

Scope, Functions, and Storage Management



Implementing Functions and Blocks

Topics

- ❑ Block-structured languages and stack storage
- ❑ In-line Blocks
 - Storage for local, global variables
- ❑ First-order functions
 - Implementing function call and return
 - Passing parameters and returning result
- ❑ Higher-order functions
 - deviations from stack discipline
 - language expressiveness => implementation complexity
- ❑ Additional issues
 - parameter passing
 - tail recursion and iteration

Blocks in C/C++

```
outer block {  
    {  
        int x = 2;  
        {  
            int y = 3;  
            x = y+2;  
        }  
    }  
}
```

inner block

- Blocks: regions of code that introduces new variables
 - Blocks are nested but not partially overlapped
 - What about jumping into the middle of a block?
 - Local variable: created by the current block
 - Global variable: created by surrounding blocks
- Storage management
 - Enter block: allocate space for variables
 - Exits block: some or all space may be deallocated

Blocks in Functional languages

- ML:

```
let fun g(y) = y + 3
in
  let
    fun h(z) = g(g(z))
  in h(3)
  end
end;
```

- Lisp:

```
( (lambda (g)
  ( (lambda (h) (h 3)) (lambda (z) (g (g z))) )
  (lambda (y) (+ y 3)) )
```

Summary of Blocks

- Blocks in common languages
 - C { ... }
 - Algol begin ... end
 - ML let ... in ... end
- Two forms of blocks
 - In-line blocks
 - Blocks associated with functions or procedures
- Topic: block-based memory management
 - Access to local variables, global variables, and function parameters

Managing Data Storage In a Block

- ❑ Local variables
 - Declared inside the current block
 - ❑ Enter block: allocate space
 - ❑ Exit block: de-allocate space
- ❑ Global variables
 - Declared in a enclosing block
 - ❑ Already allocated before entering current Block
 - ❑ Remain allocated after exiting current block
- ❑ Function parameters
 - Input parameters
 - ❑ Allocated and initialized before entering function body
 - ❑ De-allocated after exiting function body
 - Return parameters
 - ❑ Address remembered before entering function body
 - ❑ Value set after exiting function body

Scoping rules

Finding non-local (global) variables

- Global and local variables

outer block	x	0
h(3)	z	3
	x	1
g(12)	z	3

```
{ int x=0;  
  fun g(z) = x+z;  
  fun h(z) = let x = 1 in  
              g(z) end;  
  h(3)  
};
```

- Static scope

- Find global declarations in the closest enclosing blocks in program text

- Dynamic scope

- Find global variables in the most recently entered blocks

Managing Blocks

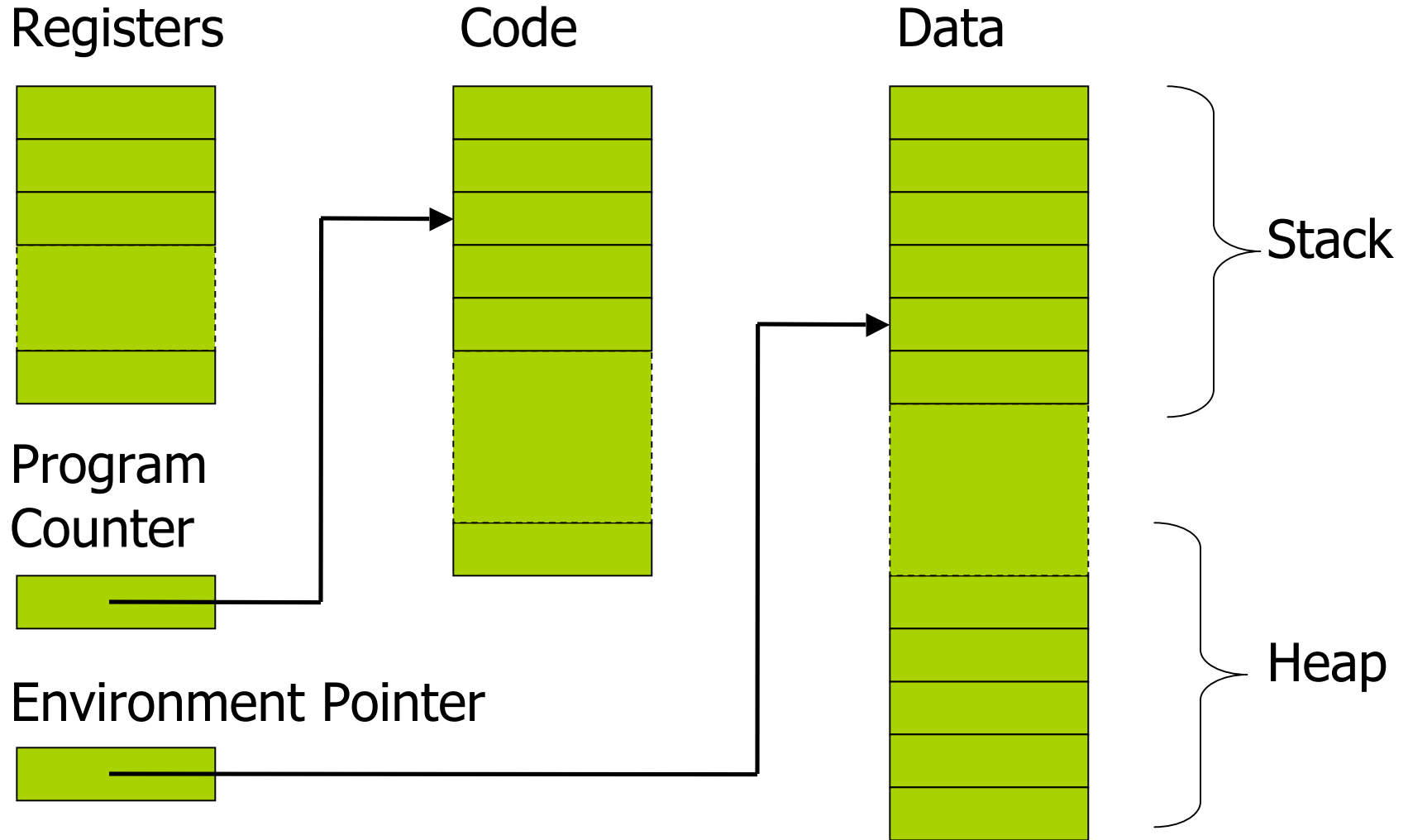
- Activation record: data structure allocated on runtime stack
 - Contains space for local variables in the block determined as compile time
 - Contains values of local variables
 - Evaluated at runtime
- Before evaluating each block, push it's activation record onto a runtime stack; after exit the block, pop activation record off stack

```
{ int x=0;  
  int y=x+1;  
    { int z=(x+y)*(x-y);  
      };  
};
```

```
Push record with space for x, y  
Set values of x, y  
  Push record for inner block  
  Set value of z  
  Pop record for inner block  
Pop record for outer block
```

May need space for variables and intermediate results like
(x+y), (x-y)

Simplified Machine Model



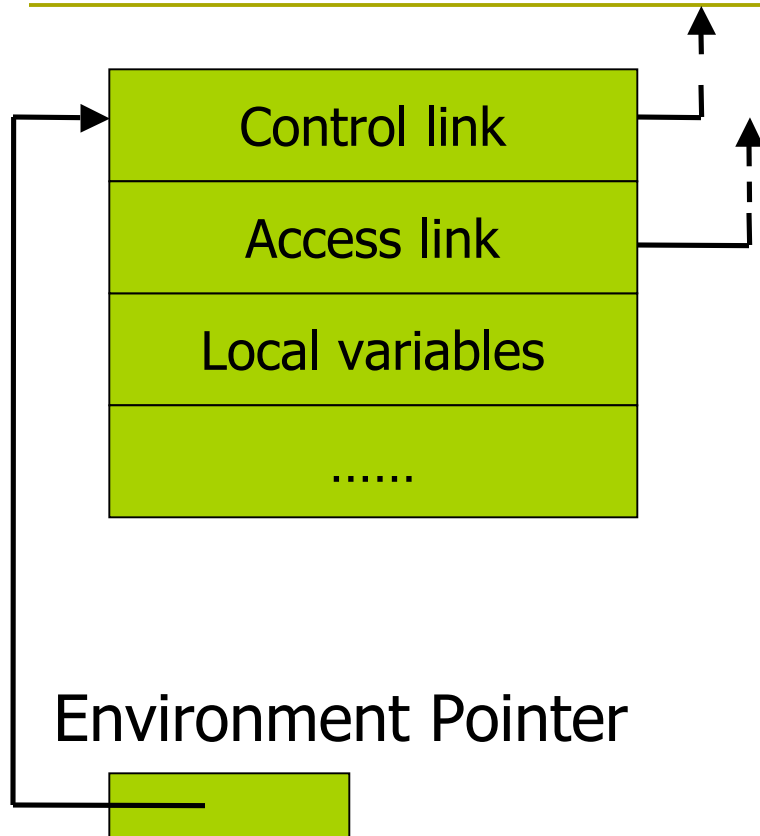
Data Storage Management

- Runtime stack
 - contains data related to block entry/exit
 - Environment pointer: current stack position
 - Block entry: add new data to stack
 - Block exit: remove outdated data
- Heap
 - Contains data of varying lifetime
 - Variables that last throughout the program
 - Data pointed to by variables on the runtime stack
- Environment pointer points to current stack position
 - Block entry: add new activation record to stack
 - Block exit: remove most recent activation record
- Registers, code segment and program counter
 - Ignore registers
 - Details of instruction set will not matter

Managing Activation Records

- Compilers generate instructions for
 - Pushing and popping activation records
 - Deciding the size of activation records
 - Connecting control links
- Finding locations of local variables
 - Calculating the offset of each local variable
 - Location = environment pointer + offset
- Finding locations of global variables
 - How to find the scope of the variable?

Activation record for inline blocks (static scoping)



- Control link
 - Link to activation record of previous (calling) block
 - Depends on dynamic behavior of program
- Access link
 - Link to activation record of immediate enclosing block
 - Depends on static form of program text
- Push record on stack
 - Set new control link to env ptr
 - Set env ptr to new record
- Pop record off stack
 - Follow current control link to reset environment pointer

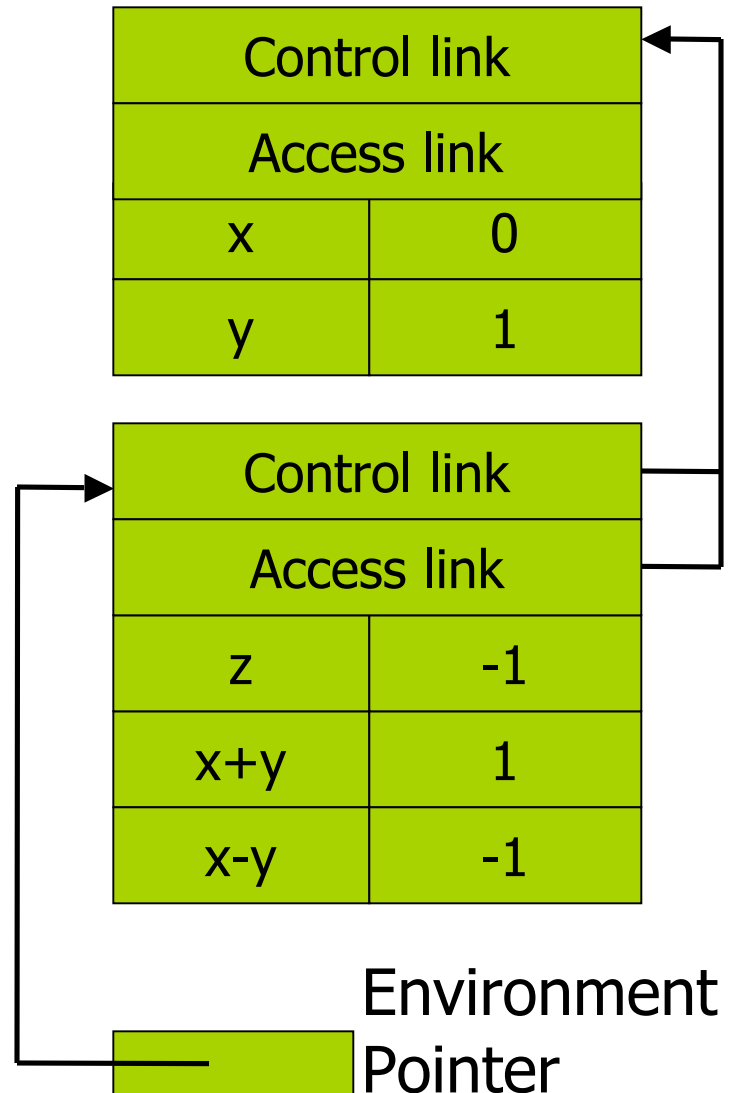
Example: inline blocks

```
{ int x=0;  
  int y=x+1;  
  { int z=(x+y)*(x-y);  
  };  
};
```

```
Push record with space for x, y  
Set values of x, y  
  Push record for inner block  
  Set value of z  
  Pop record for inner block  
Pop record for outer block
```

Control link	
Access link	
x	0
y	1

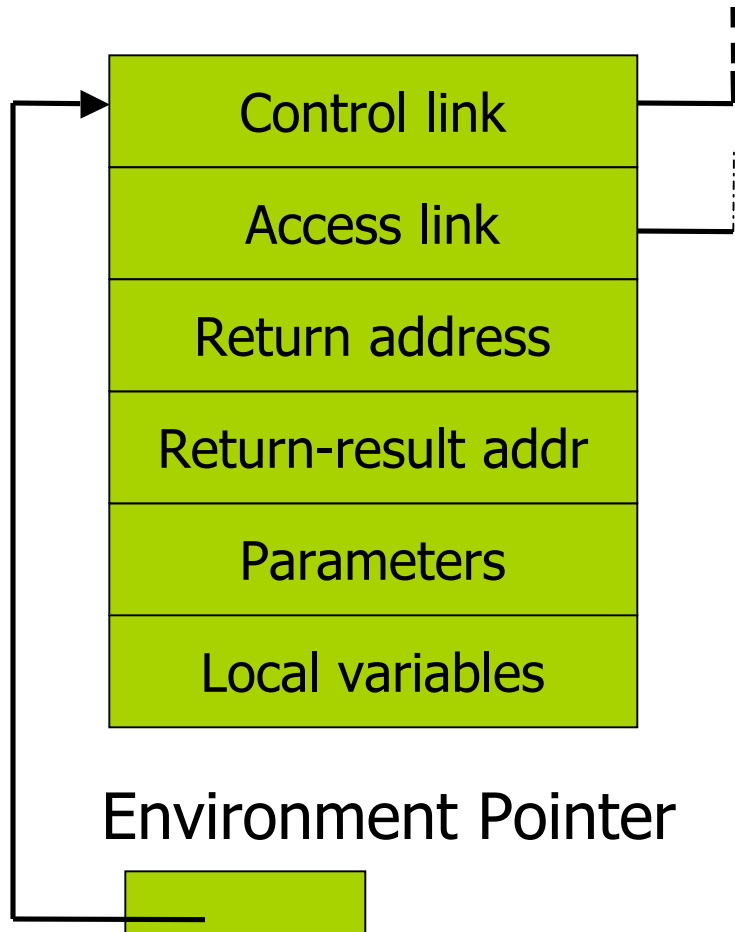
Control link	
Access link	
z	-1
x+y	1
x-y	-1



Functions and procedures

- Functions: parameterized blocks
 - `(fn (x, y) => x + y) (2,3);`
 - `((lambda (x y) (+ x y)) 2 3)`
 - Formal and actual parameters
 - `x, y` : formal parameters
 - `2, 3`: actual parameters for `x, y`
- Activation record for functions include
 - Parameters
 - Return address
 - The instruction to jump to after finishing evaluation
 - Return Result Address
 - Location to put return value before exit

Activation record for functions



□ Return address

- Where to continue execution after exit
- Pointer to code space

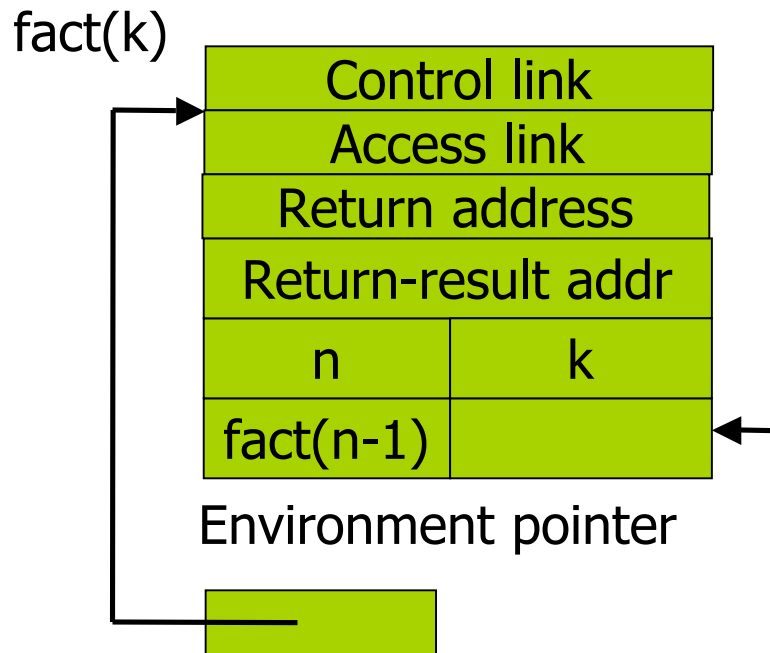
□ Return-result address

- Where to put return result
- Pointer to caller's activation record

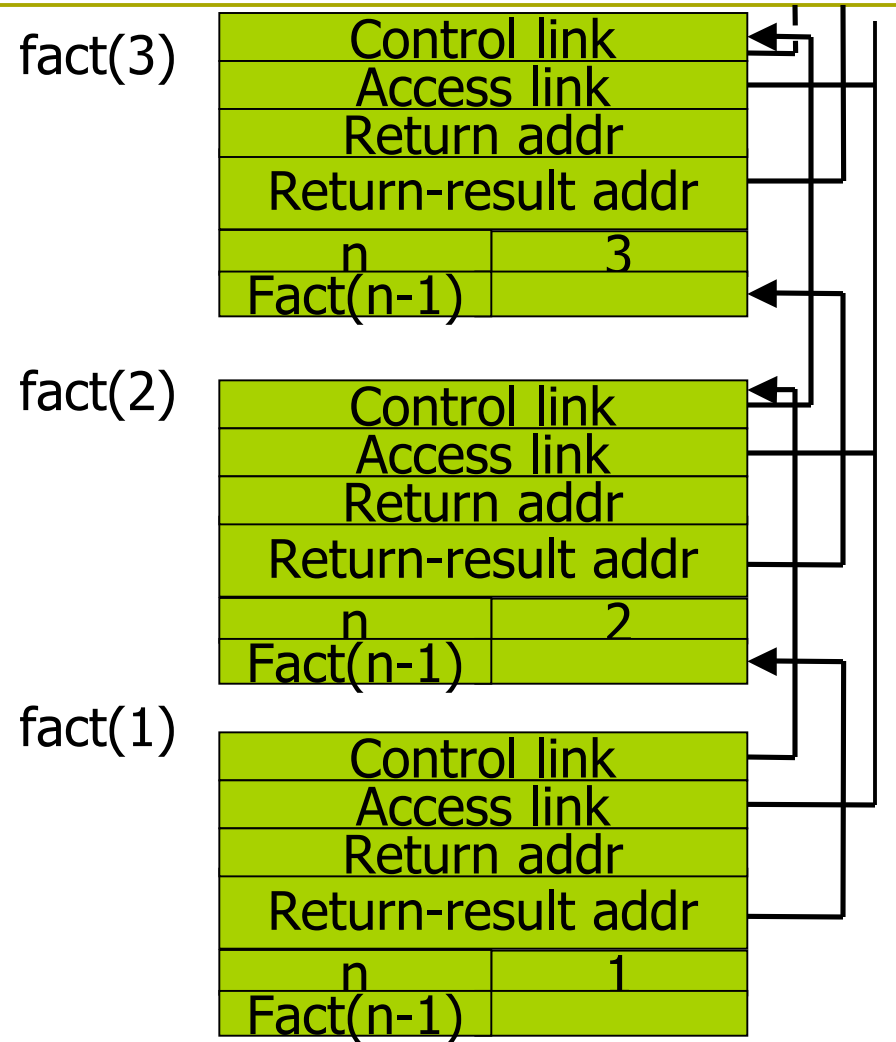
□ Parameters

- Values for formal parameters
- Initialized with actual parameters

Example: function calls



fact(n) = if $n \leq 1$ then 1
 else $n * \text{fact}(n-1)$



Function Abstraction As Values

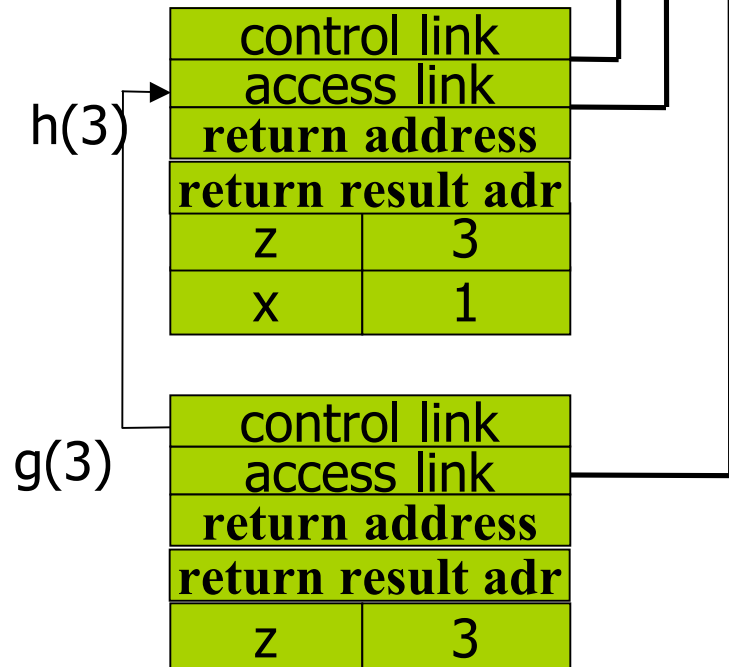
```

let val x=1;
  fun g(z) = x+z;
  fun h(z) =
    let x = 2 in
      g(z) end
in h(3)
end;
  
```

- What are values for g and h?
- How to determine their access links?
 - Nameless (inlined) blocks
 - Access link = control link
 - Named (function) blocks
 - Enclosing block of name declaration

outer block

x	1
g	...?
h	...?



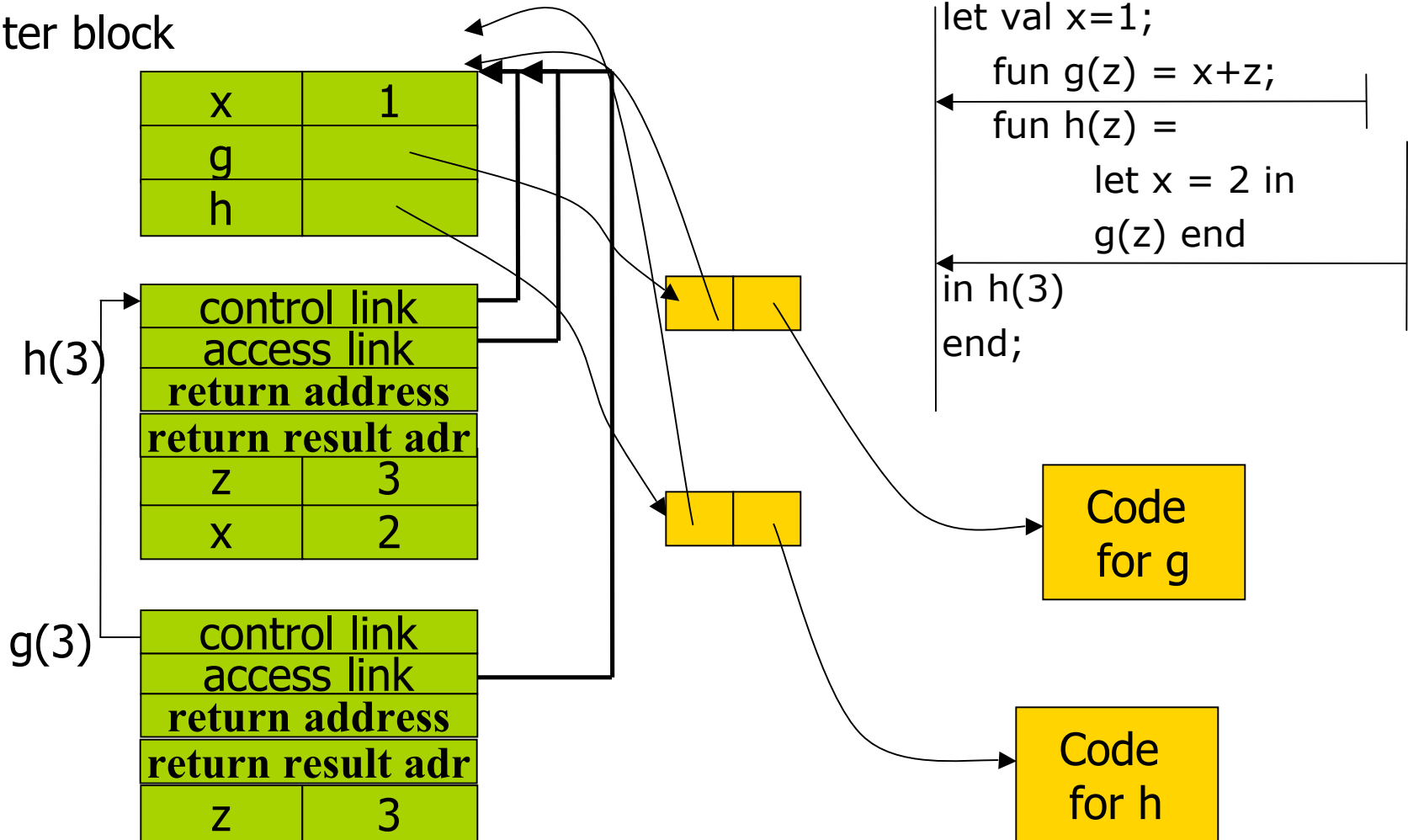
Closures

- A function value is a *closure: env, code*
 - code: a pointer to the function body in code space
 - env: current runtime stack
 - Activation records of enclosing scopes

- When a function is called,
 - Retrieve value (closure) of function using its name
 - Push a new activation record onto runtime stack
 - Set control link to environment pointer, modify environment pointer
 - Set return address, return value addr, parameters and local variables
 - Set access link to the activation record ptr in closure
 - Set program pointer to the code pointer in closure

Function Values as Closures

outer block



Summary: Function Parameters

- Use closure to maintain
 - A code pointer to the body of the function
 - A data pointer to the enclosing environment of function body
- When called, set access link data pointer from closure, then jump to the code pointer
- All access links point “up” in stack
 - May jump past activation records to find global variables
 - Still push and pop activation records using stack (last-in-first-out) order

Parameter passing

- Each function definition introduces a sequence of formal parameters that are used as local variables in the function body
 - When the function is called, each formal parameter is matched against an actual parameter
 - What is the relation between each pair of formal-actual parameter?
- Pass-by name
 - Renaming each occurrence of a formal parameter in function body with its actual parameter --- delay of evaluation
 - Used in Lambda calculus and side-effect free languages
- Pass-by-value
 - Each formal parameter is mapped to the value of its actual parameter
 - Callee cannot change values of actual parameters
- Pass-by-reference
 - Each formal parameter is mapped to the storage of its actual parameter
 - Callee can change values of actual parameters
 - Formal parameters in activation record may be aliased
 - Aliasing: two names refer to same location

Example: What is the final result?

pseudo-code

```
int f (int x)
{
  x := x+1; return x;
};
main() {
  int y = 0;
  print f(y)+y;
}
```

pass-by-ref



pass-by-value



Standard ML

```
fun f (x : int ref) =
  ( x := !x+1; !x );
val y = ref 0 : int ref;
f(y) + !y;
```

```
fun f (z : int) =
  let val x = ref z in
    x := !x+1; !x
  end;
val y = ref 0 : int ref;
f(!y) + !y;
```

Return Function as Result

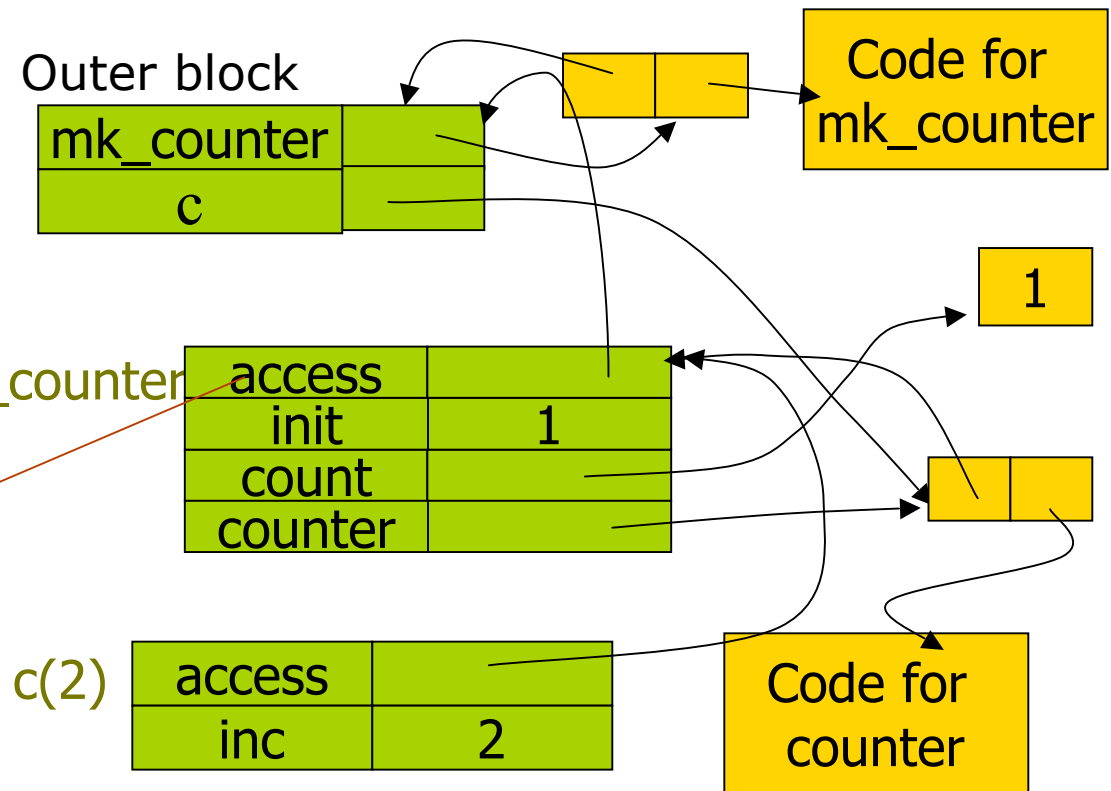
- Language feature
 - Functions that return “new” functions
 - E.g. `fun compose(f,g) = (fn x => g(f x));`
- Function “created” dynamically
 - Each function value is a closure = (env, code), where code may contain references to variables in env
 - code *not* “created” dynamically (static compilation)
- Need to save the runtime environment of function as a pointer to enclosing activation records
 - But the enclosing activation record may have been popped off the runtime stack
 - Returning functions as results not allowed in C
 - Just like returning pointers to local variables

Example(skip): Function Closure as Returned Result

```

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);
  
```

**must be saved after
being popped from
runtime stack**



Summary: Return Function Results

- Use a closure to maintain the runtime environment of a function
- May need to keep activation records of functions even after the function calls return
 - Stack (last-in-first-out) order fails!
- To support return functions as results, need to extend the standard “stack” implementation
 - Put activation records on heap
 - Invoke garbage collector as needed
 - Not as crazy as it sounds
 - May only need to search reachable data

Tail recursion

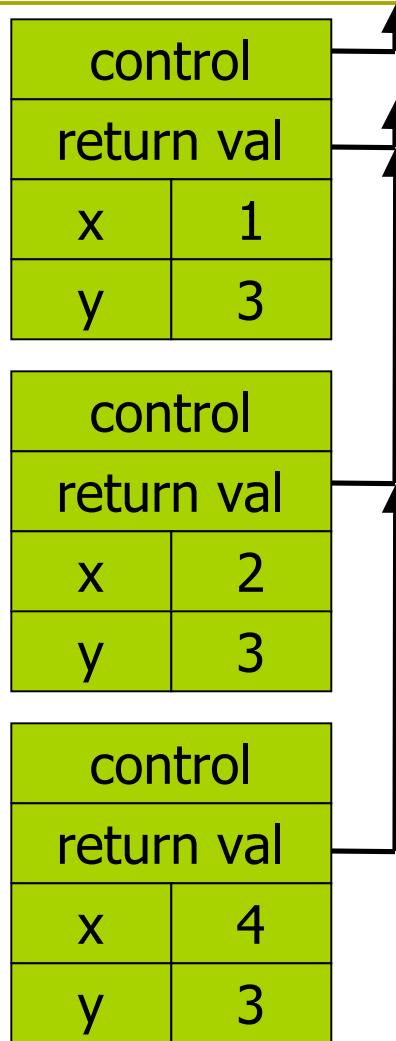
- *A function call from g to f is a tail call*
 - *if g returns the result of calling f with no further computation*
- Example
 - `fun g(x) = if x>0 then f(x) else f(x)*2`
 - **tail call** (points to `f(x)`)
 - **not a tail call** (points to `f(x)*2`)
- Optimization
 - Can pop activation record on a tail call
 - Especially useful for recursive tail call (f to f)
 - Callee's activation record has exactly same form
 - Callee can reuse activation record of the caller
 - The sequence of recursive calls can be translated into a loop

Example: what is the result?

f(1,3)

control		↑
return val		↑
x	1	
y	3	

```
fun f(x,y) = if x>y
  then x
  else f(2*x, y);
f(1,3) + 7;
```

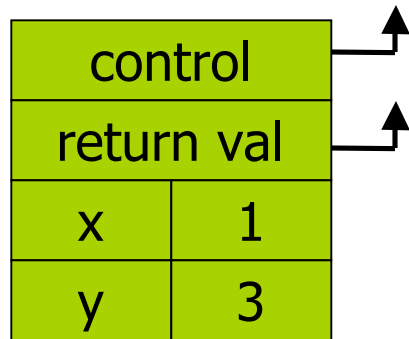


Expressed in loop:

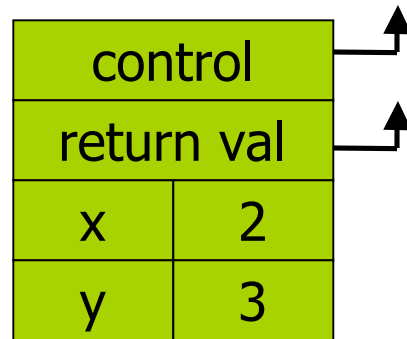
```
fun f(x,y) =
  let val z = ref x in
  while not (!z > y) do
    z := 2 * !z;
  !z
  end;
f(1,3) + 7;
```

Tail recursion elimination

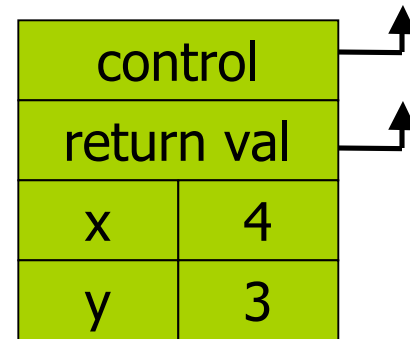
f(1,3)



f(2,3)



f(4,3)



```
fun f(x,y) = if x>y
  then x
  else f(2*x, y);
f(1,3);
```

Optimization: pop followed by push
=> reuse activation record in place

Conclusion: tail recursive function
calls are equivalent to iterative loops

Tail recursion and iteration

- ❑ Non-tail recursive function

```
fun mapList(f, nil) = nil
  | mapList(f, x::y) =
      f(x)::mapList(f,y);
```
- ❑ Tail recursive function

```
fun last(x::nil) = x
  | last(x::y) = last(y);
```
- ❑ Iteration

```
fun last(input) =
  let val y= ref input
  in while not(tl(!y)=nil) do
      y := tl(!y)
  end;
  hd(!y)
end
```

- ❑ Step1: what parameters change when making recursive calls?
 - create a reference for each changed parameter.
 - NOTE: no need to create reference for the return result
 - ❑ Tail recursion only returns at the base case
- ❑ Step2: what is the base case of recursion?
 - This is the stop condition for the while loop.
- ❑ Step3: what to do before making tail call?
 - loop body: prepare for the next tail call
- ❑ Step4: return base case value.

Summary of scope issues

- Block-structured languages use runtime stack to maintain activation records of blocks
 - Activation records contain parameters, local variables, ...
 - 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 management may fail if function returned from call
 - Closures *not* needed if functions not in nested blocks
 - Example: C