中国科学技术大学
University of Science and Technology of China

Scope, Function Calls and Storage Management

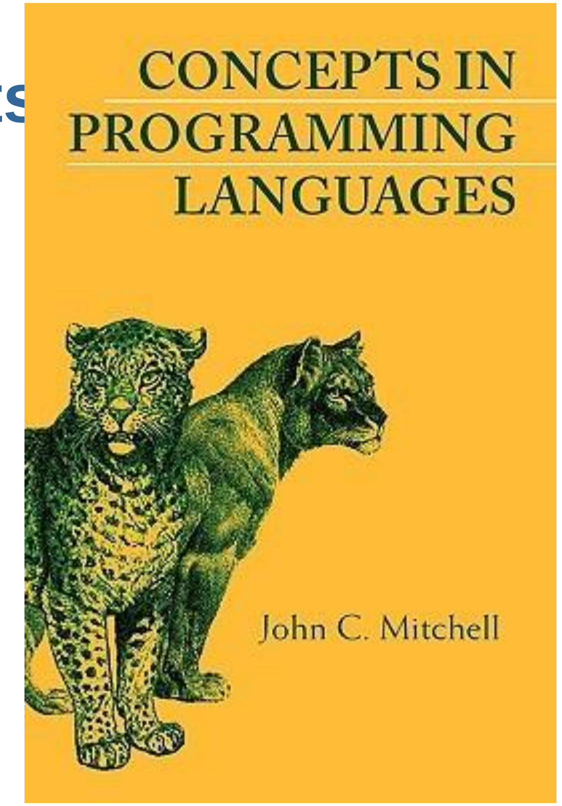
《程序语言设计和程序分析》

张昱
yuzhang@ustc.edu.cn
中国科学技术大学
计算机科学与技术学院

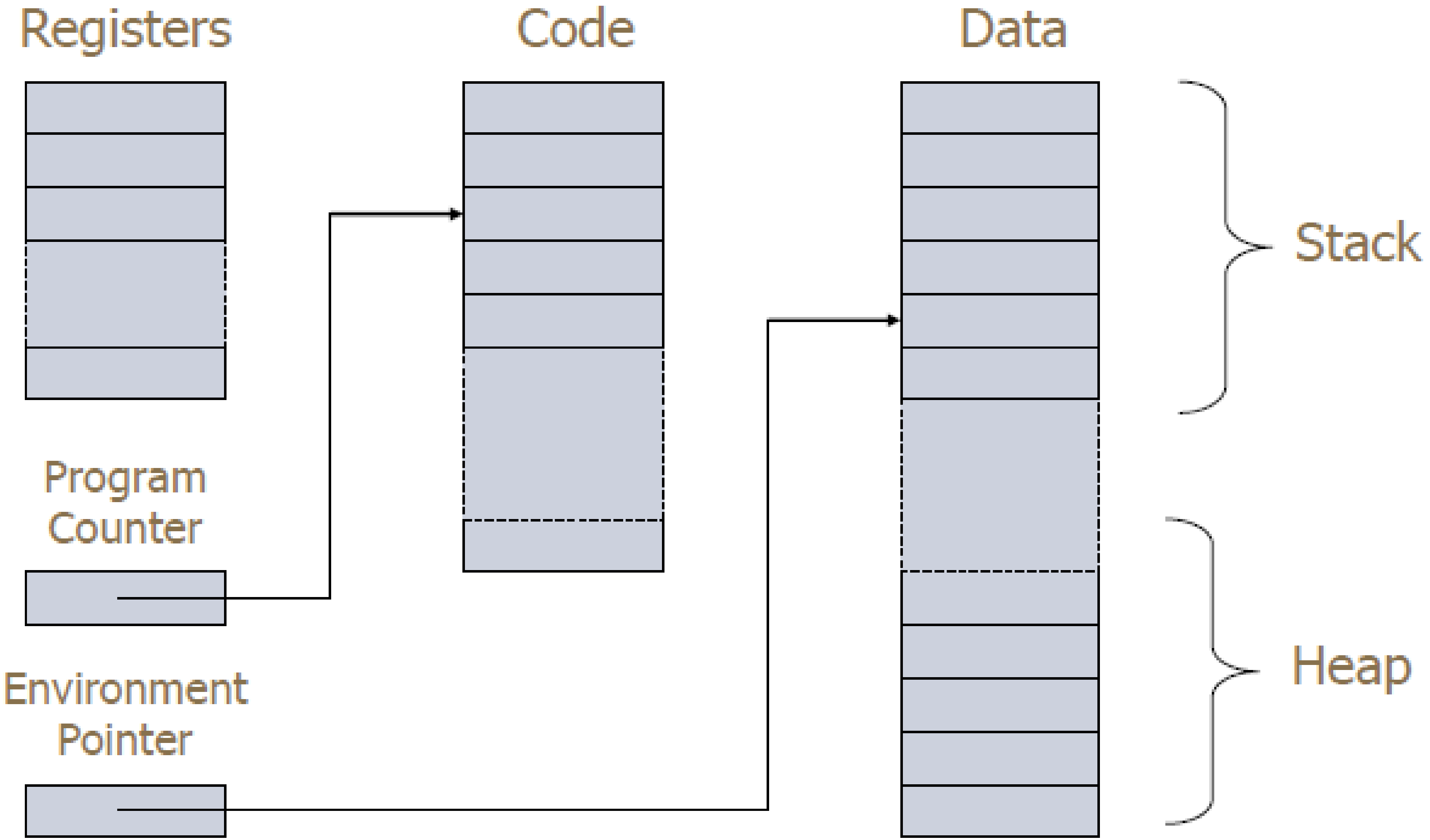
Reading

- "Concepts in Programming Languages"
  - Chapter 7: Scope, Functions, and Storage Management

- http://theory.stanford.edu/people/jcm/books.html
- https://homepages.dcc.ufmg.br/~camarao/lp/concepts

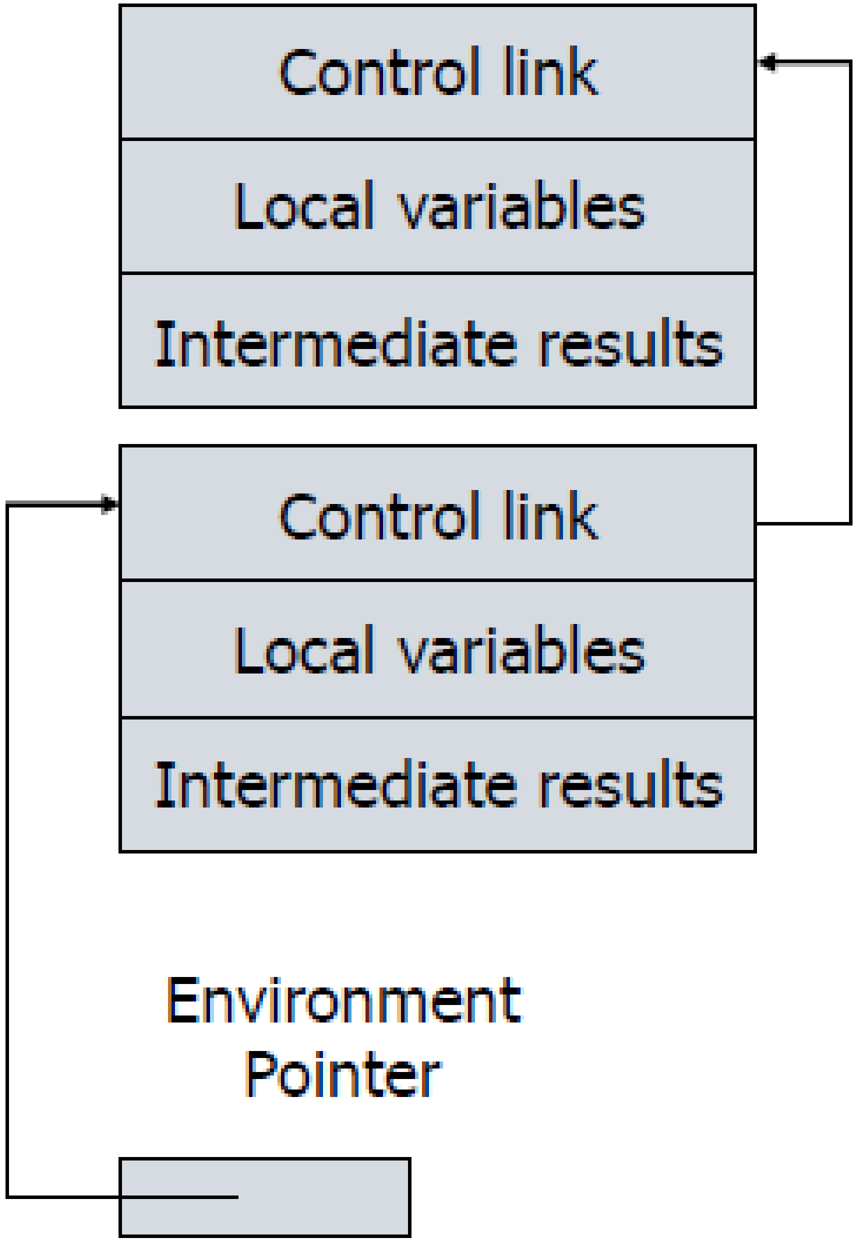


Simplified Machine Model



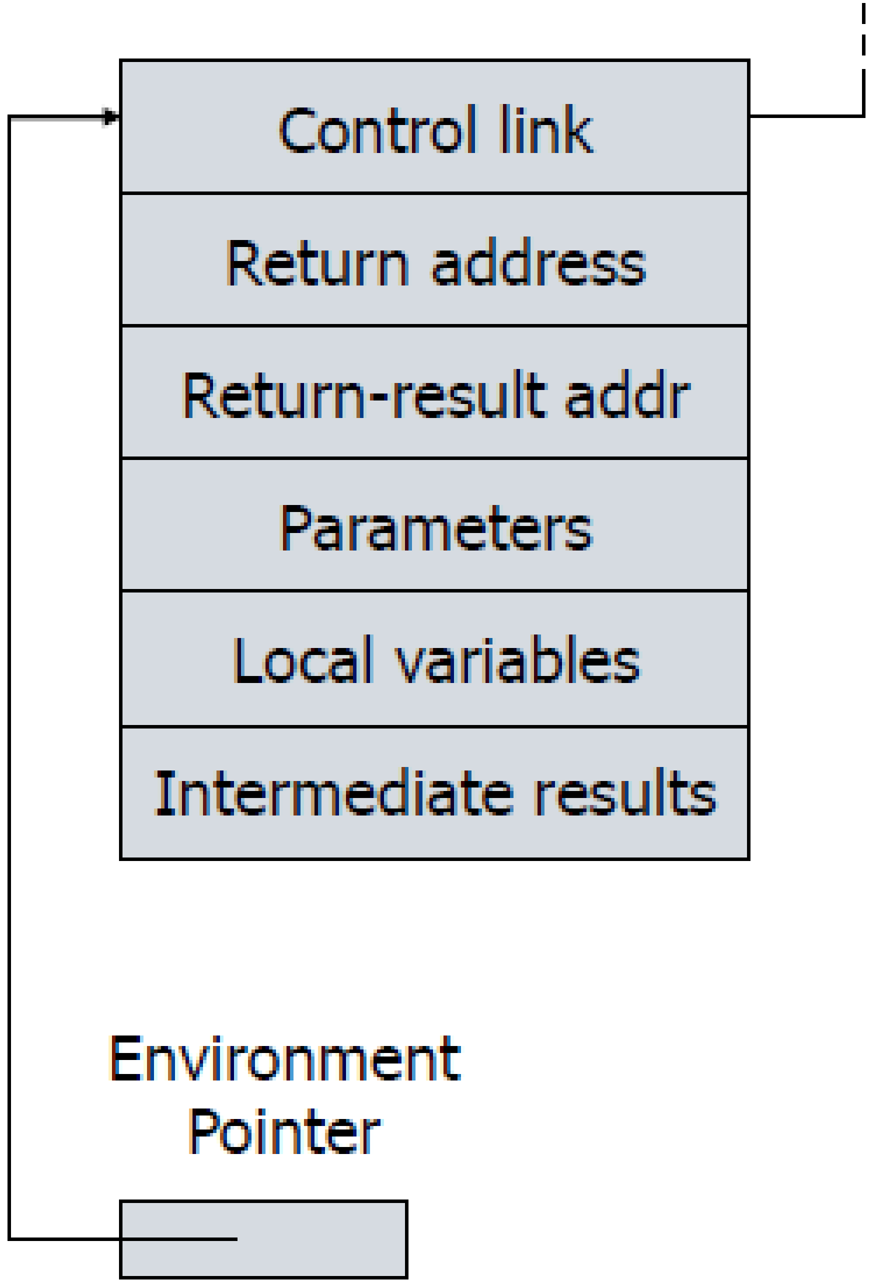
Activation Record for In-link Block

- Control link
  - Pointer to previous record on stack
- Push record on stack
  - Set new control link to point to old env ptr
  - Set env ptr to new record
- Pop record off stack
  - Follow control link of current record to reset environment pointer



Activation record for function

- Return address
  - Location of code to execute on function return
- Return-result address
  - Address in activation record of calling block to store function return val
- Parameters
  - Locations to contain data from calling block

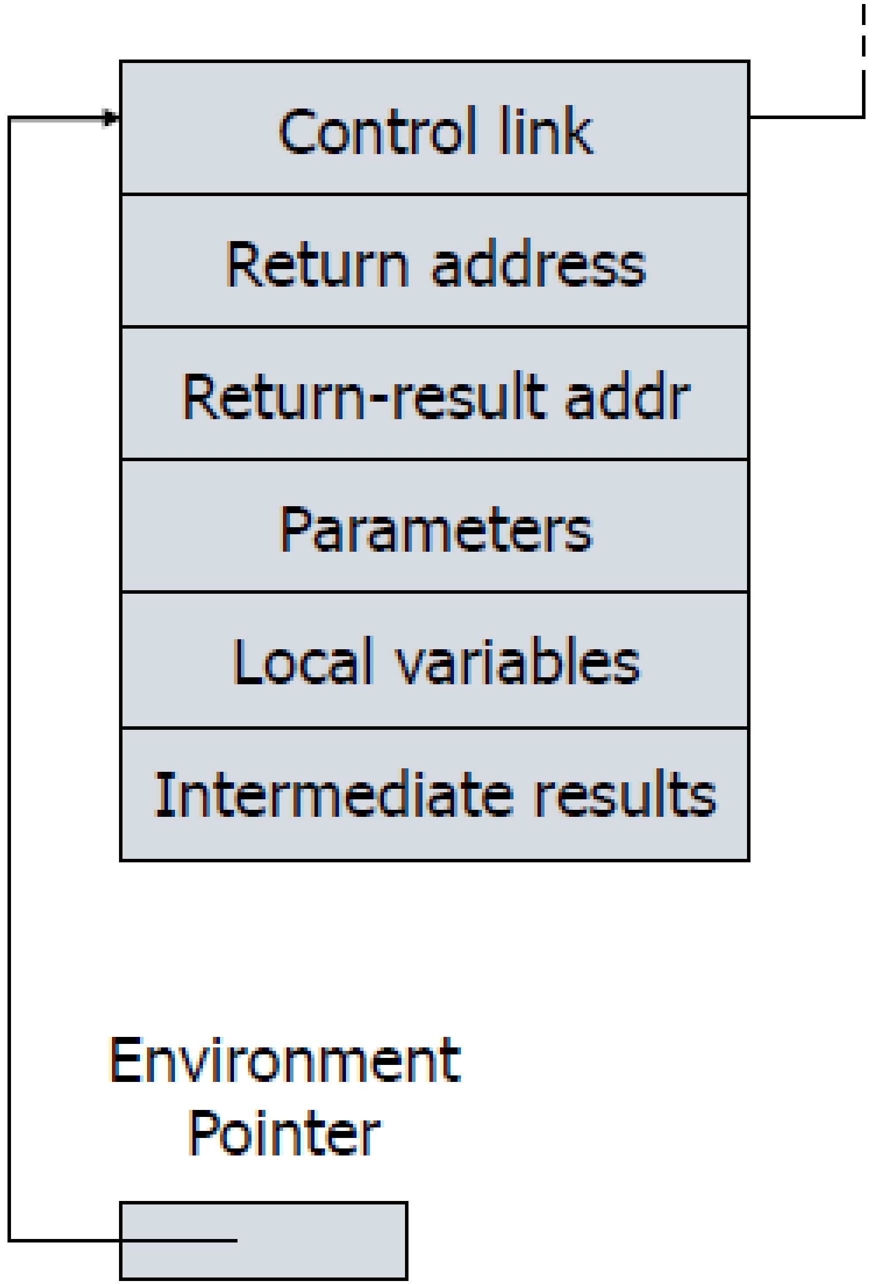


First-order Functions

- Parameter passing
  - pass-by-value: copy value to new activation record
  - pass-by-reference: copy pointer to new activation record
- Access to global variables
  - global variables are contained in an activation record higher “up” the stack
- Tail recursion
  - an optimization for certain recursive functions

Activation record for static scope

- Control link
  - Link to activation record of previous (calling) block
- Access link
  - Link to activation record of closest enclosing block in program text
- Difference
  - Control link depends on dynamic behavior of program
  - Access link depends on static form of program text



Higher-order Functions

- Language features
  - Functions passed as arguments
  - Functions that return functions from nested blocks
  - Need to maintain environment of function

  Functions as first class values
- Simpler case
  - Function passed as argument
  - Need pointer to activation record “higher up” in stack
- More complicated second case
  - Function returned as result of function call
  - Need to keep activation record of returning function

Closures

- **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**

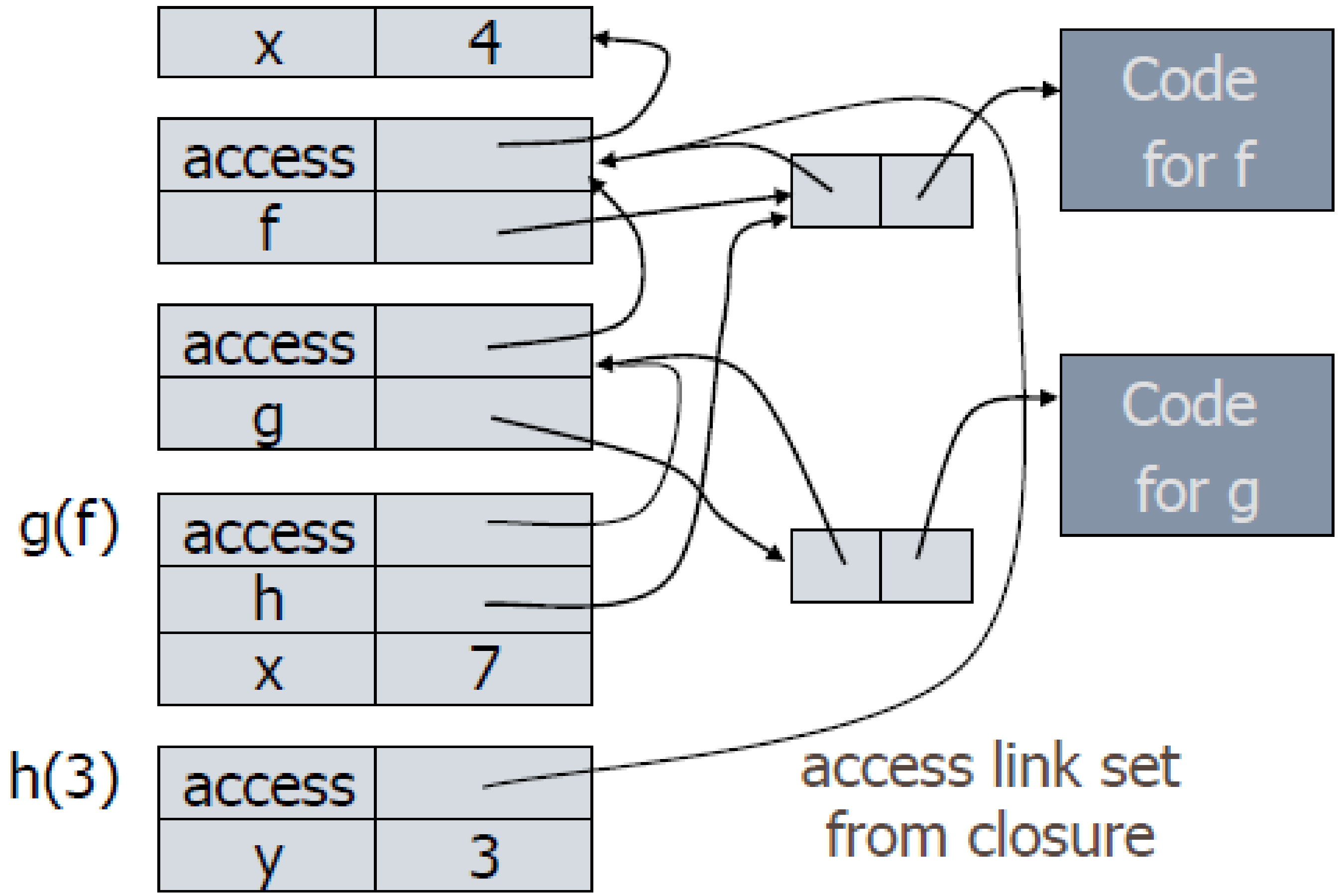
Function Argument and Closures

University of Science and Technology of China

ML

```
var x = 4;
 fun f(y) = x*y;
  fun g(h) =
   let
    var x=7
   in
    h(3) + x;
 g(f);
```

Run-time stack with access links

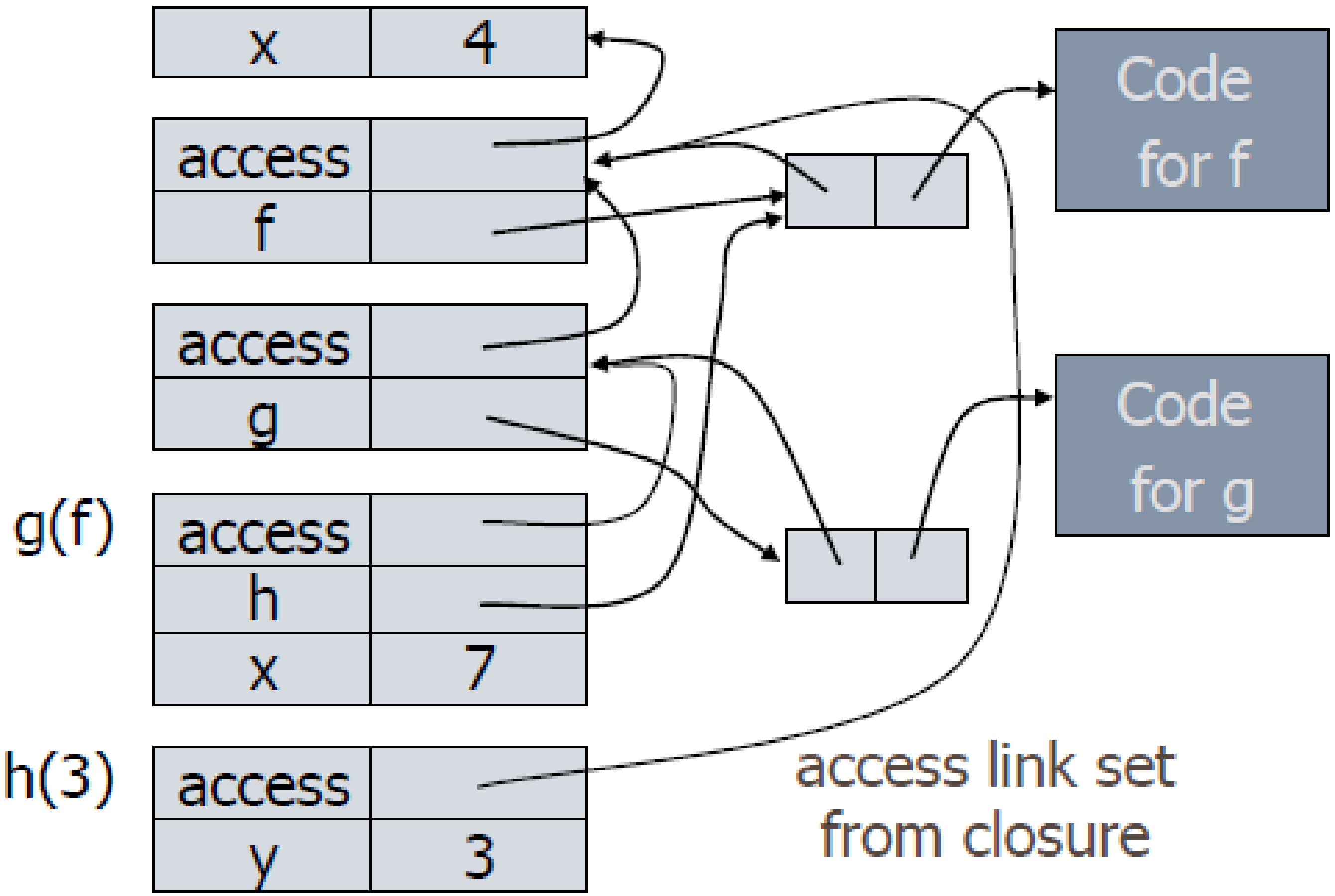


Function Argument and Closures

Lua

```
{ var x = 4;
  { function f(y)
        {return x*y;}
    { function g(h) {
       var x=7;
       return h(3) + x;
      };
      g(f);
}}}
```

Run-time stack with access links



Summary: Function Arguments

- 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 activation records to find global vars
  - Still deallocate activation records using stack (LIFO) order

Return Function as Result

- Language feature
  - Functions that return “new” functions
  - Need to maintain environment of function
- Example

```
function compose(f,g)
    { return function(x) { return g(f (x)) } };
```

- Function “created” dynamically
  - expression with free variables

    values are determined at run time
  - function value is closure = env, code
  - code *not* compiled dynamically (in most languages)

Example: Return fctn with Private State

ML

mk_counter : int → (int → int)
c : int → int

```
fun mk_counter (init : int) =
    let  val count = ref init
        fun counter(inc:int) =
            (count := !count + inc; !count)
    in
        counter
    end;
val c = mk_counter(1);
c(2) + c(2);
```

Private variable count

closure

The value is a closure

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

Example: Return fctn with private state

JS

```
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);
```

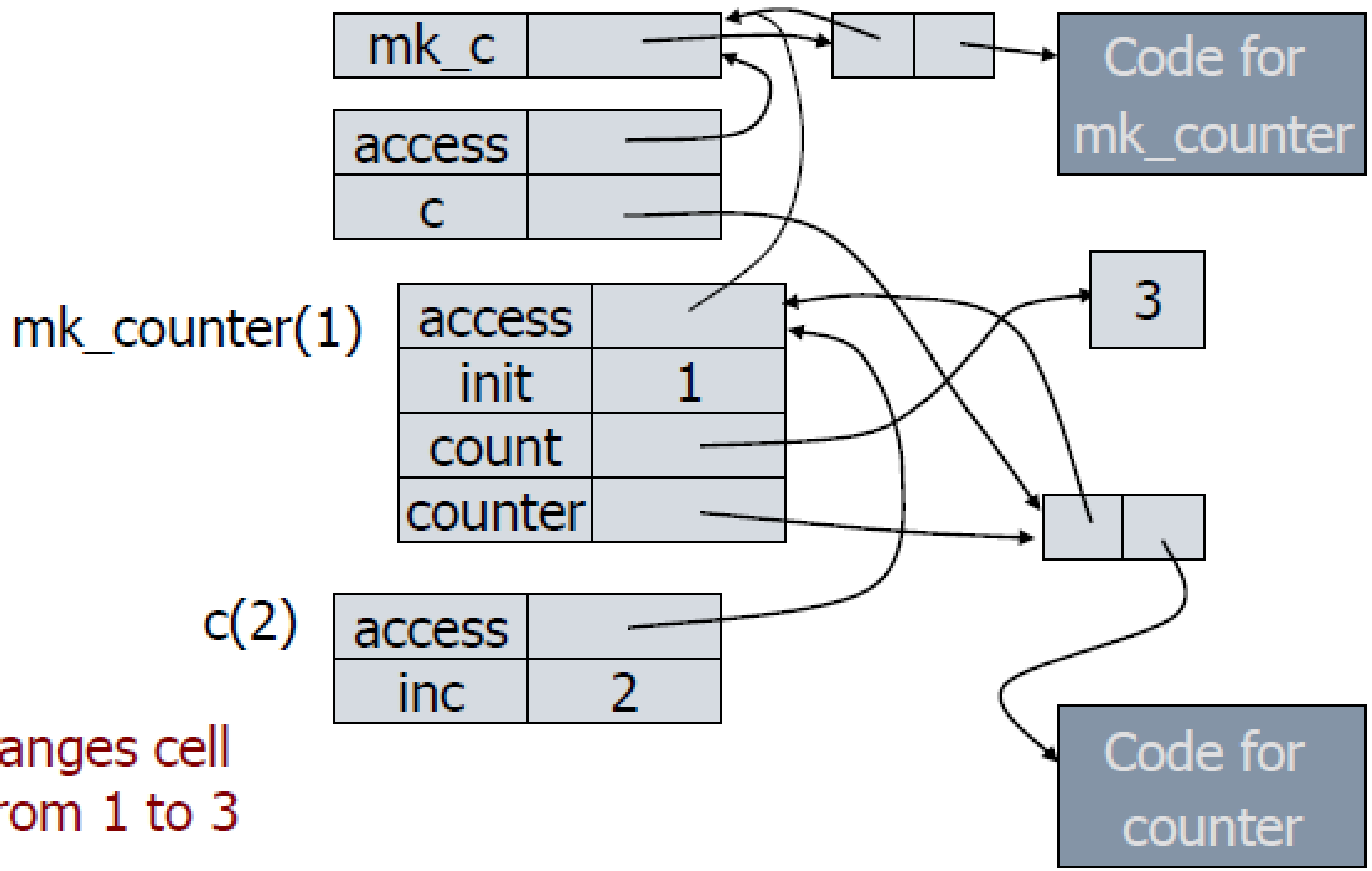
- Function to "make counter" returns a closure
- How is correct value of count determined in c(2) ?

Function Results and Closures

University of Science and Technology of China

```
fun mk_counter (init : int) =
  let val count = ref init
      fun counter(inc:int) = (count := !count + inc; !count)
      in  counter end
  end;
val c = mk_counter(1);
c(2) + c(2);
```

ML

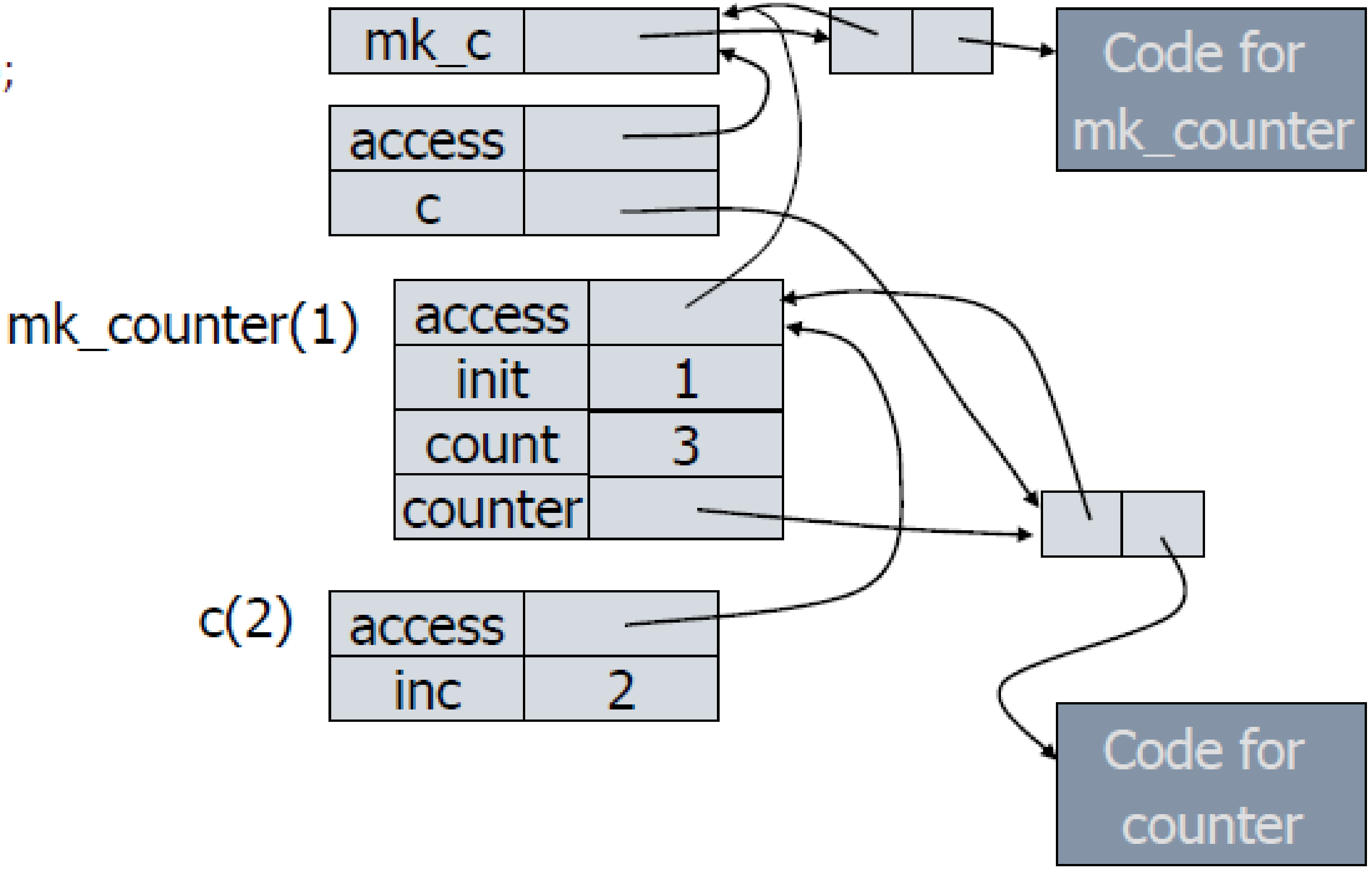


Call changes cell value from 1 to 3

Function Results and Closures

JS

```
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);
```



Summary of Scope Issues

- **Block-structured language uses stack of activation 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 do *not* needed if functions not in nested blocks**

Closures via "Upvalues"

- **Lua authors wanted lexical scoping (词法作用域/静态作用域) early on**
  - **difficult due to technical restrictions**
    - wanted to keep a simple array stack for activation records
    - one-pass compiler
- **Lua 3.1 with a compromise called *upvalues***
  - **In creating a function, make (frozen) copies of the values of any external variables used by a function.**

```
function f ()                高阶函数：void →(void→int)
  local b = 1
  return (function () return %b + 1 end)  // b是外部的局部变量, upvalue
end
return f()() --> 2       upvalue 有些像C的static局部变量
```

Full Lexical Scoping

- **Lua 5.0 got the real thing**
- **Solution: “Keep local variables in the (array-based) stack and only move them to the heap if they go out of scope while being referred by nested functions." (JUCS 11 #7)**

```
function f ()
  local b = 1
  local inc_b = (function () b = b + 1 end)
  inc_b()
  return (function () return b end)
end
return f()() --> 2
```

closure：一个匿名函数加上其可访问的upvalue

Tail Calls

- tail calls supported since 5.0
  - called function reuses the stack entry of the calling function
    - erases information from stack traces
- only for statements of the form `return f(...)`
  - `return n * fact(n-1)` does not result in a tail call

Coroutines

- *coroutines*—a general control abstraction
  - term introduced by Melvin Conway in 1963
  - has lacked a precise definition, but implies “the capability of keeping state between successive calls"
- have not been popular in mainstream languages
  - but used in Go
- classification:
  - *full coroutines* are stackful, and first-class objects
    - stackful coroutines can suspend their execution from within nested functions
  - an *asymmetric coroutine* is “subordinate” to its caller—can yield, caller can resume

Coroutines in Lua

- constraints: portability and C integration
  - cannot manipulate a C call stack in ANSI C
  - impossible: first-class continuations (as in Scheme), symmetric coroutines (e.g., in Modula-2)
- Lua 5.0 got *full asymmetric coroutines*, with `create`, `resume` and `yield` operations
  - ...and PUC-Rio guys gave proof of ample expressive power
    里约热内卢天主教大学
  - capture only a partial continuation, from `yield` to `resume` — cannot have C parts there

协程有现场保护

Coroutine Example

中国科学技术大学
University of Science and Technology of China

```
> return (string.gsub("abbc", "b",
        function (x) return "B" end))
aBBc
> return (string.gsub("abbc", "b",
        coroutine.wrap(function (x)
                coroutine.yield("B")
                coroutine.yield("C")
                end)))
aBCc
```