第十一章 运行时环境

赵银亮 西安交通大学2025

本章内容

机器代码生成器的原理：

对于三地址代码可采用模式替换方法转换为机器语言代码

运行时环境的设计：

静态区、栈区、堆区。

源程序中每个简单变量和下标变量的求值环境。

中间表示中每个临时变量的求值环境

每个函数的运行时环境

函数的可执行代码的构建：

将foo@code变换为foo@label，其中为每个名字引用都获得运行时环境支持。

11.1机器代码生成器的原理

- 将中间语言代码作为输入串，并初始化为剩余串；
- 将生成的机器代码作为输出，初始化为空；
- 重复以下步骤直到剩余串为空为止：
  - 若剩余串的某前缀与某模式匹配，则生成一个替换实例添加到输出串的尾部；
  - 若剩余串的所有前缀与所有模式都不匹配则出错。
- 返回输出串作为所生成的机器代码。

- 常见两种策略：
  - 贪心法：选择能匹配上的最大前缀进行替换；
  - 线性规划：每个模式替换对都赋予代价，生成总体代价最小的机器代码。

开发模式替换对（指令模板）

中间语言

1. 对使用的变量的数量没有限制
2. 使用较为抽象的指令用于函数调用与返回
3. IF-THEN-ELSE指令有两个目标标号（标记，label）
4. 允许任意一个整数都可作为一个算术指令的操作数

机器语言

1. 处理器的寄存器是很有限的
2. 一系列指令，分别称为调用序列与返回序列
3. 大部分处理器的条件跳转指令都只有一个目标标记，简单处理为下条指令即条件为假时目标标号
4. RISC处理器仅允许小常数做操作数

第1条属于寄存器分配器所要解决的问题
其他条都在本章来解决：从中间代码生成机器代码的简单解决思路是，将每条中间语言指令翻译为一条或多条机器代码的指令，但是也不排除一条机器指令覆盖一到多条中间语言指令的情形。

条件转移

- 条件转移在各种机器上有多种形式，一些与寄存器寄存器比较、寄存器常数比较结合起来，与IF-THEN-ELSE相似。另一些条件转移指令使用已形成并存储在专用条件寄存器或条件标志位的信息，而该条指令的下一条指令是条件为假时的目标

IF c THEN l_t ELSE l_f

```
branch_if_c l_t
jump l_f
```

IF c THEN l_t ELSE l_f

```
branch_if_not_c l_f
l_t:
```

- 如果条件太复杂机器指令不支持，可先计算条件并存储于后继跳转指令可以检测的地方，即MIPS的通用寄存器，ARM32的一组算术标志位（可由比较指令和算术指令设置）
- 通常， IF-THEN-ELSE可被翻译为至多两条指令，一条做比较另一条做条件跳转

常数

- MIPS只允许16-bit常数（小整数）做为操作数，寄存器为32-bit的或64-bit的
- 在寄存器构造更大常数需要两条指令分别装入对应高、低16位的两个小整数来完成
- ARM32仅允许8-bit小整数，那么在寄存器里构造32-bit整数需要四条指令。
- 中间语言指令使用常数，代码生成器为此进行相应处理：若该整数适合指令的常数域，则直接生成使用该常数的指令，否则需要先生成指令在寄存器里构造这个常数，然后生成使用这个寄存器的指令。
- 如果发现在循环体里每次都构造同一个常数，可将那些代码移到外边，循环体里仅保留使用那个寄存器的指令（循环不变量外提）

复杂指令

- 中间语言的大多数指令是原子的。非原子的仅分支、函数调用、返回
- 中间语言的多条指令组合成机器代码的一条指令

  $t_2 = t_1 + 256$　　`lw r3, 256(r1)`

  $t_3 = M[t_2]$
- r1和r3是分配给t_1和t_3的寄存器。
- 能合并的条件是这个t_1在后续代码中不再需要，因为合并生成的指令不会保存它的值。
- 因此需要知道一个变量的内容是否不再需要，即在某次使用后死亡还是继续活跃。考虑临时变量严格遵守一次定义一次使用，可被标记为一次使用后即死亡
- 对中间语言代码进行活跃变量分析可知道每个变量的最后一次使用是发生在哪条指令。课程中用t^{last}表示变量t在中间语言代码中的最后一次使用。

模式与替换

- 每一条机器指令都用中间语言的一至多条指令描述
- 模式pattern：中间语言的指令序列
- 替换replacement：对应的机器指令
- 从中间代码中找出与模式匹配的指令序列，将它们都做替换
- 出现在模式替换中的k、t、r_d和l，依次表示常数、变量（可进一步区分源程序变量和临时变量）、寄存器和标号（模式中源程序标号，替换实例中机器代码标号）。

$t = r_s + k$	lw r_t, $k(r_s)$
$r_t = \mathrm{M}[t^{\mathrm{last}}]$	

- 也存在中间语言的一条指令不对应单个机器指令的情形，因此一般地模式和替换都推广为指令序列

机器代码生成器

- 翻译中间语言的指令序列为机器代码的指令序列
- 贪心算法：逐次查找覆盖最大前缀的模式并进行替换（每一口都吃掉最大的一块，但最后结果不一定是最好）
- 为每个模式替换对都确定其开销，比如转移越少开销越少，就可用线性规划方法求解之

- 下页指令模板：
  - 假定已处理过大整数
  - 利用MIPS的一个小的指令子集
  - 提示如何探索和利用机器指令为该中间语言设计更多指令模板

指令模板（模式替换对）

$t = r_s + k$ $r_t = M[t^{last}]$	`lw` r_t, k(r_s)
$r_t = M[r_s]$	`lw` r_t, 0(r_s)
$r_t = M[k]$	`lw` r_t, k(R0)
$t = r_s + k$ $M[t^{last}] = r_t$	`sw` r_t, k(r_s)
$M[r_s] = r_t$	`sw` r_t, 0(r_s)
$M[k] = r_t$	`sw` r_t, k(R0)

$r_d = r_s + r_t$	`add` r_d, r_s, r_t
$r_d = r_t$	`add` r_d, R0, r_t
$r_d = r_s + k$	`addi` r_d, r_s, k
$r_d = k$	`addi` r_d, R0, k
GOTO l	`j` l
LABEL l	l:

IF r_s=r_t THEN l_1 ELSE l_2 LABEL l_2	beq r_s, r_t, l_1 l_2:
IF r_s=r_t THEN l_1 ELSE l_2 LABEL l_1	bne r_s, r_t, l_2 l_1:
IF r_s=r_t THEN l_1 ELSE l_2	beq r_s, r_t, l_1 j l_2
IF r_s<r_t THEN l_1 ELSE l_2 LABEL l_2	slt r_d, r_s, r_t bne r_d, R0, l_1 l_2:
IF r_s<r_t THEN l_1 ELSE l_2 LABEL l_1	slt r_d, r_s, r_t beq r_d, R0, l_2 l_1:
IF r_s<r_t THEN l_1 ELSE l_2	slt r_d, r_s, r_t bne r_d, R0, l_1 j l_2

贪心法机器代码生成举例

$a = a + b^{\text{last}}$ | $r_d = r_s + r_t$ | `add` r_d, r_s, r_t | `add` a, a, b

$d = c + 8$ | $r_d = r_s + k$ | `addi` r_d, r_t, k

$\mathrm{M}[d^{\text{last}}] = a$ | $t = r_s + k$ | `sw` r_t, $k(r_s)$ | `sw` a, $8(c)$

$\mathrm{M}[t^{\text{last}}] = r_t$

IF $a = c$ THEN l_1 ELSE l_2

LABEL l_2

IF r_s=r_t THEN l_1 ELSE l_2 | `beq` r_s, r_t, l_1 | `beq` a, c, l_1

LABEL l_2 | l_2: | l_2:

小结

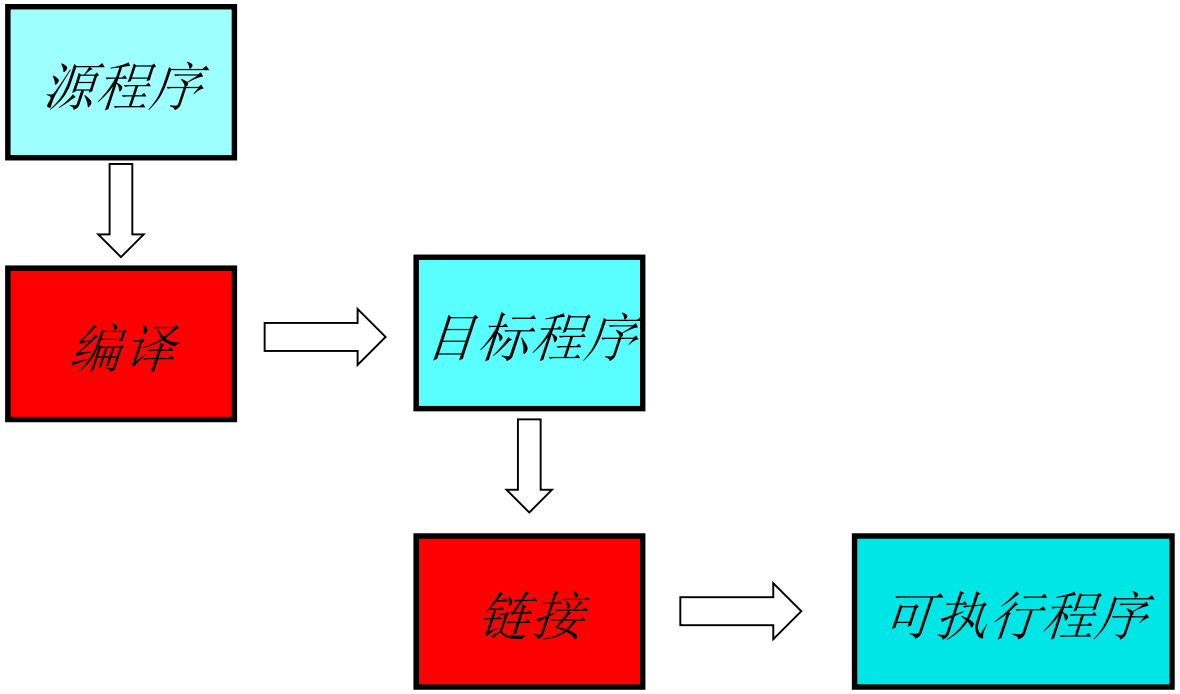
- 三地址指令应适应广泛的机器指令系统的代码生成。
- 指令模板的设计有灵活性：
  - $t = r_s + k$ | lw r_t, $k(r_s)$
    $r_t = M[t^{last}]$ |
  - $r_t = M[r_s + k]$ | lw r_t, $k(r_s)$
- 开发更多模式替换对
  - 更多MIPS指令模板
  - ARM32指令模板
  - X86指令模板
  - …

11.2名字的运行时环境设计

- 名字的运行时环境指运行时如何解决名字引用问题，是一种设计结果
- 名字有：
  - 简单变量
  - 下标变量
  - 函数
  - 源语言的标号
  - 临时变量
  - 中间语言的标号
- 相关术语：存储分配、内存映像、栈帧等等

从可执行程序说起

- 编译的终极目标是得到可执行程序

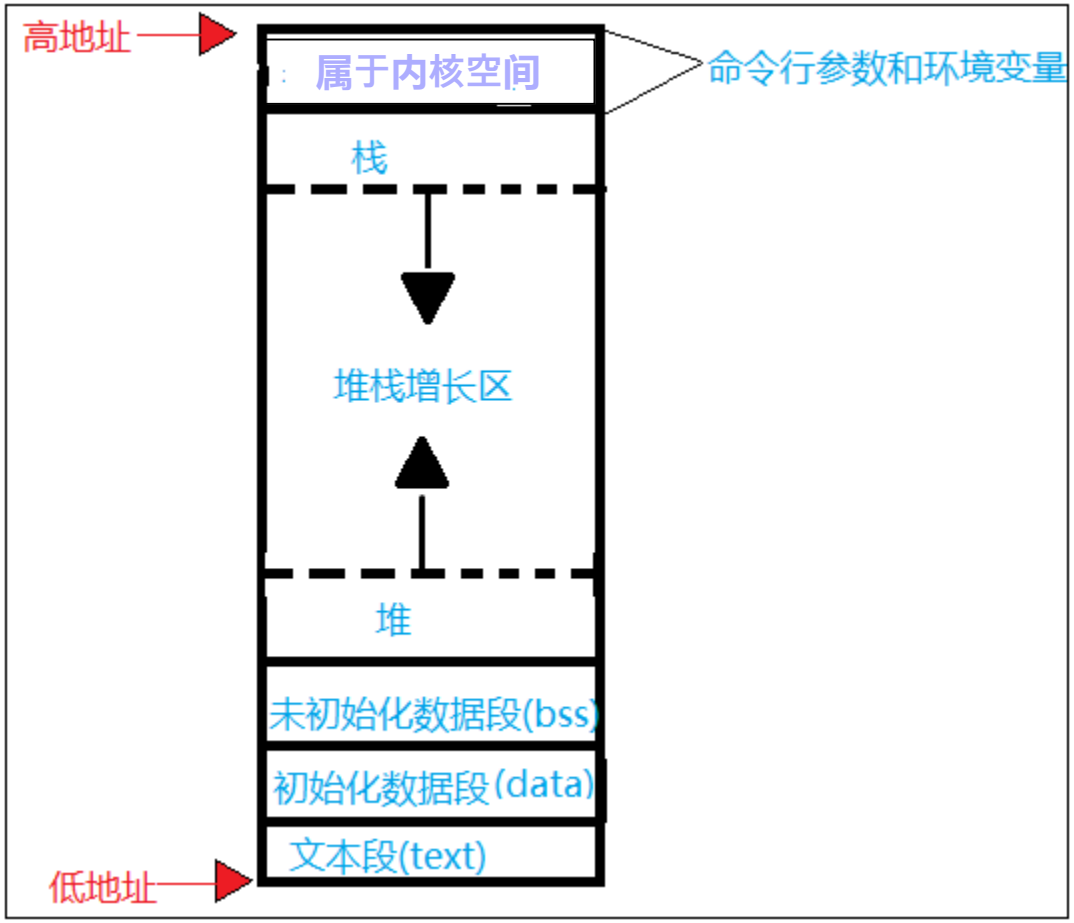


可执行程序：磁盘文件与内存映像

- 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）

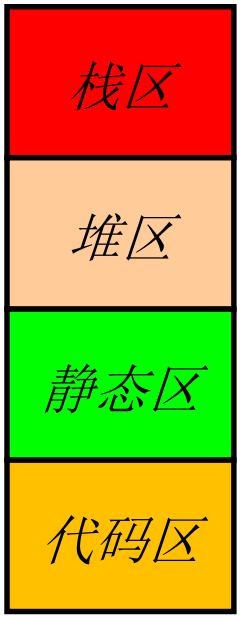
- C未初始化的全局变量和静态变量，初值为0或NULL

```
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



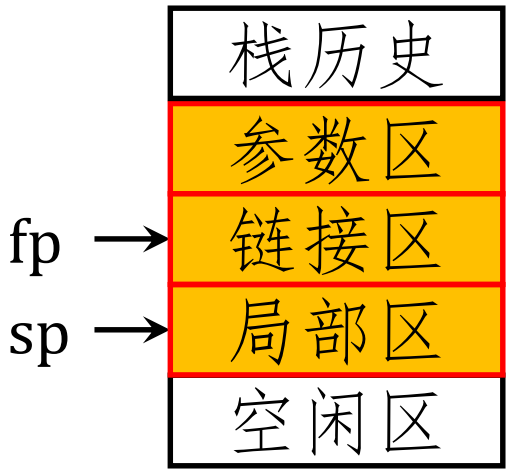
- 存储目标程序、分配存储单元给程序中的名字、存储运行时产生的中间结果、实现过程调用与返回时相关存储管理、实现动态申请与释放存储块的管理、正确访问名的值等。《运行时存储空间组织》

名字的运行时环境

- 简单变量x：x的声明宿主的最新栈帧
- 下标变量a[t]：a的声明宿主的最新栈帧
- 函数foo：foo的代码分配在代码区，foo()的运行时环境是其栈帧
- 源语言的标号label：只出现在符号表里，没有运行时环境。
- 临时变量t：t的引用宿主的最新栈帧（由于临时变量仅出现在中间代码里，通过newvar()产生，也可认为其引用宿主也即声明宿主）
- 中间语言的标号l：仅考虑在编译时已静态解决的，这些标号仅出现在指令中，在此限定下就没有它的运行时环境。

归纳：中间语言代码中出现的名字都分配在它们的声明宿主的最新栈帧里

栈帧方案



- 栈帧往地址减小方向生长
- 栈往地址减小的方向生长（图示地址习惯为下小上大）
- push/pop是基于sp的
- sp指向栈顶单元；
- fp指向栈帧对象；
- 若fp和sp指向同一个栈帧，其就是当前栈帧。

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

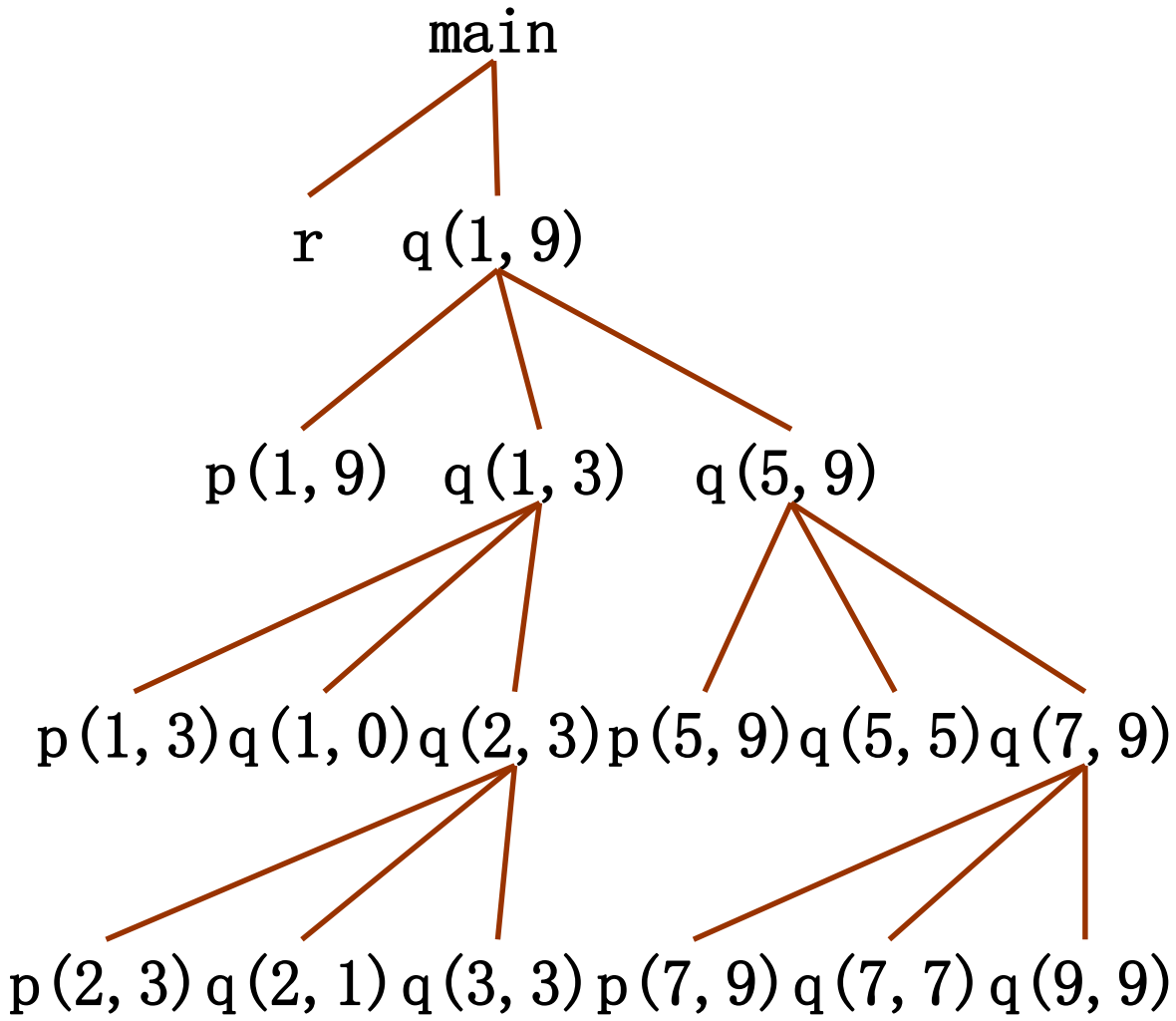
- 程序描述
  - main()调用gcd()
  - gcd()是递归的
- 则会有如下现象
  - 某时刻，活着的main活动嵌套着活着的gcd活动
  - 某时刻，一个活着的gcd活动嵌套着另一个活着的gcd活动
  - 某时刻，活着的gcd活动只有一个
  - 某时刻，活着的main活动不嵌套任何gcd活动
- 栈快照可有同一函数的多个栈帧

```
#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

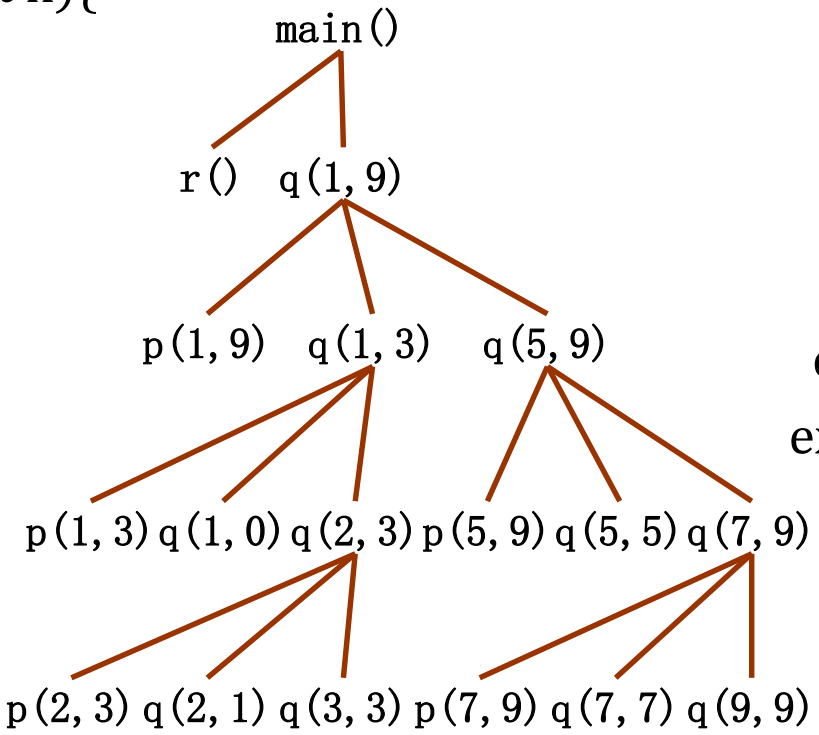



```
int a[11];
void readArray(){…
  int i;
  …}
int partition(int m, int n){
  …}
void quicksort(int m, int n){
  int i;
  if (n>m){
    i=partition(m, n);
    quicksort(m, i-1);
    quicksort(i+1, n);
  }}
main() {
  readArray();
  a[0]=-9999;
  a[10]=9999;
  quicksort(1,9);}
```

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)}
```

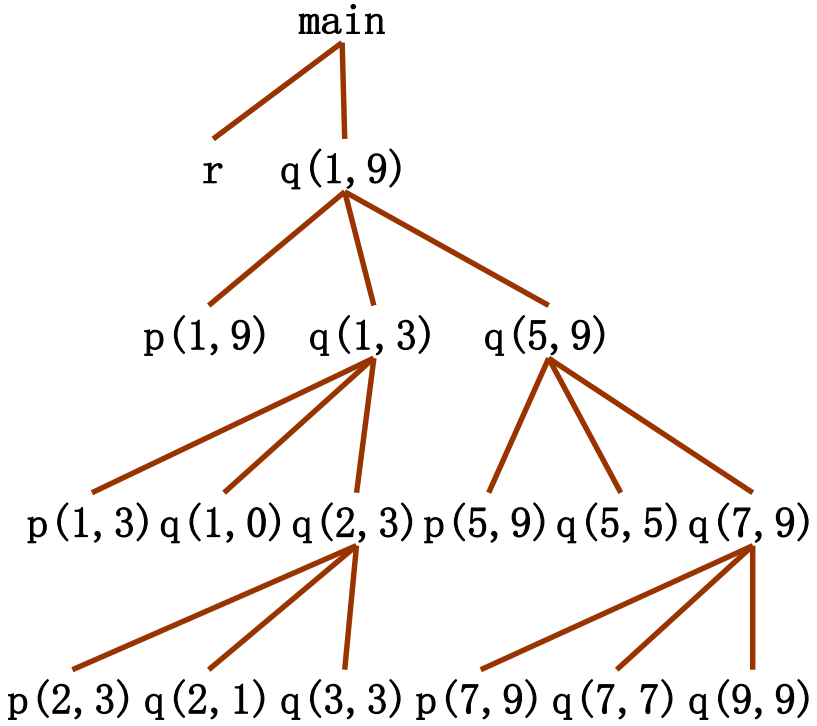
- 过程调用次序与先根遍历活动树一致；
- 过程返回次序与后根遍历活动树一致；




```
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>:(<参数2>:1 <参数1>:9) <链接区>:(<访问链> <控制链> <返址>) <局部区>:(m n i))
- 例：main()@frame:(<参数区>:NIL <链接区>:(<访问链> <控制链> <返址>) <局部区>:NIL)

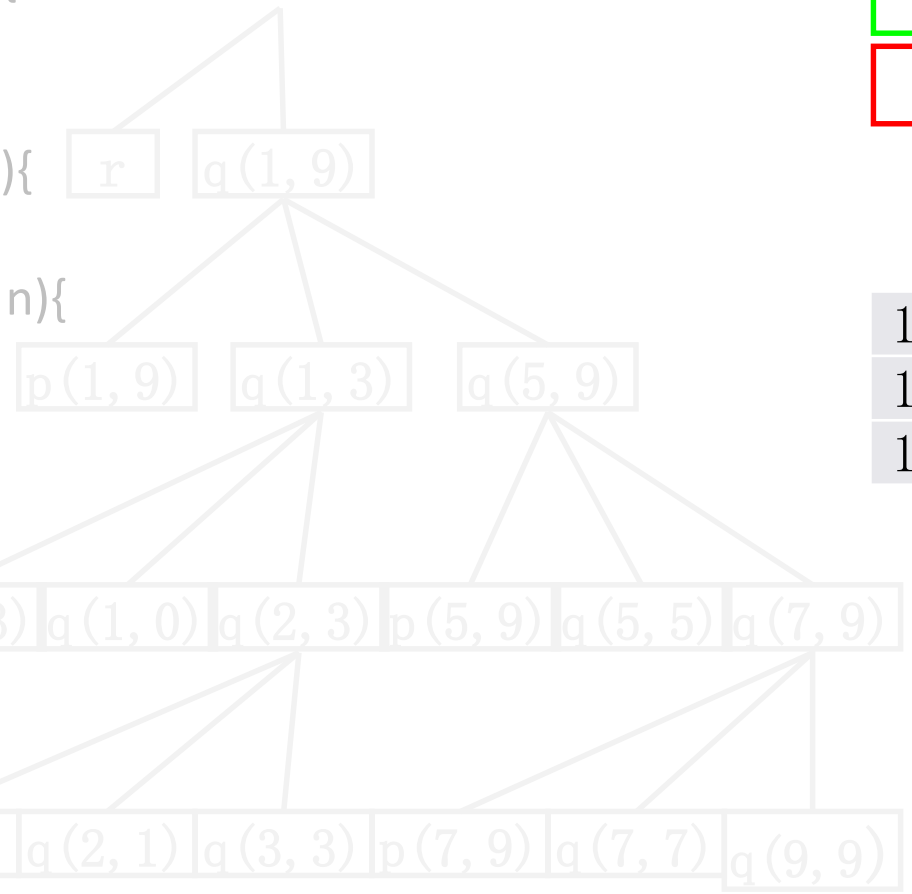
103	〈访问链〉
102	〈控制链〉
101	〈返址〉

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



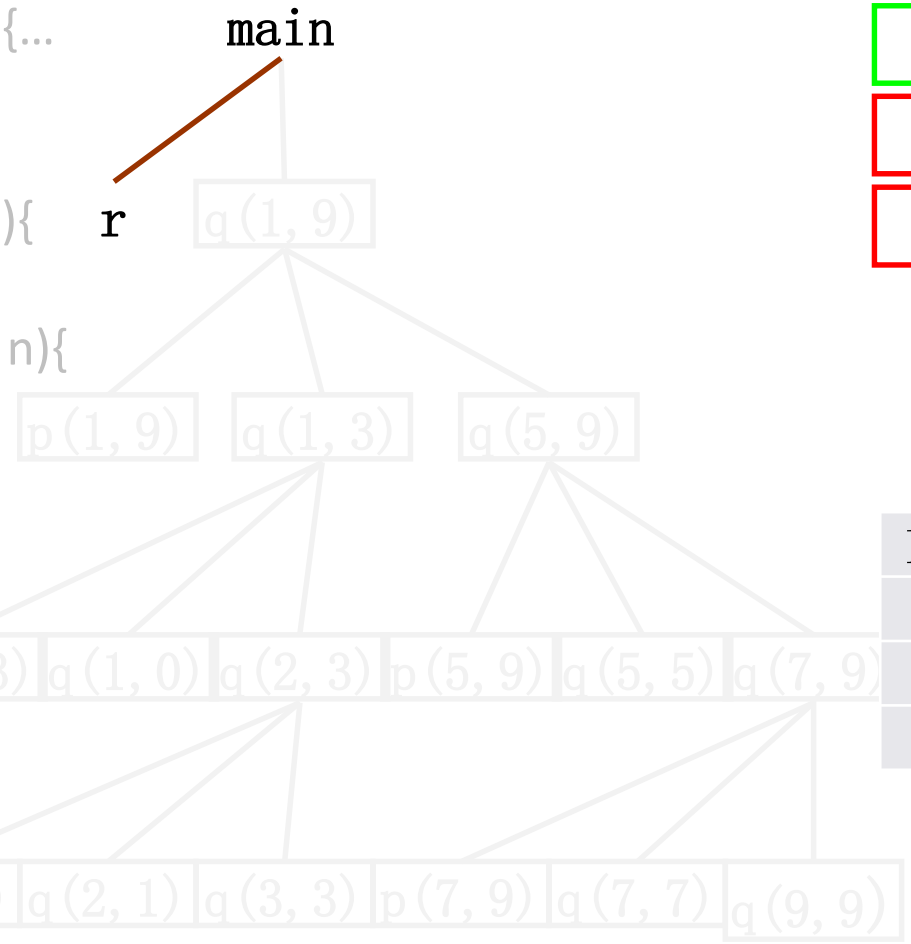
ε()@frame

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);}
```



ε()@frame

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);}
```

ε()@frame

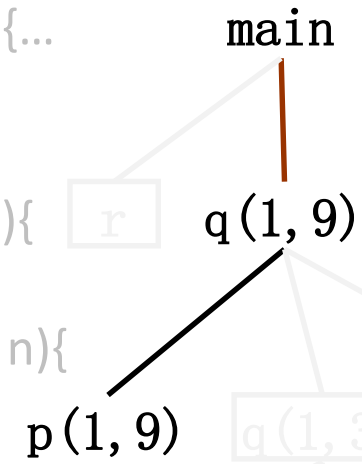
main()@frame

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);}
```



ε()@frame

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);}
```

ε()@frame

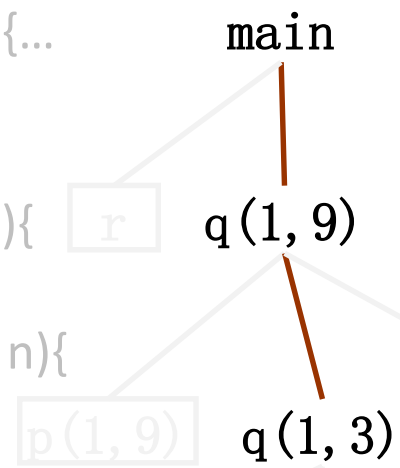
main()@frame

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);}
```



ε()@frame

main()@frame

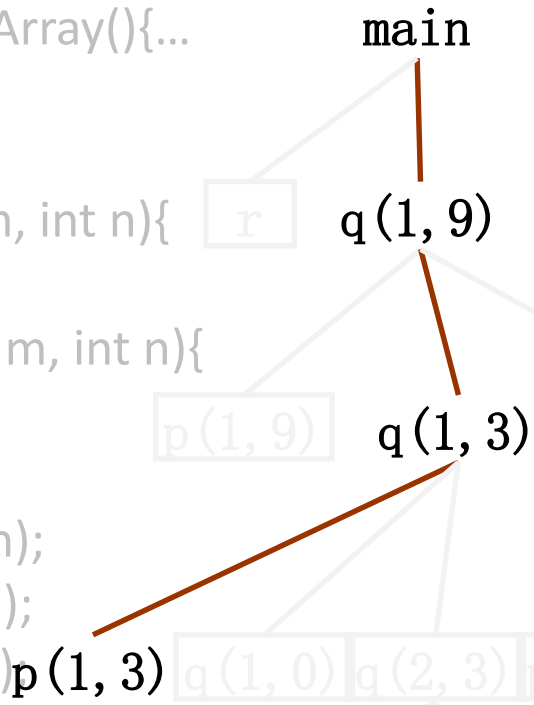
q(1,9)@frame

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);}
```

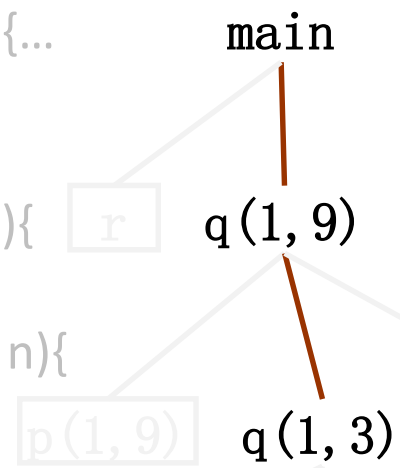


- ε()@frame
- main()@frame
- q(1,9)@frame
- q(1,3)@frame
- p(1,3)@frame

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);}
```



ε()@frame

main()@frame

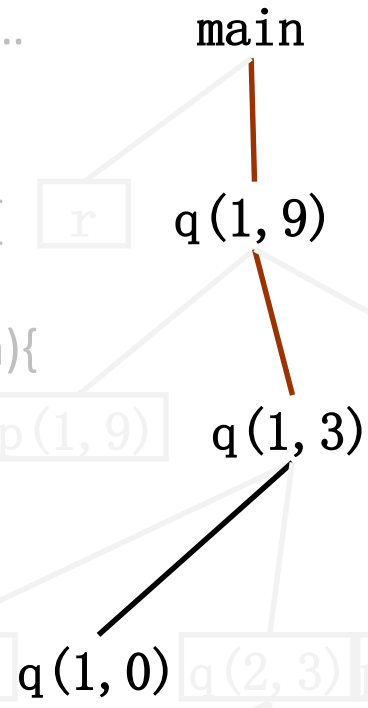
q(1,9)@frame

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);}
```



ε()@frame

main()@frame

q(1,9)@frame

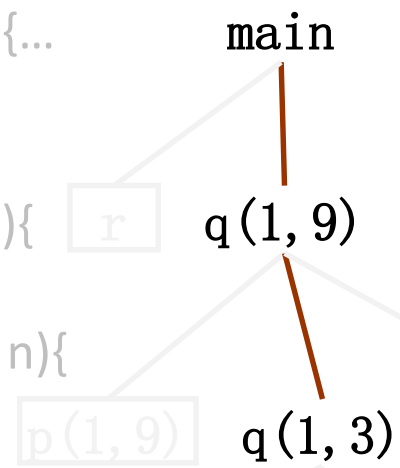
q(1,3)@frame

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);}
```



ε()@frame

main()@frame

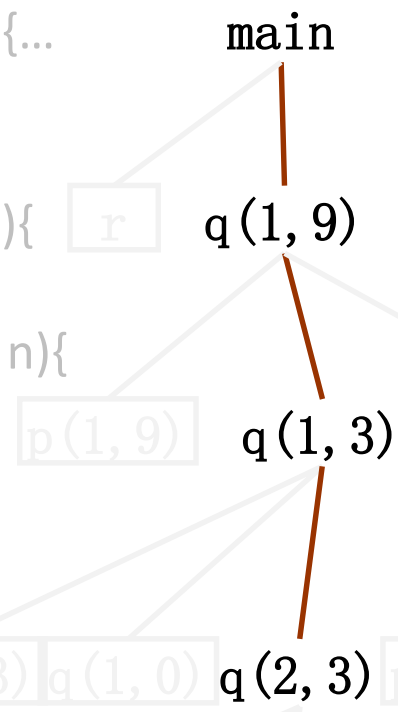
q(1,9)@frame

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);}
```



ε()@frame

main()@frame

q(1,9)@frame

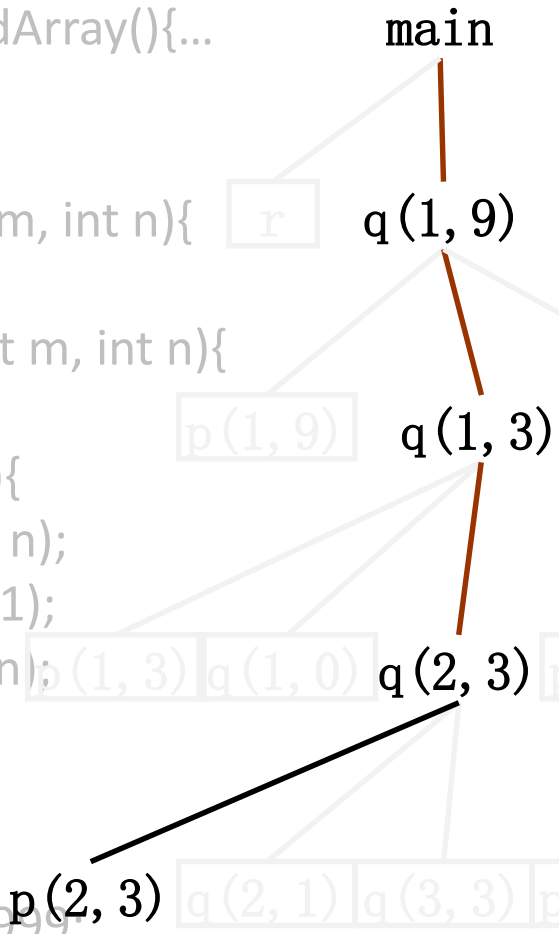
q(1,3)@frame

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);}
```



ε()@frame

main()@frame

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
80	〈返址〉
79	m
78	n
77	i
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

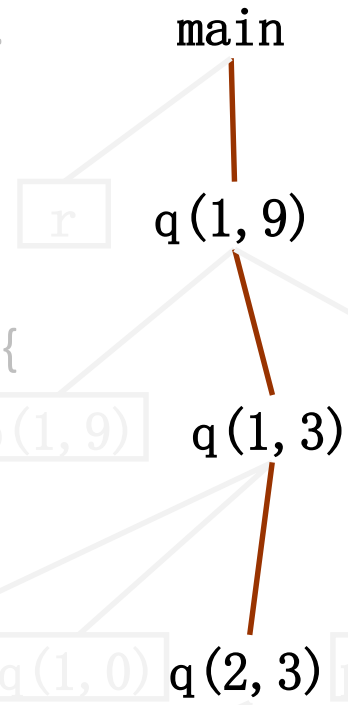
栈顶栈帧为当前栈帧，由此可知：fp=73（指向该栈帧）且sp=70（指向栈顶单元而且是当前栈帧的地址最小单元）

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);}
```



ε()@frame

main

q(1,9)@frame

q(1,3)@frame

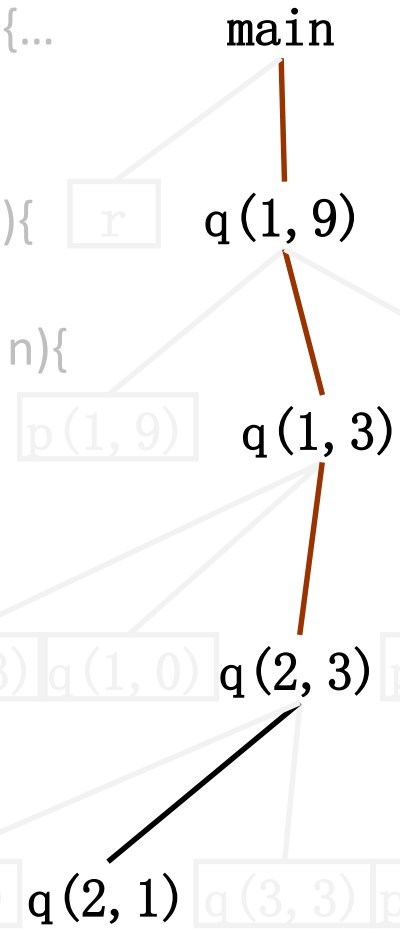
q(2,3)@frame

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);}
```

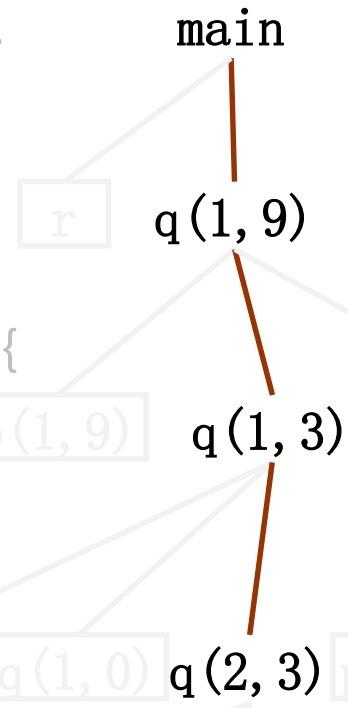


- ε()@frame
- main()@frame
- q(1,9)@frame
- q(1,3)@frame
- q(2,3)@frame
- q(2,1)@frame

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);}
```



ε()@frame

main()@frame

q(1,9)@frame

q(1,3)@frame

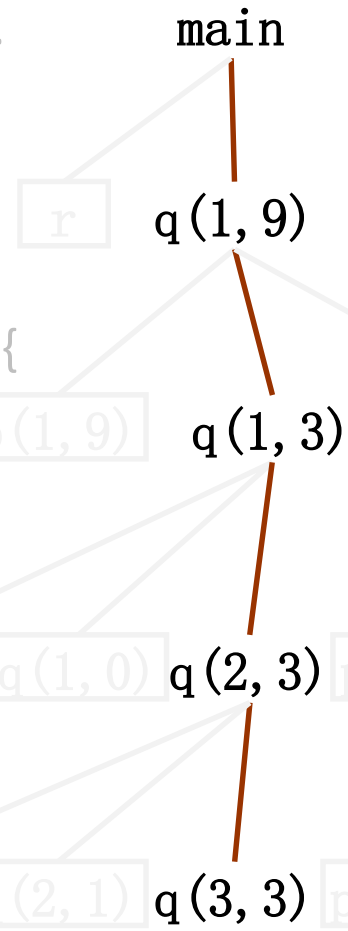
q(2,3)@frame

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);}
```

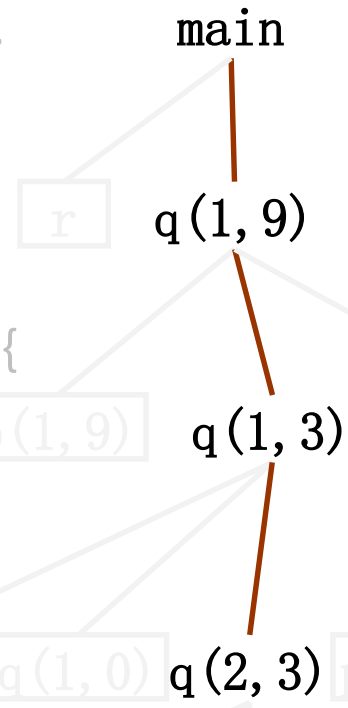


- ε()@frame
- main()@frame
- q(1,9)@frame
- q(1,3)@frame
- q(2,3)@frame
- q(3,3)@frame

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);}
```



ε()@frame

main()@frame

q(1,9)@frame

q(1,3)@frame

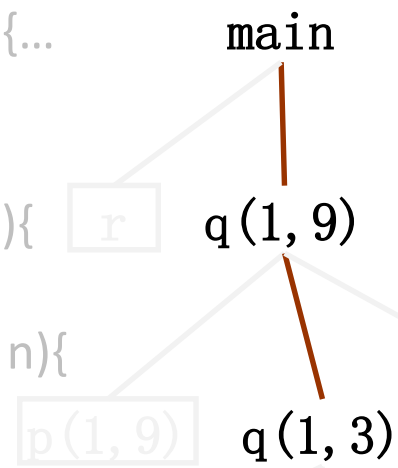
q(2,3)@frame

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);}
```



ε()@frame

main()@frame

q(1,9)@frame

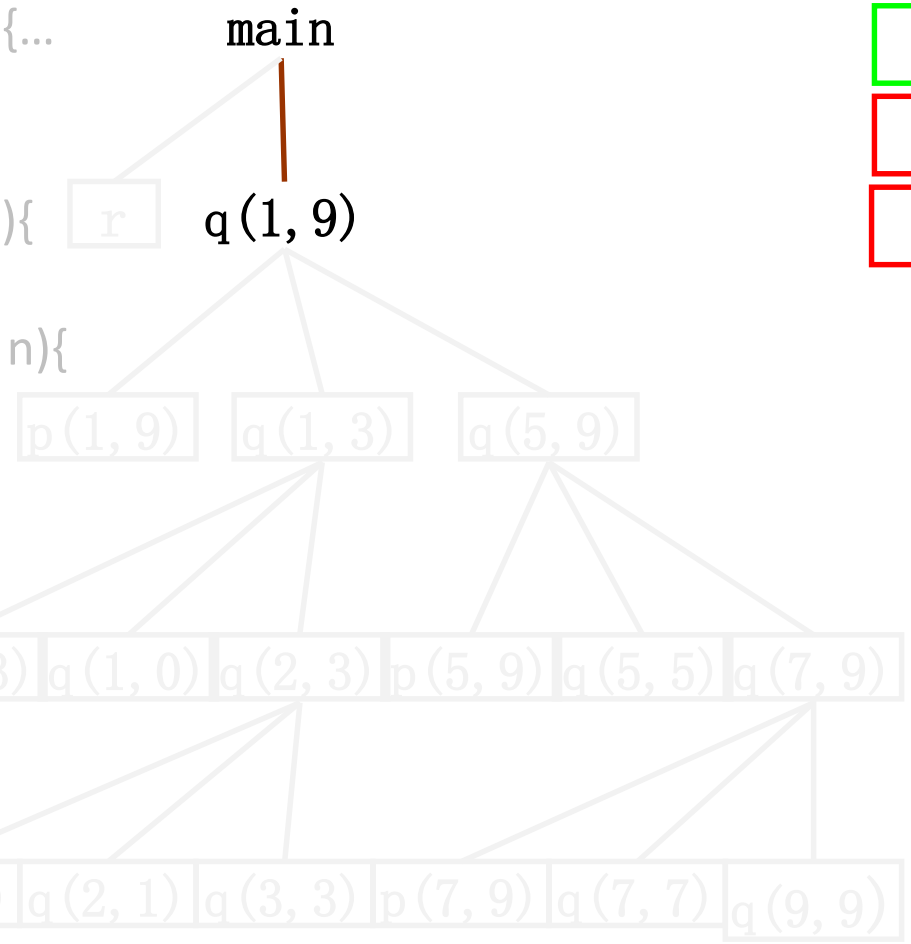
q(1,3)@frame

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);}
```



ε()@frame

main()@frame

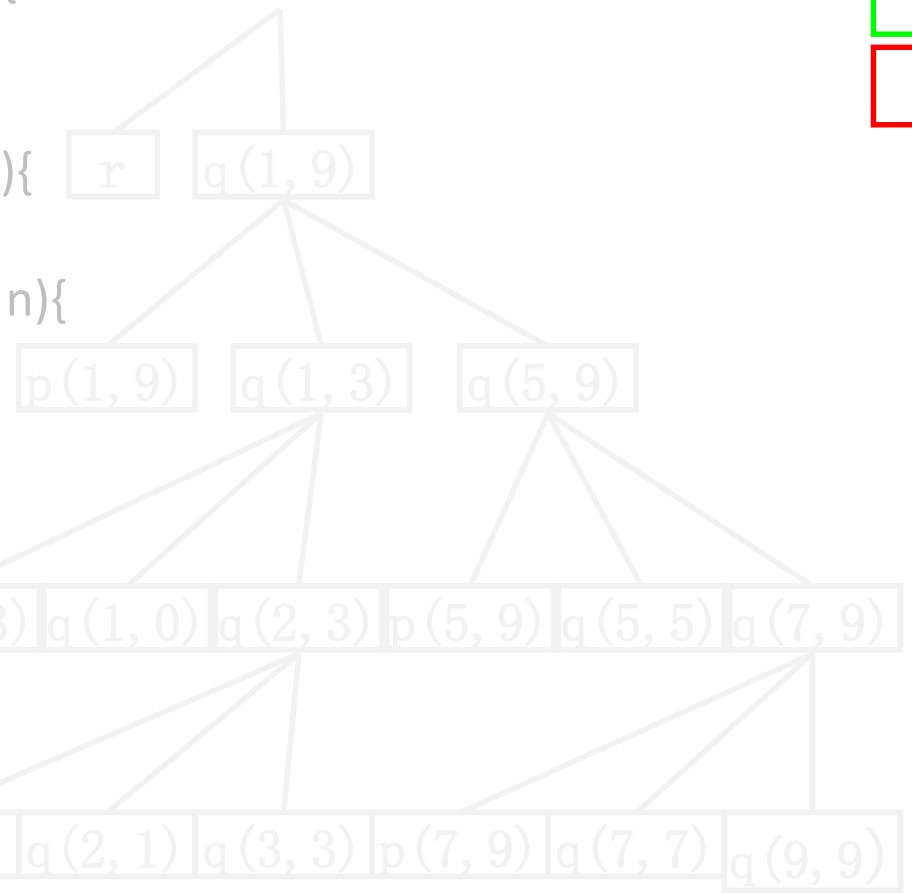
q(1,9)@frame

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);}
```



ε()@frame

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
- 过程的活动记录在它被调用时建立，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
```

允许小整数直接出现在指令中
参与模式替换？
以下红色也允许？
蓝色也可略去？

```
ε@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

ε()

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

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。

@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]

<访问链>指向词法上嵌套外层（即声明宿主）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。

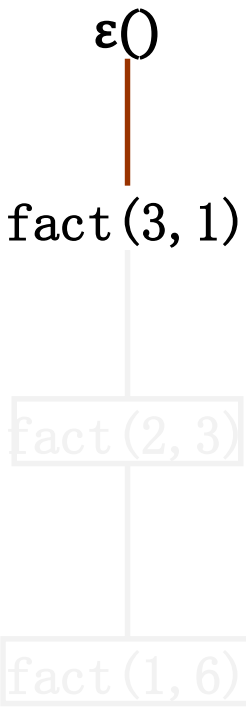
g()的一个活动记录g()@frame的访问链单元指向最新f()@frame。

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

@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]

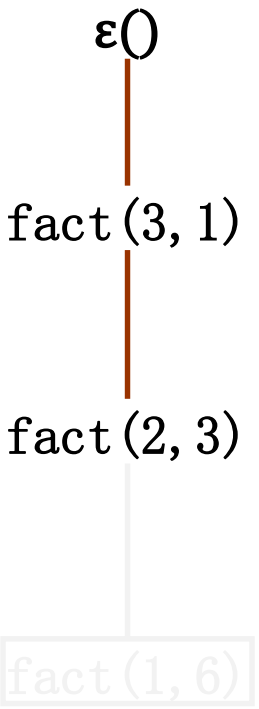
主调ε()负责创建被调fact(3,1)的栈帧
所以被调的控制链总是指向主调的栈帧。
对应代码
[PAR t6; PAR t5;
t7=CALL fact, 2]

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

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。

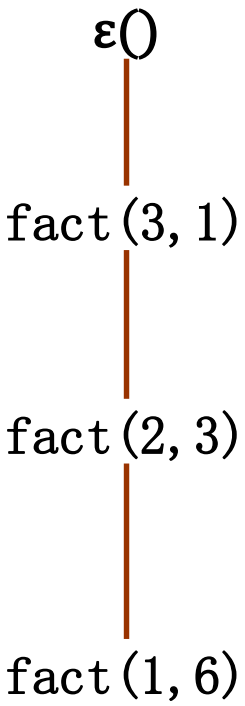
栈快照

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=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

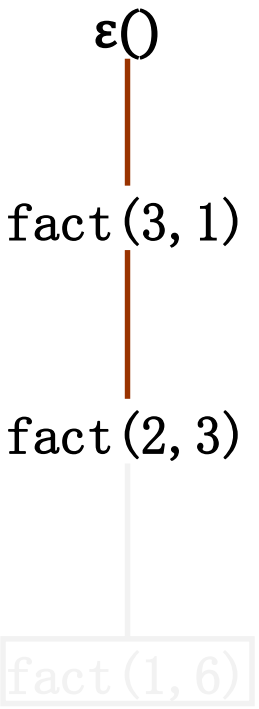
80	〈参数2〉:3
79	〈参数1〉:2
78	〈访问链〉:99
77	〈控制链〉:87
76	〈返址〉
75	n:2
74	a:3
73	t1:1
72	t2:6
71	t3
70	〈参数2〉:6
69	〈参数1〉:1
68	〈访问链〉:99
67	〈控制链〉:77
66	〈返址〉
65	n:1
64	a:6
63	t1
62	t2
61	t3

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

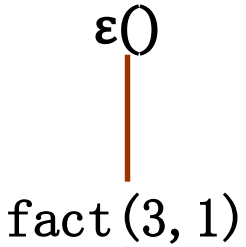
80	〈参数2〉:3
79	〈参数1〉:2
78	〈访问链〉:99
77	〈控制链〉:87
76	〈返址〉
75	n:2
74	a:3
73	t1:1
72	t2:6
71	t3:6

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。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



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

@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(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]

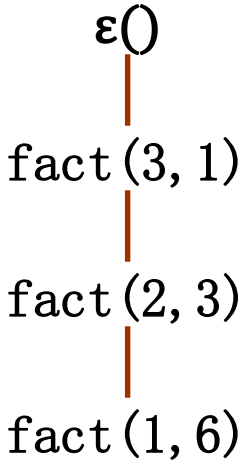
fact@code=[IF n==1
THEN l1 ELSE l2;LABEL l1;
RETURN a;GOTO l3;
LABEL l2;t1=n-1; t2=n*a;
PAR t2; PAR t1;
t3=CALL fact, 2;
RETURN t3;LABEL l3]

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

活动记录（生存期各单元取值情况）



@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]

@frame:(<访问链>:NIL <控制链>:NIL <返址> x:129 fact[1]: @frame fact[0]:fact@label t4:123 t5:3 t6:1 t7:6)

fact(3,1)@frame:(<参数2>:1 <参数1>:3 <访问链>:@frame <控制链>:@frame <返址> n:3 a:1 t1:2 t2:3 t3:6)

fact(2,3)@frame:(<参数2>:3 <参数1>:2 <访问链>:@frame <控制链>:fact(3,1)@frame <返址> n:2 a:3 t1:1 t2:6 t3:6)

fact(1,6)@frame:(<参数2>:6 <参数1>:1 <访问链>:@frame <控制链>:fact(2,3)@frame <返址> n:1 a:6 t1:_ t2:_ t3:_)

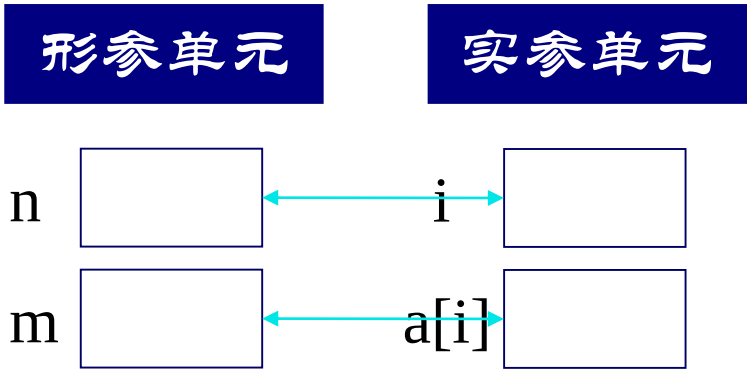
11.3 参数传递

- 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。

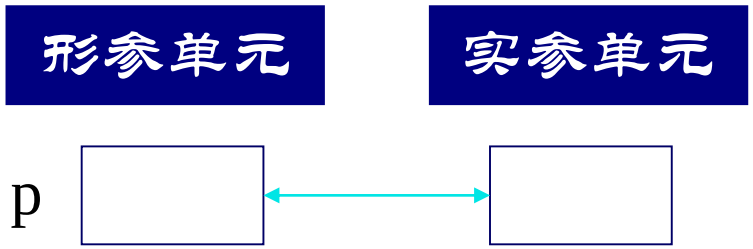


swap(i,a[i]);

```
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			

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 }
```

@table:(outer:NIL width:36 argc:0 arglist:NIL rtype:NIL level:0 code:[t6=CALL main, 0] entry:(name:i type:INT offset:4) entry:(name:A type:ARRAY base:16 dims:1 dim[0]:3 etype:INT) entry:(name:Q type:FUNC offset:24 mytab:Q@table) entry:(name:main type:FUNC offset:32 mytab:main@table) entry:(name:t6 type:TEMP offset:36))

Q@table:(outer:@table width:8 argc:1 arglist:(B) rtype:VOID level:1 code:[t1=1; A[t1]=3; i=2; write B; B=5] entry:(name:B type:INT offset:4) entry:(name:t1 type:TEMP offset:8))

main@table:(outer:@table width:20 argc:0 arglist:NIL rtype:INT level:1 code:[i=1; t2=1; A[t2]=2; t3=2; A[t3]=4; t4=A[i]; PAR t4; t5=CALL Q, 1] entry:(name:t2 type:TEMP offset:4)... entry:(name:t5 type:TEMP offset:20)

例：参数传递（传值、传址）

@table:(outer:NIL width:36 argc:0 arglist:NIL rtype:NIL level:0 code:[t6=CALL main, 0] entry:(name:i type:INT offset:4) entry:(name:A type:ARRAY base:16 dims:1 dim[0]:3 etype:INT) entry:(name:Q type:FUNC offset:24 mytab:Q@table) entry:(name:main type:FUNC offset:32 mytab:main@table) entry:(name:t6 type:TEMP offset:36))

Q@table:(outer:@table width:8 argc:1 arglist:(B) rtype:VOID level:1 code:[t1=1; A[t1]=3; i=2; write B; B=5] entry:(name:B type:INT offset:4) entry:(name:t1 type:TEMP offset:8))

main@table:(outer:@table width:20 argc:0 arglist:NIL rtype:INT level:1 code:[i=1; t2=1; A[t2]=2; t3=2; A[t3]=4; t4=A[i]; PAR t4 t5=CALL Q, 1] entry:(name:t2 type:TEMP offset:4)... entry:(name:t5 type:TEMP offset:20)

@frame:(alink:NIL clink:NIL raddr i:1/2 A[2]:4 A[1]:2/3 A[0] Q[1]:Q@label Q[0]:@frame main[1]:main@label main[0]:@frame t6:_)

main()@frame:(alink:@frame clink:@frame raddr t2:1 t3:2 t4:2 t5:)

Q(2)@frame:(arg1:2 alink:@frame clink@main()@frame raddr B:2/5 t1:1)

@frame:(alink:NIL clink:NIL raddr i:1/2 A[2]:4 A[1]:2/3/5 A[0] Q[1]:Q@label Q[0]:@frame main[1]:main@label main[0]:@frame t6:0)

main()@frame:(alink:@frame clink:@frame raddr t2:1 t3:2 t4:2 t5:_)

Q(&A[1])@frame:(arg1:&A[1] alink:@frame clink@main()@frame raddr B:2/3 t1:1)

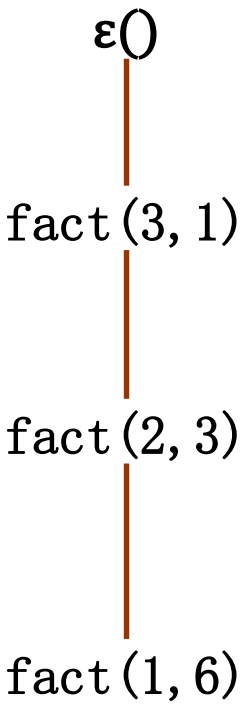
传地址举例

ε@frame

fact(3,1)@frame

fact(2,3)@frame

fact(1,6)@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
```

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〉:92
89	〈参数1〉:93
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〉:82
79	〈参数1〉:83
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〉:72
69	〈参数1〉:73
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

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，可将其它机制作为拓展。

例：形参类型为FUNPTT、ARRPTT

```
int z;
int a[10,20];
int bar(int x;){
  return x+1};
float foo(int x; int b[];
          int boo();){
  if (x>0) z=0
  else return boo(0,)};
print foo(0, a[],bar(),)
```

```
@table:(outer:NIL width:828 argc:0 arglist:NIL rtype:INT
level:0 code:[t5=0; PAR bar; PAR a; PAR t5;
t6=CALL foo@label, 3; print t6]
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)
entry:(name:t5 type:TEMP offset:824)
entry:(name:t6 type:TEMP offset:828))
```

```
foo@table:(outer:@table width:24 argc:3
arglist:(x b boo) rtype:FLO level:1
code:[IF x>R0 THEN l1 ELSE l2; LABEL l1; z=R0; GOTO l3;
LABEL l2;t3=0; PAR t3; 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)
entry:(name:t3 type:TEMP offset:20) entry:(name:t4
type:TEMP offset:24))
```

```
bar@table:(outer:@table width:12
argc:1 arglist:(x) rtype:INT level:1
code:[t1=1; t2=x+t1; RETURN t2]
entry(name:x type:INT offset:4)
entry:(name:t1 type:TEMP offset:8)
entry:(name:t2 type:TEMP offset:12)
)
```

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:NIL width:828 argc:0 arglist:NIL rtype:INT
level:0 code:[t5=0; PAR bar; PAR a; PAR t5;
t6=CALL foo@label, 3; print t6]
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)
entry:(name:t5 type:TEMP offset:824)
entry:(name:t6 type:TEMP offset:828))
```

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 bar; PAR a; PAR t5; t6=CALL foo@label, 3; print t6]

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	t3:_
279	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]:499
294	foo[1]:foo@label
293	foo[0]:_
292	t5:0
291	t6:_

foo@table:(outer:@table width:24 argc:3 arglist:(x b boo)
rtype:FLO level:1 code:[IF x>R0 THEN l1 ELSE l2; LABEL l1; z=R0; GOTO l3; LABEL l2;t3=0;
PAR t3; 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)
entry:(name:t3 type:TEMP offset:20) entry:(name:t4 type:TEMP offset:24))

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279单元没有值，随后bar返回时就有值了。

[PAR t3; t4=CALL boo, 1]

bar@table:(outer:@table width:12
argc:1 arglist:(x) rtype:INT level:1
code:[t1=1; t2=x+t1; RETURN t2]
entry(name:x type:INT offset:4)
entry:(name:t1 type:TEMP offset:8)
entry:(name:t2 type:TEMP offset:12))

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	t3:0
279	t4:1
278	<参数1>:0
277	<访问链>
276	<控制链>
275	<返址>
274	x:0
273	t1:1
272	t2:1

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 $r]将控制流回到foo中（红色CALL的下条指令）。
- bar栈帧剩下的部分276-278是由主调foo来释放的，因为这部分是它创建的。约定为sp=fp=276
- 红色CALL的下条指令负责将$v0传送到279单元中，即赋给t4

[PAR t3; t4=CALL boo, 1]

```
bar@table:(outer:@table width:12 argc:1 arglist:(x)
rtype:INT level:1 code:[t1=1; t2=x+t1; RETURN t2]
entry(name:x type:INT offset:4)
entry:(name:t1 type:TEMP offset:8)
entry:(name:t2 type:TEMP offset:12))
```

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	t3:
279	t4:1
~~278~~	~~<参数1>:0~~
~~277~~	~~<访问链>~~
~~276~~	~~<控制链>~~
~~275~~	~~<返址>~~
~~274~~	~~x:0~~
~~273~~	~~t1:1~~
~~272~~	~~t2:1~~

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

~~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~~	~~t3:_~~
~~279~~	~~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]:499
294	foo[1]:foo@label
293	foo[0]:_
292	t5:0
291	t6:1

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

foo@table:(outer:@table width:24 argc:3 arglist:(x b boo)
rtype:FLO level:1 code:[IF x>R0 THEN l1 ELSE l2; LABEL l1; z=R0; GOTO l3; LABEL l2;t3=0; PAR t3; 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)
entry:(name:t3 type:TEMP offset:20) entry:(name:t4 type:TEMP offset:24))

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]:499
294	foo[1]:foo@label
293	foo[0]:_
292	t5:0
291	t6:1

```
@table:(outer:NIL width:828 argc:0 arglist:NIL rtype:INT
level:0 code:[t5=0; PAR bar; PAR a; PAR t5;
t6=CALL foo@label, 3; print t6]
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)
entry:(name:t5 type:TEMP offset:824)
entry:(name:t6 type:TEMP offset:828))
```

例：形参类型为FUNPTT、ARRPTT

@table:(...code:[t5=0; PAR bar; PAR a; PAR t5; t6=CALL foo@label, 3; print t6] ...z type:INT...a type:ARRAY...dims:2 dim[0]:10 dim[1]:20...bar type:FUNC.....foo type:FUNC.....t5 ...t6...)

foo@table:(...arglist:(x b boo)...code:[IF x>R0 THEN l1 ELSE l2; LABEL l1; z=R0; GOTO l3; LABEL l2;t3=0; PAR t3; t4=CALL boo, 1; RETURN t4;LABEL l3] ... x type:INT... b type:ARRPTT... boo type:FUNPTT... t3...t4)

bar@table:(... arglist:(x)... code:[t1=1; t2=x+t1; RETURN t2]... x type:INT... t1...t2)

@frame:(alink:NIL clink:NIL raddr z:0 a[9,19]... a[0,0] bar[1]:bar@label bar[0]:@frame foo[1]:foo@label foo[0]:_ t5:0 t6:1)

foo(0,a,bar)@frame:(arg3:&boo[0] arg2:&a[0,0] arg1:0 alink@frame clink:@frame raddr x:0 b:&a[0,0] boo[1]:bar@lable boo[0]:@frame t3:0 t4:1)

bar@frame:(arg1:0 alink:@frame clink:foo(0,a,bar)@frame raddr x:0 t1:1 t2:1)

```
int z;
int a[10,20];
int bar(int x;){return x+1};
float foo(int x; int b[]; int boo();){
  if (x>0) z=0  else return boo(0,)};
print foo(0, a[],bar(),)
```

构建可执行代码g@label

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

[$v0=t1; j g@epilogue]

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

[sp=sp-4; M[sp]=t_m;...;
sp=sp-4; M[sp=t_1]++
[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)]

g()的引用宿主f()为g()创建活动记录

- f@code包含[PAR t_m; ...; PAR t_1; 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和fp都指向<控制链>，执行调子指令控制流进入g@code（已转换为g@label）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$ra压栈；把参数赋给形参；sp=sp-g@width
- 这部分被称为序言g@prologue

//调用序列

[sp=sp-4; M[sp]=t_m;...;
sp=sp-4; M[sp=t_1]++
[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)]

//设g@arglist=(a_1 ... a_m)
[sp=sp-4; M[sp]=$a
a_1=M[fp+8]; ...
a_m=M[fp+8+(m-1)*4]
sp=sp-g@width]

g()释放自己栈帧并将控制流转回f()

- 转换g@code中[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]类指令，所以就将共同部分作为g代码的尾声
- 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。

```
//返回序列
[$v0=t1; j g@epilogue]

//尾声
g@epilogue=[
sp=sp+g@width
sp=sp+4; $t2=M[sp]
jr $t2]
```

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。

调用序列callseq与返回序列retseq

- 将[PAR tm;...;PAR t1;v=CALL foo,m]转换为callseq：
  - 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。

- 将[RETURN t]转换为retseq：
  - 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]

11.4 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
- 写出活动树及各个活动记录

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()
```

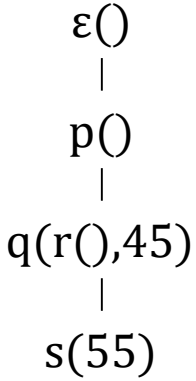
500	〈访问链〉:0	ε()
499	〈控制链〉:0	
498	〈返址〉	
497	x:15	运行结果
496	y:21	运行结果
495	q[1]:q@label	
494	q[0]	
493	p[1]:p@label	
492	p[0]	
491	〈访问链〉:499	p()
490	〈控制链〉:499	
489	〈返址〉	
488	r[1]:**r@label**	由函数声明得出
487	r[0]:**490**	对实参求值结果

486	〈参数2〉:45	q(...)
485	〈参数1〉:480	参数传递到形参中
484	〈访问链〉:499	
483	〈控制链〉:490	
482	〈返址〉	
481	s[1]:**r@label**	rt=M[fp+8]+4;s=M[rt]
480	s[0]:**490**	rt=M[fp+8];s=M[rt]
479	x:45	x=M[fp+12]
478	y	随后为76
477	〈参数1〉:55	r(55)
476	〈访问链〉:490	声明宿主恰同引用宿主
475	〈控制链〉:483	
474	〈返址〉	
473	x:55	
472	z:76	第二行的y

活动树与活动记录

在活动生存期中它的栈帧各单元内容及其变化的属性表（k-v表）表示

```
int x;
int y;
void q(int s(); int x;){
   int y;
   y=s(x+10,);
   print y};
int p(){
   int r(int x;){
      int z;
      z=x+y;
      return z};
   q(r(), x*3,)};
x=15;
y=21;
p()
```



@frame(alink:NIL clink:NIL raddr x:15 y:21 q[1]:q@label q[0]:_ p[1]:p@label p[0]:_ t:_)

p()@frame:(alink:@frame clink:@frame raddr r[1]:r@label r[0]:p()@frame t4:45 t5:_)

q(r(),45)@frame:(arg2:45 arg1:&s[0] alink:@frame clink:p()@frame raddr s[1]:r@label s[0]:p()@frame x:45 y:76 t1:55 t2:76)

r(55)@frame:(arg1:55 alink:p()@frame clink:q(r(),45)@frame raddr x:55

例：形参为数组原型、函数原型

- 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[6*x],)};
print foo(2, bar(), a[],)
```

例：形参为数组原型、函数原型

```
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[6*
x],)};
print foo(2, bar(), a[],)
```

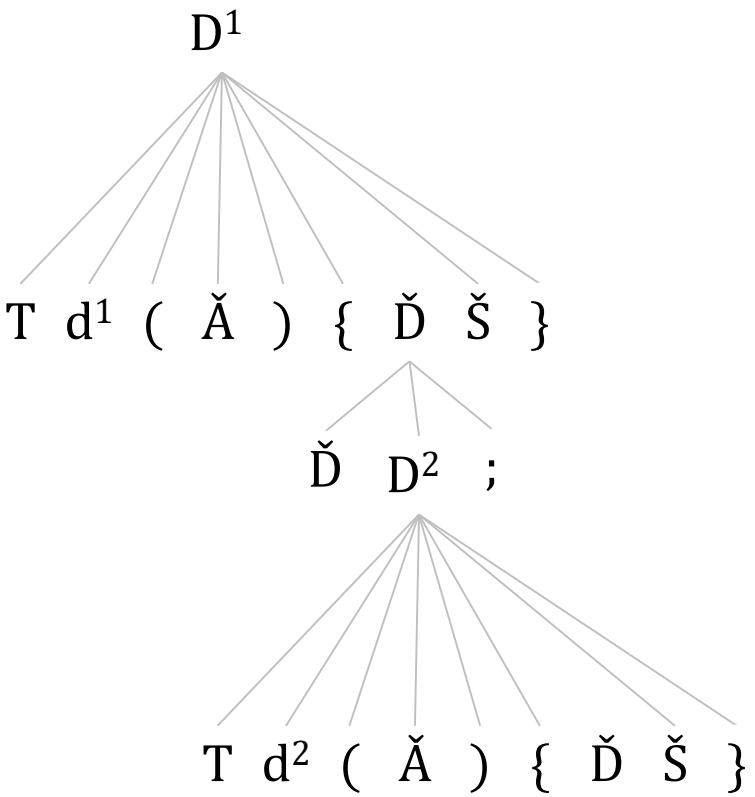
500	〈访问链〉:0
499	〈控制链〉:0
498	〈返址〉
497	x
496	z:0
495	a[9,19]:28
…	…
297	a[0,1]:1
296	a[0,0]:0
295	bar[1]:bar@label
294	bar[0]:499
293	foo[1]:foo@label
292	foo[0]:_

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	形参名单元值	实参值
形参名单元	参数单元值	复制自实参	实参单元值

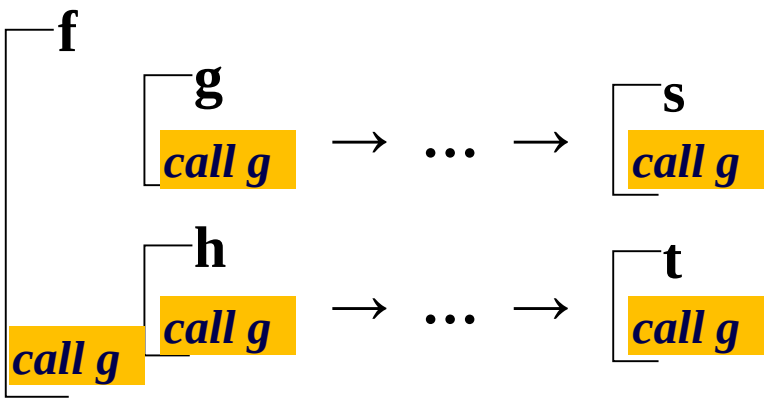
11.5构建访问链及访问非局部名字

- 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



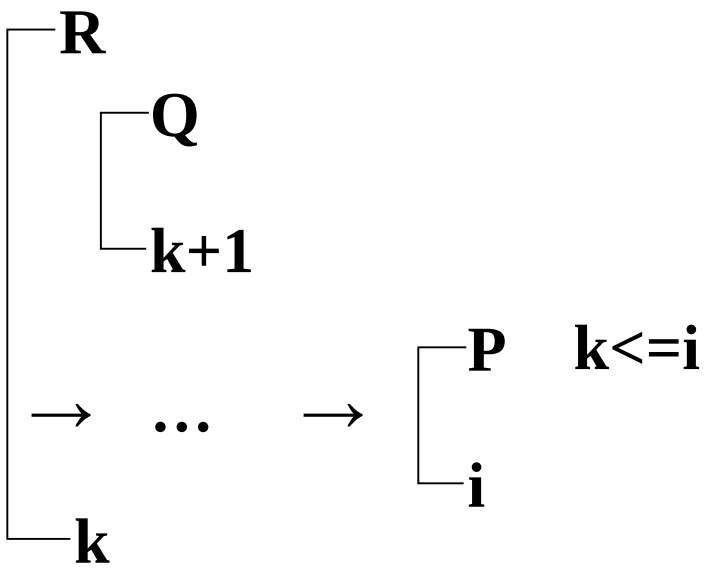
嵌套过程、作用域及嵌套层次

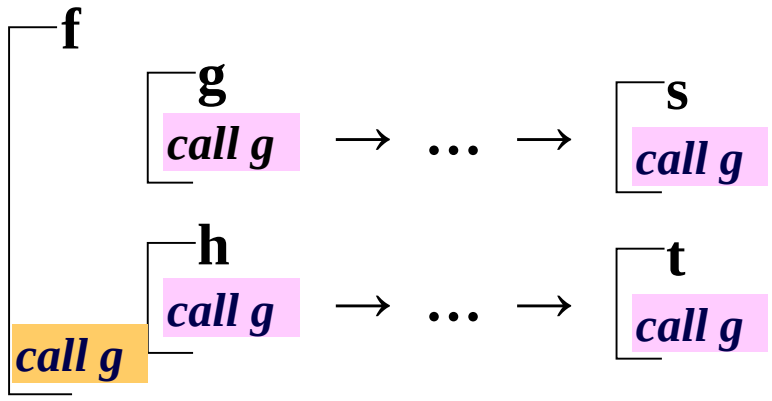
- 一个函数的名字的静态作用域是这个函数的所有引用宿主的集合
- Pascal语言的过程g()的静态作用域定义为：
  - g()的声明宿主f()；
  - g()及其声明子孙。即设为S，那么g()∈S，若r()的声明宿主为h()且h()∈S那么r()∈S。
  - 与g()有同一声明宿主的且在g()之后声明的函数以及它的声明子孙。



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

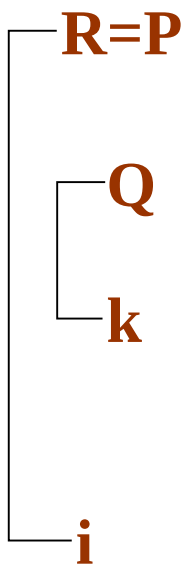
①R=P，即P是Q的声明宿主。

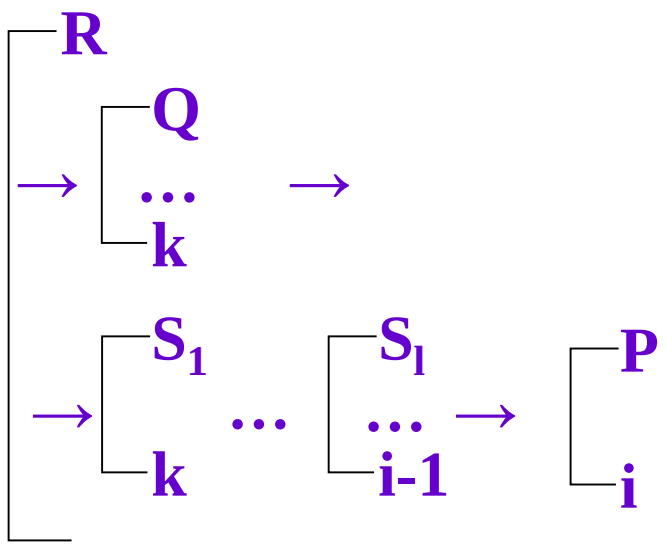
P的活动记录就是Q的声明宿主的最新活动记录。

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。

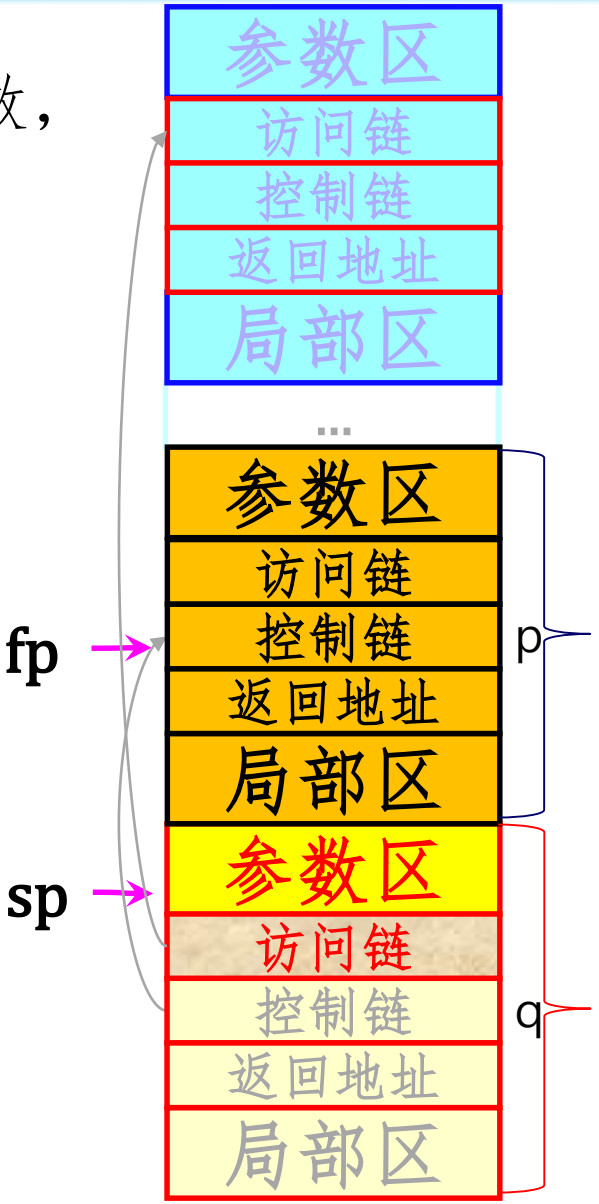
沿着主调P的活动记录的访问链走过i-k+1个（也就是找到了R的最新活动记录），那个活动记录作为Q的访问链所指。





主调p()构建被调q()的访问链单元

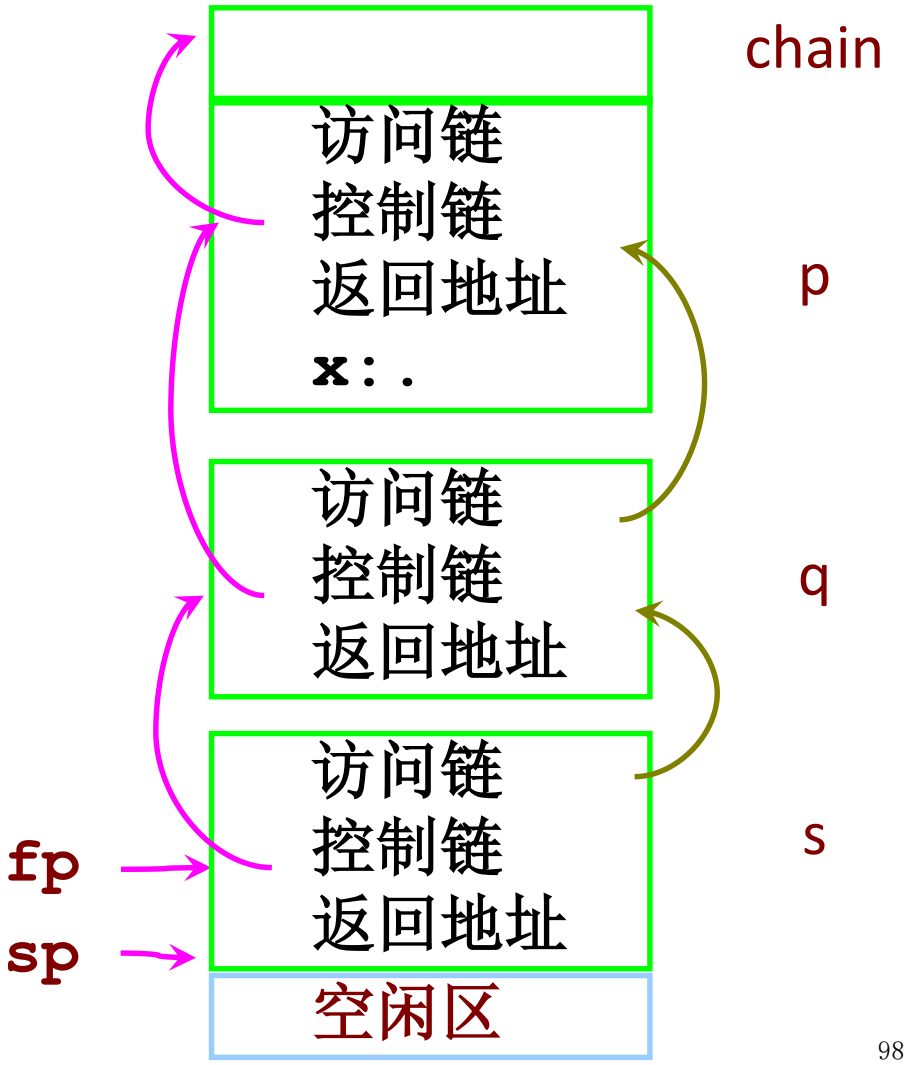
- tab为主调p的符号表，tab->level嵌套层数，sp指向参数区（参数区刚构建完成）
- tab_q=lookup1(tab, q, mytab);
  tab_r=tab_q->outer;
  if(tab_r!=tab){//情形②：i-k+1
  k=tab->level-tab_q->level+1;
  emit[rt=fp];
  for(i=0;i<k;i++)emit[rt=M[rt+4]];
  emit[sp=sp-4]; emit[M[sp]=rt]
  }else
  emit[push [fp]]//情形①：



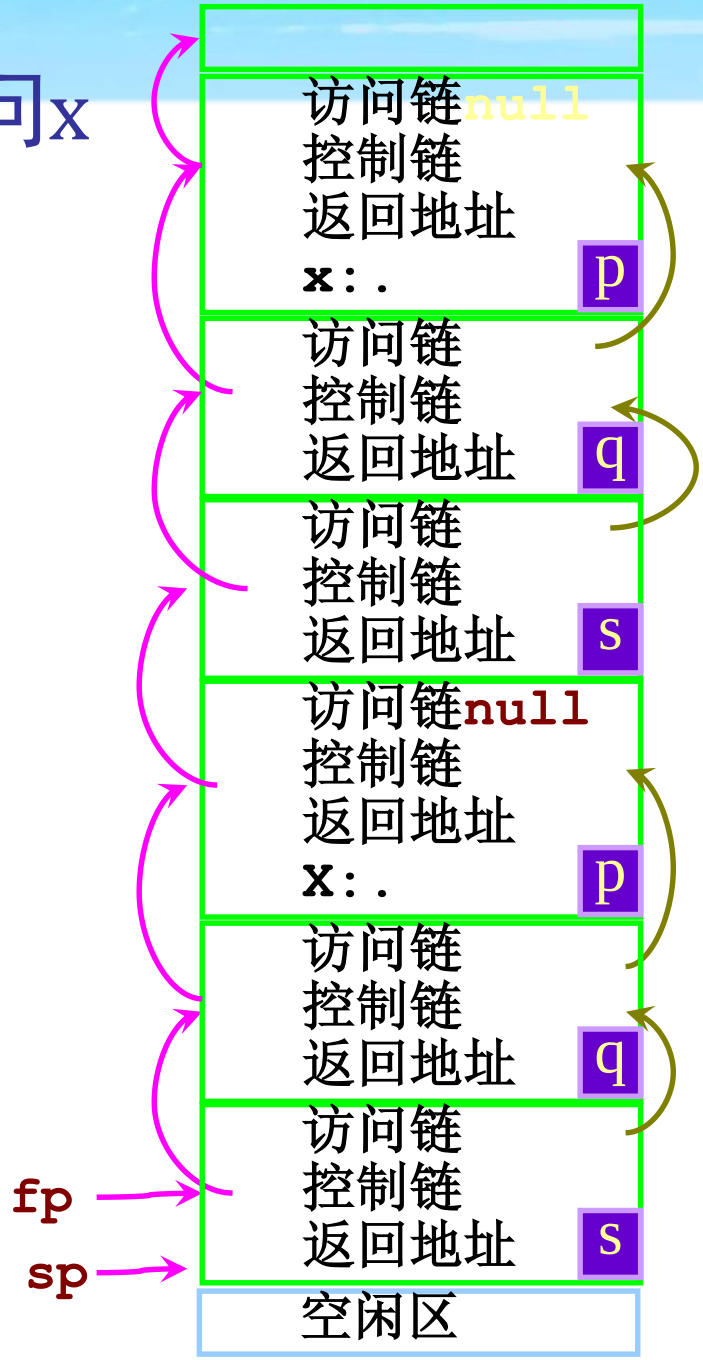
使用访问链对非局部名寻址

```
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

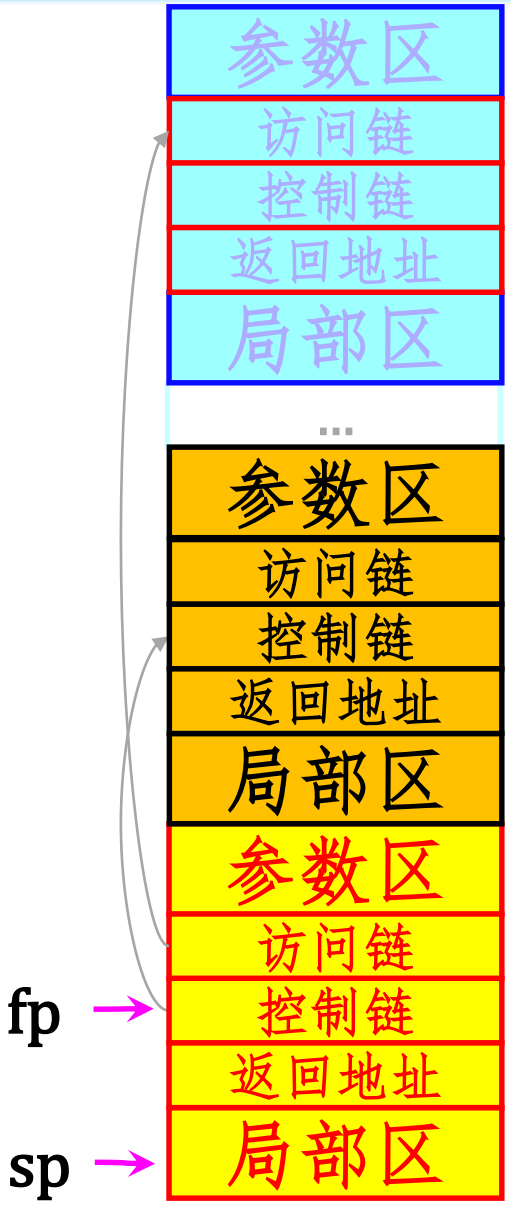
- t=x@value可转换为：
  [rt=fp; rt=M[rt+4]; rt=M[rt+4]; t=M[rt-8]]
- x@value=t可转换为：
  [rt=fp; rt=M[rt+4]; rt=M[rt+4]; M[rt-8]=t]

- 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。

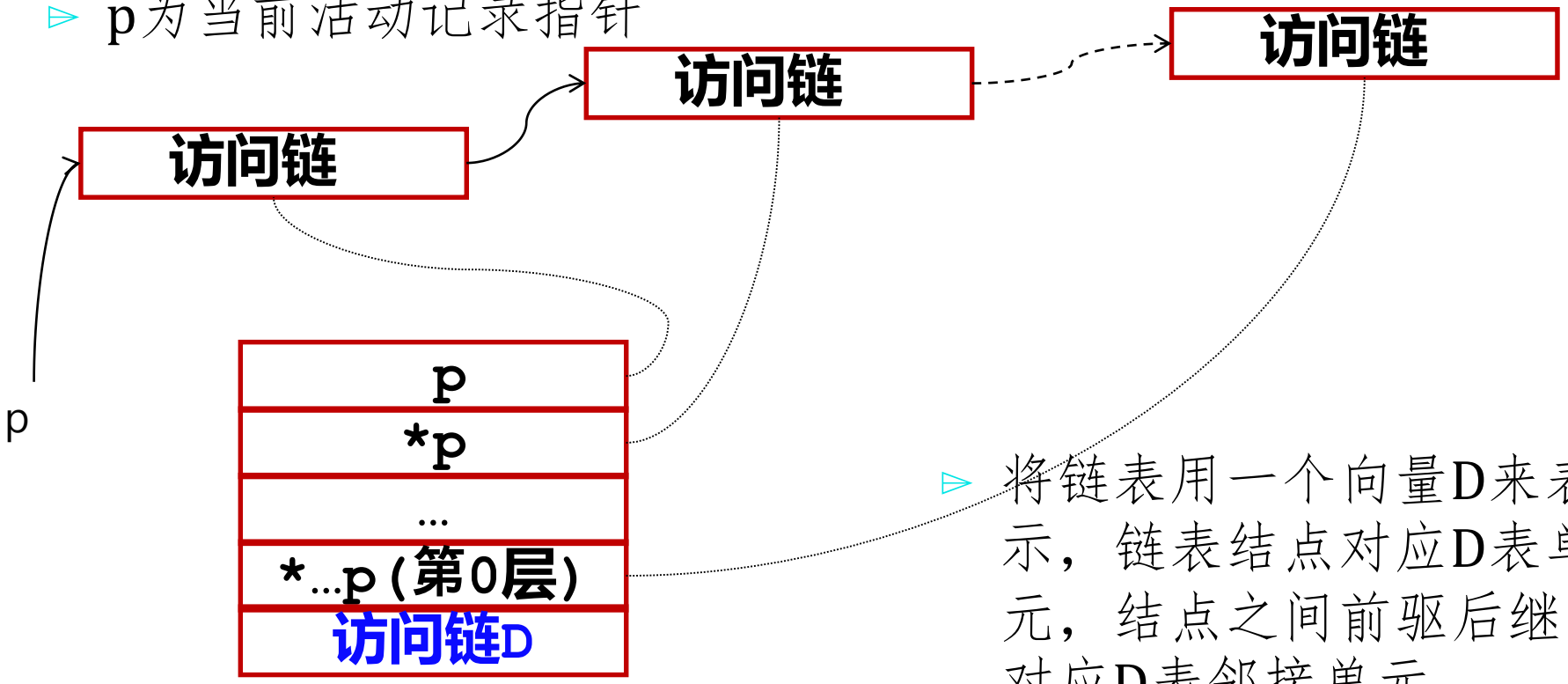
非局部名寻址

- 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



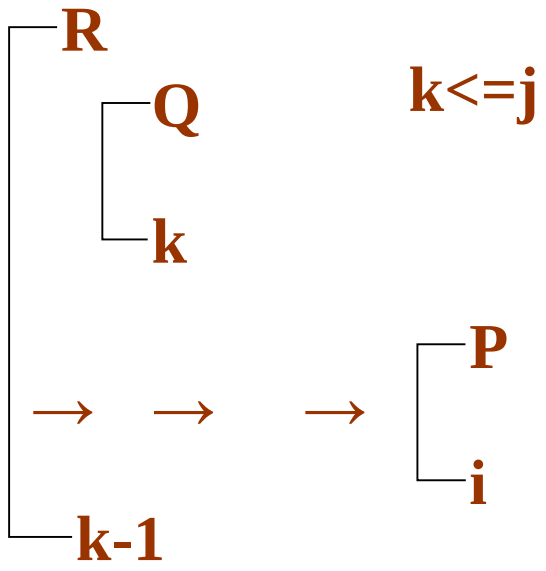
- 将链表用一个向量D来表示，链表结点对应D表单元，结点之间前驱后继对应D表邻接单元。
- 显示表D有k个元素都为指向活动记录的指针值，其中k为该过程嵌套层数

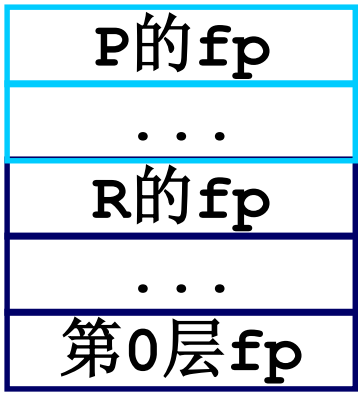
用主调的D表构造被调的D表1

- 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





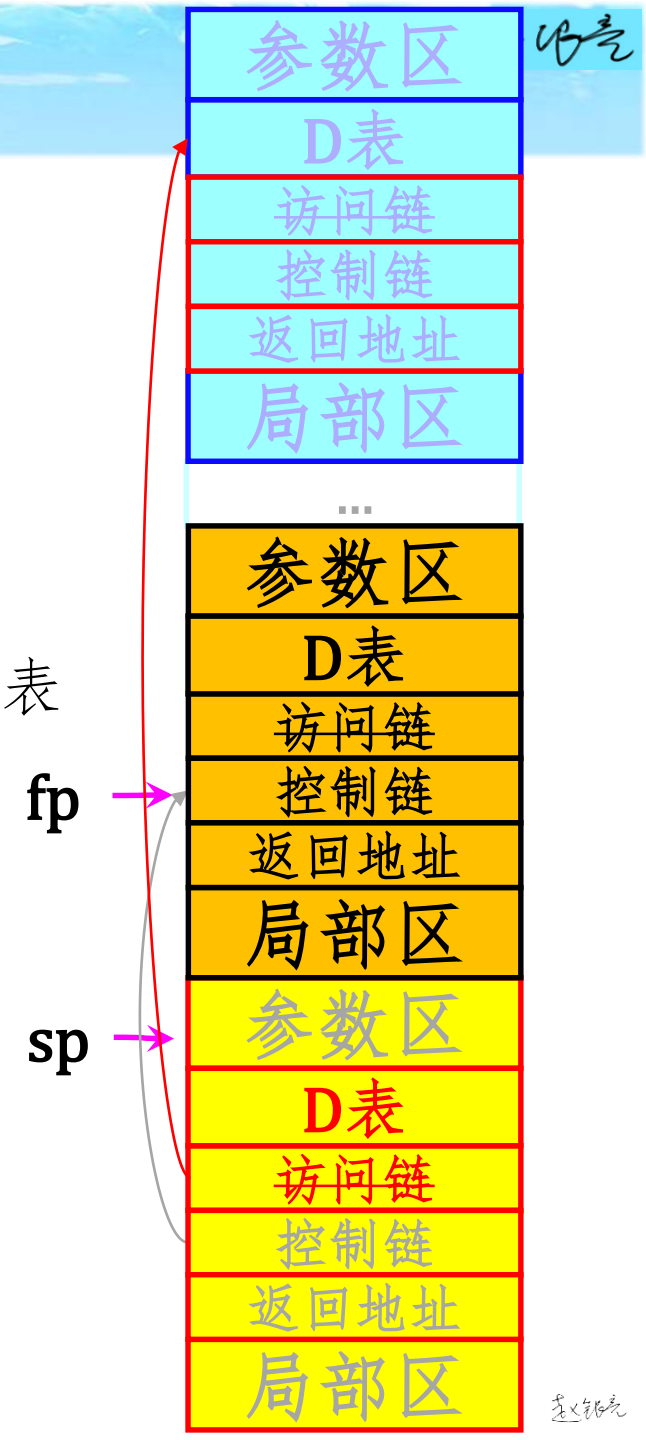
主调的D表

被调的D表

构建D表（p调用q）

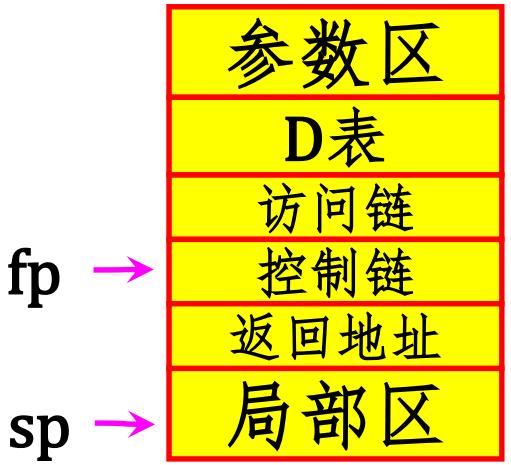
- sp指向黑参数区（参数区刚构建完成）

```
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

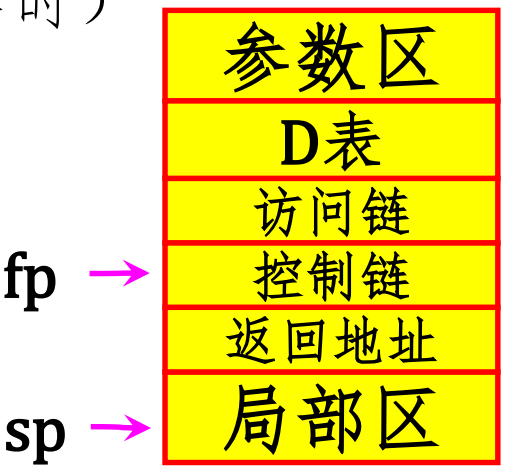
- 过程代码中，非局部名的访问地址
  - 绝对地址 = D[静态层数]+相对地址
  - 静态层数指定义那个非局部名的过程的层数
  - 相对地址为LL+偏移量



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...
```



11.6 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.
2nd edition: © Springer International Publishing AG 2017

11.7 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

- 1、习题11.2 额外包括：写出符号表、活动树、各活动记录
- 2、参照ppt第10~11页写出面向ARM32的指令模板（可选）
- 3、试针对ppt第91页程序写出所生成的可执行程序（使用MIPS或ARM指令）
- 以上作业提交时间6月9日（雨课堂）

- 大作业（六）提交时间6月26日（邮箱）
- 写程序对下列QL程序生成可执行代码（使用MIPS指令或使用ARM指令）

QL程序样例

```
int x;
int a[2, 3];
void q(int s(); int b[]; int x;){
        int y;
        y=s(x+10,);
        print y};
 int p(){
        int r(int b[];){
                int z;
                z=b[6]+x;
                return z};
        q(r(), x*3,)};
x=15;
a[1,2]=21;
p()
```