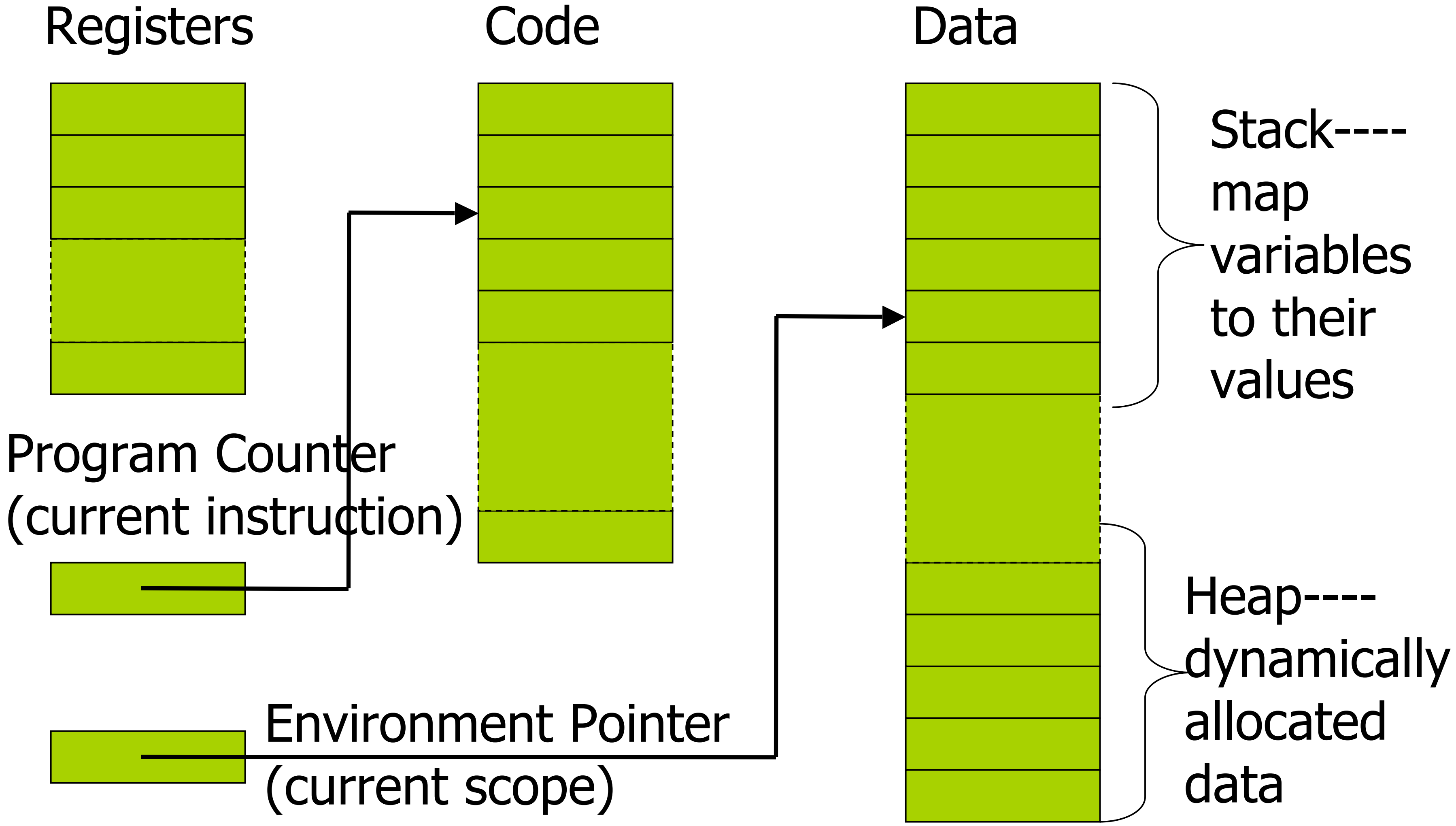
Scope, Functions, and Storage Management

Implementing Functions and Blocks

Simplified Machine Model (Compare To List Abstract Machine)



Registers

Code

Data

Stack---- map variables to their values

Program Counter (current instruction)

Environment Pointer (current scope)

Heap---- dynamically allocated data

Data Storage Management

- Runtime stack: mapping variables to their values
  - When introducing new variables: push new stores to stack
  - When variables are out of scope: pop outdated storages
- Environment pointer: current stack position
  - Used to keep track of storages of all active variables
- Heap: dynamically allocated data of varying lifetime
  - Variables that last throughout the program
  - Data pointed to by variables on the runtime stack
  - Target of garbage collection
- The code space: the whole program to evaluate
- Program counter: current/next instruction to evaluate
  - keep track of instructions being evaluated
- Registers: temporary storages for variables

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
  - Enter block: allocate space for variables
  - Exits block: some or all space may be deallocated
- Blocks are nested but not partially overlapped
  - Jumping out of a block
    - Make sure variables are freed before exiting
  - What about jumping into the middle of a block?
    - Variables in the block have not yet been allocated

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

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 previously entered 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 values
    - Address remembered before entering function body
    - Value set after exiting function body
- Scoping rules: where to find memory allocated for variables?
  - Need to find the block that introduced the variable

Parameter passing

- Each function have a number of formal parameters
  - At invocation, they are matched against actual parameters
- Pass-by name
  - Rename each occurrence of formal parameter with its actual parameter --- delay of evaluation
  - Used in Lambda calculus and side-effect free languages
- Pass-by-value
  - Replace formal parameter with value of its actual parameter
  - Callee cannot change values of actual parameters
- Pass-by-reference
  - Replace formal parameter with address of its actual parameter
  - Callee can change values of actual parameters
  - Different formal parameters may have the 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

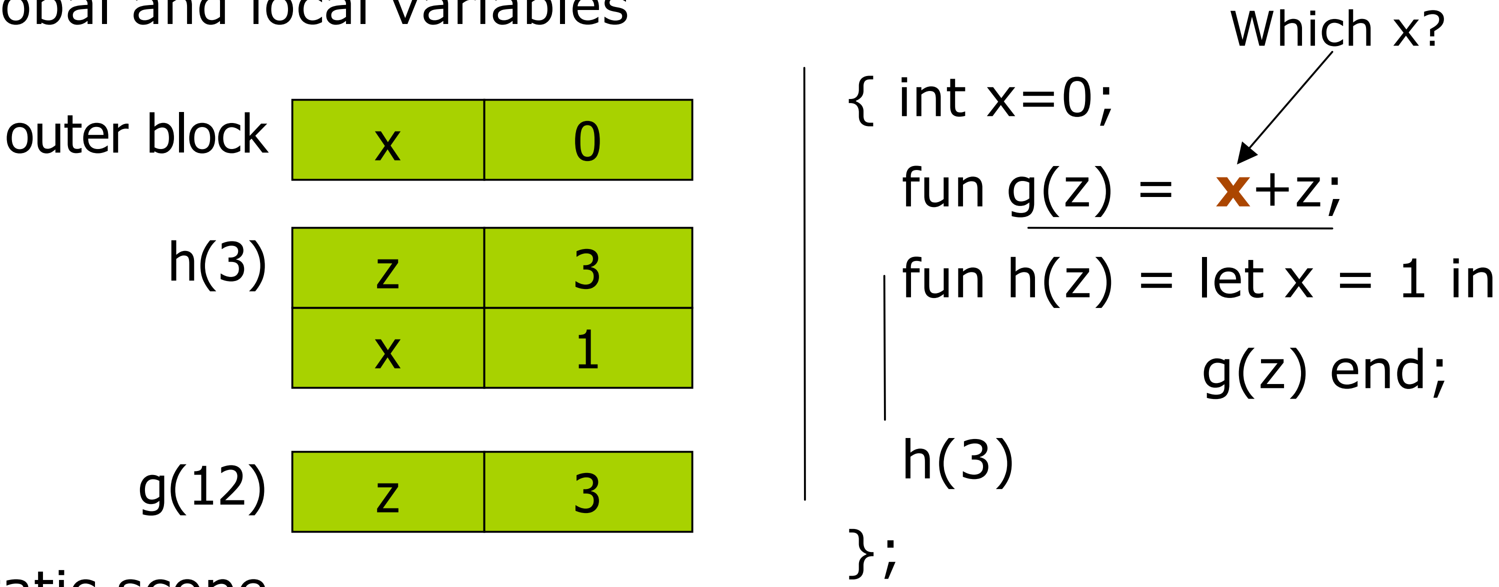
Standard ML

```
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;
```

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

Which x?

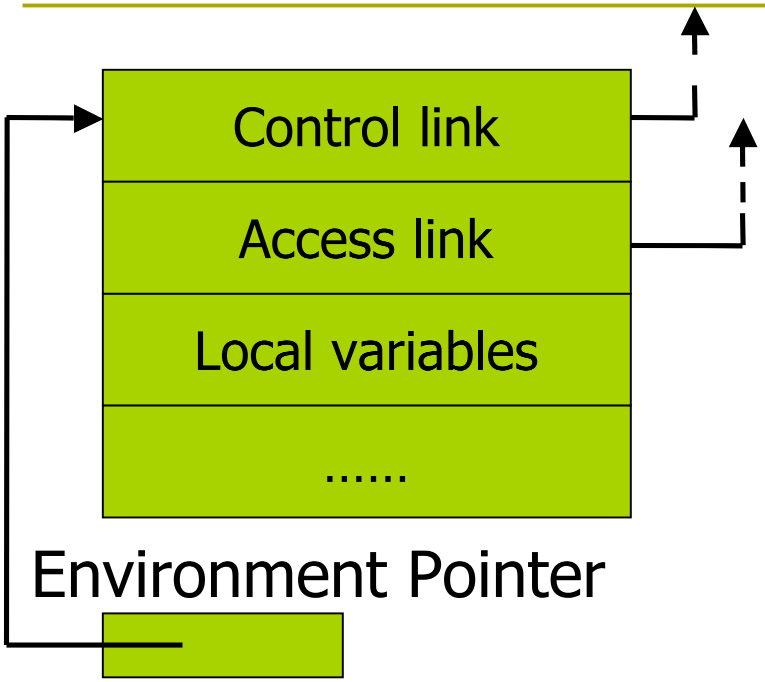
```
{ int x=0;
   fun g(z) =  x+z;
   fun h(z) = let x = 1 in
                    g(z) end;
   h(3)
};
```

- Static scope
  - Find global declarations in the closest enclosing blocks in program text
- Dynamic scope
  - Find global variables in the most recent activation record

Managing Blocks

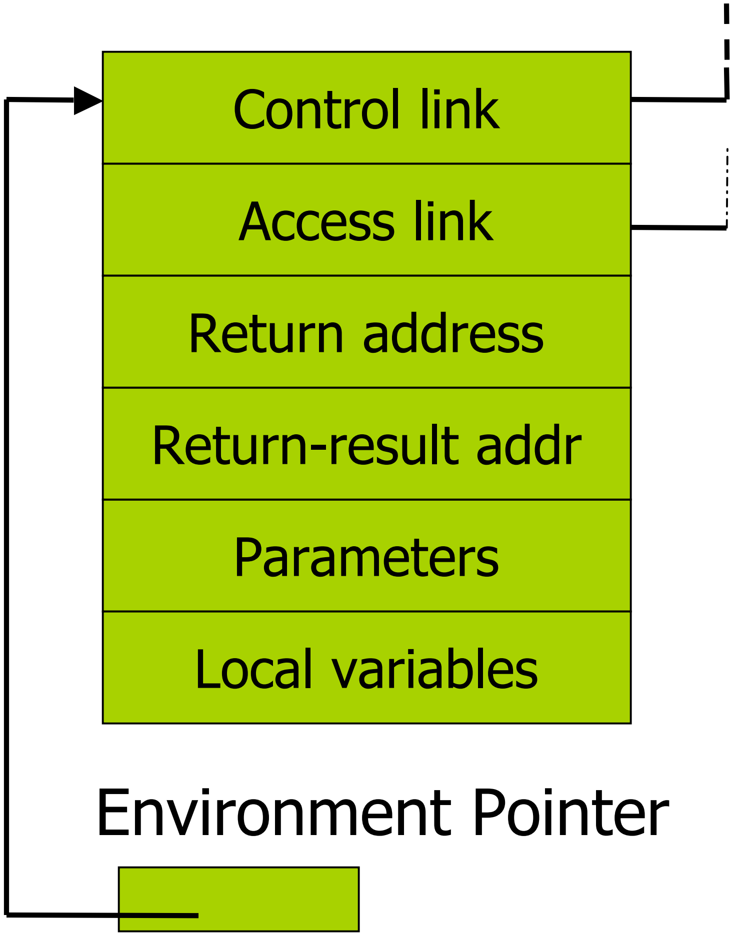
- Activation record: memory storage for each block
  - Contains values for local variables in the block
- Managing Activation Records
  - Allocated on a runtime stack: First-In-Last-Out
  - Before evaluating each block, push its activation record onto runtime stack; after exit the block, pop its activation record off stack
  - Compilers generate instructions for pushing & popping of activation records (pre-compute their sizes)
- Finding locations of local variables
  - Compiler calculate the offset of each variable
  - Dynamically find activation record of introducing block
  - Location = activation record pointer + offset

Activation Record For Inline Blocks



- Control link
  - Point to activation record of previous (calling) block
  - Depend on runtime behavior
  - Support push/pop of ARs
- Access link
  - Point to activation record of immediate enclosing block
  - Depend on static form of program
- 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

Activation Records For Functions



Control link

Access link

Return address

Return-result addr

Parameters

Local variables

Environment Pointer

- Return address
  - Where to continue execution after return
  - Pointer to the next instruction following the function call
- Return-result address
  - Where to put return result
  - Pointer to caller’s activation record
- Parameters
  - Values for formal parameters
  - Initialized with the actual parameters

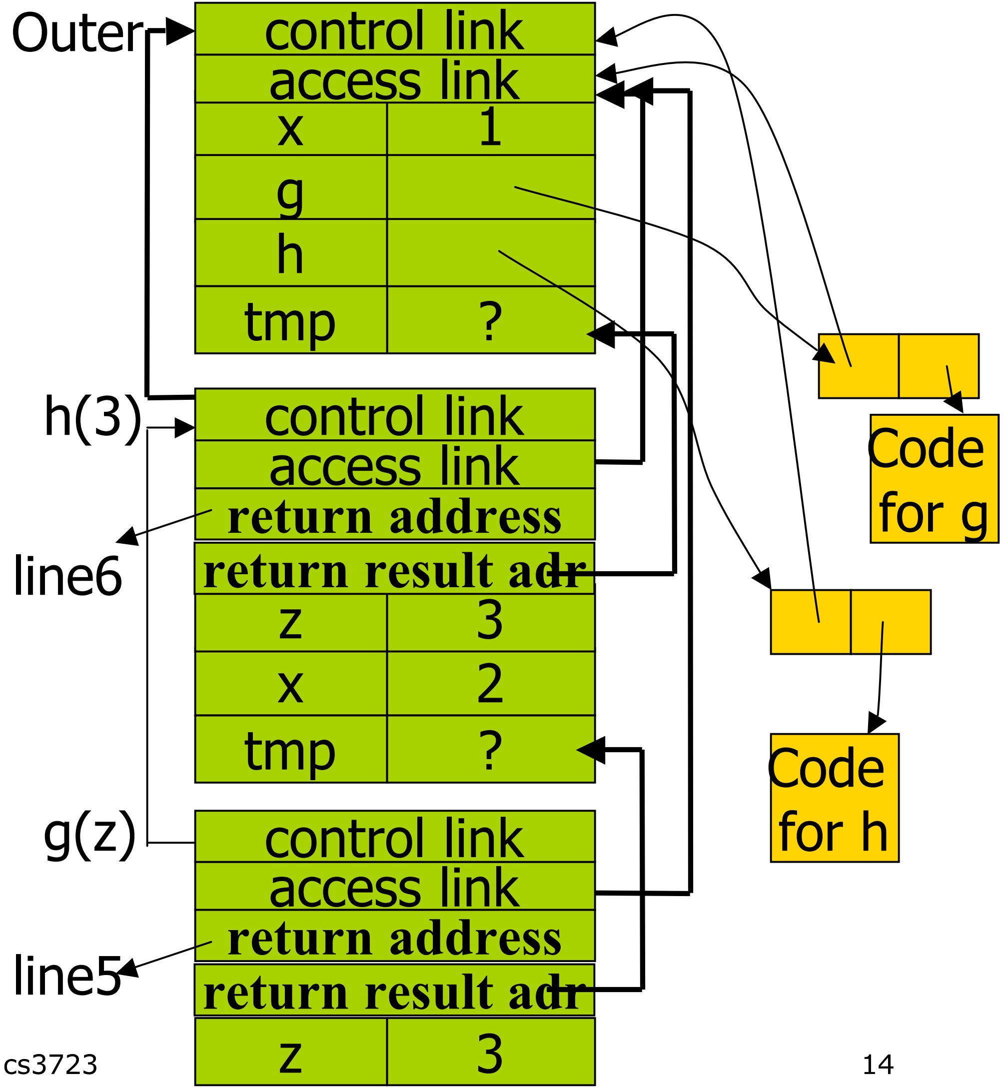
Function Abstraction As Values

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

- What are values for g,h?
- How to determine their access links?
  - Inlined blocks

    Access link = control link
  - Function blocks

    Enclosing block of the function definition



Outer

control link	
access link	
x	1
g	
h	
tmp	?

h(3)

line6

control link	
access link	
return address	
return result adr	
z	3
x	2
tmp	?

g(z)

line5

control link	
access link	
return address	
return result adr	
z	3

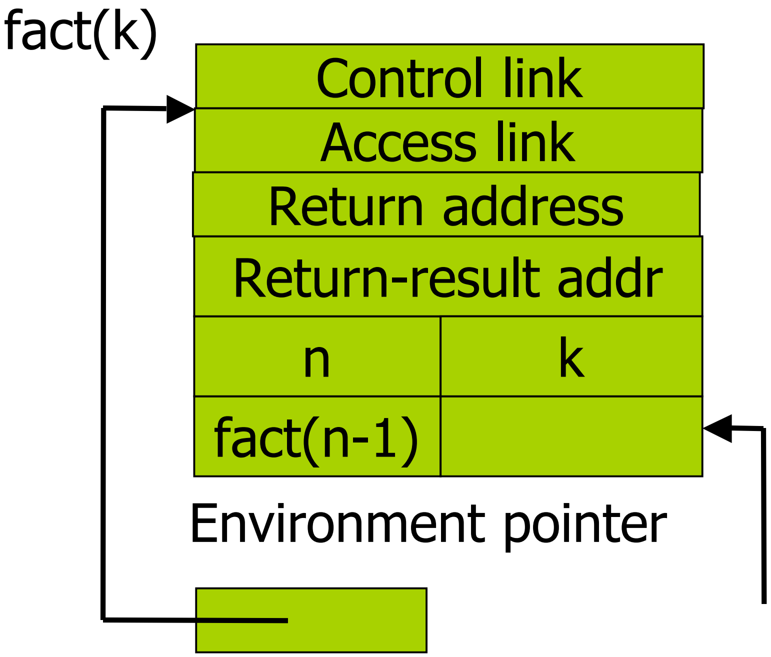
Code for g

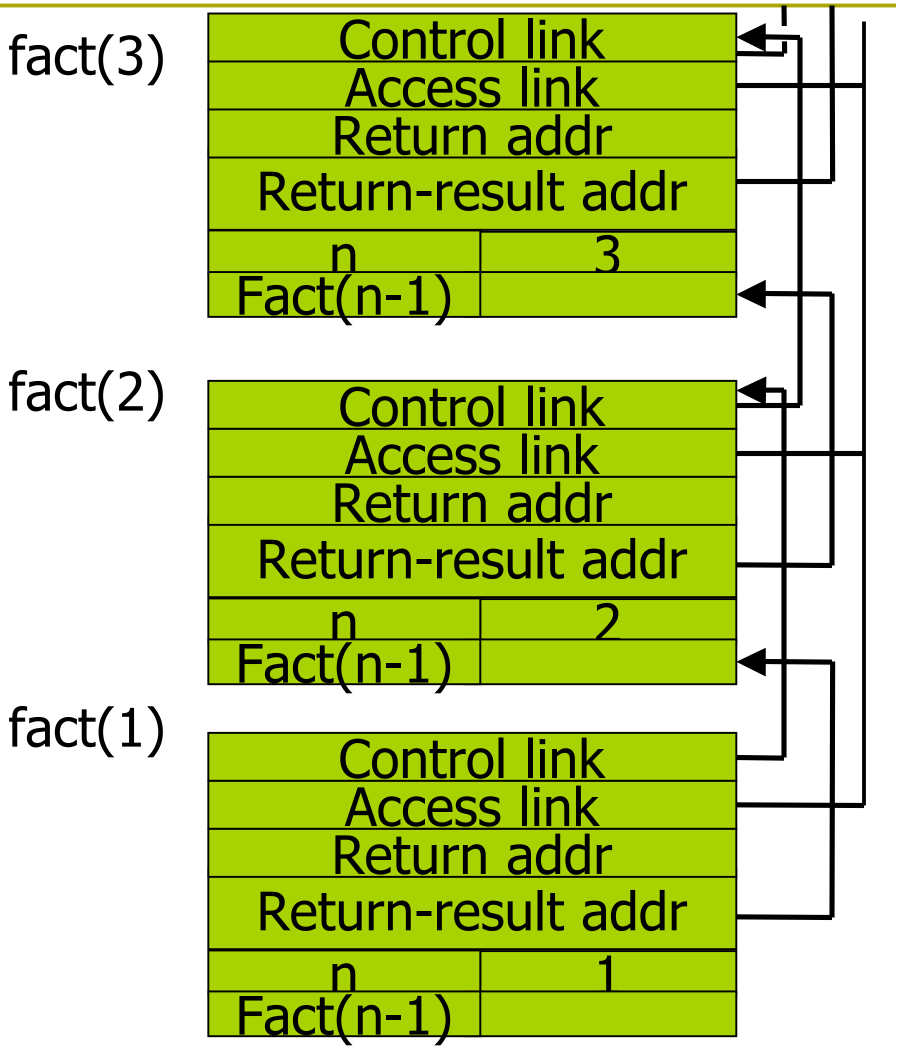
Code for h

Closures

- A function value is a closure: (env, code)
  - code: a pointer to the function body
  - env: activation record of the enclosing block
- Use closure to maintain a pointer to the static environment of a function body
  - When called, set access link from closure
- When a function is called,
  - Retrieve the closure of the function
  - Push a new activation record onto runtime stack
  - Set return address, return value addr, parameters and local variables
  - Set access link to equal to the env pointer in closure
  - Start the next instruction from code pointer in closure

Example: Function Calls





```
fact(n) = if n<= 1  then 1
          else n * fact(n-1)
```

Return Function as Result

- Language feature: functions that return new functions
  - E.g. **`fun compose(f,g) = (fn x => g(f x));`**
  - Each function value is a closure = (env, code), where code may contain references to variables in env
  - Code is not "created" dynamically (static compilation)
- Use a closure to save the runtime environment of function
  - Environment: pointer to enclosing activation records
  - But the enclosing activation record may have been popped off the runtime stack
  - Returning functions as results is not allowed in C
    - Just like returning pointers to local variables
- Need to extend the standard "stack" implementation
  - Put activation records on heap
  - Invoke garbage collector as needed
  - Not as crazy as is sounds

Tail Call And Tail Recursion

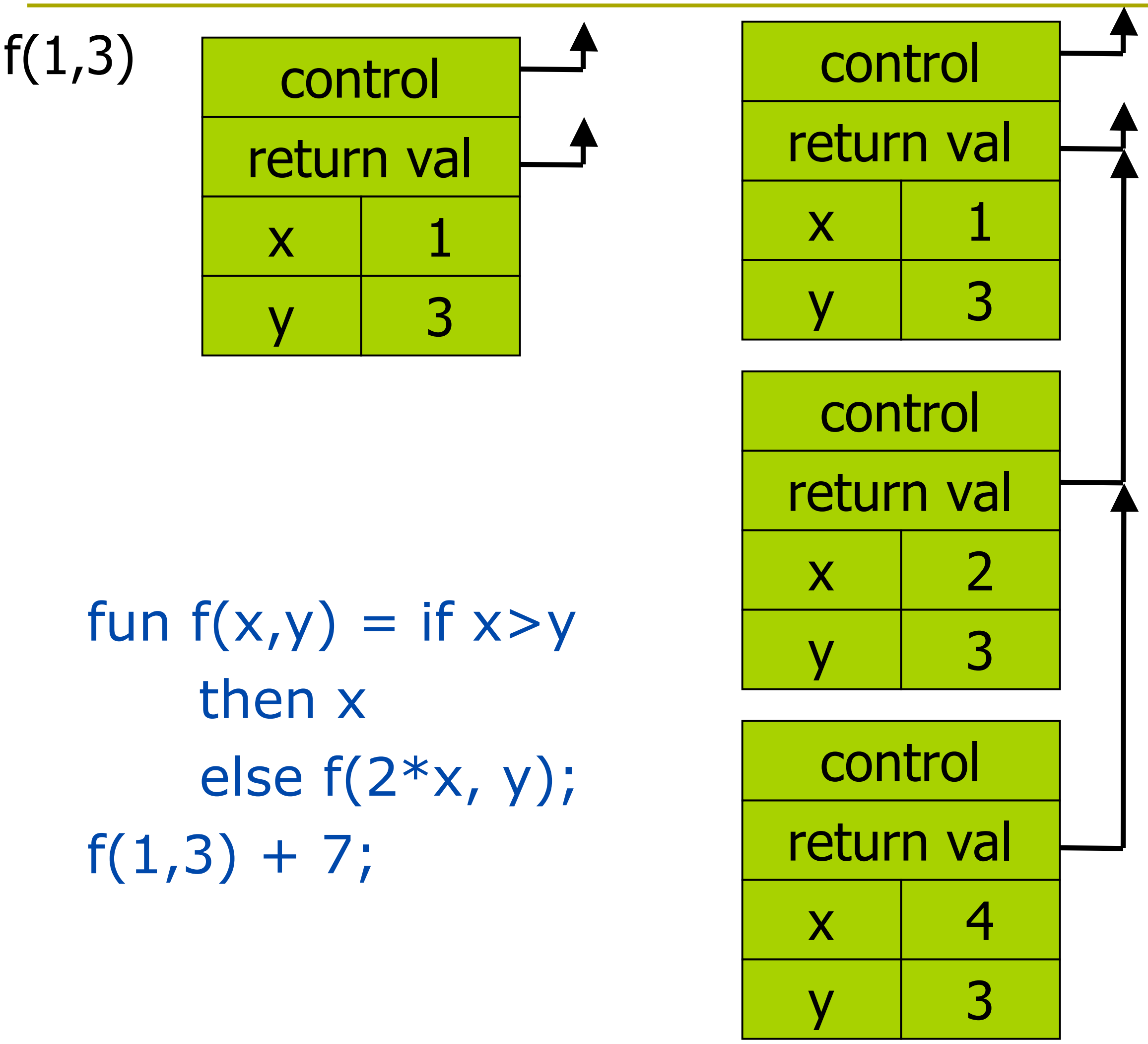
- *A function call from g to f is a tail call*
  - *if g returns the result of calling f with no further computation*
- Example

  tail call

  not a tail call

  - fun g(x) = if x>0 then f(x) else f(x)*2
- Optimization
  - Can pop activation record on a tail call
  - Especially useful for a tail recursive call (f to f)
    - Callee's activation record has exactly same form
    - Callee can reuse activation record of the caller
    - A sequence of tail recursive calls can be translated into a loop

Example: what is the result?



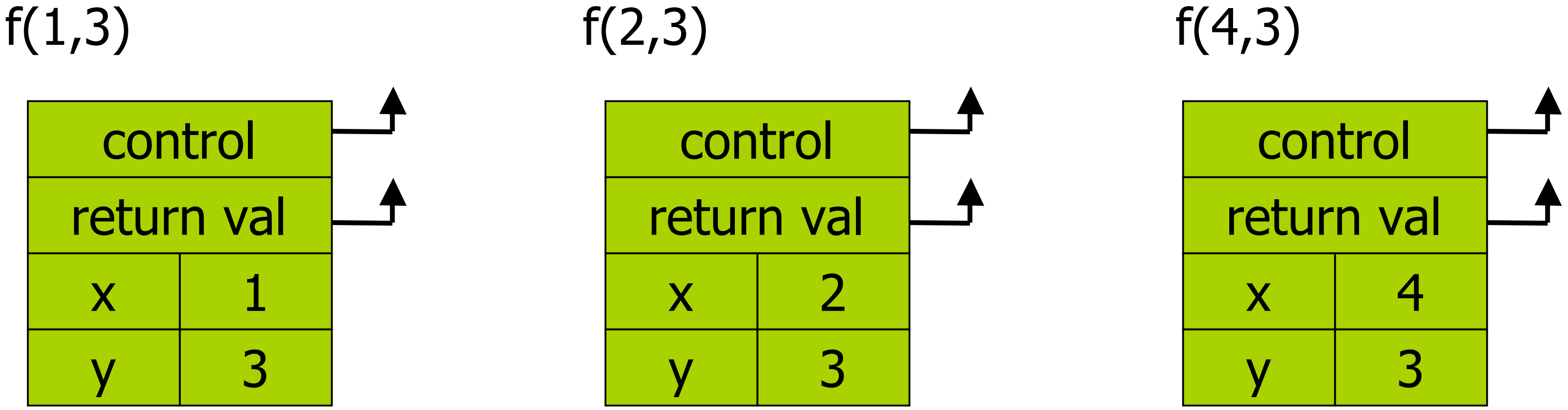
```
fun f(x,y) = if x>y
      then x
      else f(2*x, y);
f(1,3) + 7;
```

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)

control	
return val	
x	1
y	3

f(2,3)

control	
return val	
x	2
y	3

f(4,3)

control	
return val	
x	4
y	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

- 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

- 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
  - Env pointer of closure used when function is called
  - Runtime stack management may fail if functions are returned as result
  - Closures is *not* needed if functions are not in nested blocks
    - Example: C