变量作用域和函数

冯新宇

部分内容来源于Stanford CS242 (2012) 课件

主要内容

- 代码块和变量作用域
- 内联代码块和活动记录
- 函数调用
  - 静态作用域和动态作用域
- 高阶函数和闭包
  - 函数作为参数
  - 函数作为返回值

代码块

注意：仓颉本身不支持代码块，此代码仅为示意

```
{
  var x: Int = 3
  {
    var y: Int = x + 4
  }
}
```

独立出现-内联

```
while(x < n){
  var counter: Int = 0
  …
  if (counter > 100){
    var tmp: Int = 0
    …
  }
}
```

伴随特定语法-内联

```
func f(x: Int, y: Int): Int {
  var counter: Int = 0
  …
}
…
f(3, 5)
f(4, 8)
```

作为函数体

内联（in-line）：代码出现的位置就是它被执行的位置

非内联

代码块

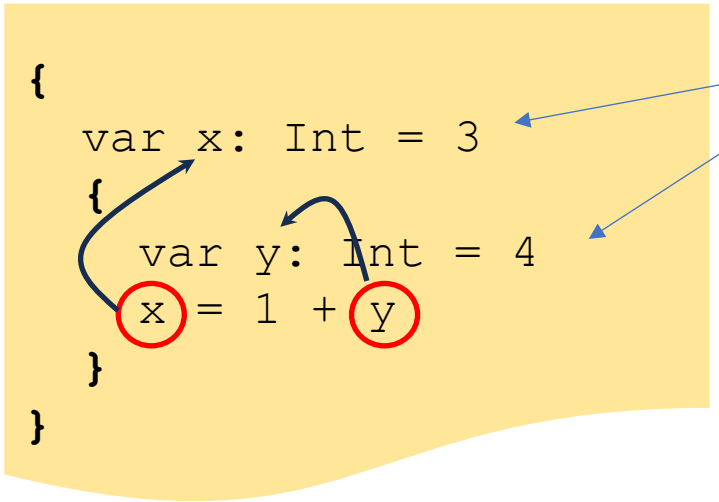
```
{
  var x: Int = 3
  {
    var y: Int = x + 4
  }
}
```

```
begin_A:
  …
  …
begin_B
  …
end_A
  …
end_B
```

可以并列和嵌套，但不能部分重叠

局部变量和内存管理

- 嵌套代码块和局部变量



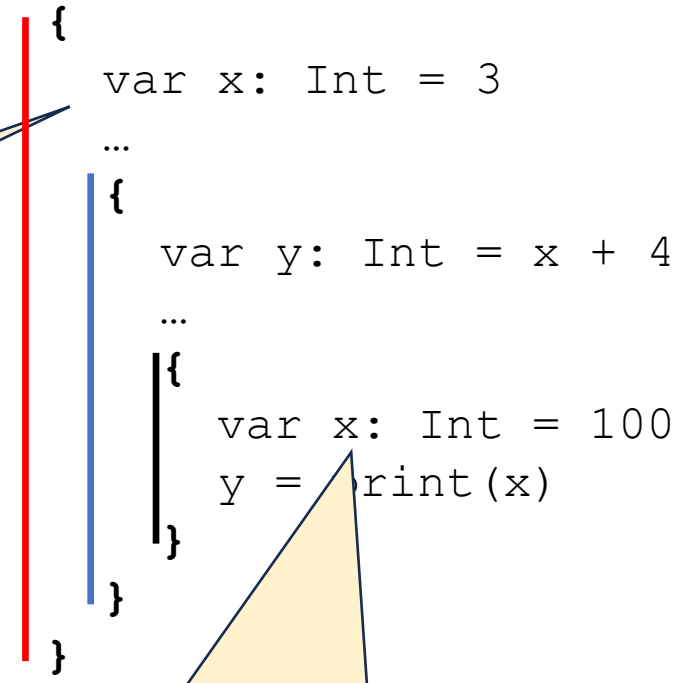
在不同的块中声明的变量

内部块中，访问“局部变量”y
和非局部变量x

- 内存管理
  - 进入代码块：分配空间，存储本代码块中声明的变量
  - 推出代码块：释放空间（局部变量消亡）

作用域（scope）和生存期（lifetime）

- （声明的）作用域
  - 可以访问该声明的代码区域
- （变量的）生存期/生命周期
  - 为变量分配的存储空间的存在的时间段



外部块中x声明的作用域为红色代码块中 挖掉 黑色 部分后剩余的部分

外部代码块中x的生存期为整个外部代码块的执行时间（从进入代码块到退出的时间），包括内部x的生存期。

作用域 不等于 生存期

内部的x声明隐藏/遮盖了外部的x声明

内联代码块的活动记录

- 活动记录
  - 保存在运行时栈上的数据结构
  - 存储局部变量的值

```
{
  var x: Int = 0
  var y: Int = x + 1
  {
    var z: Int = (x + y) * (x - y)
  }
}
```

活动记录压栈，为x、y分配内存空间

计算并保存x、y的值

内层代码块动记录压栈，为z分配内存空间

计算并保存z的值

内层代码块的活动记录退栈

外层代码块的活动记录退栈

内联代码块的活动记录

- 控制链
  - 指向栈上前一个活动记录的指针
- 活动记录压栈
  - 把环境指针赋值给新的控制链
  - 设置环境指针，指向新的活动记录
- 活动记录退栈
  - 把当前活动记录的控制链赋值给环境指针

例子

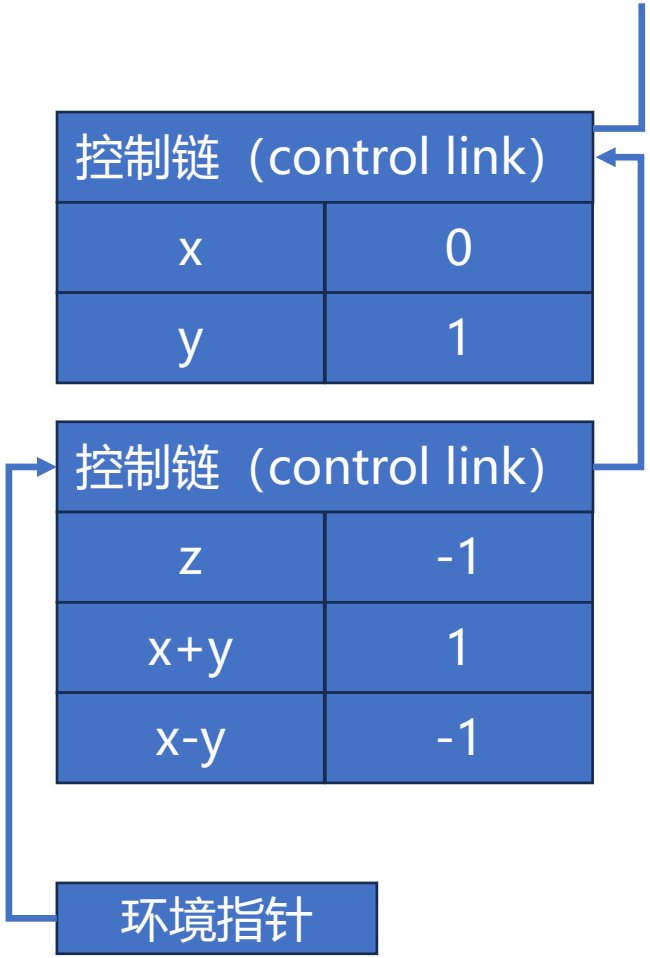
```
{
  var x: Int = 0
  var y: Int = x + 1
  {
    var z: Int = (x + y) * (x - y)
  }
}
```

活动记录压栈，为x、y分配内存空间

计算并保存x、y的值

- 内层代码块动记录压栈，为z分配内存空间
- 计算并保存z的值
- 内层代码块的活动记录退栈

外层代码块的活动记录退栈



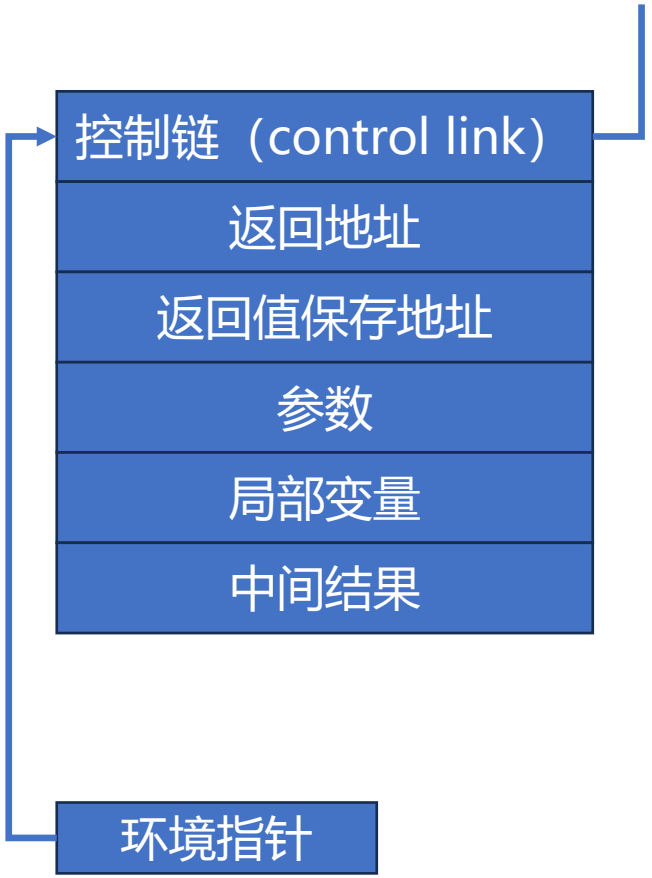
函数调用和活动记录

- 函数定义

```
func f(a1: t1, …, an: tn): tr {
    … // 局部变量声明
    … // 函数体
    return …
}
```

- 活动记录需要包括以下内容
  - 返回地址
  - 参数
  - 局部变量，中间结果
  - 函数返回时，在上一级活动中保存返回值的地址

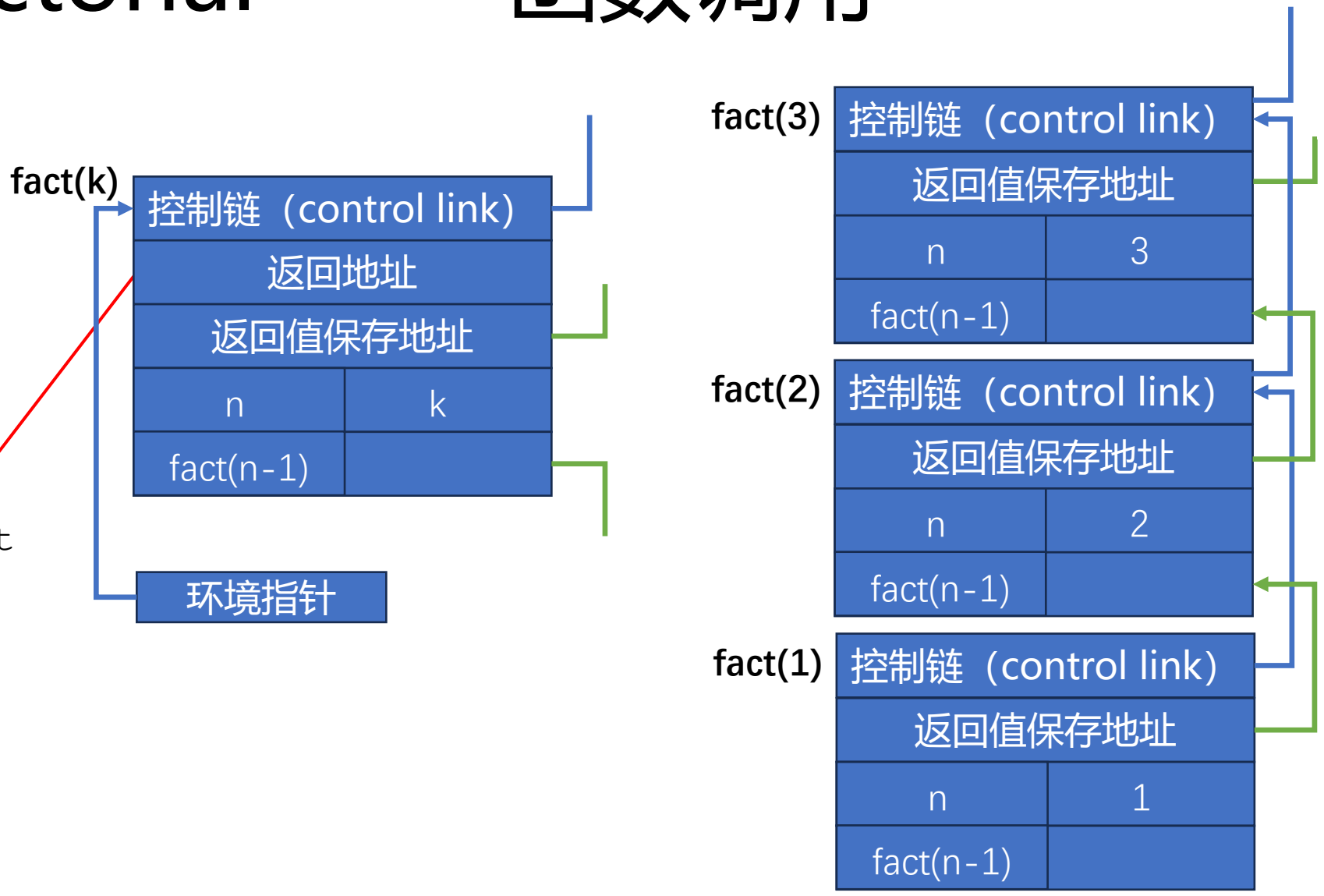
函数调用和活动记录



- 返回地址
  - 函数返回后，跳转回调用者的代码地址
- 返回值保存地址
  - 上级调用者活动记录中保存函数返回值的地址
- 参数
  - 保存调用者传递的参数

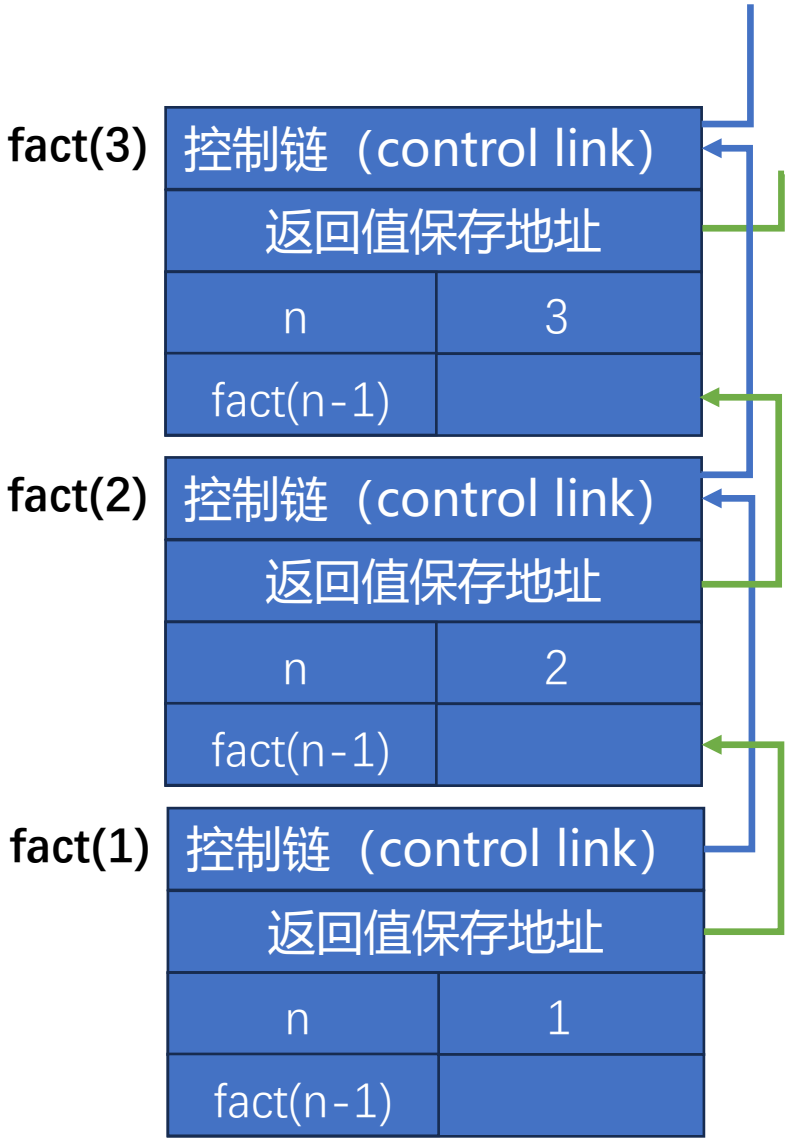
例子：factorial —— 函数调用

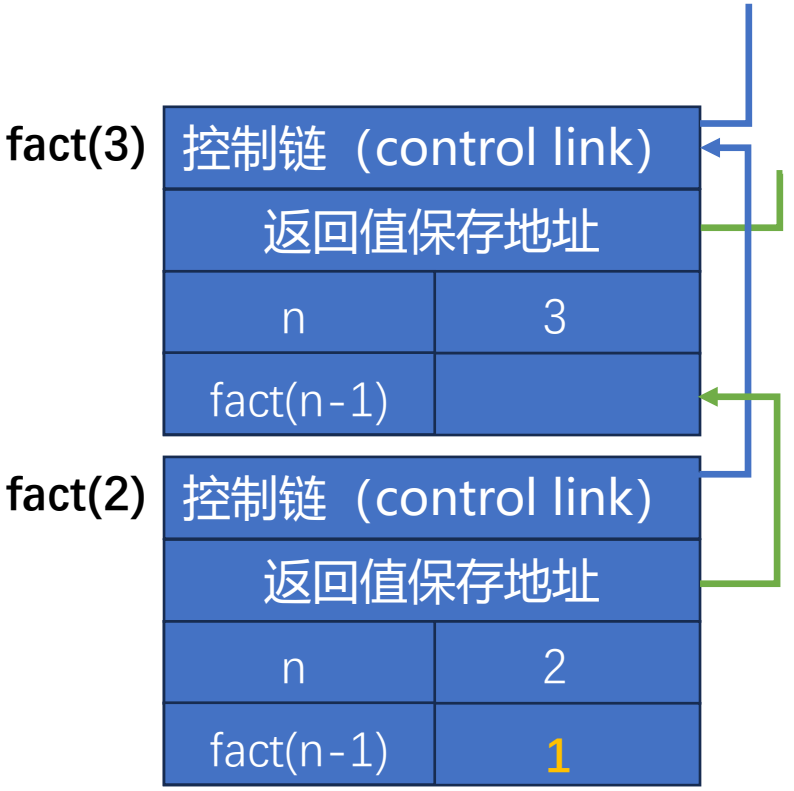
```
func fact(n: Int): Int
{
    if (n <= 1) {
        1
    } else {
        n * fact (n-1)
    }
}
```



例子：factorial —— 函数返回

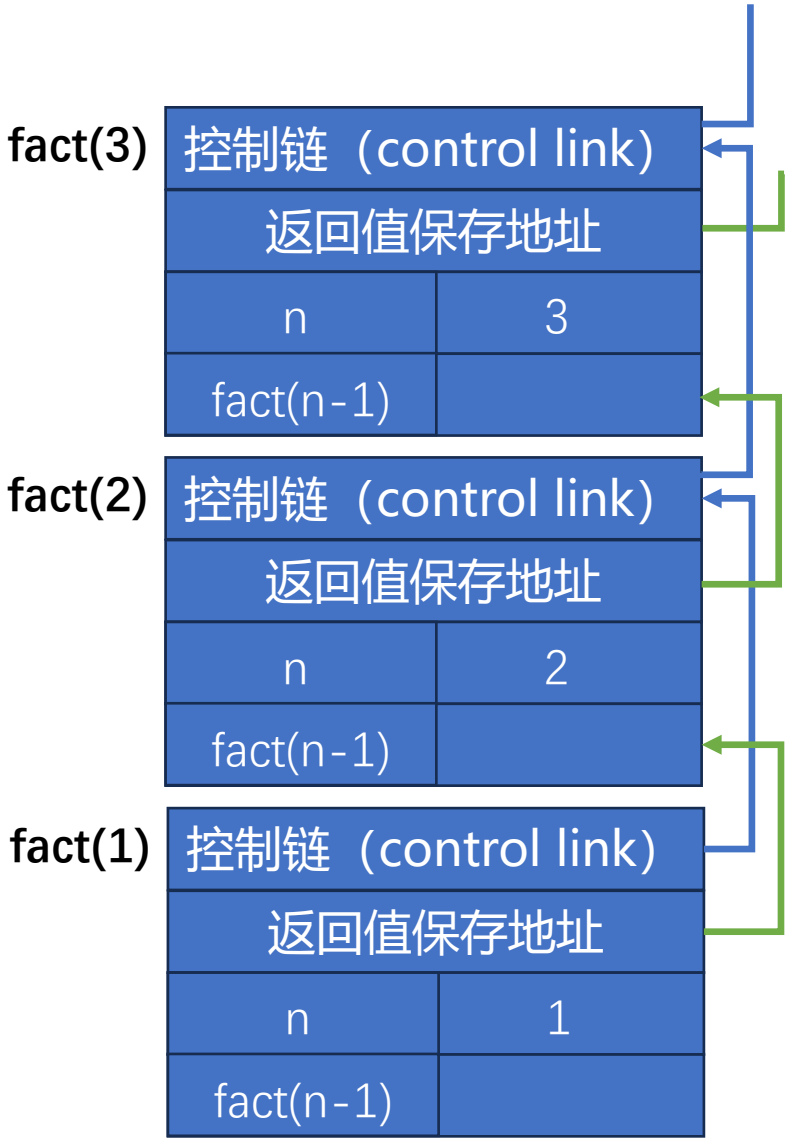
```
func fact(n: Int): Int
{
    if (n <= 1) {
        1
    } else {
        n * fact (n-1)
    }
}
```





例子：factorial —— 函数返回

```
func fact(n: Int): Int
{
    if (n <= 1) {
        1
    } else {
        n * fact (n-1)
    }
}
```



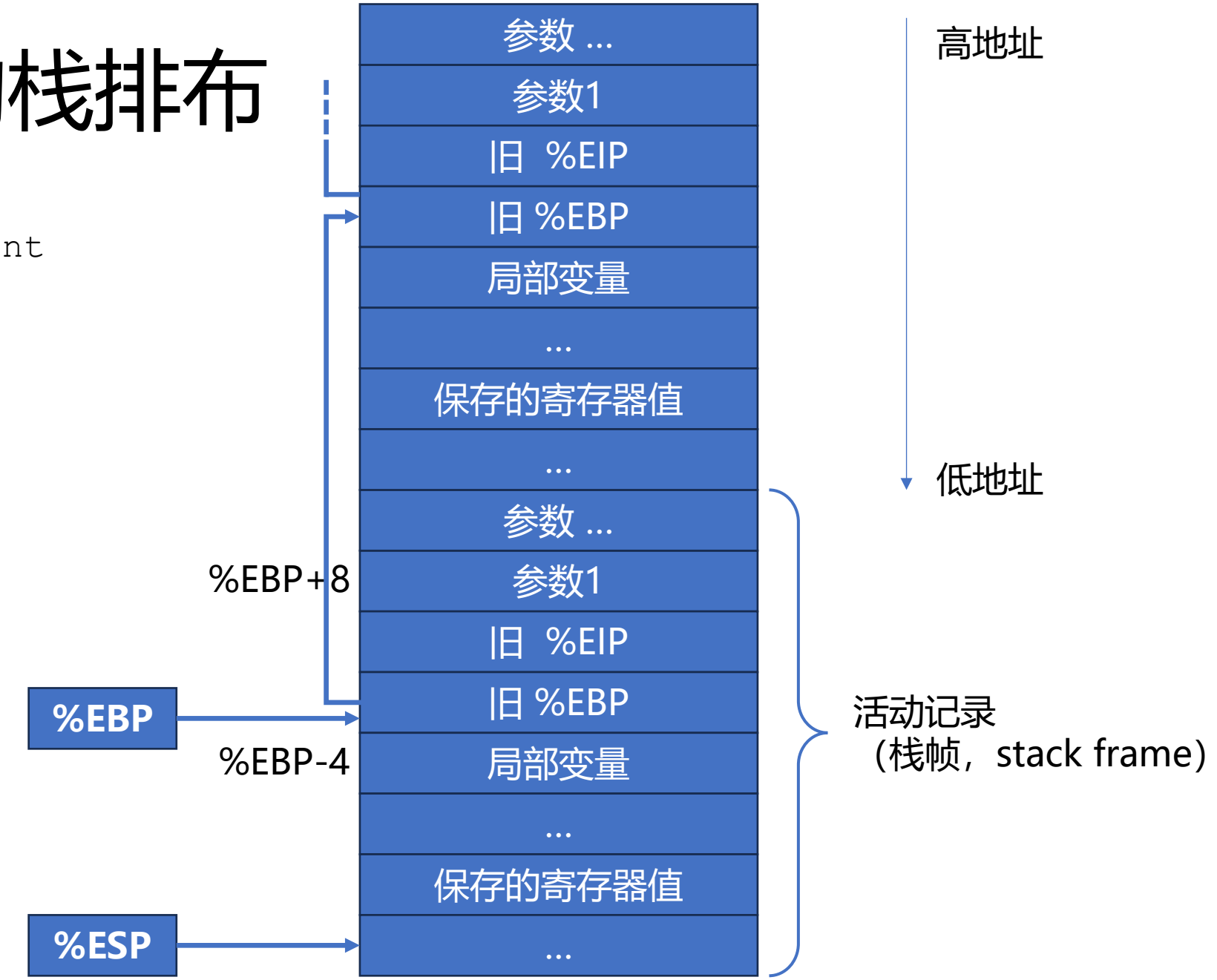
编译型语言的栈排布

```
func f(a1: Int, a2: Int): Int
{
    let x1: Int = …
    let x2: Int = …
    …
    return …
}
```

活动记录存储在连续的栈空间

编译器根据函数定义，提前计算好活动记录的大小，以及每个单元相对于位移

访问相应单元只需要采用相对于%ESP / %EBP的相对地址



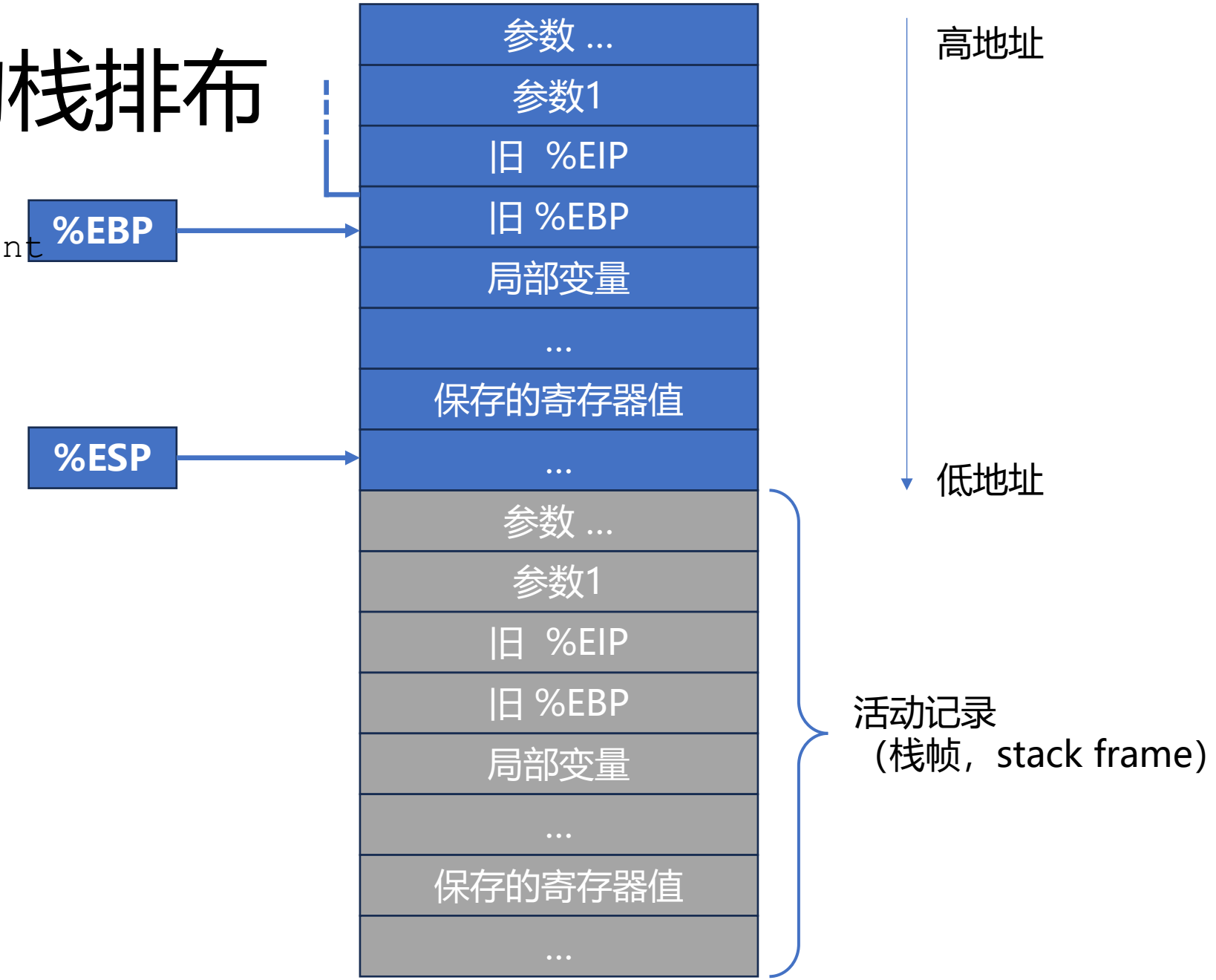
编译型语言的栈排布

```
func f(a1: Int, a2: Int): Int
{
    let x1: Int = …
    let x2: Int = …
    …
    return …
}
```

活动记录存储在连续的栈空间

编译器根据函数定义，提前计算好活动记录的大小，以及每个单元相对于位移

访问相应单元只需要采用相对于%ESP / %EBP的相对地址



一阶函数调用 —— 参数传递

- 基本术语：左值和右值
  - 赋值语句： y = x + 3
  - 左边的标识符代表其地址，称为左值
  - 右边的标识符代表其地址中保存的值，称为右值

- 参数传递：传引用（pass-by-reference）
  - 把参数的左值（地址）保存在活动记录中
  - 被调用函数（callee）可以通过该地址为调用者（caller）中的变量赋值

- 参数传递：传值（pass-by-value）
  - 把右值拷贝一份保存在活动记录中
  - Callee不能修改caller中的变量值
  - 避免指针别名

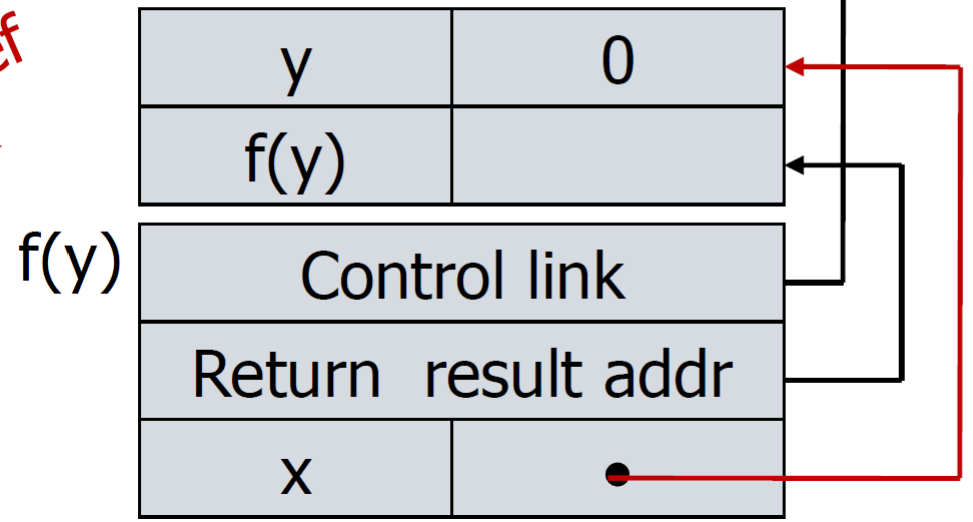
仓颉中仅支持传值，与大多数语言保持一致

pseudo-code

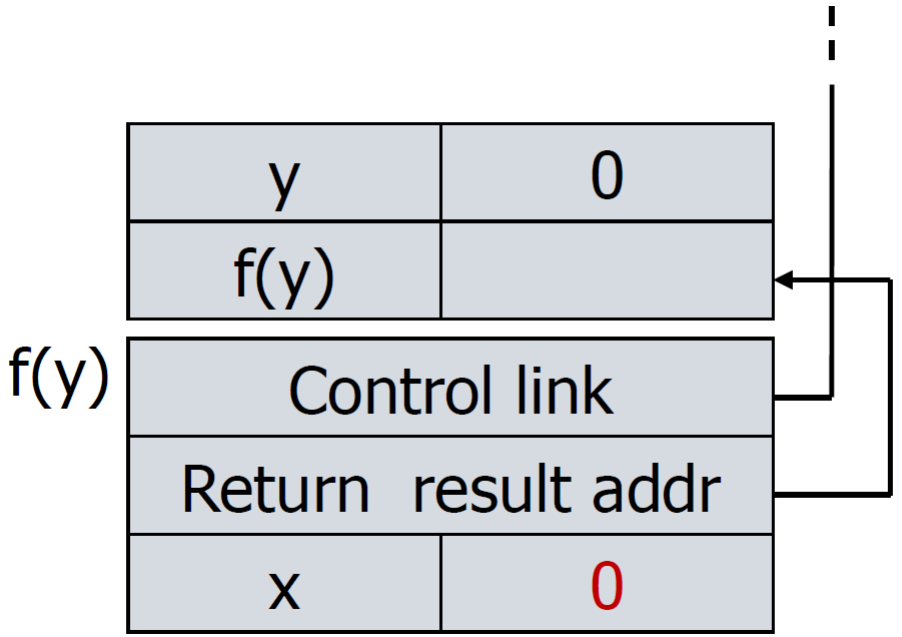
```
function f (x) =
    { x = x+1; return x;  }
var y = 0;
print (f(y)+y);
```

activation records

pass-by-ref

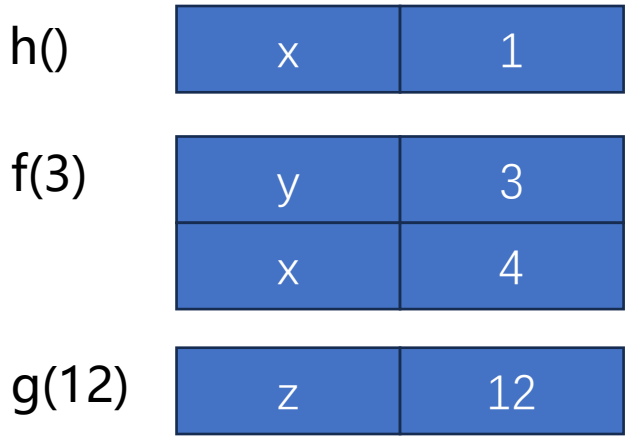


pass-by-value



非局部变量的访问

```
func h(): Unit
{
    let x = 1
    func g(z: Int){ x + z }
    func f(y){
        var x = y + 1
        g(y * x)
    }
    f(3)
}
```

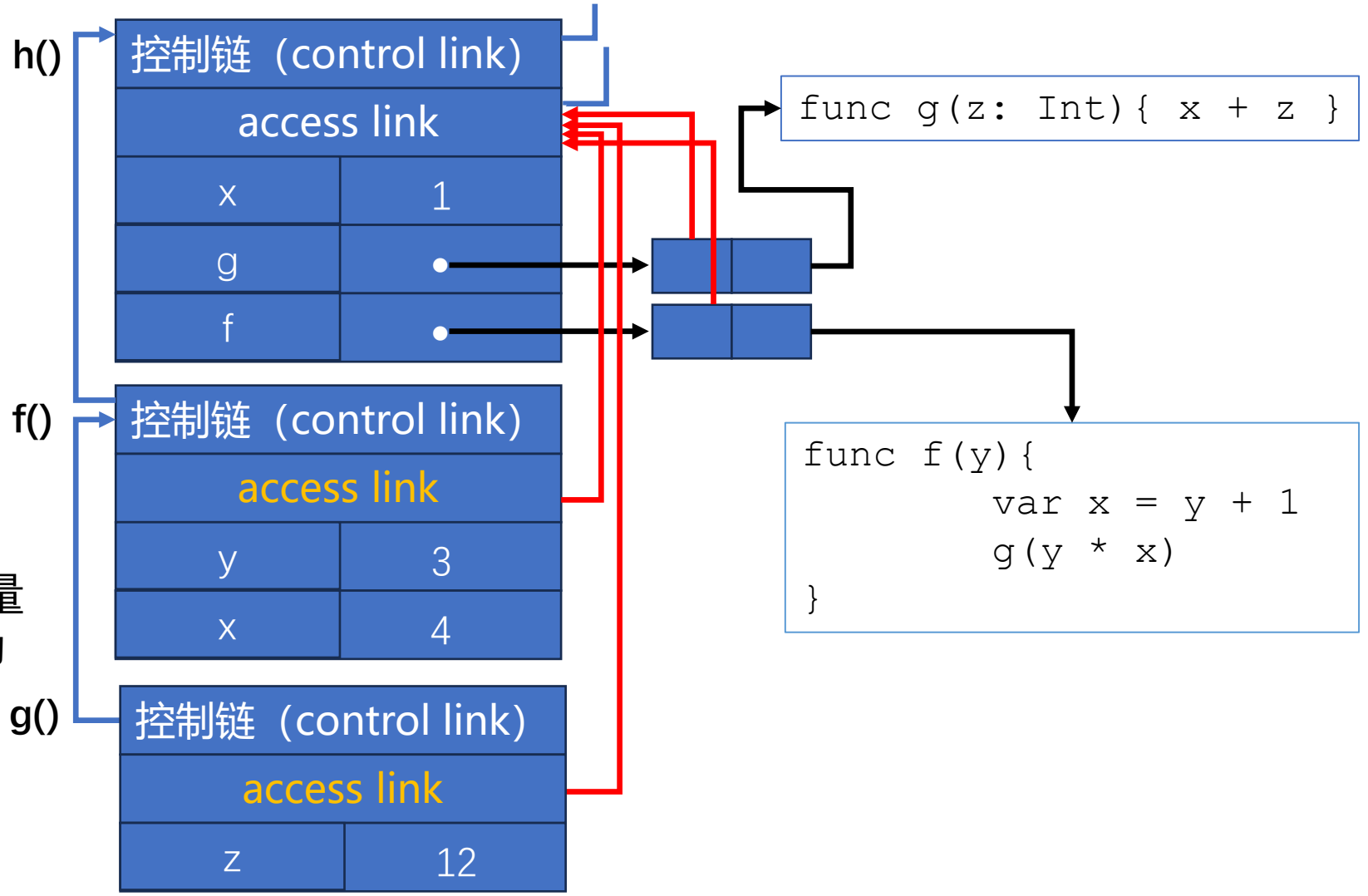


- 函数g中的 “x + z” 将访问哪个 x？
  - **静态作用域：**访问g定义处所属的活动记录中的值
    - g(12)返回 13
  - **动态作用域：**访问g的调用者的活动记录中的值
    - g(12)返回16

静态作用域和access link

```
func h(): Unit
{
    let x = 1
    func g(z: Int){ x + z }
    func f(y){
        var x = y + 1
        g(y * x)
    }
    f(3)
}
```

- 使用access link 访问非局部变量
  - Access link总是指向声明该函数的活动记录



尾调用和尾递归优化

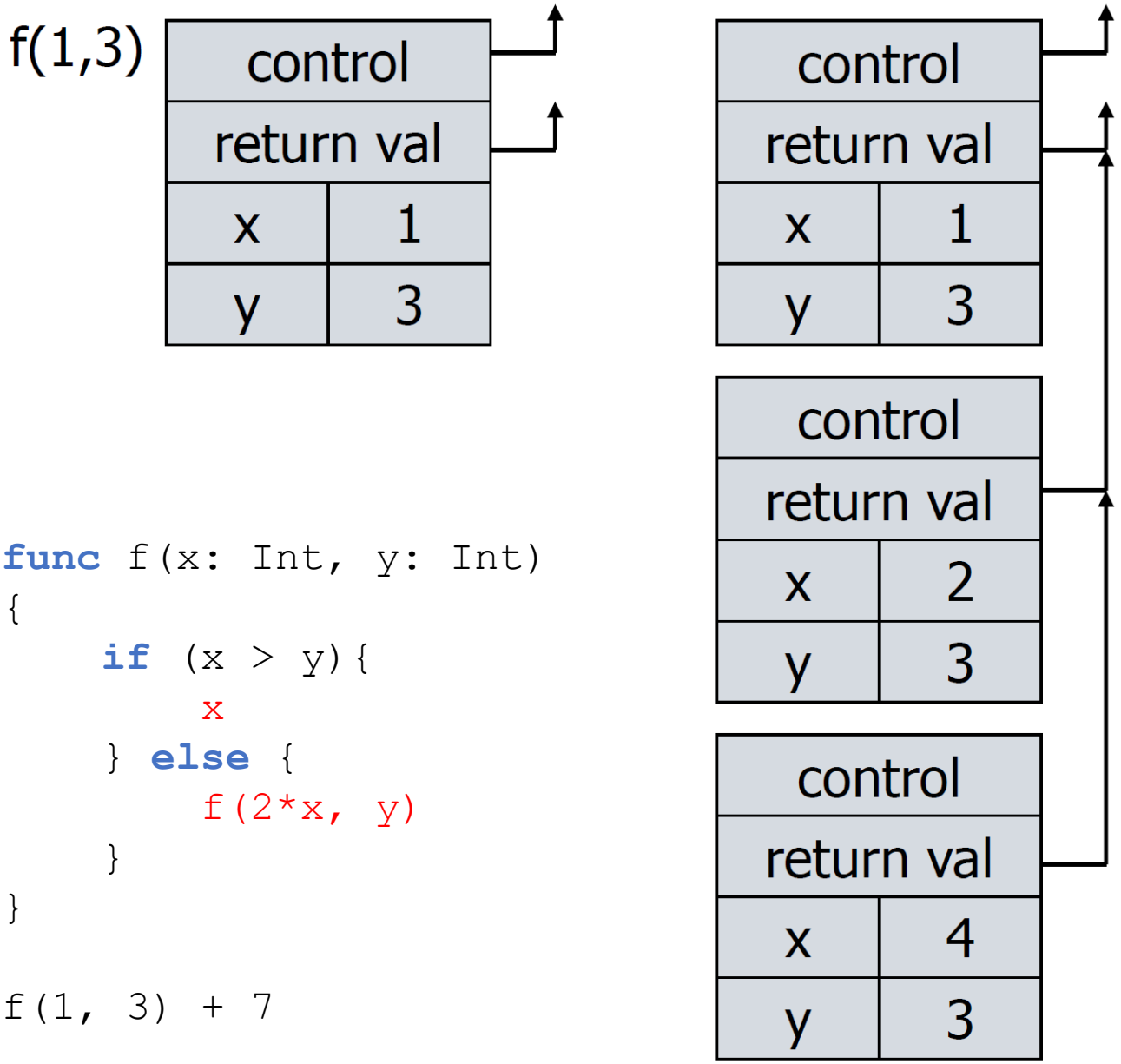
- 函数g对函数f的调用是尾调用（tail call），如果：
  - 调用函数f是g中的最后一个动作（并把f的返回值直接作为自己的返回值）

```
func g(x: Int)
{
    if (x > 0){
        f(x)
    } else {
        -5 * f(x)
    }
}
```

尾调用

不是尾调用，虽然出现在函数体代码的最后

- 编译优化
  - 如果g对f的调用是尾调用，可以在调用f的时候就把g的活动记录退栈
  - 对尾递归（tail recursion）尤其有用，因为caller和callee的活动记录格式一样
    - 被调用者可以直接复用调用者的活动记录，无需退栈/压栈



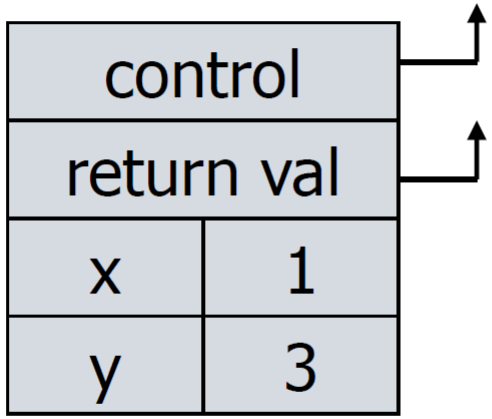

```
func f(x: Int, y: Int)
{
    if (x > y){
        x
    } else {
        f(2*x, y)
    }
}

f(1, 3) + 7
```

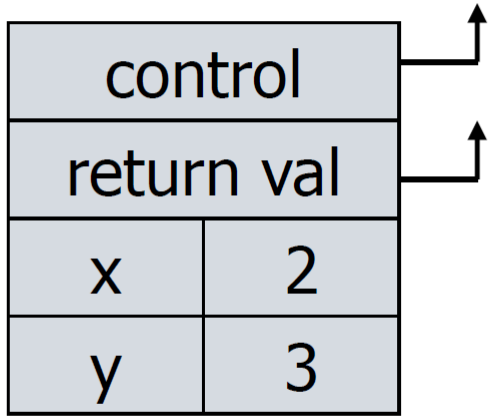
计算大于y的最小的2的指数

- 编译优化
  - 将返回值地址设置为caller的返回值地址
  - 同样，将control link、返回地址都复用caller的值
    - 返回时，跨过caller，返回caller的caller（的caller ...）

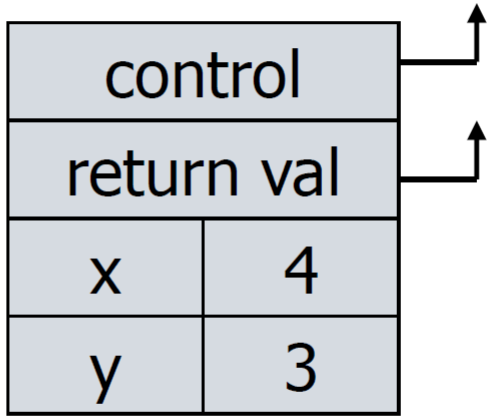
f(1,3)



f(2,3)



f(4,3)



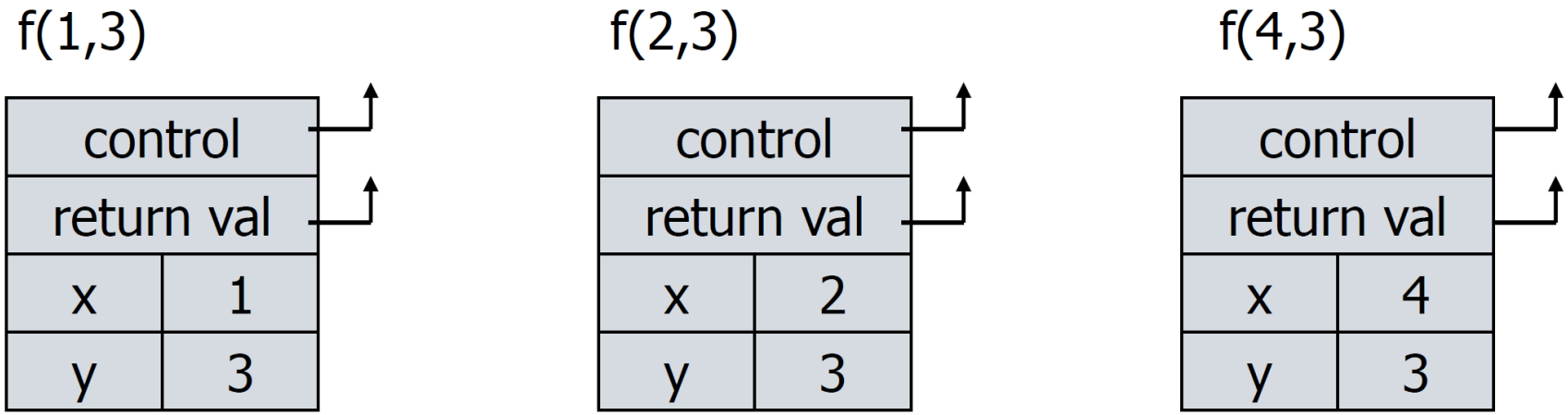

```
func f(x: Int, y: Int)
{
    if (x > y){
        x
    } else {
        f(2*x, y)
    }
}

f(1, 3) + 7
```

计算大于y的最小的2的指数

- 编译优化
  - 先将caller的活动记录退栈，再把callee压栈 = 原地复用caller的活动记录
- 效果
  - 尾递归优化后效果等同于循环

尾递归和循环




```
func f(x: Int, y: Int)
{
    if (x > y){
        x
    } else {
        f(2*x, y)
    }
}

f(1, y)
```

```
func g(y: Int)
{
    var x = 1
    while (!x > y){
        x = 2 * x
    }
    return x
}

g(y)
```

初始值

终止条件判断

循环体

高阶函数

- 语言特性
  - 函数本身作为其他函数的参数或者返回值
  - 有时候称作“函数作为一等公民（first-class citizen）”
  - 实现中需要保存函数的“环境”

- 简单情况
  - 函数作为参数
  - 需要同时传递access link，指向栈上函数定义处的活动记录

- 更复杂的情况
  - 函数作为返回值
  - 需要赋值并额外保存函数定义处的活动记录

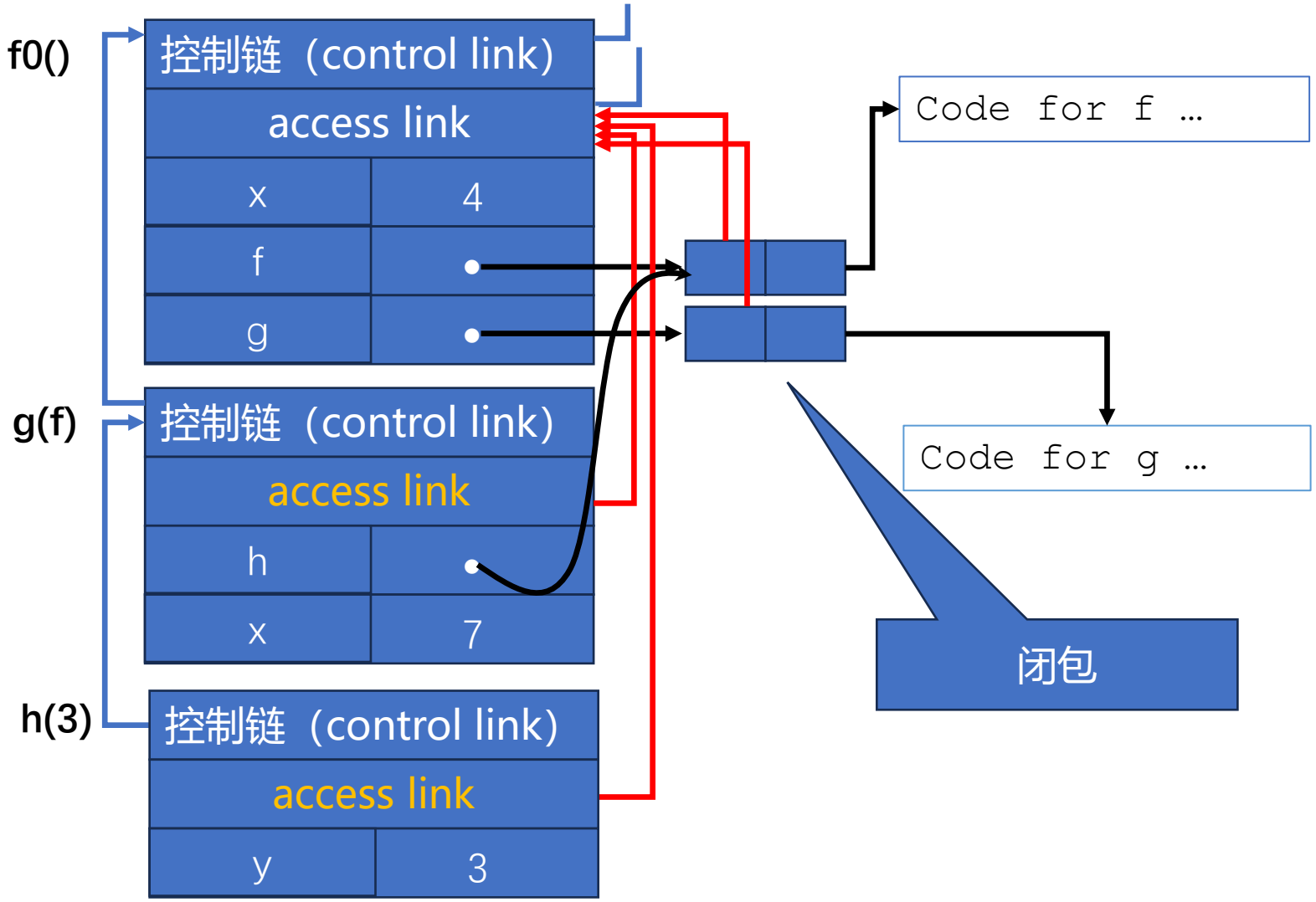
函数作为参数

```
func f0() {
    let x = 4
    func f(y: Int){ x * y }
    func g(h){
        var x = 7
        h(3) + x
    }
    g(f)
}
f0()
```

两个x的取值各是多少?

函数参数和闭包

```
func f0() {
    let x = 4
    func f(y: Int){ x * y }
    func g(h){
        var x = 7
        h(3) + x
    }
    g(f)
}
f0()
```



闭包

- Function value is pair *closure* = ⟨*env*, *code* ⟩
- When a function represented by a closure is called,
  - Allocate activation record for call (as always)
  - Set the access link in the activation record using the environment pointer from the closure

小结：函数作为参数

- Use closure to maintain a pointer to the static environment of a function body
- When called, set access link from closure
- All access links point “up” in stack
  - May jump past activ records to find global vars
  - Still deallocate activ records using stack (lifo) order

函数作为返回值

Lambda表达式

```
func comp(f: A -> B, g: B -> C){
    return { x => g(f(x)) }
}
```

```
func comp(f: A -> B, g: B -> C){
    func builder(x){
        return g(f(x))
    }
    return builder
}
```

- 语言特性
  - 函数返回“新的”函数
  - 需要维护函数的“环境”
- 函数被动态创建
  - 被创建的函数包含非局部变量（“捕获”非局部变量）
  - 函数值为闭包<env, code>
  - 闭包动态创建，但代码不会动态生成和编译

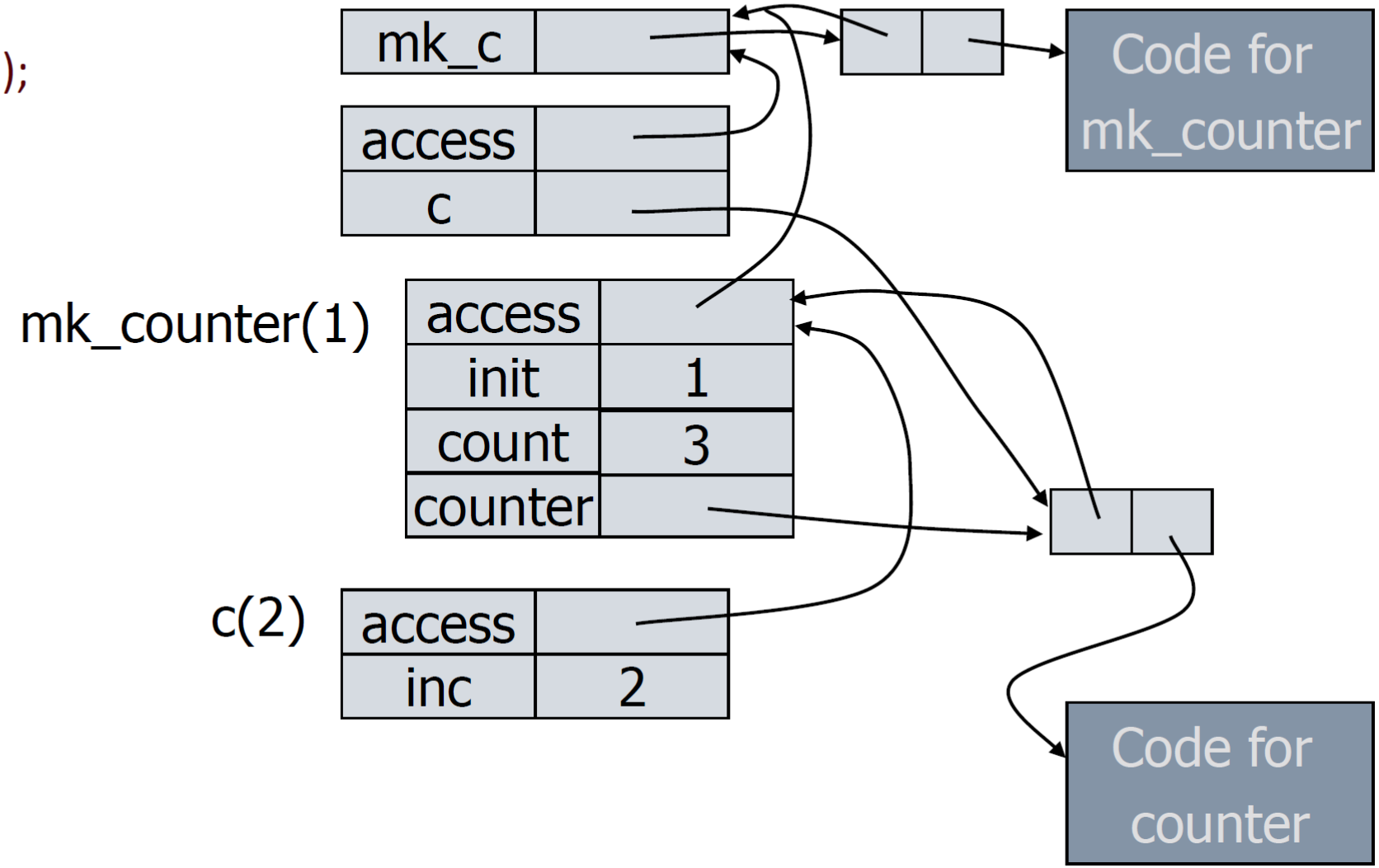
函数（作为返回值）带有私有数据

Javascript code为例

```
function mk_counter (init) {
    var count = init;
    function counter(inc) {count=count+inc; return
    count};
    return counter};
var c  = mk_counter(1);
c(2) + c(2);
```

- Function to “make counter” returns a closure
- How is correct value of count determined in c(2) ?

```
function mk_counter (init) {
    var count = init;
    function counter(inc) {count=count+inc; return count};
    return counter};
var c  = mk_counter(1);
c(2) + c(2);
```



小结：函数作为返回值

- Use closure to maintain static environment
- May need to keep activation records after return
  - Stack (lifo) order fails!
- Possible “stack” implementation
  - Forget about explicit deallocation
  - Put activation records on heap
  - Invoke garbage collector as needed
  - Not as totally crazy as is sounds

    May only need to search reachable data

小结

- Block-structured lang uses stack of activ records
  - Activation records contain parameters, local vars, …
  - Also pointers to enclosing scope
- Several different parameter passing mechanisms
- Tail calls may be optimized
- Function parameters/results require closures
  - Closure environment pointer used on function call
  - Stack deallocation may fail if function returned from call
  - Closures *not* needed if functions not in nested blocks