成为必然。



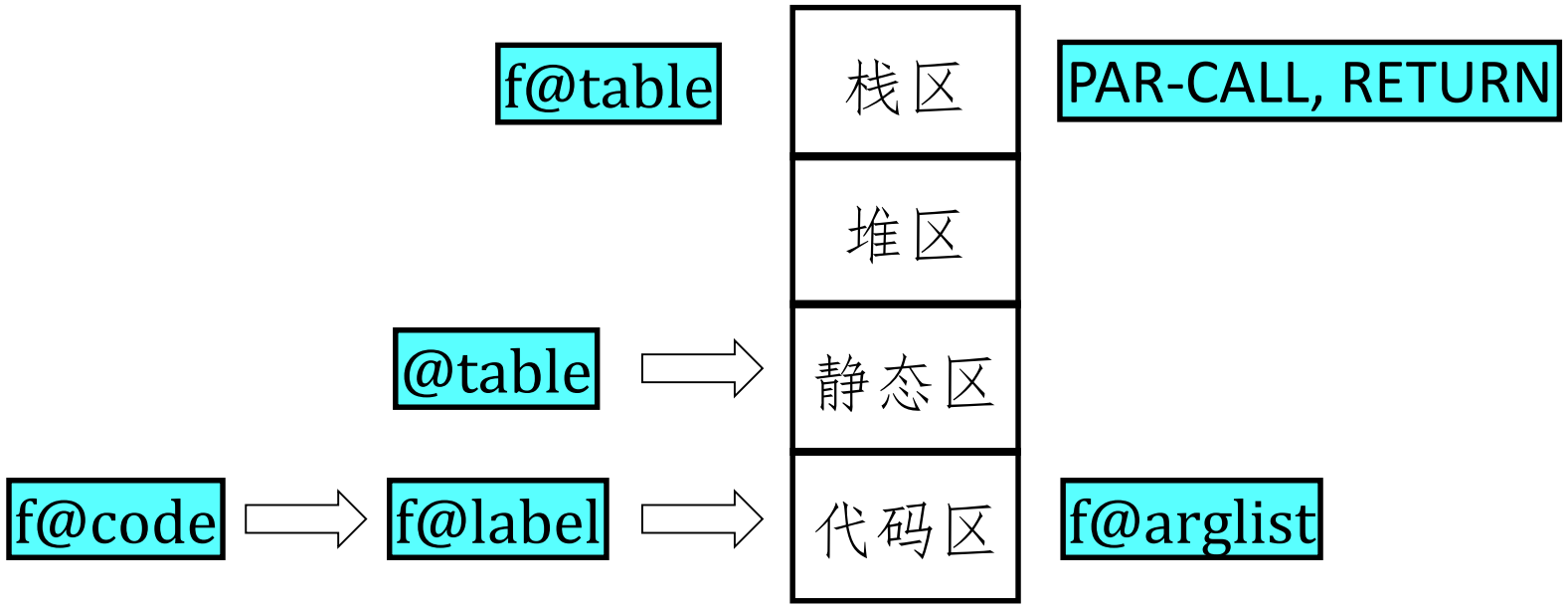
符号表到内存映像的映射示意

- 给名分配存储单元
  存放它的值
- 一些名的值多次更新
- 一些名的值永不更新
- 在引用名时找到
  正确的存储单元
- 名的值计算表现为
  环境的动态变化
- 当源程序被翻译为
  中间语言代码后一切
  都在符号表中
- 符号表映射为内存
  映像是必然的。



本章任务

- ~~构建~~代码区：所有函数代码一次性存储在代码区
- ~~构建~~静态区：无名函数（最外层函数）声明的名字一次性分配在静态区
- 栈区管理：栈帧（又称为活动记录），参数传递、局部和非局部名引用
- 堆区管理



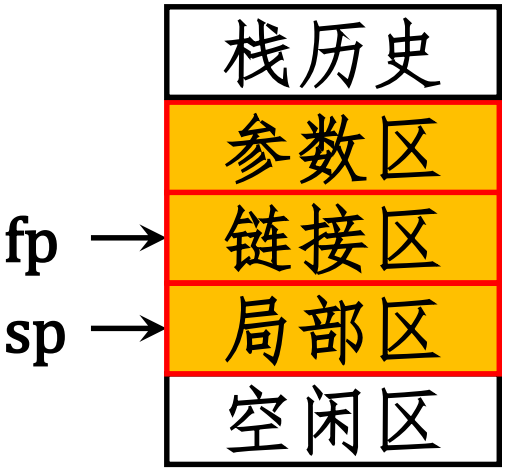
目标机提供的支持

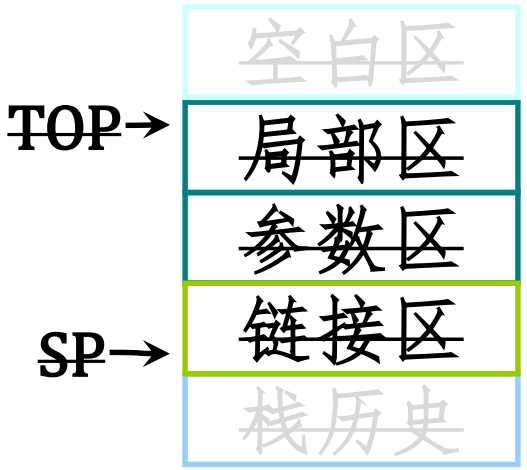
- 目标语言MIPS、X86、...（寻址、对齐、栈操作、转子等）
- 内存单元用一维数组M[MEMSIZE]表示
  - 代码区 从code_lb开始按地址增加方向，假定不溢出
  - 静态区 从static_lb开始按地址增加方向，假定不溢出
  - 堆区 从heap_lb开始按地址增加方向，假定不溢出
  - 栈区 从stack_lb开始按地址减小方向，假定不溢出
- 对于内存单元，采用变址寻址方式
  - M[基址+偏移量表达式]
  - 比如：[t=sp+200;M[t]...]和[M[fp-4]...]但无[M[rs+rt]...]
- 对于求值环境，采用栈帧实现方式：
  - 栈顶指针sp指向当前栈顶端（指向已用单元）。
  - 栈帧指针fp指向当前栈帧。

MIPS指令系统

- $R0 恒0
- $v0,$v1 返回值
- $a0-$a4 参数传递
- $t0-$t7 临时
- $sp
- $fp
- $ra 返回地址
- R型指令
- I型指令：基址寻址，PC相对寻址、立即数寻址
- j与jal

栈帧方案





- 惯例：地址下小上大
- push/pop是基于sp的
- fp被看作指向栈帧对象；
- sp被看作指向一个栈单元；
- fp和sp指向同一个栈帧的，就是当前栈帧。

- ~~惯例：地址下小上大~~
- ~~SP指向当前栈帧起点；~~
- ~~SP用于变址访问栈单元；~~
- ~~TOP指向另一端；~~
- ~~push/pop是基于TOP~~

9.1 程序运行时过程的活动

- 程序执行起点
  - C从main()开始
  - Fortran从主块开始
  - Pascal从主程序（即最外层分程序）开始
  - 我们的语言从最外层开始
- 过程的活动：过程的一次执行（从被调开始到返回结束）
- 程序在运行时任一时刻，可有多个活动是活着的，它们之间都是嵌套关系，没有并列关系的。
- 运行时快照：指定时刻的内存映像（包括栈内容）
- 运行时栈快照：指定时刻的栈内容（全部活着的活动的栈帧内容）

例：过程的活动

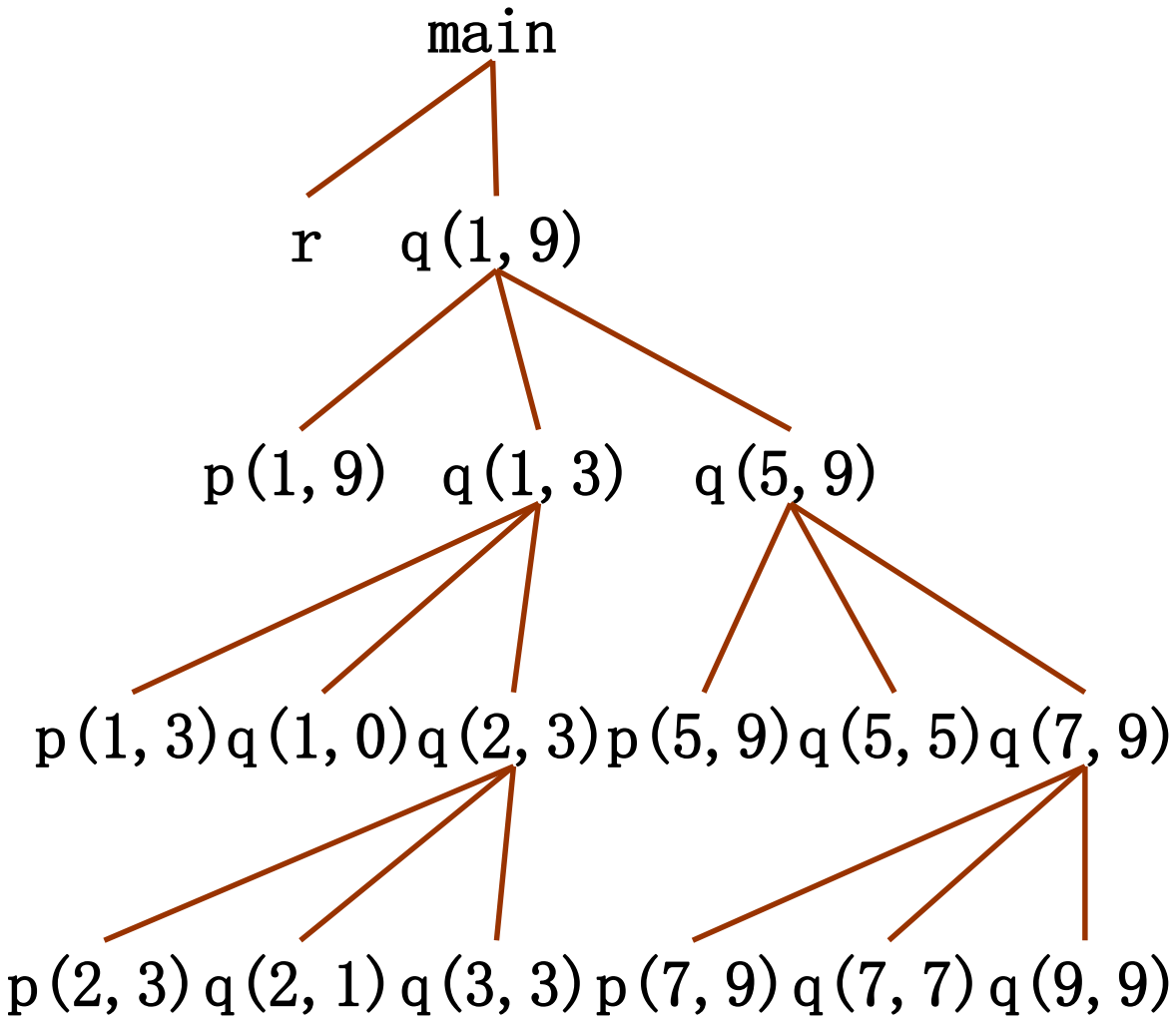
- 程序描述
  - main()调用gcd()
  - gcd()是递归的
- 则会有如下现象
  - 某时刻，活着的main活动嵌套着活着的gcd活动
  - 某时刻，一个活着的gcd活动嵌套着另一个活着的gcd活动
  - 某时刻，活着的gcd活动只有一个
  - 某时刻，活着的main活动不嵌套任何gcd活动
- 栈快照可有同一函数的多个栈帧

```
#include <stdio.h>
int x,y;
int gcd(int u, int v){
    if(v==0)return u;
    else return gcd(v,u%v);
}
main(){
    scanf("%d%d",&x,&y);
    printf("%d\n",gcd(x,y));
    return 0;
}
```

过程活动的生命期

- 如果过程p在它的一个活动中调用过程q，那么q的该次活动必定在p的活动结束之前结束。
- 正常与异常二种情形：
  - q活动正常结束，那么基本上在任何语言中，控制流返回到p中的调用q点之后继续。
  - 忽略异常处理。
- 因此，我们用一个树来表示在整个程序运行期间的所有过程的活动，这棵树被称为活动树。
- 活动树的每个结点是一个活动
- 树中父子结点是嵌套关系，父活动的生命期在时间上包含子活动的生命期
- 树中兄弟结点是并列关系，它们的生命期不相交，它们各自后代之间在生命期上也不相交。

过程的活动树

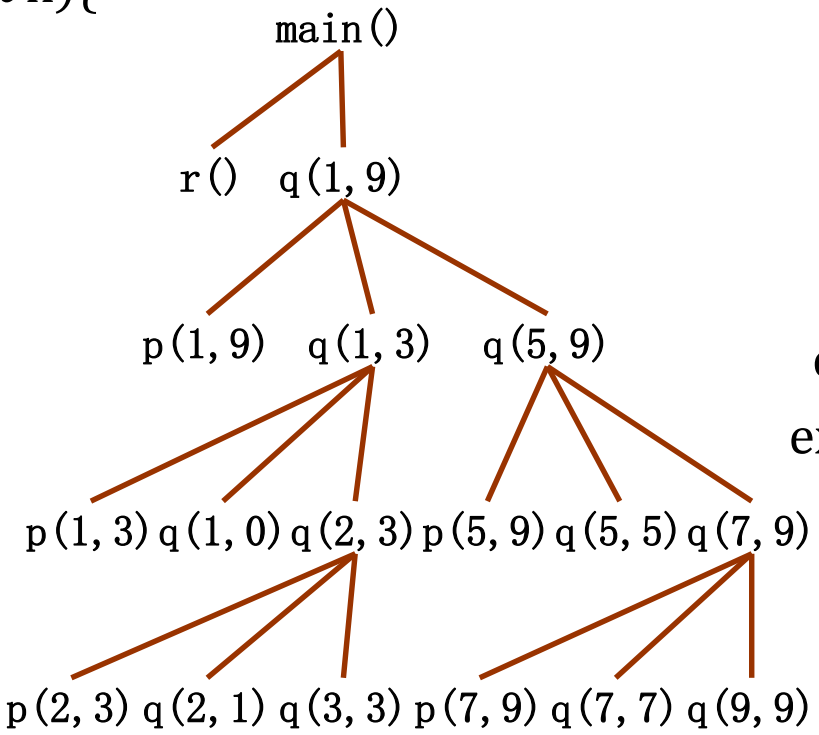



```
int a[11];
void readArray(){…
  int i;
  …}
int partition(int m, int n){
  …}
void quicksort(int m, int n){
  int i;
  if (n>m){
    i=partition(m, n);
    quicksort(m, i-1);
    quicksort(i+1, n);
  }}
main() {
  readArray();
  a[0]=-9999;
  a[10]=9999;
  quicksort(1,9);}
```

活动树的流程特点

```
int a[11];
void readArray(){...
  int i;
  ...}
int partition(int m, int n){
  ...}
void quicksort(int m, int n){
  int i;
  if (n>m){
    i=partition(m, n);
    quicksort(m, i-1);
    quicksort(i+1, n)
  }}
main() {
  readArray();
  a[0]=-9999;
  a[10]=9999;
  quicksort(1,9)}
```

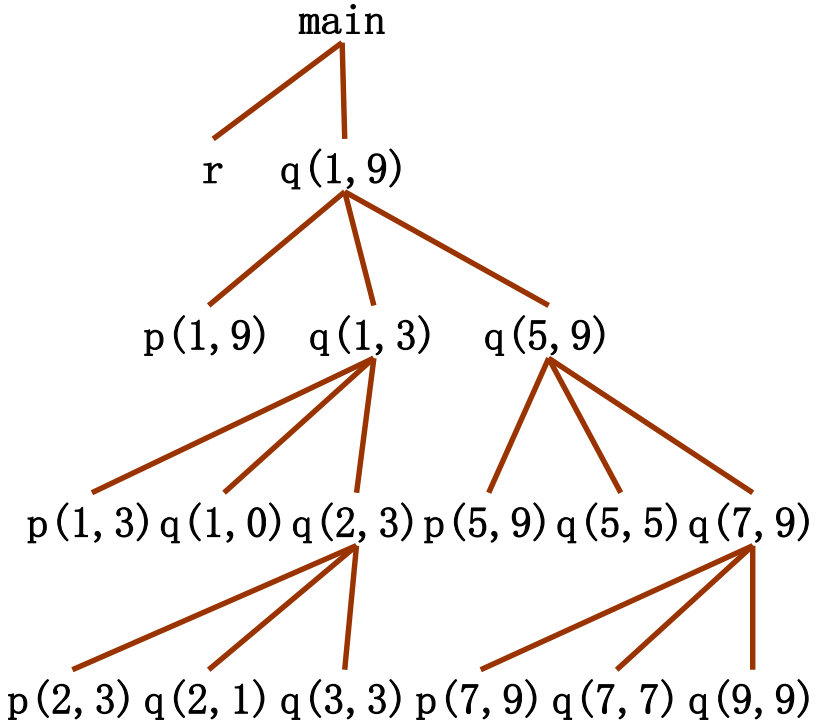
- 过程调用次序与先根遍历活动树一致；
- 过程返回次序与后根遍历活动树一致；




```
enter main()
 enter readArray()
 exit readArray()
 enter quicksort(1,9)
  enter partition(1,9)
  exit partition(1,9)
  enter quicksort(1,3)
  ...
  exit quicksort(1,3)
  enter quicksort(5,9)
  ...
  exit quicksort(5,9)
 exit quicksort(1,9)
exit main()
```

活动树与程序行为的关系

- 假定当前活动对应为树中结点N，那么当前尚未结束的活动就是N及其祖先，这些活动之间的调用关系就是由根到N的路径上结点顺序，返回次序则相反。



- 得到以下结论：
  - 过程需要存储空间实现它的环境实例；
  - 过程被调时分配它的存储空间，返回时释放；
  - 栈式管理的栈帧满足活动树上过程的存储空间要求。

小结

- 过程p的活动是从调用p时开始，p返回时为止，这个时间段被称为该过程活动的生命期。
- 过程活动之间存在嵌套和并列关系，如果某活动A和B的生命期是嵌套的，那么过程主调的本次活动嵌套被调的本次活动。
- 如果过程是递归的，那么该过程的某次活动可能嵌套在上一次活动之内部。
- 对于并列的活动A和B而言，它们的生命期没有重叠，故在时间上不会同时活着。

栈快照

- 例：过程q的一个环境实例，参数区两个整型变量m和n分别对应实参1和9。局部区中只有一个局部变量i，为整型。图中没有表明链接关系，事实上包括主调（控制链）以及有关名字引用（访问链）和控制流转移（返回地址）方面的链接关系。略去断点保护。
- 例：q(1,9)@frame:(<参数区>:(m:1 n:9) <链接区>:(<访问链><控制链><返址>) <局部区>:(i))
- 例：main()@frame:(<参数区>:NIL <链接区>:(<访问链><控制链><返址>) <局部区>:NIL)

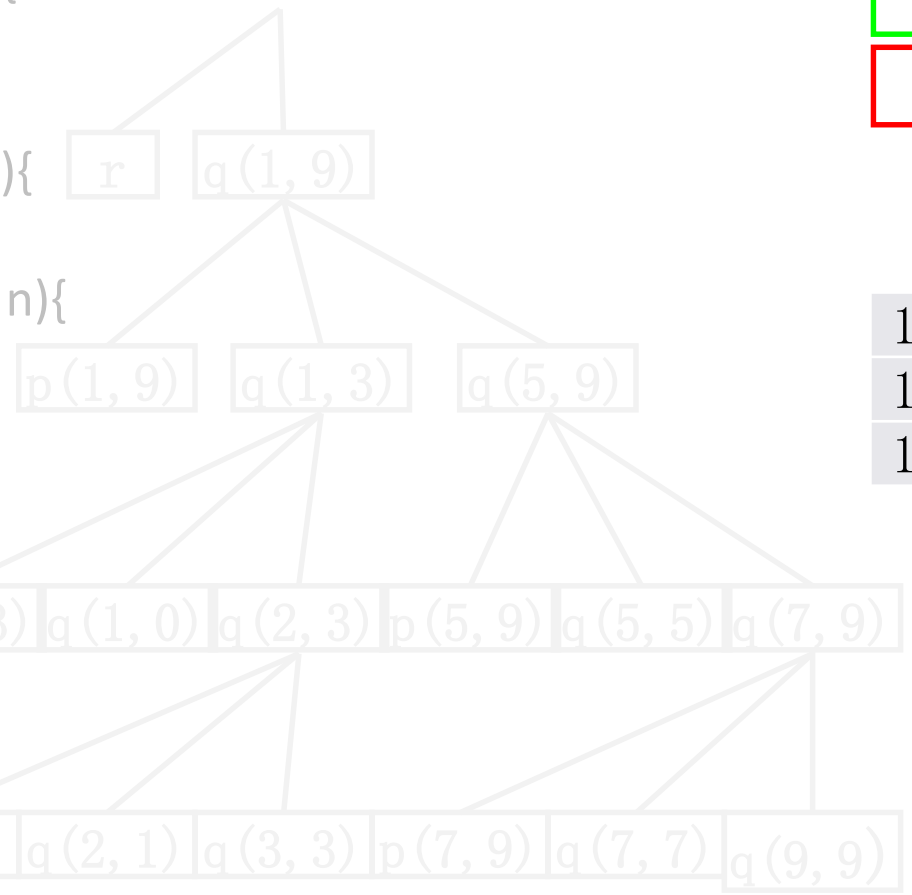
103	〈访问链〉
102	〈控制链〉
101	〈返址〉

100	〈参数2〉:9
99	〈参数1〉:1
98	〈访问链〉
97	〈控制链〉
96	〈返址〉
95	m
94	n
93	i

栈快照

```
int a[11];
void readArray(){…
  int i;
  …}
int p(int m, int n){
  …}
void q(int m, int n){
  int i;
  if (n>m){
    i=p(m, n);
    q(m, i-1);
    q(i+1, n);
  }}
main() {
  r();
  a[0]=-9999;
  a[10]=9999;
  q(1,9);}
```

main



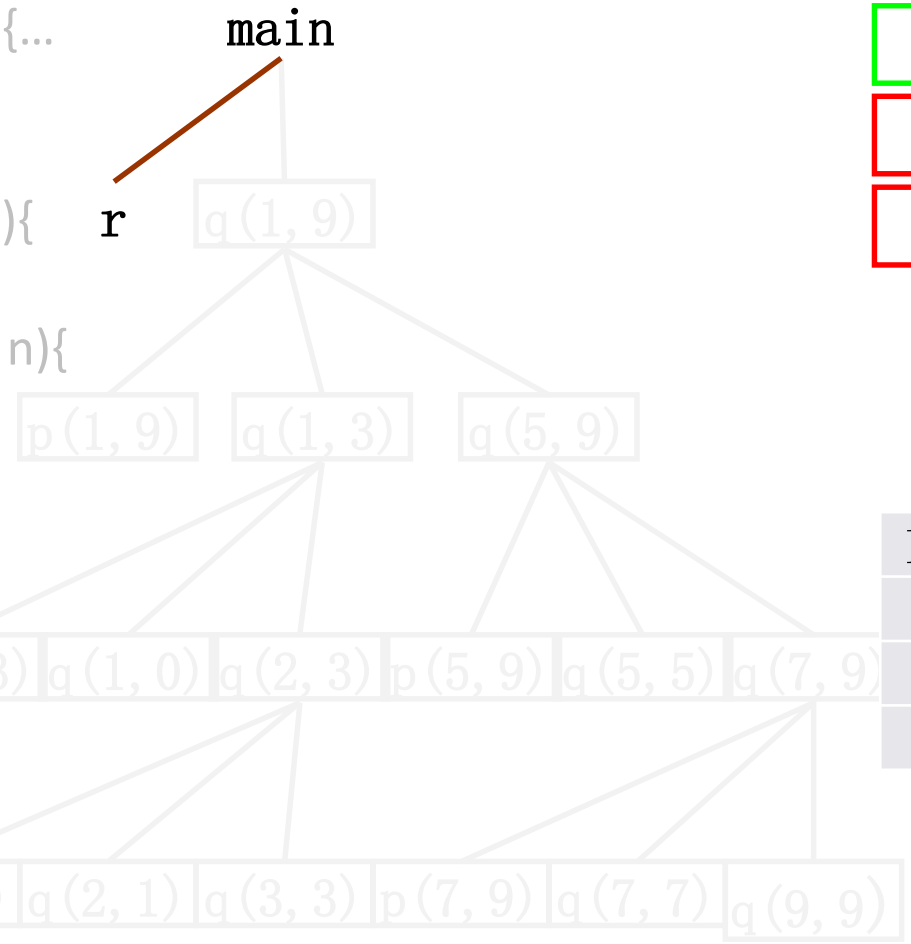
NULL;int a[11]

main()@frame

103	〈访问链〉
102	〈控制链〉
101	〈返址〉

栈快照

```
int a[11];
void readArray(){…
  int i;
  …}
int p(int m, int n){
  …}
void q(int m, int n){
  int i;
  if (n>m){
    i=p(m, n);
    q(m, i-1);
    q(i+1, n);
  }}
main() {
  r();
  a[0]=-9999;
  a[10]=9999;
  q(1,9);}
```



NULL;int a[11]

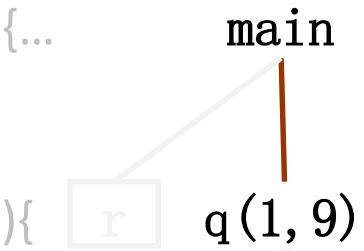
main()@frame

r()@frame

100	〈访问链〉
99	〈控制链〉
98	〈返址〉
97	i

栈快照

```
int a[11];
void readArray(){…
  int i;
  …}
int p(int m, int n){
  …}
void q(int m, int n){
  int i;
  if (n>m){
    i=p(m, n);
    q(m, i-1);
    q(i+1, n);
  }}
main() {
  r();
  a[0]=-9999;
  a[10]=9999;
  q(1,9);}
```



NULL;int a[11]

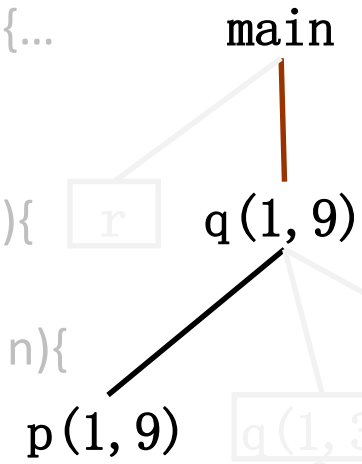
main()@frame

q(1,9)@frame

100	〈参数2〉:9
99	〈参数1〉:1
98	〈访问链〉
97	〈控制链〉
96	〈返址〉
95	m
94	n
93	i

栈快照

```
int a[11];
void readArray(){…
  int i;
  …}
int p(int m, int n){
  …}
void q(int m, int n){
  int i;
  if (n>m){
    i=p(m, n);
    q(m, i-1);
    q(i+1, n);
  }}
main() {
  r();
  a[0]=-9999;
  a[10]=9999;
  q(1,9);}
```



NULL;int a[11]

main()@frame

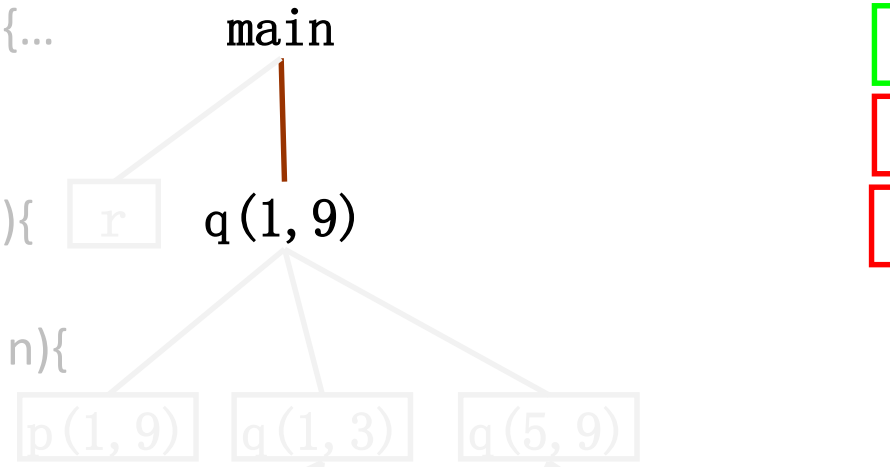
q(1,9)@frame

p(1,9)@frame

92	〈参数2〉:9
91	〈参数1〉:1
90	〈访问链〉
89	〈控制链〉
88	〈返址〉
87	m
86	n

栈快照

```
int a[11];
void readArray(){…
  int i;
  …}
int p(int m, int n){
  …}
void q(int m, int n){
  int i;
  if (n>m){
    i=p(m, n);
    q(m, i-1);
    q(i+1, n);
  }}
main()@frame() {
  r();
  a[0]=-9999;
  a[10]=9999;
  q(1,9);}
```



NULL;int a[11]

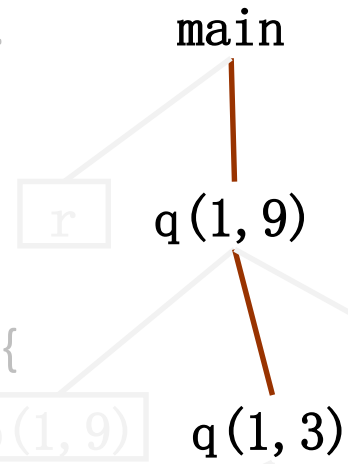
main()@frame

q(1,9)@frame

100	〈参数2〉:9
99	〈参数1〉:1
98	〈访问链〉
97	〈控制链〉
96	〈返址〉
95	m
94	n
93	i

栈快照

```
int a[11];
void readArray(){…
  int i;
  …}
int p(int m, int n){
  …}
void q(int m, int n){
  int i;
  if (n>m){
    i=p(m, n);
    q(m, i-1);
    q(i+1, n);
  }}
main() {
  r();
  a[0]=-9999;
  a[10]=9999;
  q(1,9);}
```



NULL;int a[11]

main()@frame

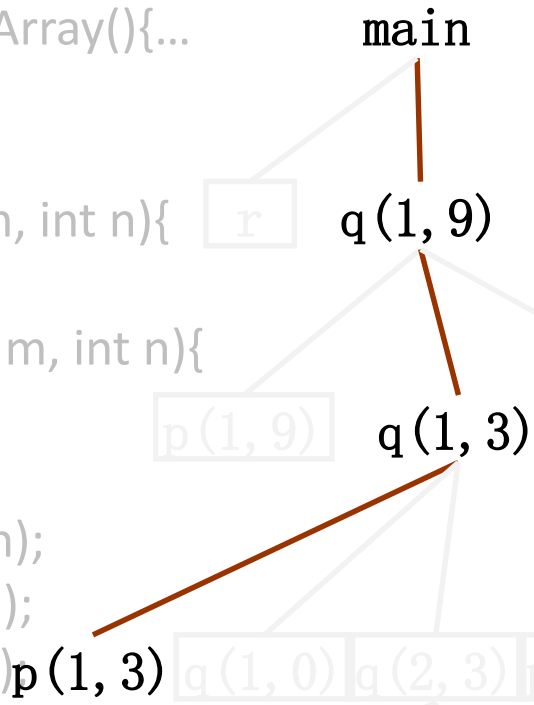
q(1,9)@frame

q(1,3)@frame

92	〈参数2〉:3
91	〈参数1〉:1
90	〈访问链〉
89	〈控制链〉
88	〈返址〉
87	m
86	n
85	i

栈快照

```
int a[11];
void readArray(){…
  int i;
  …}
int p(int m, int n){
  …}
void q(int m, int n){
  int i;
  if (n>m){
    i=p(m, n);
    q(m, i-1);
    q(i+1, n);
  }}
main() {
  r();
  a[0]=-9999;
  a[10]=9999;
  q(1,9);}
```



NULL;int a[11]

main()@frame

q(1,9)@frame

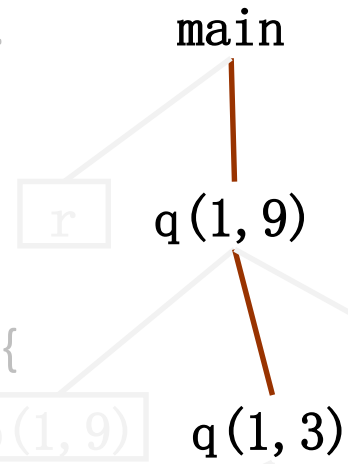
q(1,3)@frame

p(1,3)@frame

84	〈参数2〉:3
83	〈参数1〉:1
82	〈访问链〉
81	〈控制链〉
80	〈返址〉
79	m
78	n

栈快照

```
int a[11];
void readArray(){…
  int i;
  …}
int p(int m, int n){
  …}
void q(int m, int n){
  int i;
  if (n>m){
    i=p(m, n);
    q(m, i-1);
    q(i+1, n);
  }}
main() {
  r();
  a[0]=-9999;
  a[10]=9999;
  q(1,9);}
```



NULL;int a[11]

main()@frame

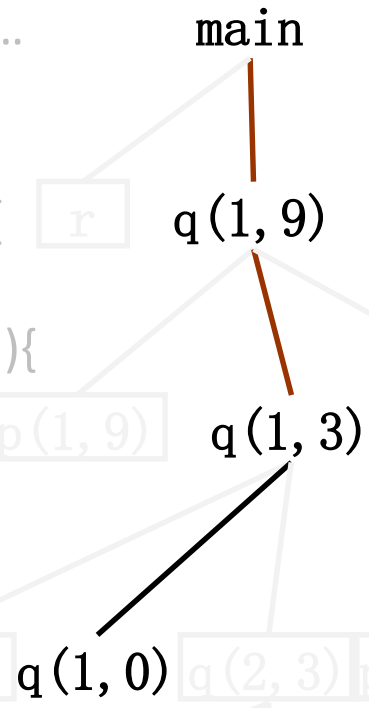
q(1,9)@frame

q(1,3)@frame

92	〈参数2〉:3
91	〈参数1〉:1
90	〈访问链〉
89	〈控制链〉
88	〈返址〉
87	m
86	n
85	i

栈快照

```
int a[11];
void readArray(){...
  int i;
  ...}
int p(int m, int n){
  ...}
void q(int m, int n){
  int i;
  if (n>m){
    i=p(m, n);
    q(m, i-1);
    q(i+1, n);
  }}
main() {
  r();
  a[0]=-9999;
  a[10]=9999;
  q(1,9);}
```



NULL;int a[11]

main()@frame

q(1,9)@frame

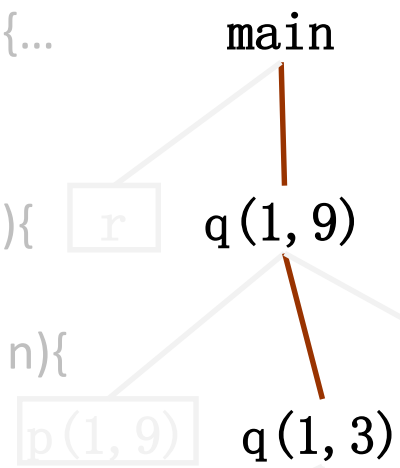
q(1,3)@frame

q(1,0)@frame

84	〈参数2〉:0
83	〈参数1〉:1
82	〈访问链〉
81	〈控制链〉
80	〈返址〉
79	m
78	n
77	i

栈快照

```
int a[11];
void readArray(){…
  int i;
  …}
int p(int m, int n){
  …}
void q(int m, int n){
  int i;
  if (n>m){
    i=p(m, n);
    q(m, i-1);
    q(i+1, n);
  }}
main() {
  r();
  a[0]=-9999;
  a[10]=9999;
  q(1,9);}
```



NULL;int a[11]

main()@frame

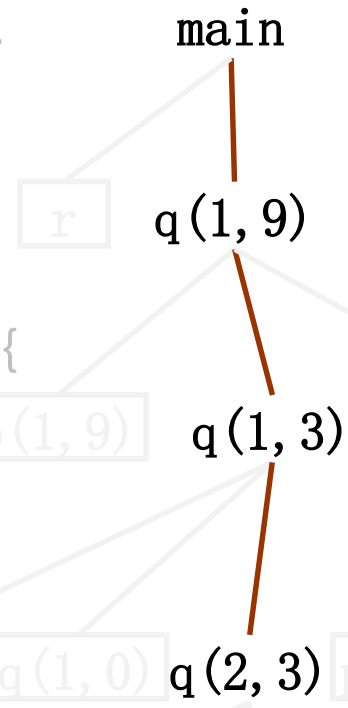
q(1,9)@frame

q(1,3)@frame

92	〈参数2〉:3
91	〈参数1〉:1
90	〈访问链〉
89	〈控制链〉
88	〈返址〉
87	m
86	n
85	i

栈快照

```
int a[11];
void readArray(){…
  int i;
  …}
int p(int m, int n){
  …}
void q(int m, int n){
  int i;
  if (n>m){
    i=p(m, n);
    q(m, i-1);
    q(i+1, n);
  }}
main() {
  r();
  a[0]=-9999;
  a[10]=9999;
  q(1,9);}
```



NULL;int a[11]

main()@frame

q(1,9)@frame

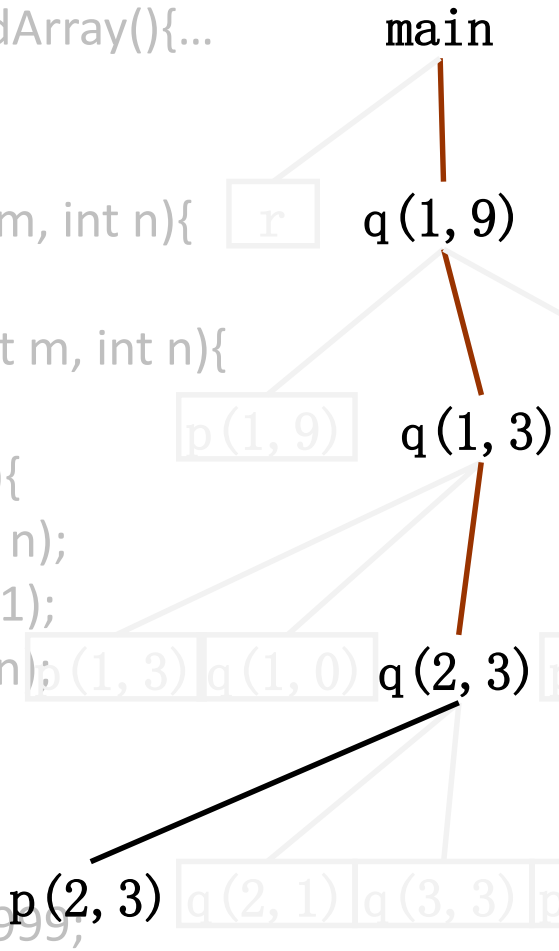
q(1,3)@frame

q(2,3)@frame

84	〈参数2〉:3
83	〈参数1〉:2
82	〈访问链〉
81	〈控制链〉
80	〈返址〉
79	m
78	n
77	i

栈快照

```
int a[11];
void readArray(){…
  int i;
  …}
int p(int m, int n){
  …}
void q(int m, int n){
  int i;
  if (n>m){
    i=p(m, n);
    q(m, i-1);
    q(i+1, n);
  }}
main() {
  r();
  a[0]=-9999;
  a[10]=9999;
  q(1,9);}
```



NULL;int a[11]

main

q(1,9)@frame

q(1,3)@frame

q(2,3)@frame

p(2,3)@frame

76	〈参数2〉:3
75	〈参数1〉:2
74	〈访问链〉
73	〈控制链〉
72	〈返址〉
71	m
70	n

栈快照细节

103	〈访问链〉
102	〈控制链〉:0
101	〈返址〉
100	〈参数2〉:9
99	〈参数1〉:1
98	〈访问链〉
97	〈控制链〉:102
96	〈返址〉
95	m
94	n
93	i
92	〈参数2〉:3
91	〈参数1〉:1
90	〈访问链〉
89	〈控制链〉:97
88	〈返址〉
87	m
86	n
85	i

84	〈参数2〉:3	
83	〈参数1〉:2	
82	〈访问链〉	
81	〈控制链〉:89	← fp
80	〈返址〉	
79	m	
78	n	
77	i	← Sp
76	〈参数2〉:3	
75	〈参数1〉:2	
74	〈访问链〉	
73	〈控制链〉:81	← fp
72	〈返址〉	
71	m	
70	n	← sp

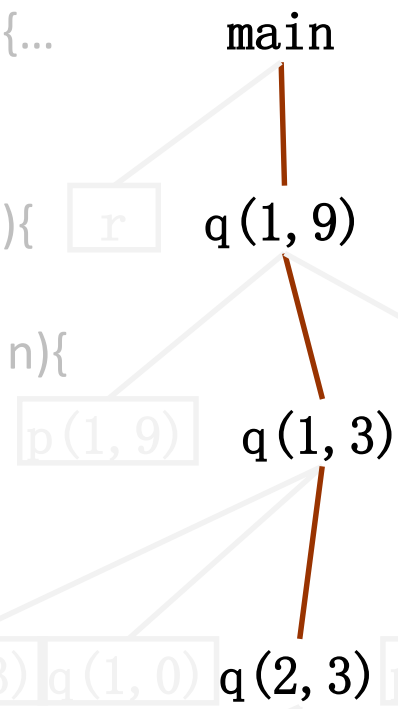
栈顶栈帧为当前栈帧，由此可知：fp=73（指向该栈帧）且sp=70（指向栈顶单元而且是当前栈帧的地址最小单元）

指向某个栈帧意味着指针值为这个栈帧的控制链单元地址。

被调过程的控制链指向主调过程的栈帧。

栈快照

```
int a[11];
void readArray(){…
  int i;
  …}
int p(int m, int n){
  …}
void q(int m, int n){
  int i;
  if (n>m){
    i=p(m, n);
    q(m, i-1);
    q(i+1, n);
  }}
main() {
  r();
  a[0]=-9999;
  a[10]=9999;
  q(1,9);}
```



NULL;int a[11]

main

q(1,9)@frame

q(1,3)@frame

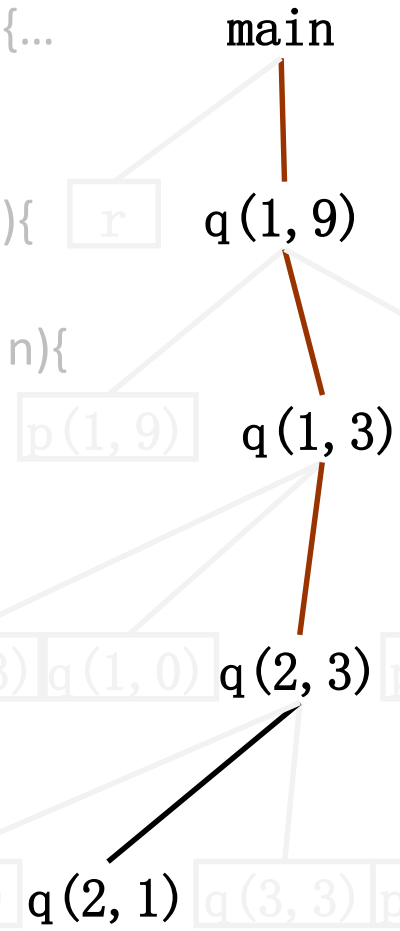
q(2,3)@frame

fp=81

sp=77

栈快照

```
int a[11];
void readArray(){...
  int i;
  ...}
int p(int m, int n){
  ...}
void q(int m, int n){
  int i;
  if (n>m){
    i=p(m, n);
    q(m, i-1);
    q(i+1, n);
  }}
main() {
  r();
  a[0]=-9999;
  a[10]=9999;
  q(1,9);}
```



NULL;int a[11]

main()@frame

q(1,9)@frame

q(1,3)@frame

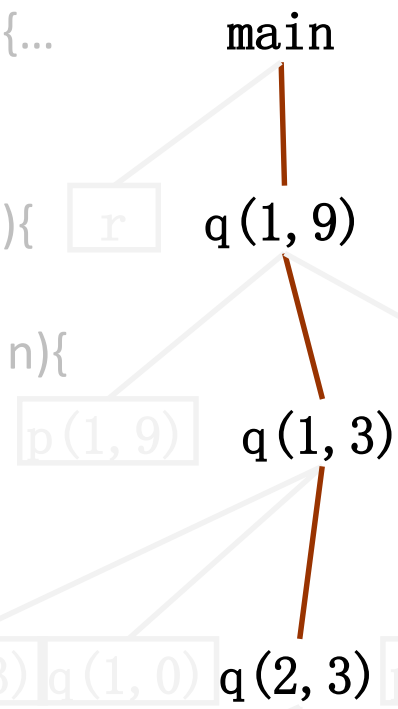
q(2,3)@frame

q(2,1)@frame

76	〈参数2〉:3
75	〈参数1〉:2
74	〈访问链〉
73	〈控制链〉:81
72	〈返址〉
71	m
70	n
69	i

栈快照

```
int a[11];
void readArray(){…
  int i;
  …}
int p(int m, int n){
  …}
void q(int m, int n){
  int i;
  if (n>m){
    i=p(m, n);
    q(m, i-1);
    q(i+1, n);
  }}
main() {
  r();
  a[0]=-9999;
  a[10]=9999;
  q(1,9);}
```



NULL;int a[11]

main()@frame

q(1,9)@frame

q(1,3)@frame

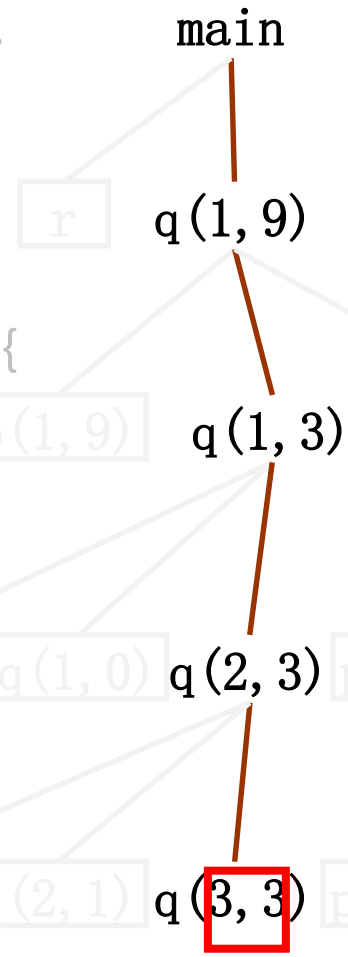
q(2,3)@frame

fp=81

sp=77

栈快照

```
int a[11];
void readArray(){...
  int i;
  ...}
int p(int m, int n){
  ...}
void q(int m, int n){
  int i;
  if (n>m){
    i=p(m, n);
    q(m, i-1);
    q(i+1, n);
  }}
main() {
  r();
  a[0]=-9999;
  a[10]=9999;
  q(1,9);}
```



NULL;int a[11]

main()@frame

q(1,9)@frame

q(1,3)@frame

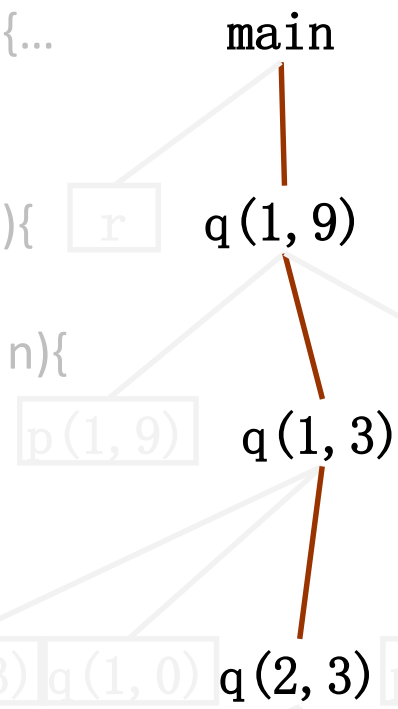
q(2,3)@frame

q(3,3)@frame

76	〈参数2〉:3
75	〈参数1〉:3
74	〈访问链〉
73	〈控制链〉:81
72	〈返址〉
71	m
70	n
69	i

栈快照

```
int a[11];
void readArray(){…
  int i;
  …}
int p(int m, int n){
  …}
void q(int m, int n){
  int i;
  if (n>m){
    i=p(m, n);
    q(m, i-1);
    q(i+1, n);
  }}
main() {
  r();
  a[0]=-9999;
  a[10]=9999;
  q(1,9);}
```



NULL;int a[11]

main()@frame

q(1,9)@frame

q(1,3)@frame

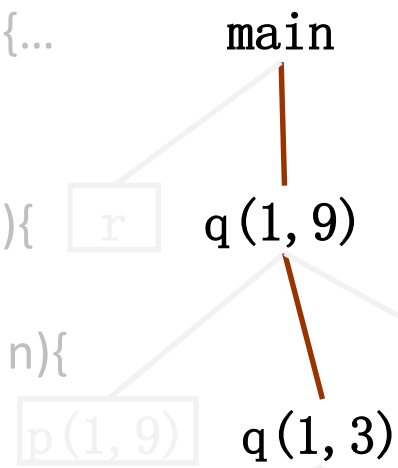
q(2,3)@frame

fp=81

sp=77

栈快照

```
int a[11];
void readArray(){…
  int i;
  …}
int p(int m, int n){
  …}
void q(int m, int n){
  int i;
  if (n>m){
    i=p(m, n);
    q(m, i-1);
    q(i+1, n);
  }}
main() {
  r();
  a[0]=-9999;
  a[10]=9999;
  q(1,9);}
```



NULL;int a[11]

main()@frame

q(1,9)@frame

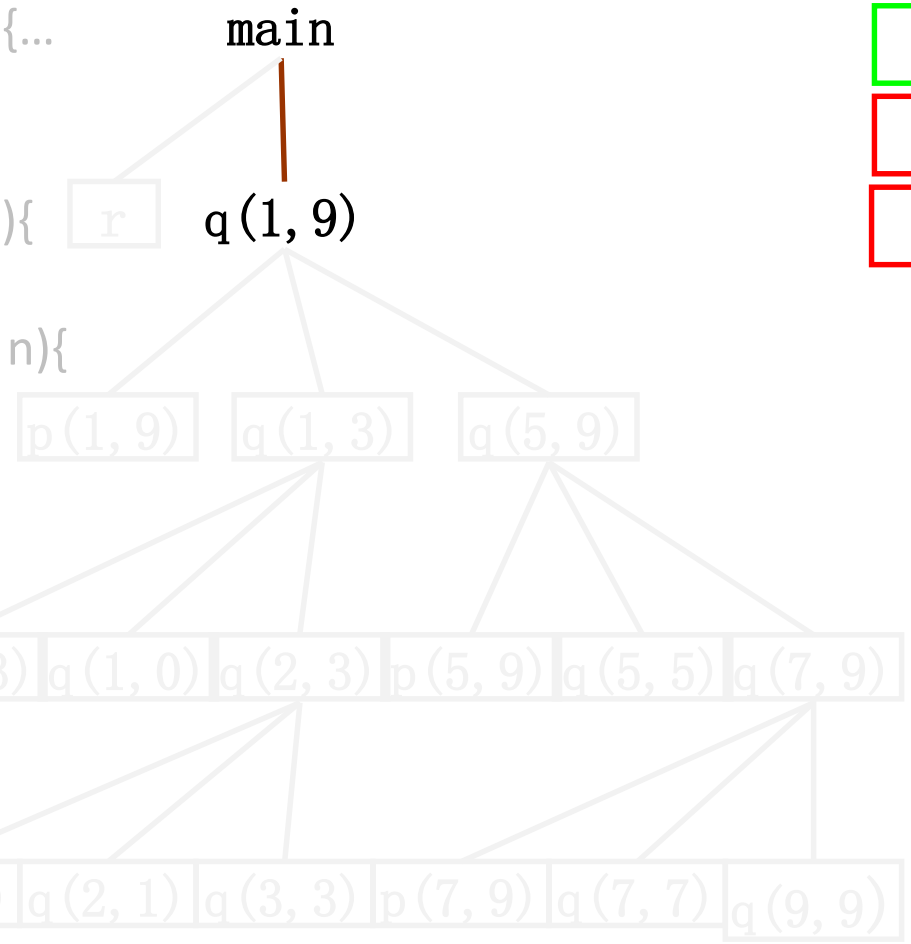
q(1,3)@frame

fp=89

sp=85

栈快照

```
int a[11];
void readArray(){…
  int i;
  …}
int p(int m, int n){
  …}
void q(int m, int n){
  int i;
  if (n>m){
    i=p(m, n);
    q(m, i-1);
    q(i+1, n);
  }}
main() {
  r();
  a[0]=-9999;
  a[10]=9999;
  q(1,9);}
```



NULL;int a[11]

main()@frame

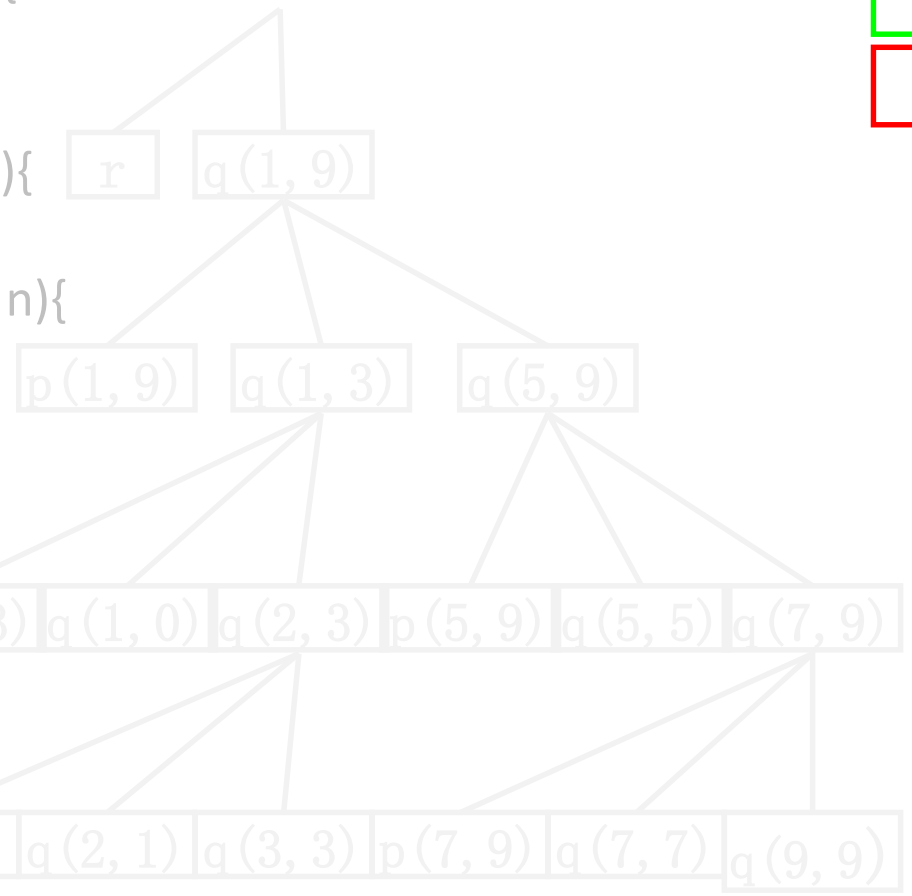
q(1,9)@frame

fp=97

sp=93

栈快照

```
int a[11];
void readArray(){…
  int i;
  …}
int p(int m, int n){
  …}
void q(int m, int n){
  int i;
  if (n>m){
    i=p(m, n);
    q(m, i-1);
    q(i+1, n);
  }}
main() {
  r();
  a[0]=-9999;
  a[10]=9999;
  q(1,9);}
```



NULL;int a[11]

main()@frame

fp=102

sp=101

观 察

- 活动记录是栈上连续的区域，对应于过程一次调用
- 过程被调用时建立，过程返回时释放
- 活动记录中为过程的参数和局部变量提供存储单元
- 当前执行哪个过程，它的活动记录就在栈顶位置

- 主调过程的活动记录与被调过程的活动记录是相邻的
- 主调应该创建被调活动记录
- 从被调活动记录能知道主调活动记录，以便返回时完成被调过程活动记录的释放，同时也能将控制流返回到主调。

- 对过程调用深度没有限制。
- 采用健值对方式描述栈帧单元，未给出值的是因为值待确定，而最终都有值，反映程序执行轨迹。

例：Fact程序运行时环境（栈）

```
int x;
int fact(int n; int a;){
    if (n==1) return a
    else return fact (n-1, n*a,)
};
x=123+fact(3,1,);
print x
```

```
@code=[
    t4=123; t5=3; t6=1;
    PAR t6; PAR t5;
    t7=CALL fact, 2;
    x=t4+t7;
    PRINT x]
```

```
fact@code=[
    IF n==1 THEN l1 ELSE l2;
    LABEL l1; RETURN a;
    GOTO l3;
    LABEL l2;
    t1=n-1; t2=n*a;
    PAR t2; PAR t1;
    t3=CALL fact, 2;
    RETURN t3;
    LABEL l3]
```

Fact程序的符号表

- @table:(outer:NIL width: argc:0 arglist:NIL rtype:VOID level:0 code:@code
  entry:(name:x type:INT offset:4)
  entry:(name:fact type FUNC offset:12 mytab:fact@table)
  entry:(name:t4 type: TEMP offset:16)
  entry:(name:t5 type: TEMP offset:20)
  entry:(name:t6 type: TEMP offset:24)
  entry:(name:t7 type: TEMP offset:28))
- fact@table:(outer:@table width: 20
  argc:2 arglist:(n a) rtype:INT level:1 code:fact@code
  entry:(name:n type:INT offset:4)
  entry:(name:a type: INT offset:8)
  entry:(name:t1 type: TEMP offset:12)
  entry:(name:t2 type: TEMP offset:16)
  entry:(name:t3 type: TEMP offset:20))

栈快照

ε@frame

```
int x;
int fact(int n; int a;){
    if (n==1) return a
    else return fact (n-1, n*a,)
};
x=123+fact(3,1,);
print x
```

@code=[t4=123;
t5=3;t6=1;PAR t6;PAR t5;
t7=CALL fact, 2;x=t4+t7;
PRINT x]

fact@code=[IF n==1
THEN l1 ELSE l2;LABEL l1;
RETURN a;GOTO l3;
LABEL l2;t1=n-1; t2=n*a;
PAR t2; PAR t1;
t3=CALL fact, 2;
RETURN t3;LABEL l3]

ε()

100	<访问链>:0
99	<控制链>:0
98	<返址>
97	x
96	fact[1]
95	fact[0]
94	t4:123
93	t5:3
92	t6:1
91	t7

用途是：存放返回时下一层返回给它的值（自底向上的回溯过程使用）

层次链接

无名栈帧对于程序的一次运行只有一个。可由编译静态创建，换句话说作为程序初始化之一。

宿主为无名函数的名字声明都是全局的，可分配在静态区。是一个意思。

<访问链>指向词法上嵌套外层的最新活动记录。

设f()声明是g()声明的直接外层，即f@table中有该g名字的登记项。

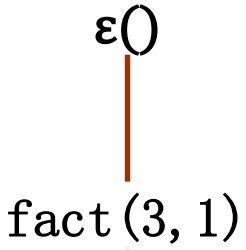
g()的一个活动记录g()@frame的访问链单元指向最新f()@frame。

所谓最新是指从g()@frame出发沿着控制链首次遇到的那个f()@frame

栈快照

ε@frame

fact(3,1)@frame



100	〈访问链〉:0
99	〈控制链〉:0
98	〈返址〉
97	x
96	fact[1]
95	fact[0]
94	t4:123
93	t5:3
92	t6:1
91	t7
90	〈参数2〉:1
89	〈参数1〉:3
88	〈访问链〉:99
87	〈控制链〉:99
86	〈返址〉
85	n:3
84	a:1
83	t1:2
82	t2:3
81	t3

```
int x;
int fact(int n; int a;){
    if (n==1) return a
    else return fact (n-1, n*a,)
};
x=123+fact(3,1,);
print x
```

主调ε()负责创建被调fact(3,1)的栈帧
所以被调的控制链总是指向主调的栈帧。
对应代码
[PAR t6; PAR t5;
t7=CALL fact, 2]

@code=[t4=123;
t5=3;t6=1;PAR t6;PAR t5;
t7=CALL fact, 2;x=t4+t7;
PRINT x]

fact@code=[IF n==1
THEN l1 ELSE l2;LABEL l1;
RETURN a;GOTO l3;
LABEL l2;t1=n-1; t2=n*a;
PAR t2; PAR t1;
t3=CALL fact, 2;
RETURN t3;LABEL l3]

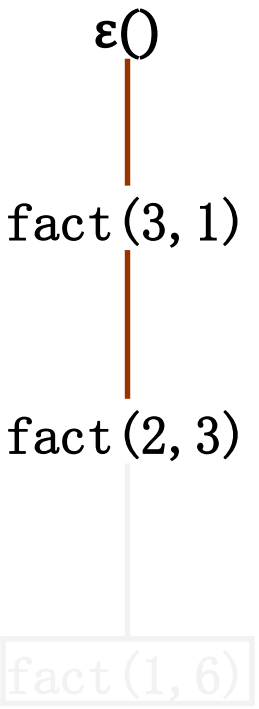
```
int fact (n : 3; a : 1)
{

}
```

栈快照

ε@frame

fact(3,1)@frame

fact(2,3)@frame



@code=[t4=123;
t5=3;t6=1;PAR t6;PAR t5;
t7=CALL fact, 2;x=t4+t7;
PRINT x]

fact@code=[IF n==1
THEN l1 ELSE l2;LABEL l1;
RETURN a;GOTO l3;
LABEL l2;t1=n-1; t2=n*a;
PAR t2; PAR t1;
t3=CALL fact, 2;
RETURN t3;LABEL l3]

```
int x;
int fact(int n; int a;){
   if (n==1) return a
   else return fact (n-1, n*a,)
};
x=123+fact(3,1,);
print x
```

100	〈访问链〉:0
99	〈控制链〉:0
98	〈返址〉
97	x
96	fact[1]
95	fact[0]
94	t4:123
93	t5:3
92	t6:1
91	t7
90	〈参数2〉:1
89	〈参数1〉:3
88	〈访问链〉:99
87	〈控制链〉:99
86	〈返址〉
85	n:3
84	a:1
83	t1:2
82	t2:3
81	t3

Fact函数的定义是在全局之下，因此visit指向99

但是fact(2,3)由fact(3,1)调用

故控制链指向87

80	〈参数2〉:3
79	〈参数1〉:2
78	〈访问链〉:99
77	〈控制链〉:87
76	〈返址〉
75	n:2
74	a:3
73	t1:1
72	t2:6
71	t3

主调fact(3,1)创建被调fact(2,3)的栈帧是在执行代码**[PAR t2; PAR t1; t3=CALL fact, 2]**时完成的。

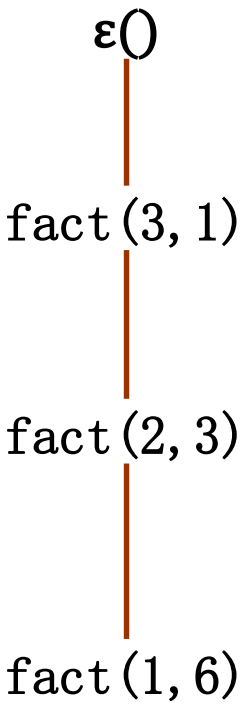
栈快照

ε@frame

fact(3,1)@frame

fact(2,3)@frame

fact(1,6)@frame



@code=[t4=123;
t5=3;t6=1;PAR t6;PAR t5;
t7=CALL fact, 2;x=t4+t7;
PRINT x]

fact@code=[IF n==1
THEN l1 ELSE l2;LABEL l1;
RETURN a;GOTO l3;
LABEL l2;t1=n-1; t2=
PAR t2; PAR t1;
t3=CALL fact, 2;
RETURN t3;LABEL l3]

```
int x;
int fact(int n; int a;){
    if (n==1) return a
    else return fact (n-1, n*a,)
};
x=123+fact(3,1,);
print x
```

100	<访问链>:0
99	<控制链>:0
98	<返址>
97	x
96	fact[1]
95	fact[0]
94	t4:123
93	t5:3
92	t6:1
91	t7
90	<参数2>:1
89	<参数1>:3
88	<访问链>:99
87	<控制链>:99
86	<返址>
85	n:3
84	a:1
83	t1:2
82	t2:3
81	t3

此时n==1，该考虑返回了！

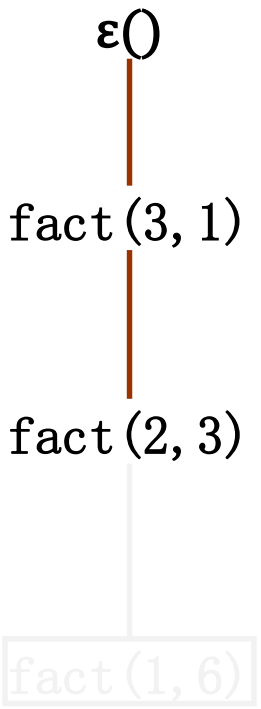
80	<参数2>:3
79	<参数1>:2
78	<访问链>:99
77	<控制链>:87
76	<返址>
75	n:2
74	a:3
73	t1:1
72	t2:6
71	t3
70	<参数2>:6
69	<参数1>:1
68	<访问链>:99
67	<控制链>:77
66	<返址>
65	n:1
64	a:6
63	t1
62	t2
61	t3

栈快照

ε@frame

fact(3,1)@frame

fact(2,3)@frame



@code=[t4=123;
t5=3;t6=1;PAR t6;PAR t5;
t7=CALL fact, 2;x=t4+t7;
PRINT x]

fact@code=[IF n==1
THEN l1 ELSE l2;LABEL l1;
RETURN a;GOTO l3;
LABEL l2;t1=n-1; t2=n*a;
PAR t2; PAR t1;
t3=CALL fact, 2;
RETURN t3;LABEL l3]

100	〈访问链〉:0
99	〈控制链〉:0
98	〈返址〉
97	x
96	fact[1]
95	fact[0]
94	t4:123
93	t5:3
92	t6:1
91	t7
90	〈参数2〉:1
89	〈参数1〉:3
88	〈访问链〉:99
87	〈控制链〉:99
86	〈返址〉
85	n:3
84	a:1
83	t1:2
82	t2:3
81	t3

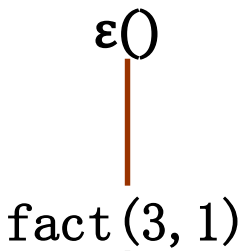
80	〈参数2〉:3
79	〈参数1〉:2
78	〈访问链〉:99
77	〈控制链〉:87
76	〈返址〉
75	n:2
74	a:3
73	t1:1
72	t2:6
71	t3:6

被调fact(1,6)执行[RETURN a]返回到主调fact(2,3)中，并得到返回值1。6
主调将其保存在t3中。被调在返回前释放了栈帧。

栈快照

ε@frame

fact(3,1)@frame



@code=[t4=123;
t5=3;t6=1;PAR t6;PAR t5;
t7=CALL fact, 2;x=t4+t7;
PRINT x]

fact@code=[IF n==1
THEN l1 ELSE l2;LABEL l1;
RETURN a;GOTO l3;
LABEL l2;t1=n-1; t2=n*a;
PAR t2; PAR t1;
t3=CALL fact, 2;
RETURN t3;LABEL l3]

100	〈访问链〉:0
99	〈控制链〉:0
98	〈返址〉
97	x
96	fact[1]
95	fact[0]
94	t4:123
93	t5:3
92	t6:1
91	t7
90	〈参数2〉:1
89	〈参数1〉:3
88	〈访问链〉:99
87	〈控制链〉:99
86	〈返址〉
85	n:3
84	a:1
83	t1:2
82	t2:3
81	t3:6

被调fact(2,3)执行[RETURN t3]返回到主调fact(3,1)中，返回值为2。同时负责释放了自己的栈帧。

栈快照

ε@frame

ε()

100	<访问链>:0
99	<控制链>:0
98	<返址>
97	x:6
96	fact[1]
95	fact[0]
94	t4:123
93	t5:3
92	t6:1
91	t7:6

fact(3, 1)

fact(2, 3)

fact(1, 6)

@code=[t4=123;
t5=3;t6=1;PAR t6;PAR t5;
t7=CALL fact, 2;x=t4+t7;
PRINT x]

fact@code=[IF n==1
THEN l1 ELSE l2;LABEL l1;
RETURN a;GOTO l3;
LABEL l2;t1=n-1; t2=n*a;
PAR t2; PAR t1;
t3=CALL fact, 2;
RETURN t3;LABEL l3]

被调fact(3,1)执行[RETURN t3]返回到主调ε() 中，返回值为6。同时负责释放了自己的栈帧。

然后主调将6赋给t7，继续直到执行打印指令后程序结束。

隐含着在@code结束处有一个stop指令

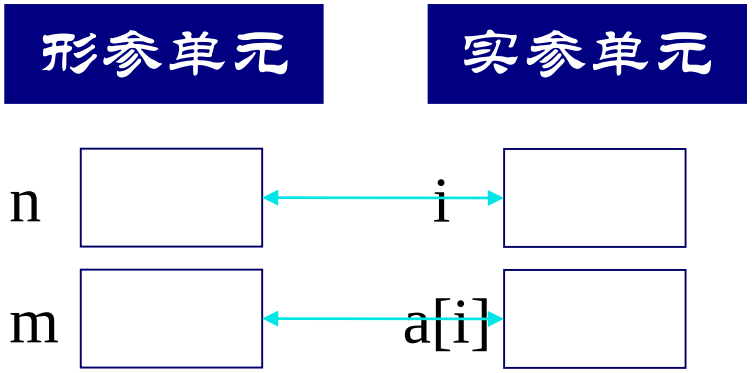
9.2 参数传递

- 统一的形参单元，一般设计为一个指针大小，还应该也是一个寄存器尺寸，设为4。
- 参数顺序与参数单元地址增大的方向一致，即PAR指令为倒序排列。
- 不允许可变参数的函数，所以简化掉了参数计数单元。即CALL指令中的参数个数n对同一个函数是固定的。
- 按照参数传递机制确定形参单元的内容，比如值传递机制意味着实参的值被存入形参单元中（在构建栈帧参数区的时候实现）。
- 传递给形参的值超长是可能的，比如实参为数组、函数等的情形，对超长参数处理待后续讨论。

参数传递机制

形参类型	对应实参
简单变量	简单变量，如x
简单变量	临时变量，如t5
数组原型	数组名字，如a[]
函数原型	函数名字，如g()

- 在被调的形参与实参有对应情况下控制流进入被调过程。
- 参数传递机制实现此对应。这与实参单元和形参单元都有关，可有几种参数传递方式。

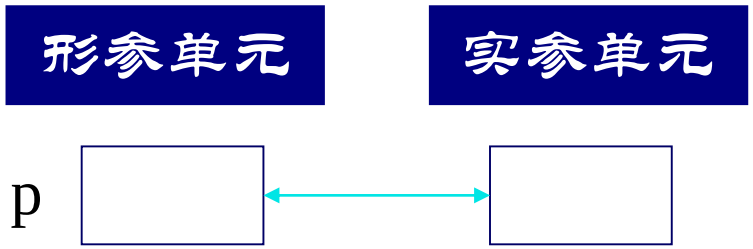


swap(i,a[i]);

```
procedure swap(n,m:real);
    var j:real;
    begin
        j:=n; n:=m; m:=j;
    end
```

参数传递机制

- 传地址(call-by-reference)：实参作为左值其地址放到形参单元中；被调过程中凡是引用形参p均解释为引用*p
- 传值(call-by-value)：实参值放到形参单元中
- 得结果(call-by-result)：调用时按照传值方式传递实参值到形参单元中，返回时要将形参单元内容拷贝到对应实参单元中。
- 传名(call-by-name)：类似于宏扩展，将实参看做字符串，替换掉被调过程中对应形参的每一次出现；执行被调过程时是执行替换后的代码。



例：参数传递

```
1  int i;
2  int A[3];
3  void Q(int B) {
4      A[1] = 3;
5      i = 2;
6      write(B);
7      B = 5;
8  }
9  int main() {
10     i = 1;
11     A[1] = 2;
12     A[2] = 4;
13     Q(A[i]);
14 }
```

传地址：实参A[1]，其值为2，形参B值为&A[1]，A[1]值修改为3，引用B解释为引用A[1]，故输出3，之后修改B解释为修改A[1]为5

传值：实参A[1]的值2，形参B的值为2，修改A[1]值为3，输出B值为2

得结果：实参A[1]的值2，形参B的值为2，修改A[1]值为3，输出B值为2，返回的时候B值等于5，需要考回给实参单元，故A[1]的值为5

传名：被调过程Q体中B的出现都要替换为A[i]，执行的是替换后的代码{A[1] = 3; i = 2; write(A[i]); A[i] = 5;}，所以输出A[2]值4，返回后A[1]值为3，A[2]值为5（注意其他方式中该值均为4）

参数传递机制的实现

53	A[2]	4					4					4			
52	A[1]	2	3				2	3	5			2	3	5	
51	A[0]														
50	i	1	2				1	2				1	2		
					2					3					2
200	<访问链>:0														
199	<控制链>:0														
198	<返址>														
197	<参数1>	2					52					2(52)			
196	<访问链>:0														
195	<控制链>:199														
194	<返址>														
193	B		2	5				52					2	5	
		main	Q		wr		mian	Q		wr		main	Q		wr
栈快照		传值					传地址					得结果			

参数传递机制的实现

- 在主调执行PAR指令序列时完成，就是将PAR x转换为一段指令完成如下功能：
- 查符号表，如果x的类型为INT、FLO、TEMP，那么传值[push x]，传地址[push &x]，得结果[push x; push &x]
- 查符号表如果x的类型为ARRAY，对应形参类型为ARRPTT，那么传地址[push &x]（这是传值机制的处理）。若若要真正的传值需要做较多工作：在局部区分配数组x大小区域，将实参数组元素都复制到该区域，首址push到形参单元中（适用于类型为ARRAY的形参，已超出了本课范围）。
- 查符号表如果实参x的类型为FUNC，那么传值[push &x'; x'=x]，其中x'是与实参x对应的形参，类型为FUNPTT；对于传地址机制实现为[push &x]
- 本课程后续仅针对传值进行讨论，可将其它机制作为思考。

例：形参类型为FUNPTT、ARRPTT

```
int z;
int a[10,20];
int bar(int x){
  return x++};
float foo(int x; int b[];
          int boo();){
  if (x >0) z=sqrt(b[0],)
  else return boo(z),};
print foo(0, a[],bar(),)
```

@table:
(outer:NULL width:820 argc:0 arglist:NIL rtype:INT
code:[t5=0; PAR t5; PAR a; PAR bar;t6=CALL foo@label, 3]
entry:(name:z type:INT offset:4)
entry:(name:a type:ARRAY base:804 etype:INT dims:2 dim[0]:10 dim[1]:20)
entry:(name:bar type:FUNC offset:812 mytab:bar@table)
entry:(name:foo type:FUNC offset:820 mytab:foo@table))

foo@table:(outer:@table width:16 argc:3
arglist:(x b boo) rtype:FLO
code:[IF x>0 THEN l1 ELSE l2;
LABEL l1; t1=0; t2=b[t1];PAR t2; t3=CALL sqrt, 1;
z=t3; GOTO l3;
LABEL l2;PAR z; t4=CALL boo, 1;RETURN t4;LABEL l3]
entry:(name:x type:INT offset:4)
entry:(name:b type:ARRPTT offset:8 etype:INT)
entry:(name:boo type:FUNPTT offset:16 rtype:INT))

bar@table:(outer:@table width:4
argc:1 arglist:(x) rtype:INT
code:[x=x+1; RETURN x]
entry(name:x type:INT offset:4)

程序执行起点为最外层函数代码

- 设程序中变量的未初始化的值都是0
- 设最外层过程的栈帧初始化在栈上
- 函数bar和foo的代码都在代码区，bar@label和foo@label分别是它们的首址
- t5和t6的值是执行过程代码时保存的
- 该快照是控制流到CALL指令时的

500	<访问链>:0
499	<控制链>:0
498	<返址>
497	z:0
496	a[9,19]:0
495	a[9,18]:0
…	…
297	a[0,0]:0
296	bar[1]:bar@label
295	bar[0]:_
294	foo[1]:foo@label
293	foo[0]:_
292	t5:0
291	t6:_

```
@table:
(outer:NULL width:820 argc:0 arglist:NIL rtype:INT
code:[t5=0; PAR t5; PAR a; PAR bar;t6=CALL foo@label, 3]
entry:(name:z type:INT offset:4)
entry:(name:a type:ARRAY base:804 etype:INT dims:2
 dim[0]:10 dim[1]:20)
entry:(name:bar type:FUNC offset:812 mytab:bar@table)
entry:(name:foo type:FUNC offset:820  mytab:foo@table))
```

主调ε()与被调foo

①数组参数仍然按照传地址方式

②[PAR bar]将fp值和bar@label（查@table得）分别传送到281和282中，并将<参数3>单元290置为指向boo[0]即281

③主调创建foo栈帧286至290其余由被调分配，约定sp=fp=286

④281中的499栈帧用于[CALL boo]指令创建bar栈帧的访问链（待后）

@code=[t5=0; PAR t5; PAR a; PAR bar; t6=CALL foo@label, 3]

290	〈参数3〉:281
289	〈参数2〉:297
288	〈参数1〉:0
287	〈访问链〉
286	〈控制链〉
285	〈返址〉
284	x:0
283	b:297
282	boo[1]:bar@label
281	boo[0]:499
280	t1:_
279	t2:_
278	t3:_
277	t4:_

500	〈访问链〉:0
499	〈控制链〉:0
498	〈返址〉
497	z:0
496	a[9,19]:0
495	a[9,18]:0
…	…
297	a[0,0]:0
296	bar[1]:bar@label
295	bar[0]:_
294	foo[1]:foo@label
293	foo[0]:_
292	t5:0
291	t6:_

foo@table:(outer:@table width:16 argc:3 arglist:(x b boo) rtype:FLO
code:[IF x>0 THEN l1 ELSE l2;LABEL l1; t1=0; t2=b[t1];PAR t2; t3=CALL sqrt, 1;
z=t3; GOTO l3; LABEL l2;PAR z; t4=CALL boo, 1;RETURN t4;LABEL l3]
entry:(name:x type:INT offset:4) entry:(name:b type:ARRPTT offset:8 etype:INT)
entry:(name:boo type:FUNPTT offset:16 rtype:INT))

主调foo与被调bar

- 在281找到对实参bar求值的环境499（最外层栈帧），构建被调bar的栈帧，其中会用到499栈帧构建访问链。
- 临时变量不会自动初始化
- <参数k>单元内容是主调构建的
- 局部区单元内容是初始化和被调的代码在运行中形成的。
- 与时间相关所以用快照表示。
  注意到277单元没有值，随后bar返回时就有值了。

[PAR z; t4=CALL boo, 1]

bar@table:(outer:@table width:4
argc:1 arglist:(x) rtype:INT
code:[x=x+1; RETURN x]
entry(name:x type:INT offset:4)

290	〈参数3〉:281
289	〈参数2〉:297
288	〈参数1〉:0
287	〈访问链〉
286	〈控制链〉
285	〈返址〉
284	x:0
283	b:297
282	boo[1]:bar@label
281	boo[0]:499
280	t1:_
279	t2:_
278	t3:_
277	t4:_
276	〈参数1〉:0
275	〈访问链〉
274	〈控制链〉
273	〈返址〉
272	x:1

被调bar返回到主调foo

- bar代码[RETURN x]将结果x值保存到约定寄存器$v0，并取出<返址>到$r，并释放栈帧后半部分（图中蓝底所示）最后通过[JR $r]将控制流回到foo中（红色CALL的下条指令）。
- bar栈帧剩下的部分274-276是由主调foo来释放的，因为这部分是它创建的。约定为sp=fp=274
- 红色CALL的下条指令负责将$v0传送到277单元中，即赋给t4

[PAR z; t4=CALL boo, 1]

bar@table:(outer:@table width:4
argc:1 arglist:(x) rtype:INT
code:[x=x+1; RETURN x]
entry(name:x type:INT offset:4)

290	<参数3>:281
289	<参数2>:297
288	<参数1>:0
287	<访问链>
286	<控制链>
285	<返址>
284	x:0
283	b:297
282	boo[1]:bar@label
281	boo[0]:499
280	t1:
279	t2:
278	t3:
277	t4:1
~~276~~	~~<参数1>:0~~
~~275~~	~~<访问链>~~
~~274~~	~~<控制链>~~
~~273~~	~~<返址>~~
~~272~~	~~x:1~~

被调foo返回到主调ε()

foo代码[RETURN t4]将结果t4值保存到约定寄存器$v0，并取出<返址>到$r，并释放栈帧后半部分（图中蓝底所示）最后通过[JR $r]将控制流回到ε()中（红色CALL的下条指令）。

foo栈帧剩下的部分286-290是由主调ε()来释放的，因为这部分是它创建的。约定为sp=fp=286

红色CALL的下条指令负责将$v0传送到291单元中，即赋给t6

@code=[t5=0; PAR t5; PAR a; PAR bar; t6=CALL foo@label, 3]

290	〈参数3〉:281
289	〈参数2〉:297
288	〈参数1〉:0
287	〈访问链〉
286	〈控制链〉
~~285~~	~~〈返址〉~~
~~284~~	~~x:0~~
~~283~~	~~b:297~~
~~282~~	~~boo[1]:bar@label~~
~~281~~	~~boo[0]:499~~
~~280~~	~~t1:_~~
~~279~~	~~t2:_~~
~~278~~	~~t3:_~~
~~277~~	~~t4:1~~

500	〈访问链〉:0
499	〈控制链〉:0
498	〈返址〉
497	z:0
496	a[9,19]:0
495	a[9,18]:0
…	…
297	a[0,0]:0
296	bar[1]:bar@label
295	bar[0]:_
294	foo[1]:foo@label
293	foo[0]:_
292	t5:0
291	t6:1

foo@table:(outer:@table width:16 argc:3 arglist:(x b boo) rtype:FLO
code:[IF x>0 THEN l1 ELSE l2;LABEL l1; t1=0; t2=b[t1];PAR t2; t3=CALL sqrt, 1;
z=t3; GOTO l3; LABEL l2;PAR z; t4=CALL boo, 1;RETURN t4;LABEL l3]
entry:(name:x type:INT offset:4) entry:(name:b type:ARRPTT offset:8 etype:INT)
entry:(name:boo type:FUNPTT offset:16 rtype:INT))

ε()函数返回意味着程序执行结束

- ε()栈帧不释放，因为是静态的
- ε()栈帧也可分配到全局静态区
- 在红色代码的最后，隐含着有一个[STOP]指令，让程序停机

500	<访问链>:0
499	<控制链>:0
498	<返址>
497	z:0
496	a[9,19]:0
495	a[9,18]:0
…	…
297	a[0,0]:0
296	bar[1]:bar@label
295	bar[0]:_
294	foo[1]:foo@label
293	foo[0]:_
292	t5:0
291	t6:1

```
@table:
(outer:NULL width:820 argc:0 arglist:NIL rtype:INT
code:[t5=0; PAR t5; PAR a; PAR bar;t6=CALL foo@label, 3]
entry:(name:z type:INT offset:4)
entry:(name:a type:ARRAY base:804 etype:INT dims:2
 dim[0]:10 dim[1]:20)
entry:(name:bar type:FUNC offset:812 mytab:bar@table)
entry:(name:foo type:FUNC offset:820  mytab:foo@table))
```

g()的引用宿主f()为g()创建活动记录

- f@code包含[PAR tm; ...; PAR t1; v=CALL g, m]
- fp和sp指向f()的栈帧
- f()负责构建<参数区>、<访问链>和<控制链>

[sp=sp-4; M[sp]=tm;...;
sp=sp-4; M[sp=t1]++
[构建访问链]++
[sp=sp-4; M[sp]=fp; fp=sp]
++
[JAL g@label;]++
[v=$v0; fp=M[sp];
sp=sp+(8+4*g@argc)]

- sp和fp都指向<控制链>，执行调子指令控制流进入g@code同时构建<返址>
- 控制流回来后sp和fp都指向<控制链>保存结果到v，fp指向f栈帧，修改sp释放剩余部分。

- g()完成剩余部分的构建，即sp=sp-g@width
- 这部分被称为序言g@prologue

[sp=sp-4; M[sp]=$a;
sp=sp-g@width]

g()释放自己栈帧并将控制流转回f()

- g@code包含[RETURN t1]
- fp和sp指向g()的栈帧
- g()保存返回结果，
  释放<局部区>，
  取下<返址>，
  控制流转移到<返址>

```
[$v0=t1
sp=sp+g@width
$t2=M[sp]; sp=sp+4
JR $t2]
```

- 可能存在多个[RETURN t]型指令，就将共同部分作为g代码的尾声部分
- 对于g@code最后一条指令不是RETURN指令，有了尾声也就没有问题了。

```
[$v0=t1; J g@epilogue]

g@epilogue=[
sp=sp+g@width
$t2=M[sp]; sp=sp+4
JR $t2]
```

g@label全貌

g@label=g@prologue++转换后的g@code++g@epilogue
=[sp=sp-4; M[sp]=$a; sp=sp-g@width]
++
转换后的g@code
++
sp=sp+g@width; sp=sp+4; $t2=M[sp]; JR $t2]

[$v0=t1; J g@epilogue]

f@label=f@prologue++转换后的f@code
++f@epilogue
=[sp=sp-4; M[sp]=$a; sp=sp-f@width]
++
转换后的f@code
++
sp=sp+f@width; sp=sp+4; $t2=M[sp];
JR $t2]

[sp=sp-4; M[sp]=tm;...;
sp=sp-4; M[sp=t1]++
[构建访问链]++
[sp=sp-4; M[sp]=fp;
fp=sp]++
[JAL g@label;]++
[v=$v0; fp=M[sp];
sp=sp+(8+4*g@argc)]

函数的序言和尾声

- 函数序言如g@prologue是加在函数原始代码即g@code的入口处的代码段，其功能：
  - 保存寄存器到栈顶部分（省略）；
  - 保存返回地址；
  - 将参数单元内容赋给形参变量；g@arglist=(a_0 … a_{m-1}) [a_0=M[fp+(8+4*0); … ; a_m=M[fp+(8+4*(m-1))]
  - 构建局部区（事实上只是分配局部区空间）。
- 函数尾声如g@epilogue是在函数返回原始代码如g@code的出口处（最后一条指令后）添加的代码段，其功能：（注意非RETURN返回则$v0无意义）
  - 恢复保存的寄存器（略）；
  - 释放局部区；
  - 弹出返回地址，并让控制流转移去往该地址。

函数调用与返回的代码段

- 调用代码序列（对应于PAR/v=CALL代码段）的功能：
  - 构建参数区；
  - 构建链接区（返址除外）；
  - 转子被调过程；//[jal g@label]
  - （子过程返回至此）将$v0赋给v；//[v=$v0]
  - 释放链接区（剩余部分）和参数区，fp和sp指向调用过程的栈帧。

- 返回代码序列（对应于[RETURN t]代码）的功能：
  - 将[RETURN t]返回结果保存在$v0中；//[$v0=t]
  - 转移到本函数的尾声执行。//[J g@epilogue]

9.3 函数作为参数

```
1 int x;
2 int y;
3 void q(int s(); int x;){
4         int y;
5         y=s(x+10);
6         print y};
7 int p(){
8         int r(int x;){
9                 int z;
10                z=x+y;
11                return z};
12        q(r(), x*3,)};
13 x=15;
14 y=21;
15 p()
```

- 设栈底单元地址500，不考虑临时变量，在源程序上进行模拟执行。
- 写出执行到第11行return语句时栈快照

例：函数参数

```
1 int x;
2 int y;
3 void q(int s(); int x;){
4           int y;
5           y=s(x+10,);
6           print y}
7 int p(){
8           int r(int x,){
9                       int z;
10                      z=x+y;
11                      return z};
12          q(r(), x*3,}
13 x=15;
14 y=21;
15 p()
```

规律：写在参数区的就要“向后面寻找”（addr更小）

Q (r ( ), x * 3)

500	<访问链>:0
499	<控制链>:0
498	<返址>
497	x:15
496	y:21
495	q[1]
494	q[0]
493	p[1]
492	p[0]

ε

全局作用域

运行结果

运行结果

在全局作用域下，p()被调用

P内部定义了一个函数(r())

491	<访问链>:499
490	<控制链>:499
489	<返址>
488	r[1]:r@label
487	r[0]:490

p

R函数只有一个参数x，且此x在上一级被定义，因此r[0] => 490

显而易见的，r[1] => r@label

对实参求值结果

实参求值的当前栈帧

p ( )

在p函数中调用了q()
q()在全局作用域下被定义，在p()中被调用

486	<参数2>:45
485	<参数1>:480
484	<访问链>:499
483	<控制链>:490
482	<返址>
481	s[1]:r@label
480	s[0]:490
479	x:45
478	y

q

Q函数内部定义了一个函数s()作为形参

参数传递的是地址

实参是r()和x*3
r() 必须要向上一级寻找！（按常理是指向487行，但是实际是见下）
x*3显而易见是15x3=45

对q()内部定义的s()进行说明(s本身作为r的形参)

s[0]在这里指的就是r[0]，指向490行
s[1]在这里就是r[1]。直接指向r@label

在这里x作为形参，指代的就是15x3=45
Y尚未被计算

随后为76

477	<参数1>:55
476	<访问链>:490
475	<控制链>:483
474	<返址>
473	x:55
472	z:76

r

声明宿主就是引用宿主

y=s(x+10) 它定义在p内部(s实际上是r，定义在p内)，调用在q内部（q调用s(x+10)）参数是45+10

第二行的y

例：形参为数组原型、函数原型

- 更正习题10.4 数组原型按照一维数组处理

```
int x; float z;
int a[10,20]; //初始化值为a[i,j]=i+j
float bar(int y;){
        float x;
        x=y*PI;
        return x};
float foo(int x; float boo(); int arr[];){
        if (x==0)z=boo(arr[1],)
        else return boo(arr[15*x],)};
print foo(2, bar(), a[],)
```

例：形参为数组原型、函数原型

全局作用域

```
int x; float z;
int a[10,20];
//初始化值为a[i,j]=i+j
float bar(int y;){
  float x;
  x=y*PI;
  return x};
float foo(int x; float boo();
            int arr[];){
  if (x==0)z=boo(arr[1],)
  else return boo(arr[*x],)};
print foo(2, bar(), a[],)
```

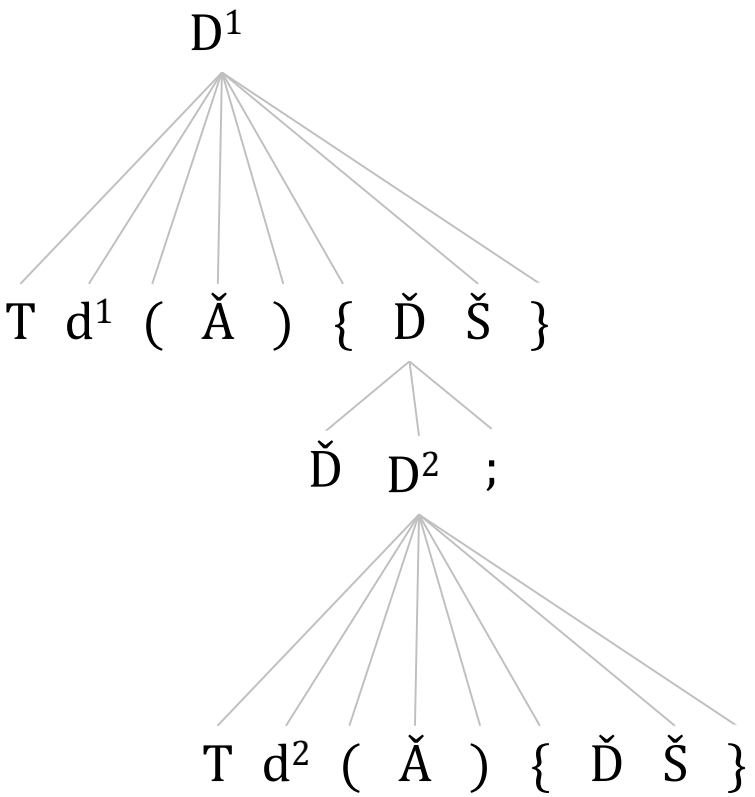
500	〈访问链〉:0
499	〈控制链〉:0
498	〈返址〉
497	x
496	z:0
495	a[9,19]:28
…	…
297	a[0,1]:1
296	a[0,0]:0
295	bar[1]:bar@label
294	bar[0]
293	foo[1]:foo@label
292	foo[0]

291	〈参数3〉:296
290	〈参数2〉:283
289	〈参数1〉:2
288	〈访问链〉
287	〈控制链〉
286	〈返址〉
285	x:2
284	boo[1]:bar@label
283	boo[0]:499
282	arr:296
281	〈参数1〉:11
280	〈访问链〉:499
279	〈控制链〉:287
278	〈返址〉
277	y:11
276	x:34.56

传值	*数组原型*	*函数原型*	*简单变量*
参数单元	实参地址	形参名单元地址	实参值
形参名单元	参数单元值	复制自实参	实参单元值

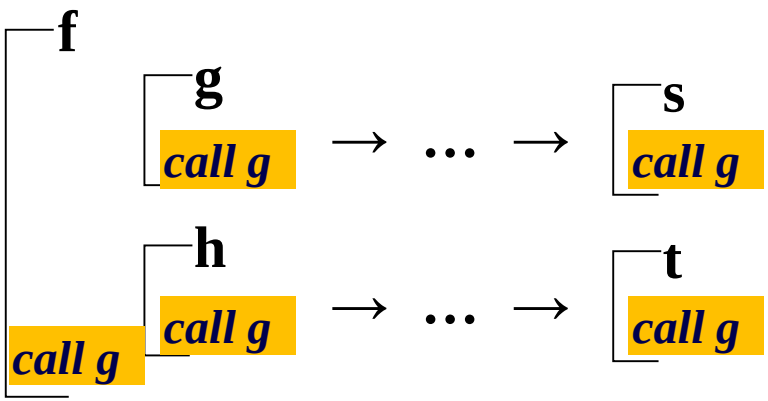
9.4构建访问链及访问非局部名字

- 函数声明的嵌套与并列关系：
- 右图函数d^1声明是D^1，d^2声明是D^2，文法均为D→T d(Ă){ĎŠ}
- 函数d^1声明嵌套函数d^2声明，说d^1是d^2的直接外层，也说函数d^2的声明宿主是函数d^1，简述为d^2是d^1的声明宿主
- 如果有主调f与被调g这样的函数调用，则说函数g的引用宿主是函数f，简述为f是g的引用宿主
- 任意函数的声明宿主只有一个，而引用宿主可以多个。如果名为g的函数有多个声明宿主，说明各个声明宿主都有名为g的局部函数，各是不同的函数



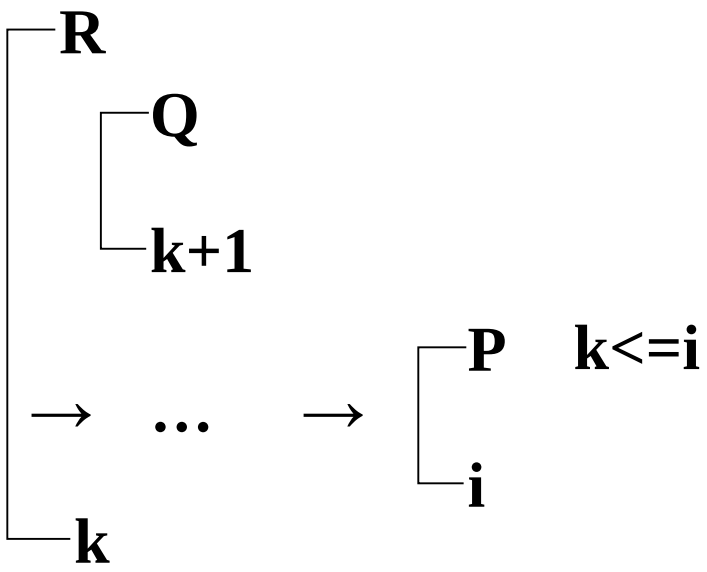
嵌套过程、作用域及嵌套层次

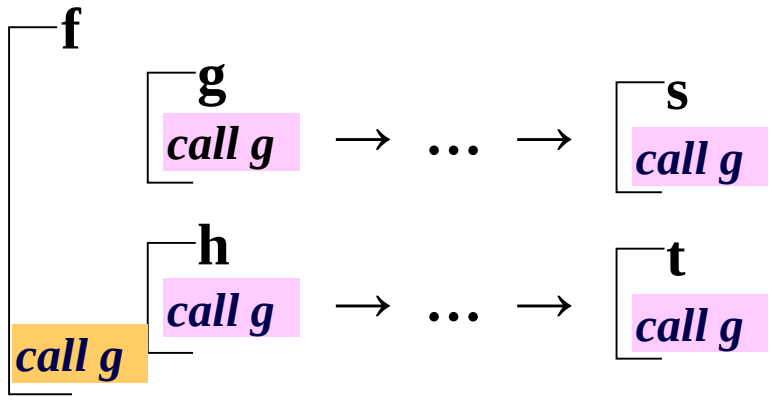
- 一个函数的名字的静态作用域是这个函数的所有引用宿主的集合
- Pascal语言的过程g()的静态作用域定义为：
  - g()的声明宿主f()；
  - g()及其声明子孙。即设为S，那么g()∈S，若r()的声明宿主为h()且h()∈S那么r()∈S。
  - 与g()有同一声明宿主的且在g()之后声明的函数以及它的声明子孙。



作用域分析

- 函数Q的引用宿主是函数P，声明宿主是函数R，那么P如何构建Q的访问链？
  - R就是Q的声明宿主P (k=i)，此时与引用宿主相同
  - R是P的外层(k<i)，换句话说P是R的声明子孙
  - 其它情况P不是Q的作用域，不可能调用Q





由主调P的访问链构造主调Q的访问链

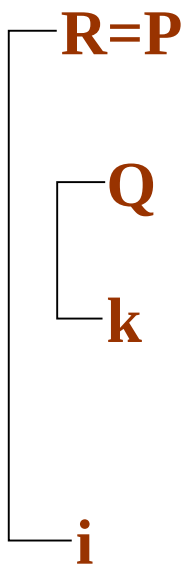
①R=P，即P是Q的声明宿主。

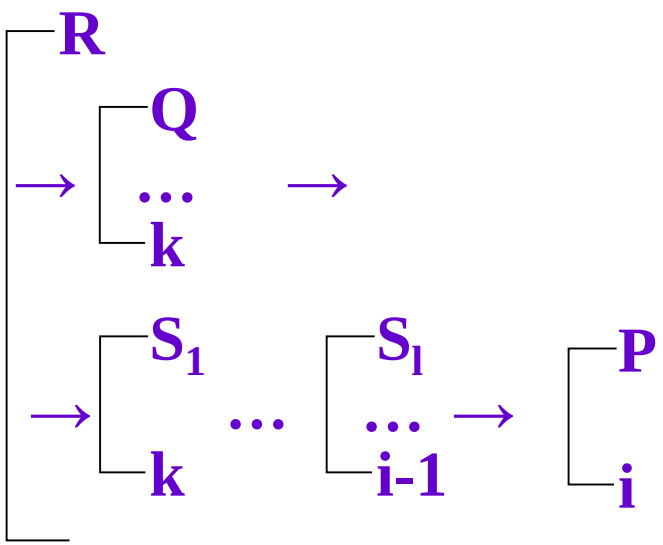
P的活动记录就是Q的声明宿主的最新活动记录。

最新栈帧是指沿控制链最近。

②Q的引用宿主P是声明宿主R的声明子孙。

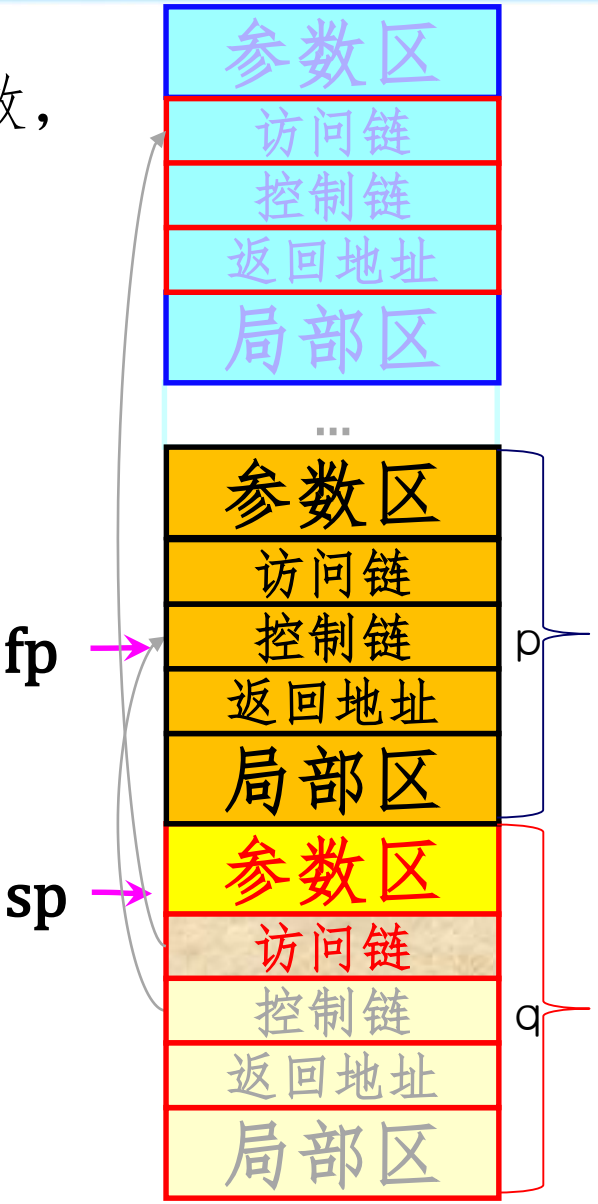
沿着主调P的活动记录的访问链走过i-k+1个（也就是找到了R的最新活动记录），那个活动记录作为Q的访问链所指。





主调p()构建被调q()的访问链单元

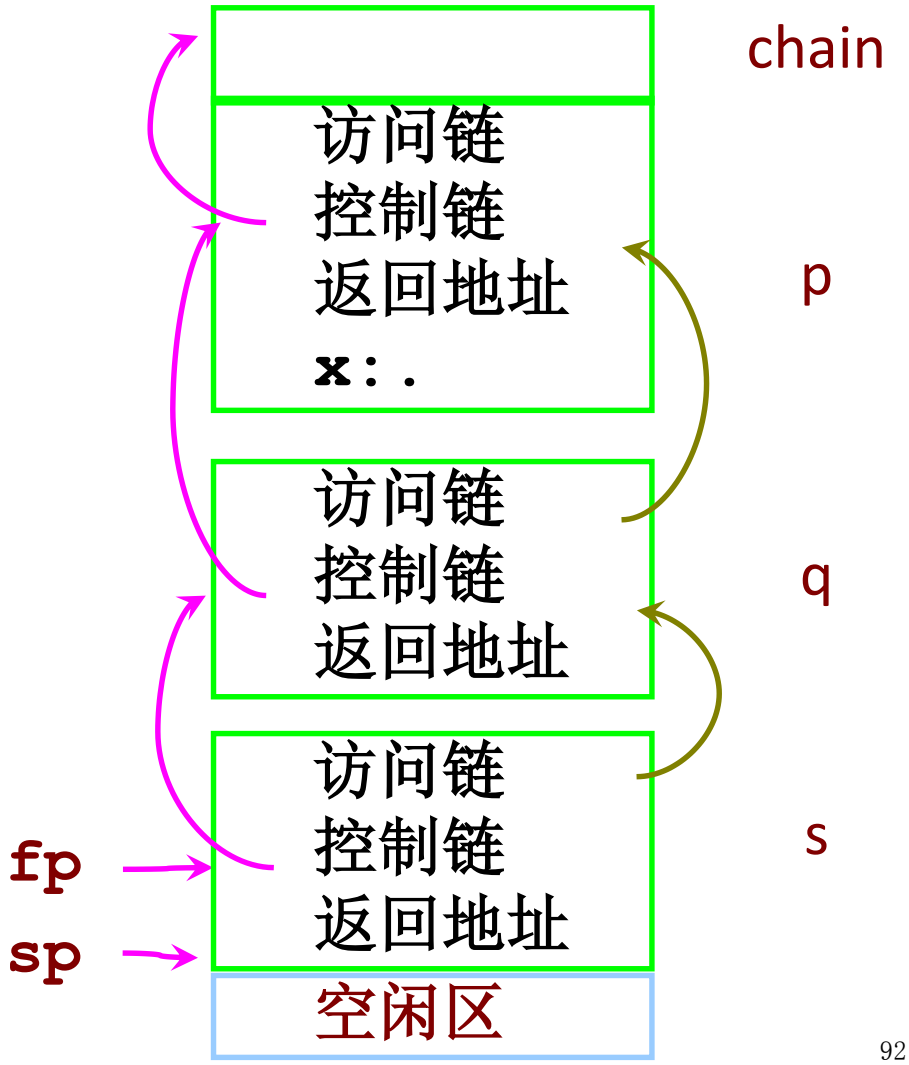
- tab为主调p的符号表，tab->level嵌套层数，sp指向参数区（参数区刚构建完成）
- tab_q=lookup1(tab, q, mytab);
  tab_r=tab_q->outer;
  if(tab_r!=tab){//情形②：i-k+1
  k=tab->level-tab_q->level+1;
  emit[rt=fp];
  for(i=0;i<k;i++)emit[rt=M[rt+4]];
  emit[sp=sp-4]; emit[M[sp]=rt]
  }else
  emit[push [fp]]//情形①：



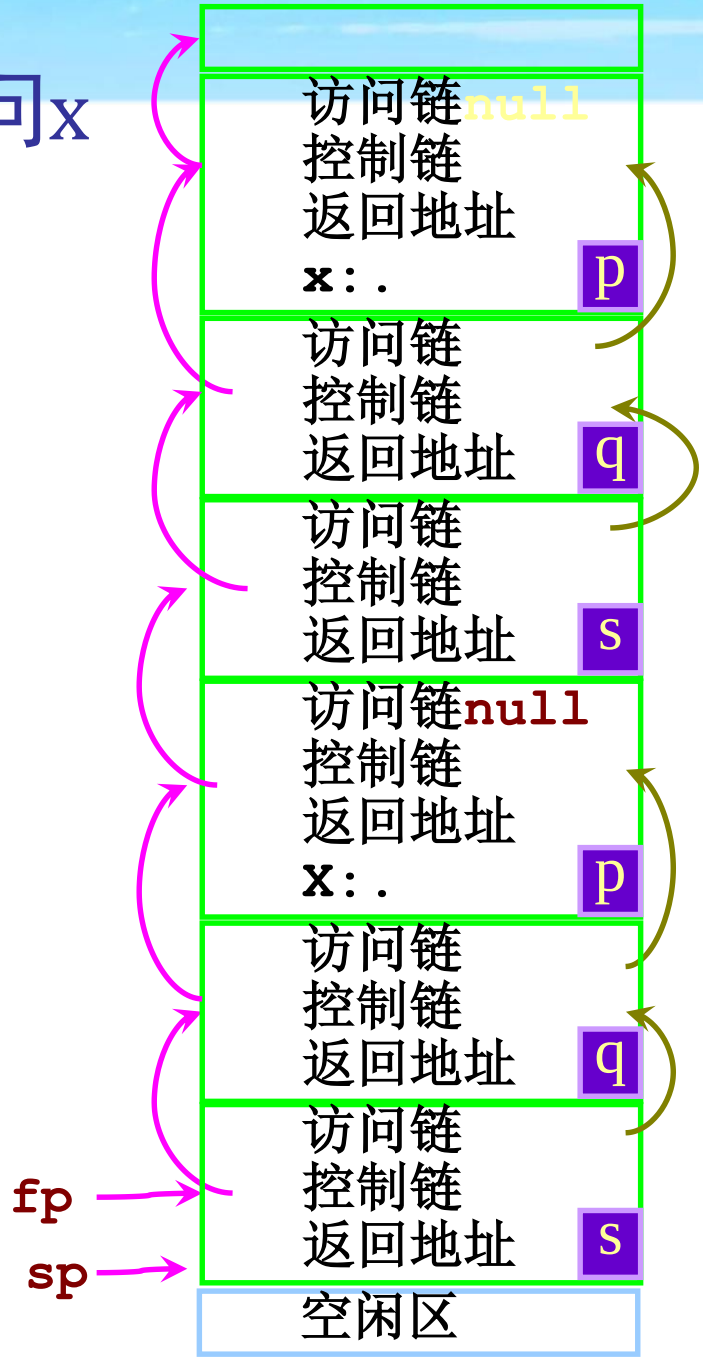
使用访问链对非局部名寻址

```
program chain;
  procedure p;
    var x:integer;
    procedure q;
          procedure s;
          begin
                x:=2;
                ...
                if ... then p;
          end;
    begin
          s;
    end;
 begin
    q;
 end;
begin
    p;
end
```

```
rt=fp
rt=M[rt+4]
rt=M[rt+4]
x值： M[rt-8]
```



s中访问x




```
rt=fp
rt=M[rt+4]
rt=M[rt+4]
```

x值：M[rt-8]

一般情形，主调层比被调层多k的话，需要沿着访问链k次装载rt寄存器，第一次使用fp进行，其余k-1次用rt

使用全局名表示名引用

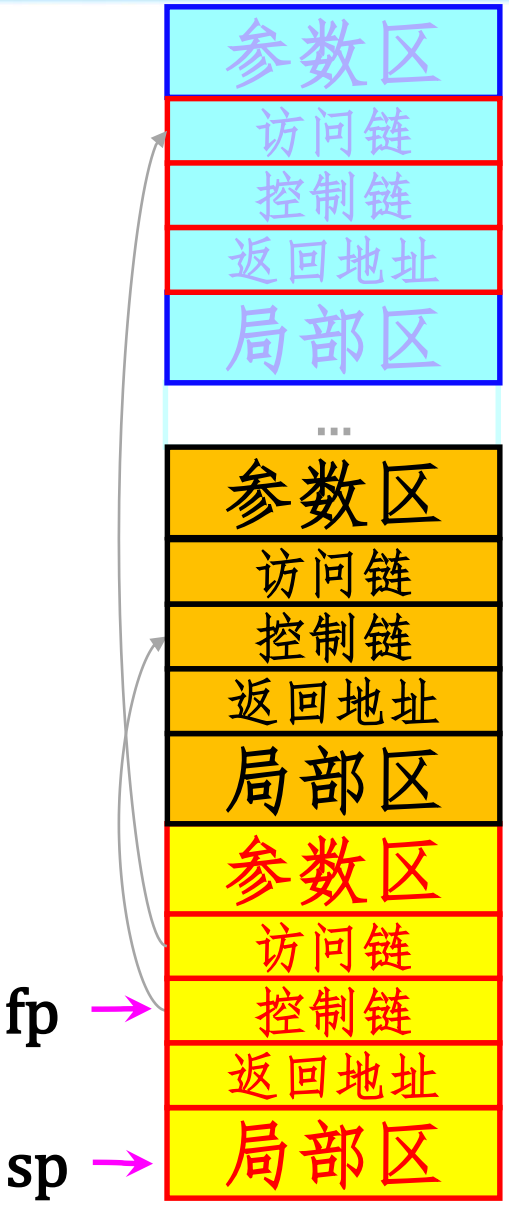
- t=x@value可转换为：
  [rt=fp; rt=M[rt+4]; rt=M[rt+4]; t=M[rt-8]]
- x@value=t可转换为：
  [rt=fp; rt=M[rt+4]; rt=M[rt+4]; M[rt-8]=t]

- t=x@addr可转换为：
  [rt=fp; rt=M[rt+4]; rt=M[rt+4]; t=rt-8]
- x@addr=t可转换为：
  错误

- 无论局部名还是非局部名的引用，都可采用以上表示形式。
- 如果通过重名分析解决了重名，就可以不再考虑重名，有助于减轻负担。

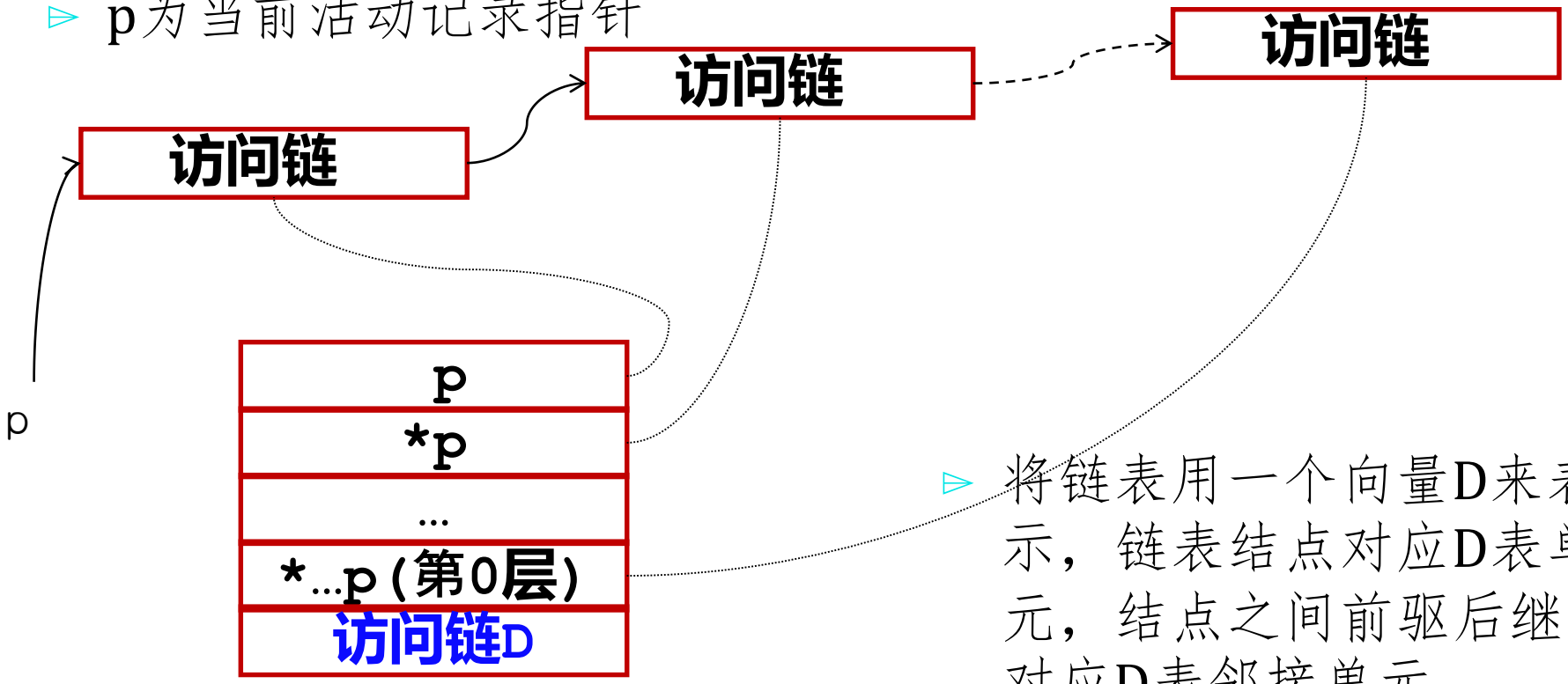
非局部名寻址

- 针对代码中的局部变量x进行如下处理（编译时）并产生代码替换。
- tab1=tab; //当前符号表
  emit[rt=fp];
  while(tab1!=NULL){
  if(lookup1(tab1, x)!=UNBOUND){
  offs=lookup1(tab1, x, offset);
  subst[?x, M[rt-?(offs+LL)]];
  break} else {
  tab1=tab1->outer;
  if(tab1==NULL)error();//没找到x
  emit[rt=M[rt+4]]//沿访问链往外
  }} //令emit[]输出指令都插入到
  //出现x的指令前边



访问链优化为显示表

- p为当前活动记录指针



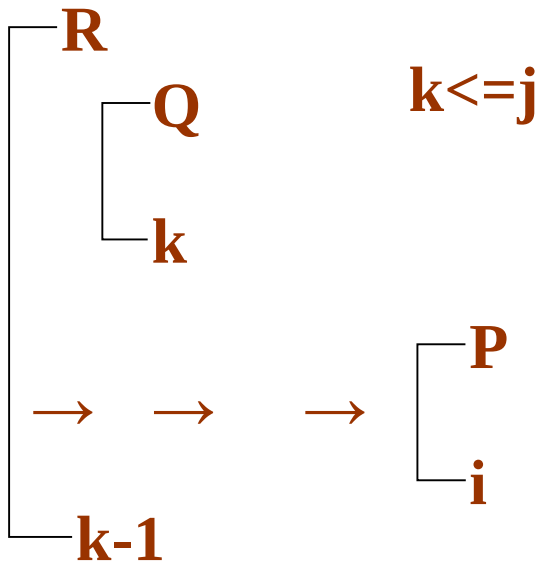
- 将链表用一个向量D来表示，链表结点对应D表单元，结点之间前驱后继对应D表邻接单元。
- 显示表D有k个元素都为指向活动记录的指针值，其中k为该过程嵌套层数

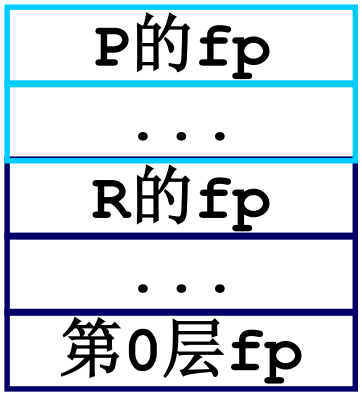
用主调的D表构造被调的D表1

- R就是P：主调的活动记录就是被调访问链所指那个
- P的D表++指向Q的指针

由主调的D表构造被调的D表2

- R是P的外层：那么沿着当前活动记录的访问链走过i-k+1个（也就是找到了R的最新活动记录），那个活动记录作为Q的访问链所指向的
- P的D表中的前k个 外加指向Q的指针





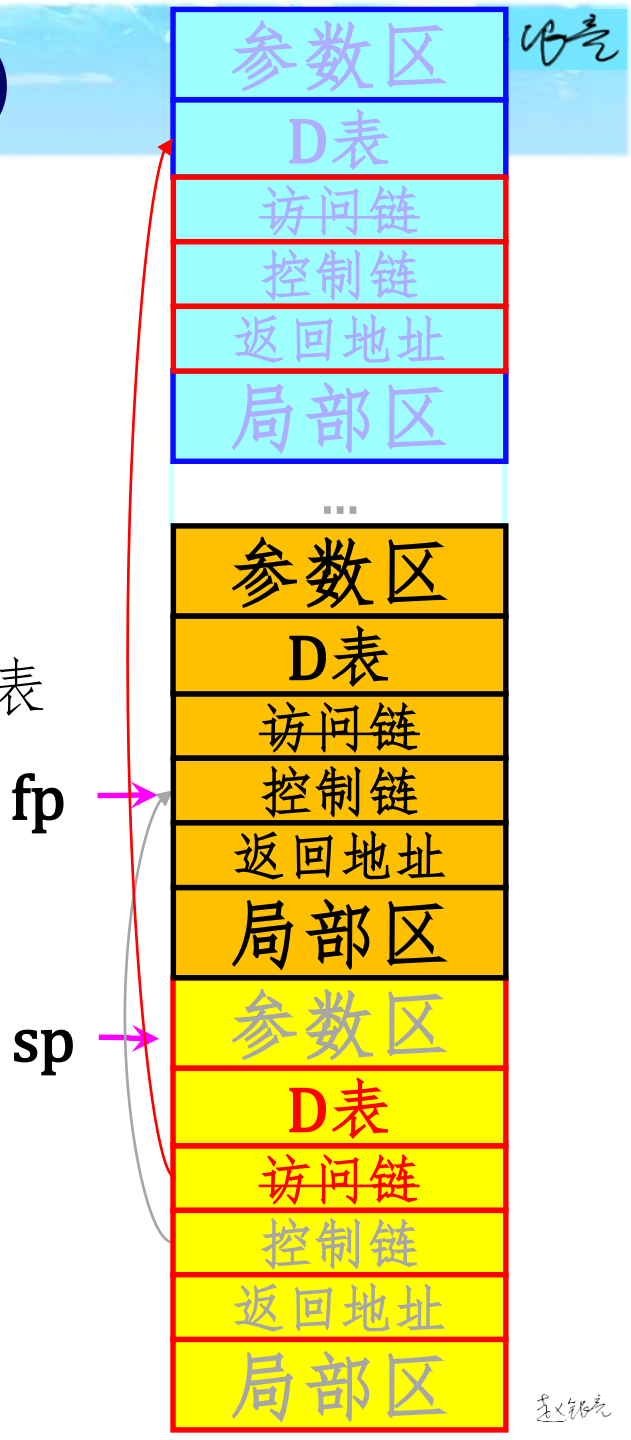
主调的D表

被调的D表

构建D表（p调用q）

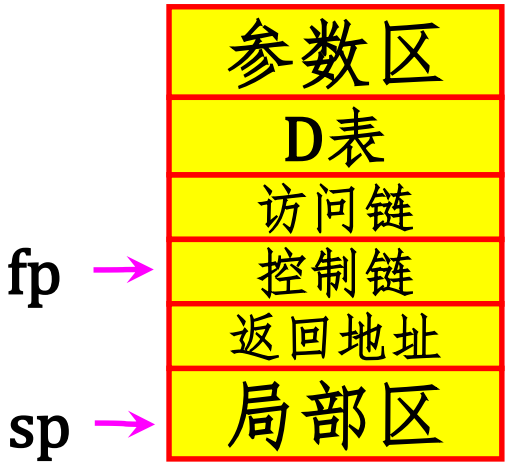
- sp指向黑参数区（参数区刚构建完成）

tab_q=lookup(q,mytab);tab_r=tab_q->outer;
//复制主调D表前k个字到红D表前k个字
k=tab_q->level;//红D表长度为k+1
emit[sp=sp -?((k+1)*4)];//指向红D表开始
emit[rs=fp+8];//指向黑D表开始
for(i=0;i<k;i++){//复制黑D表k个字到红D表
emit[rt=M[rs+?(i*4)]];
emit[M[sp+?(i*4)]=rt]}
emit[M[sp+?(k*4)]=?(sp-8)];
//自己fp



非局部名寻址

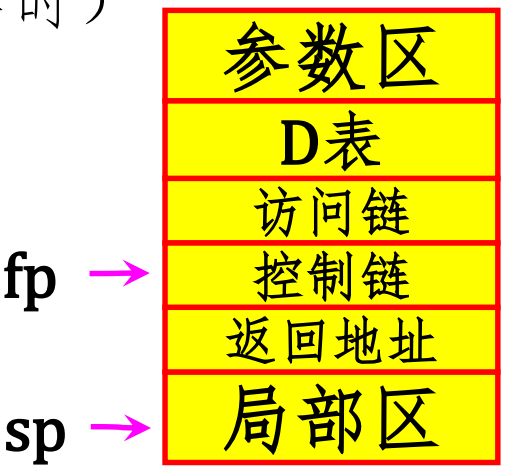
- 过程代码中，非局部名的访问地址
  - 绝对地址 = D[静态层数]+相对地址
  - 静态层数指定义那个非局部名的过程的层数
  - 相对地址为LL+偏移量



使用D表的非局部名寻址

- 针对代码中的变量x进行如下处理（编译时）并产生代码替换（替换范围tab->code)。

```
tab1=tab;
while(tab1!=NULL){
 if(lookup1(tab1, x)!=UNBOUND){
   k=tab1->level;//x所在层的层数
   emit[rt=M[fp+ ?(LD+k*4)]];//LD=8
   offs=lookup1(tab1, x, offset);
   subst[?x, M[rt- ?(LL+offs)]]; //把x替换为局部区x单元
   break}
 else {
   tab1=tab1->outer;//继续查找x所在层
   if(tab1==NULL)error()
 } //LD为D表至fp的距离（D表偏移量）
}//LL为局部区至fp的距离（局部区偏移量）；LA参数区...
```



9.6 构建代码区

- 对程序中声明的每一个函数f，对f@code进行代码转换，具体转换内容及方法已经逐一介绍了，转换后就得到f的可执行代码，即f@label，保存在f登记项的offset双字区第一个字中。
- 当然，涉及到以下我们未做介绍的工作：
  - 中间代码优化
  - 基于指令模板的目标代码生成
  - 寄存器分配
  - 目标代码优化
- 将可执行代码映射到代码区，过程名与代码入口点关联起来以便生成这样的代码它能转移到该入口点实现对该过程代码的执行。
  - 控制流进入被调过程[jal g@label]
  - 控制流从被调过程返回[jr $t]

参考书

Torben Ægidius Mogensen
Datalogisk Institut
Københavns Universitet
Copenhagen
Denmark
Introduction to Compiler Design.

9.7 堆式动态存储分配

- 显式的动态申请
  - Pascal的new和dispose语句
  - C的malloc和free语句
  - Java的new语句
- 显式的动态释放
  - Pascal的new和dispose语句
  - C的malloc和free语句
- 隐式的动态释放
  - Java的Garbage Collection
  - Lisp的Garbage Collection

堆式动态存储分配的实现

- 定长块管理
- 变长块管理
  - 首次适应法
  - 最佳适应法
  - 最差适应法

变长块管理中空闲块选择算法的比较

- 最佳适应法
  - 请求分配的内存块大小范围较广的情况
  - 有可能产生很小的碎片
  - 保留更大的块以满足大尺寸申请
  - 分配、释放均要查链表
- 最差适应法
  - 请求分配的内存块大小范围较窄的情况
  - 保持块由大到小次序
- 首次适应法
  - 有随机性，介于二者之间
  - 保持块由小到大次序

隐式存储回收(Garbage Collection)

- Mark-Sweep方法
- Stop-Copy方法
- 实时的方法

小结

- 内存映像及栈帧
- 过程活动、生命期、嵌套并列关系
- 活动记录、参数区、链接区、局部区
- 栈快照、栈指针sp与fp
- 构建活动记录
- 函数序言、尾声、调用代码序列
- 参数传递
- 构建访问链、D表、非局部名的访问
- 函数作为参数的处理
- 代码区、静态区、堆区

- 作业P269-271：4, 5, 6, 9

作业

- 对于下列程序，写出当执行导到return语句时的栈快照。提示：就所给源程序而言不涉及临时变量，只需模拟执行得出栈快照内容即可。假定栈快照的起始单元地址为500。

```
1 int x;
2 int y;
3 void q(int s(); int x;){
4         int y;
5         y=s(x-6,)}
7 int p(){
8         int r(int x;){
9                 int a[5];
10                a[1]=x+y;
11                return a[1]};
12        q(r(), x*y,}
13 x=5;
14 y=2;
15 p()
```