CS 242

2012

Scope, Function Calls and Storage Management

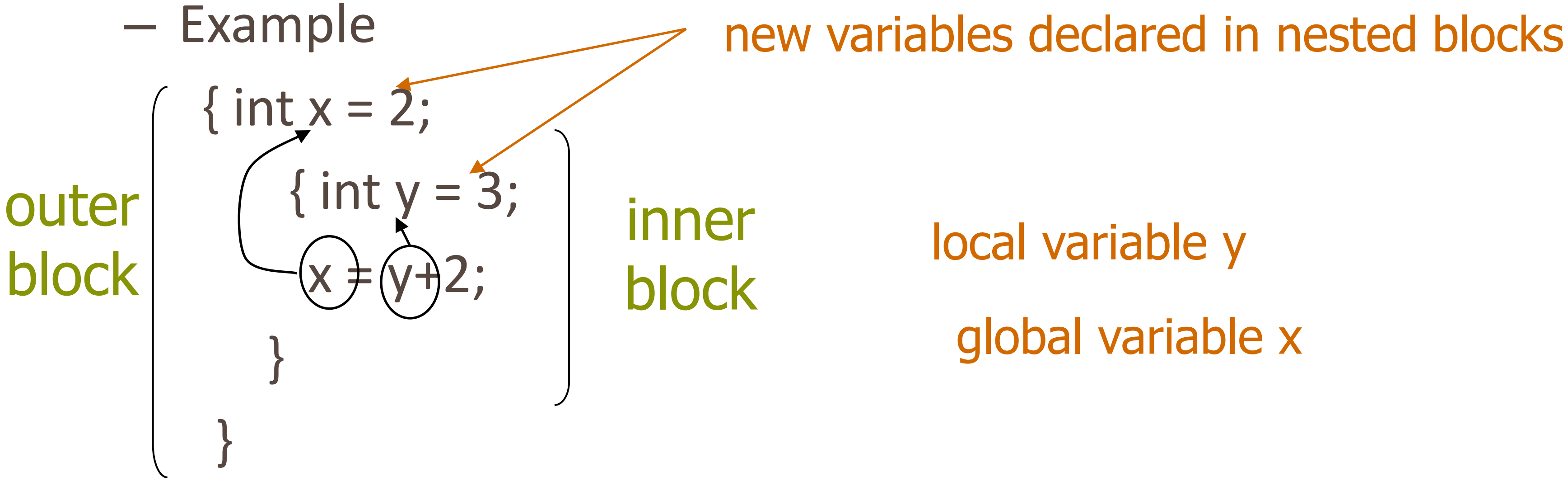
Reading: Chapter 7, Concepts in Programming Languages

Topics

- Block-structured languages and stack storage
- In-line Blocks
  - activation records
  - storage for local, global variables
- First-order functions
  - parameter passing
  - tail recursion and iteration
- Higher-order functions
  - deviations from stack discipline
  - language expressiveness => implementation complexity

Block-Structured Languages

- Nested blocks, local variables
  - Example



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

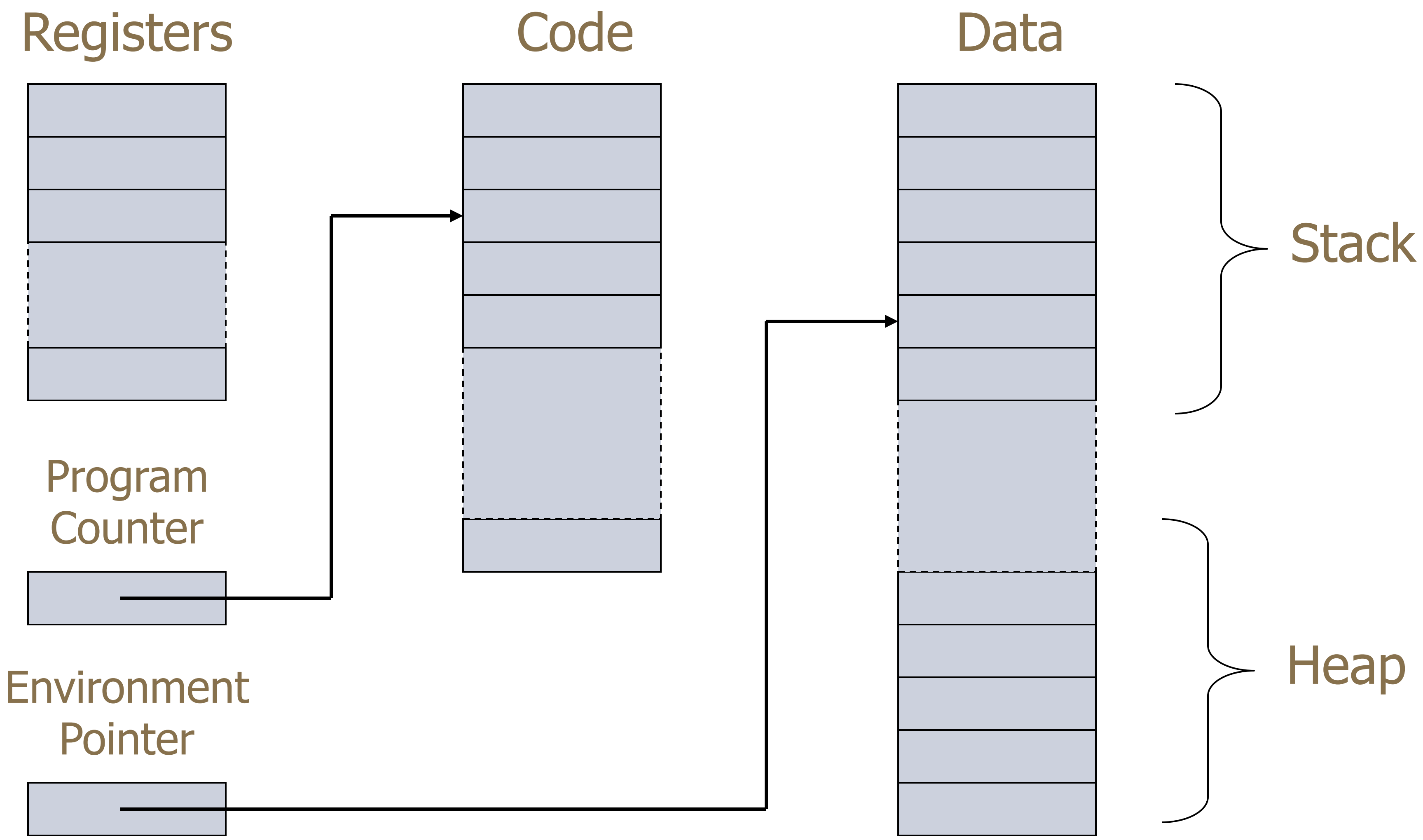
  - Storage management
    - Enter block: allocate space for variables
    - Exits block: some or all space may be deallocated

Examples

- Blocks in common languages
  - C, ~~JavaScript~~ * { … }
  - Algol begin … end
  - ML, Haskell 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, parameters, global variables

* JavaScript functions provide blocks

Simplified Machine Model



Interested in Memory Mgmt Only

- Registers, Code segment, Program counter
  - Ignore registers
  - Details of instruction set will not matter
- Data Segment
  - Stack contains data related to block entry/exit
  - Heap contains data of varying lifetime
  - Environment pointer points to current stack position
    - Block entry: add new activation record to stack
    - Block exit: remove most recent activation record

Some basic concepts

- Scope
  - Region of program text where declaration is visible
- Lifetime
  - Period of time when location is allocated to program

```
{ int x = … ;
    {  int y = … ;
        {  int x = … ;
            ….
         };
    };
};
```

Inner declaration of x hides outer one.

Called "hole in scope"

Lifetime of outer x includes time when inner block is executed

Lifetime $\neq$ scope

Lines indicate "contour model" of scope.

In-line Blocks

- Activation record
  - Data structure stored on run-time stack
  - Contains space for local variables
- Example

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

Push record with space for x, y

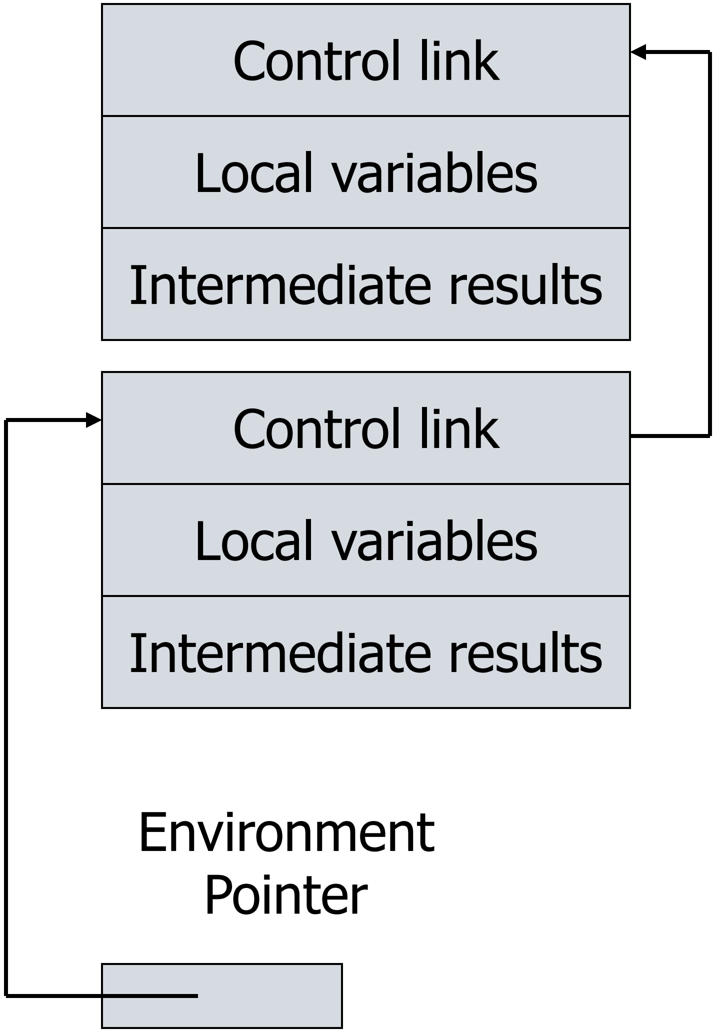
Set values of x, y

> 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)

Activation record for in-line block


Control link

Local variables

Intermediate results

Control link

Local variables

Intermediate results

Environment Pointer

- 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

Can be optimized away, but assume not for purpose of discussion.

Example

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

Push record with space for x, y

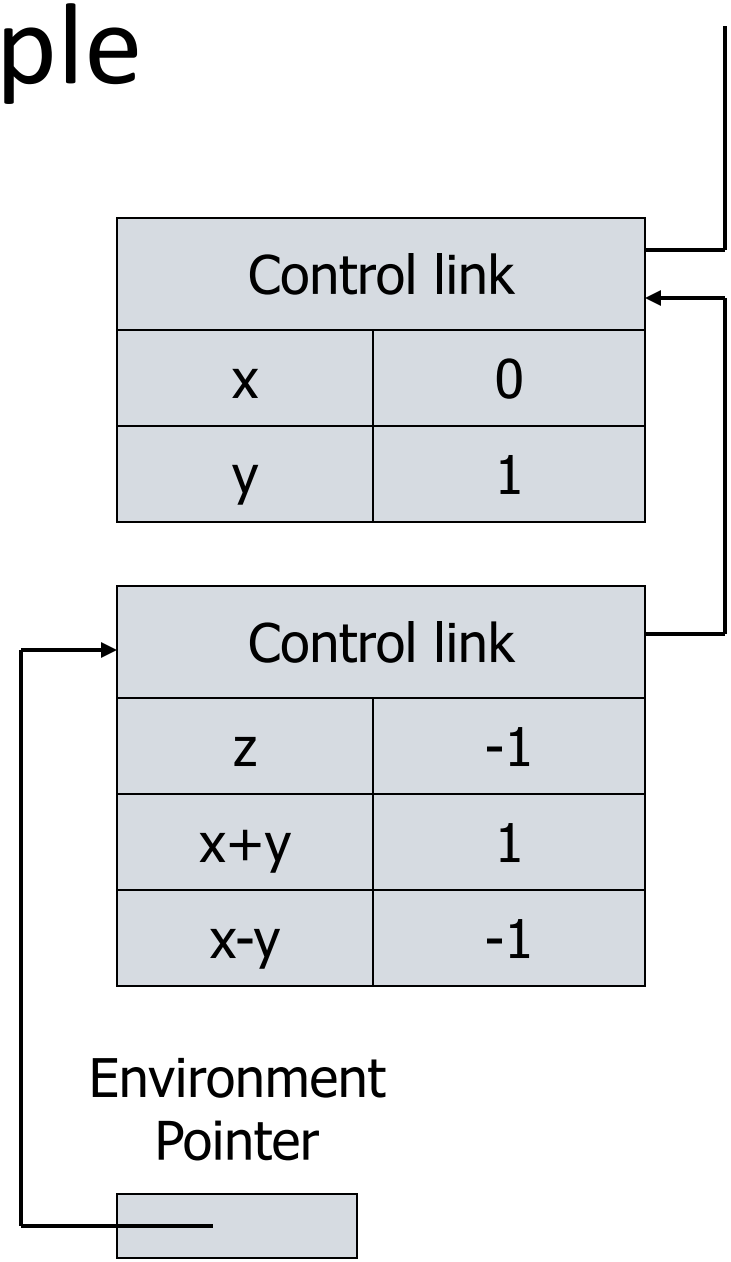
Set values of x, y

    Push record for inner block

    Set value of z

    Pop record for inner block

Pop record for outer block



Control link	
x	0
y	1

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

Environment Pointer

Scoping rules

- Global and local variables

  x, y are local to outer block

  z is local to inner bock

  x, y are global to inner block

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

- Static scope

  global refers to declaration in closest enclosing block

- Dynamic scope

  global refers to most recent activation record

These are same until we consider function calls.

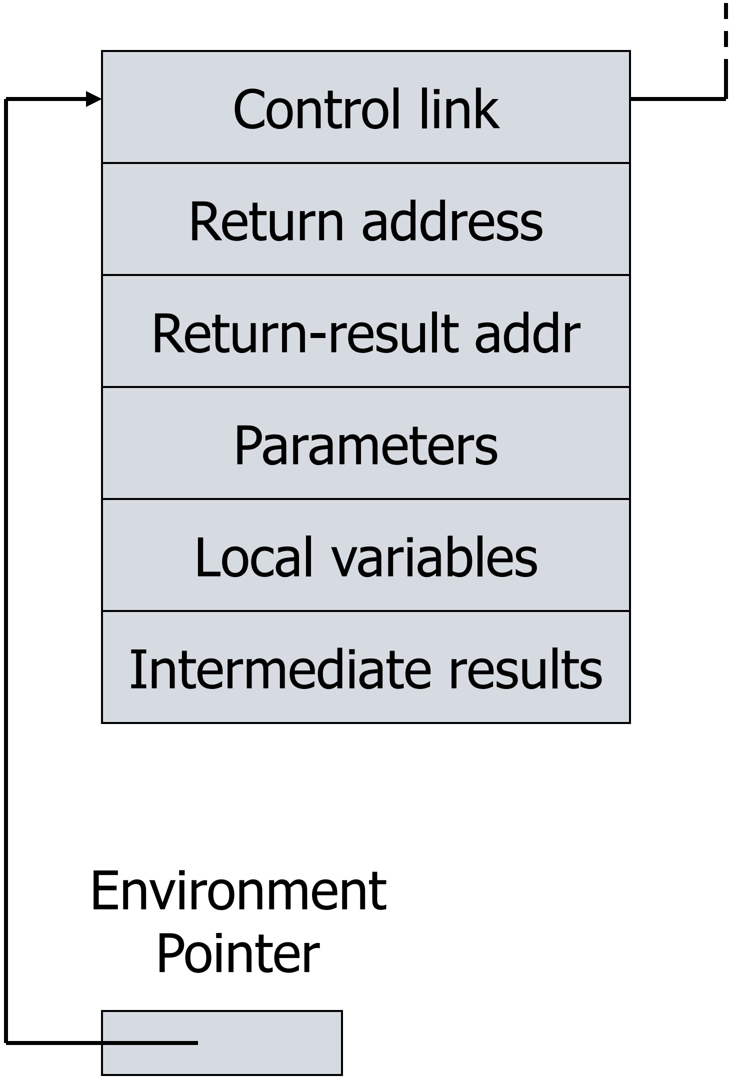
Functions and procedures

- Syntax of procedures (Algol) and functions (C)

```
procedure P (<pars>)          <type> function f(<pars>)
   begin                      {
       <local vars>                <local vars>
       <proc body>                 <function body>
   end;                       }
```

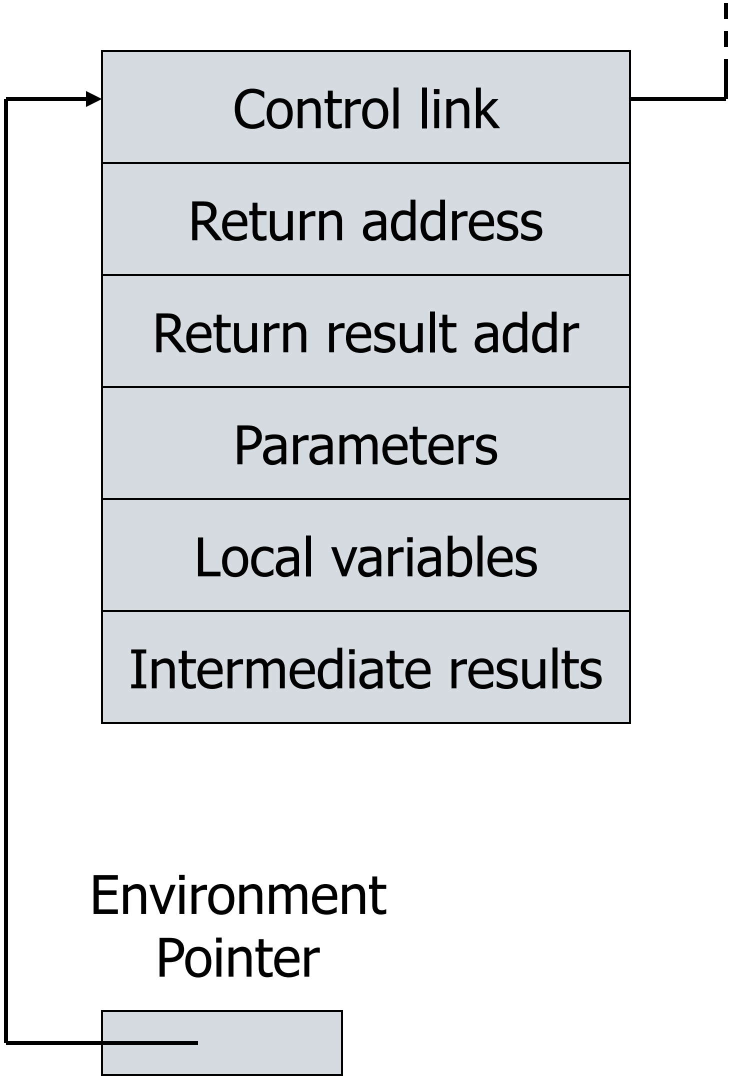
- Activation record must include space for
  - parameters
  - return address
  - local variables, intermediate results
  - return value (an intermediate result)
  - location to put return value on function exit

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

Example



- Function

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

  – Return result address

  – location to put fact(n)

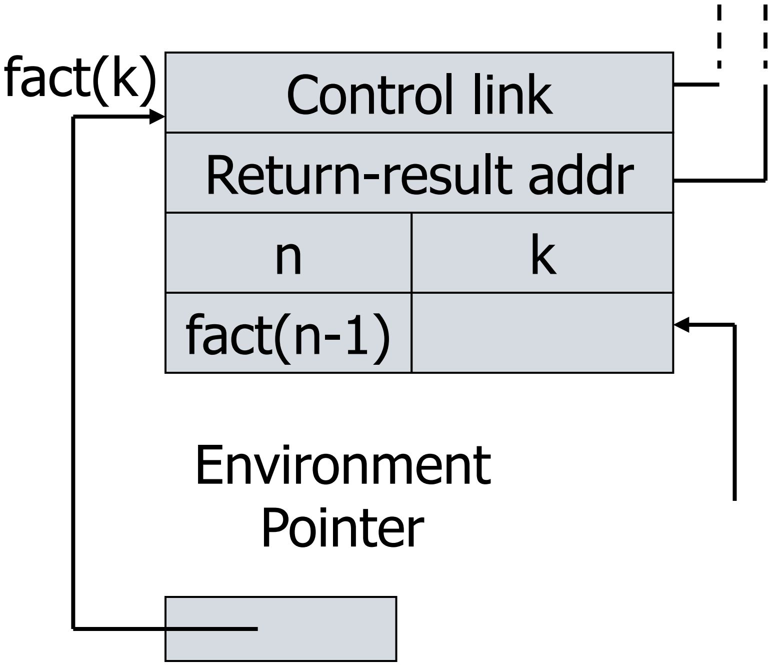
- Parameter

  – set to value of n by calling sequence

- Intermediate result

  – locations to contain value of fact(n-1)

Function call



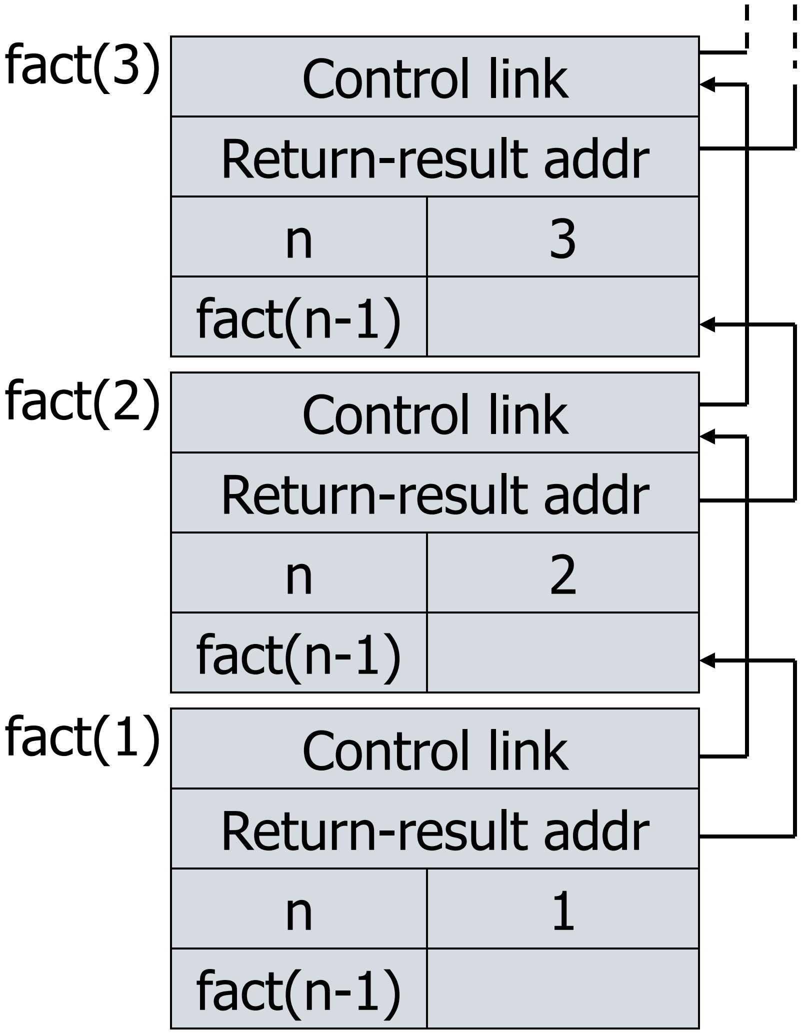
fact(k)

Control link

Return-result addr

n	k
fact(n-1)	

Environment Pointer



fact(3)

Control link

Return-result addr

n	3
fact(n-1)	

fact(2)

Control link

Return-result addr

n	2
fact(n-1)	

fact(1)

Control link

Return-result addr

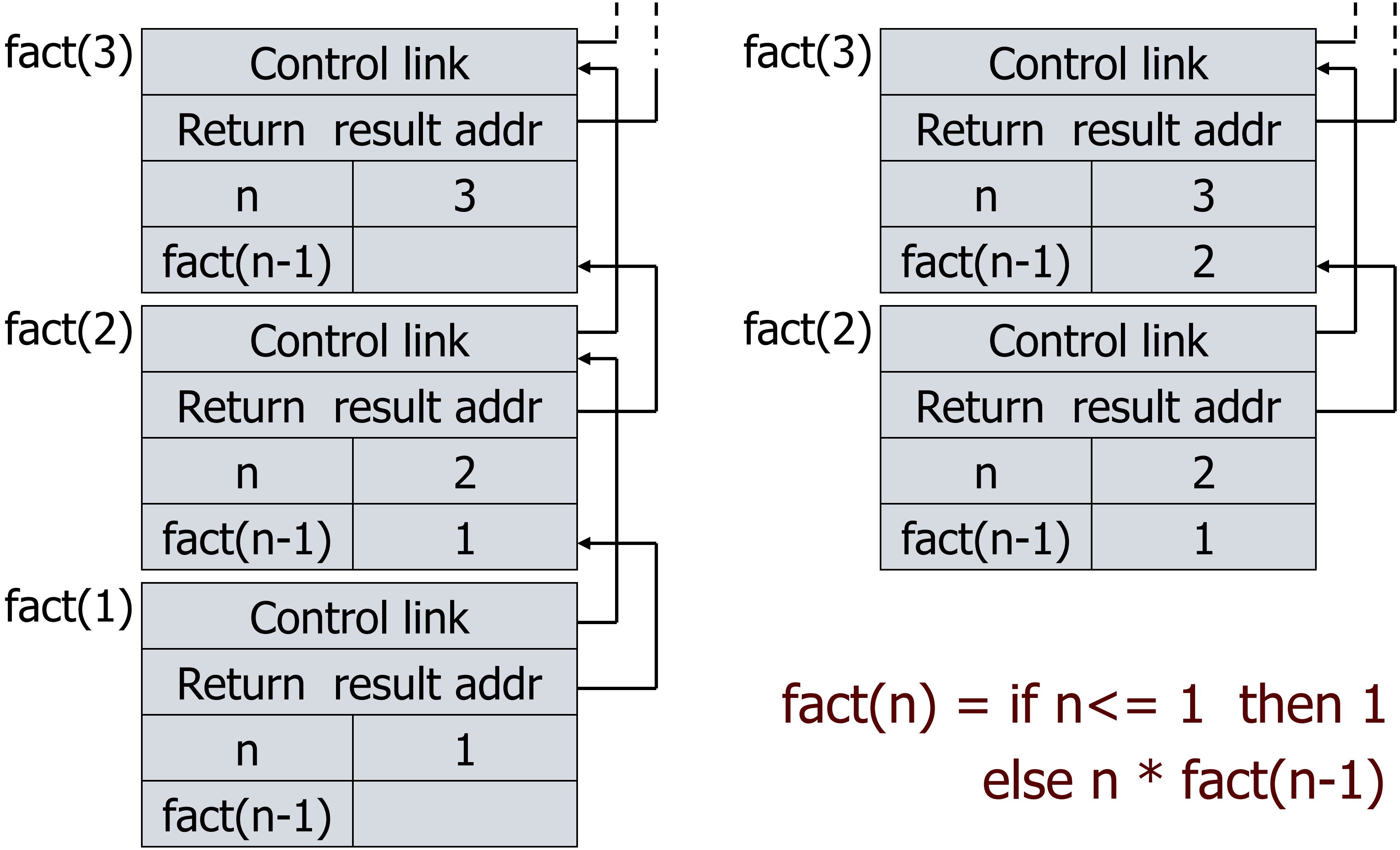
n	1
fact(n-1)	

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

Return address omitted; would be ptr into code segment

Function return next slide →

Function return



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

Topics for first-order functions

- Parameter passing
  - pass-by-value: copy value to new activation record
  - pass-by-reference: copy ptr 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

See this yourself: write factorial and run under debugger

Parameter passing

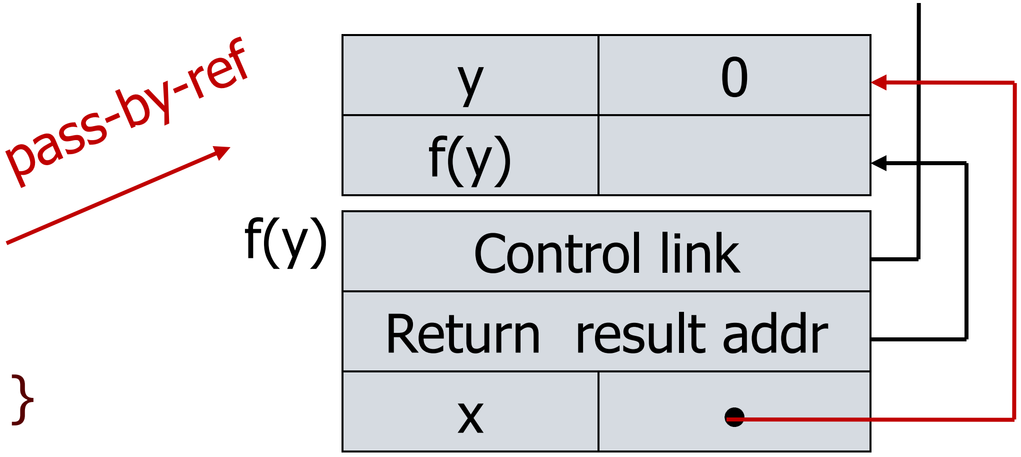
- General terminology: L-values and R-values
  - Assignment `y := x+3`
    - Identifier on left refers to location, called its L-value
    - Identifier on right refers to contents, called R-value
- Pass-by-reference
  - Place L-value (address) in activation record
  - Function can assign to variable that is passed
- Pass-by-value
  - Place R-value (contents) in activation record
  - Function cannot change value of caller's variable
  - Reduces aliasing (alias: two names refer to same loc)

Example

pseudo-code

activation records

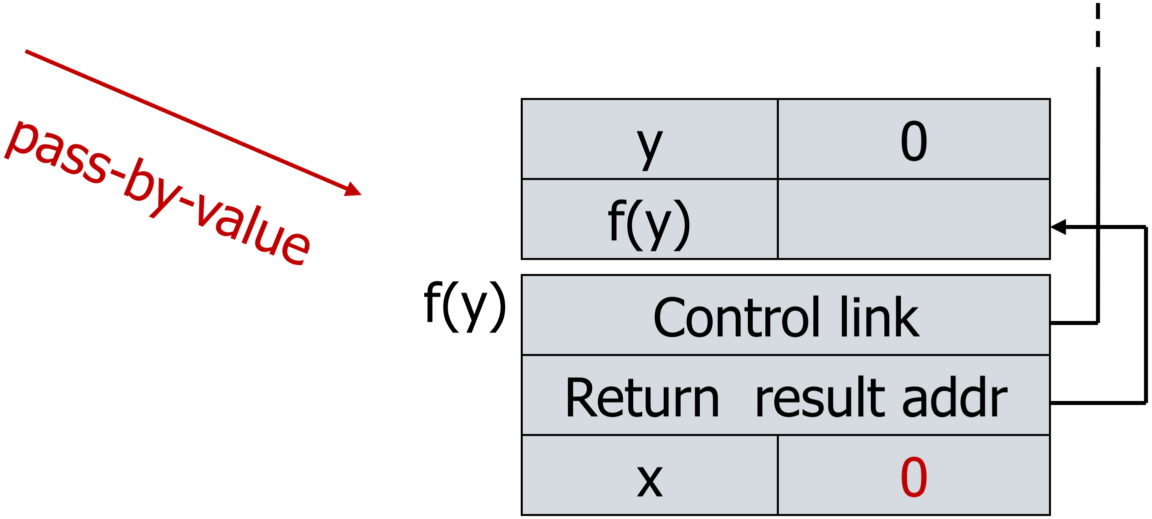
```
function f (x) =
    { x = x+1; return x;  }
var y = 0;
print (f(y)+y);
```



pass-by-ref

f(y)

y	0
f(y)	
Control link	
Return result addr	
x	●



pass-by-value

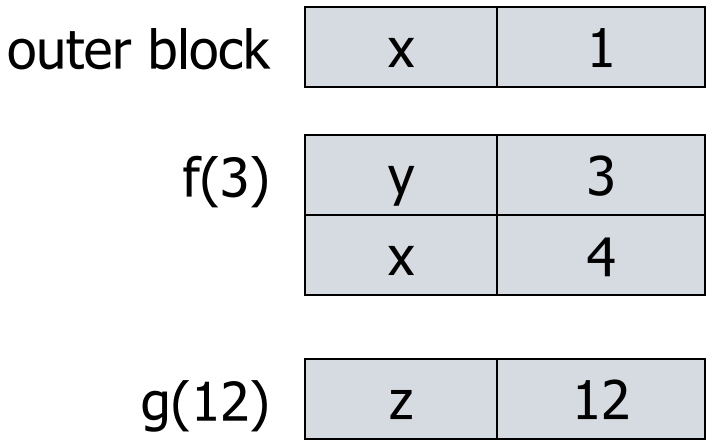
f(y)

y	0
f(y)	
Control link	
Return result addr	
x	0

Access to global variables

- Two possible scoping conventions
  - Static scope: refer to closest enclosing block
  - Dynamic scope: most recent activation record on stack
- Example

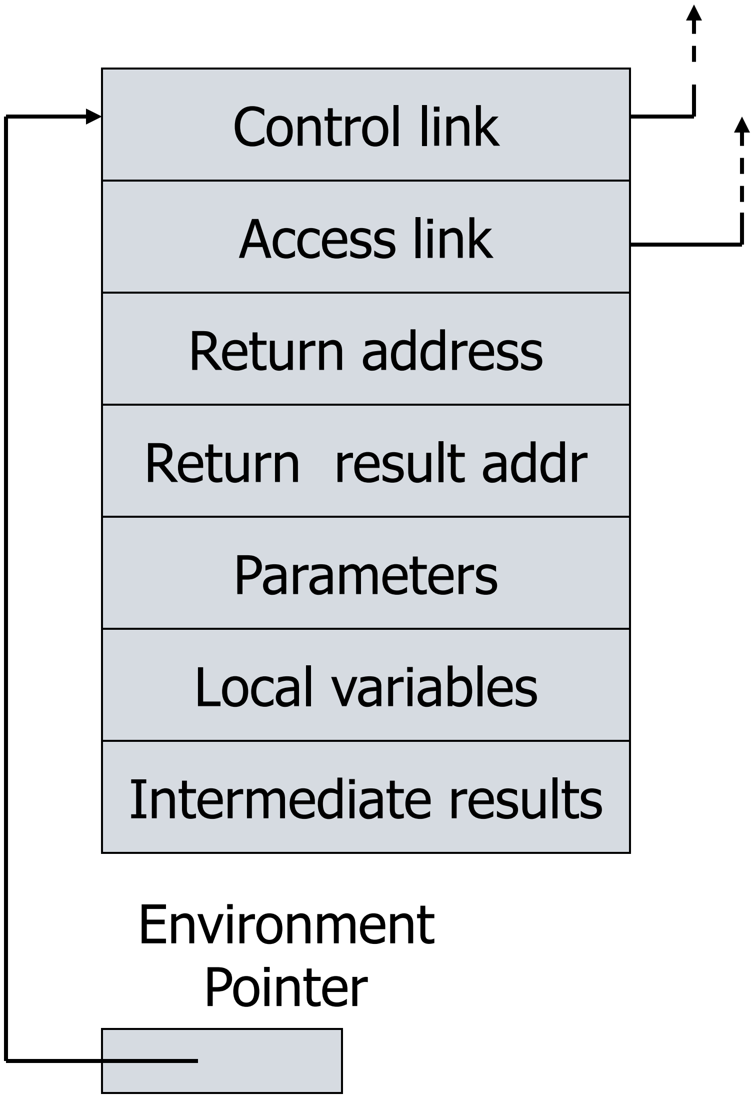
```
var x=1;
function g(z) { return x+z; }
function f(y) {
    var x = y+1;
    return g(y*x);
}
f(3);
```



outer block	x	1
f(3)	y	3
	x	4
g(12)	z	12

Which x is used for expression x+z ?

Activation record for static scope



Control link

Access link

Return address

Return result addr

Parameters

Local variables

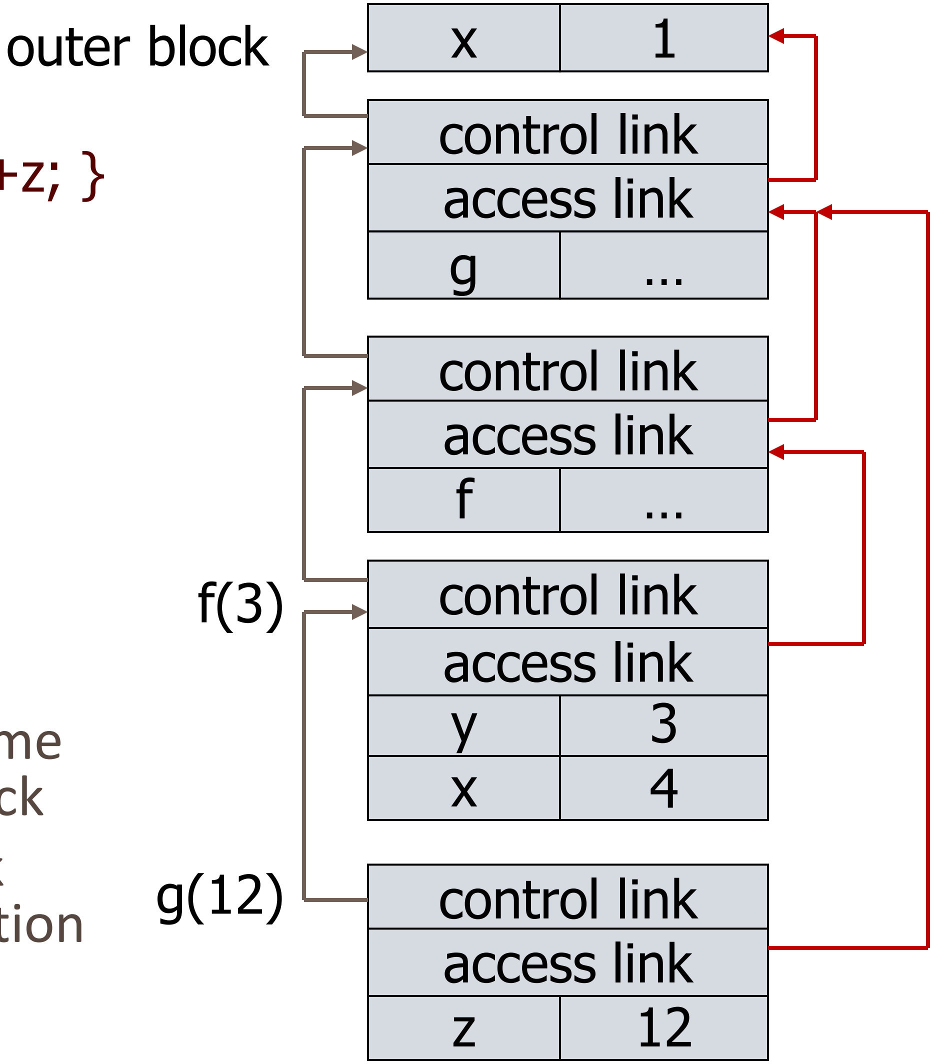
Intermediate results

Environment Pointer

- 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 prog
  - – Access link depends on static form of program text

Static scope with access links

```
var x=1;
  function g(z) = { return x+z; }
    function f(y) =
      { var x = y+1;
        return g(y*x); }
    f(3);
```

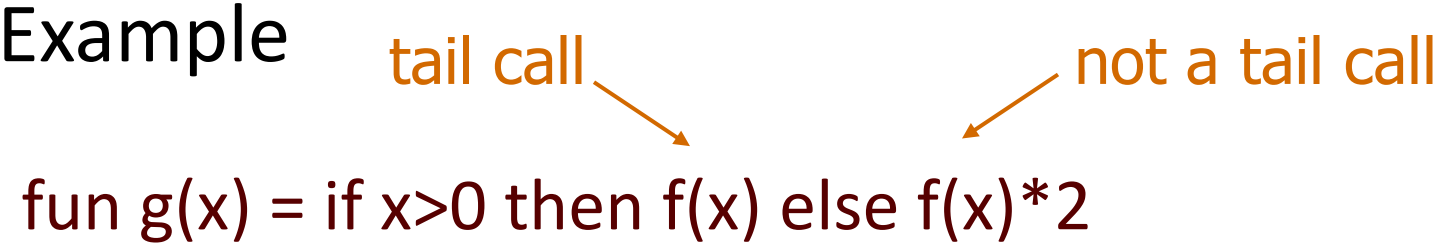


Use access link to find global variable:

- Access link is always set to frame of closest enclosing lexical block
- For function body, this is block that contains function declaration

Tail recursion (first-order case)

- Function g makes a *tail call* to function f if
  - Return value of function f is return value of g
- Example



tail call

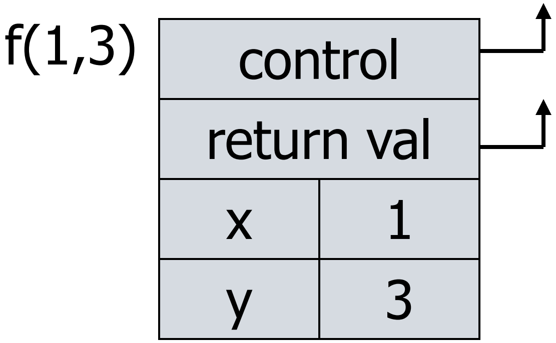
not a tail call

```
fun g(x) = if x>0 then f(x) else f(x)*2
```

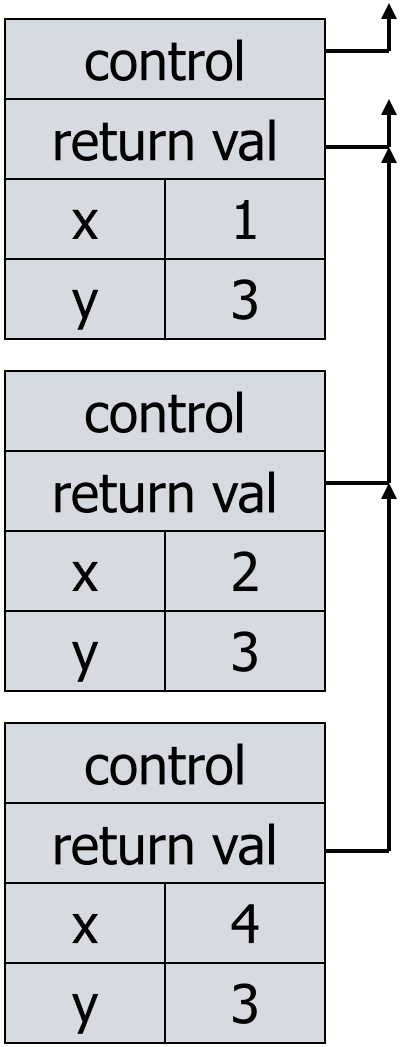
- Optimization
  - Can pop activation record on a tail call
  - Especially useful for recursive tail call
    - next activation record has exactly same form

Example

Calculate least power of 2 greater than y



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



Optimization

- Set return value address to that of caller

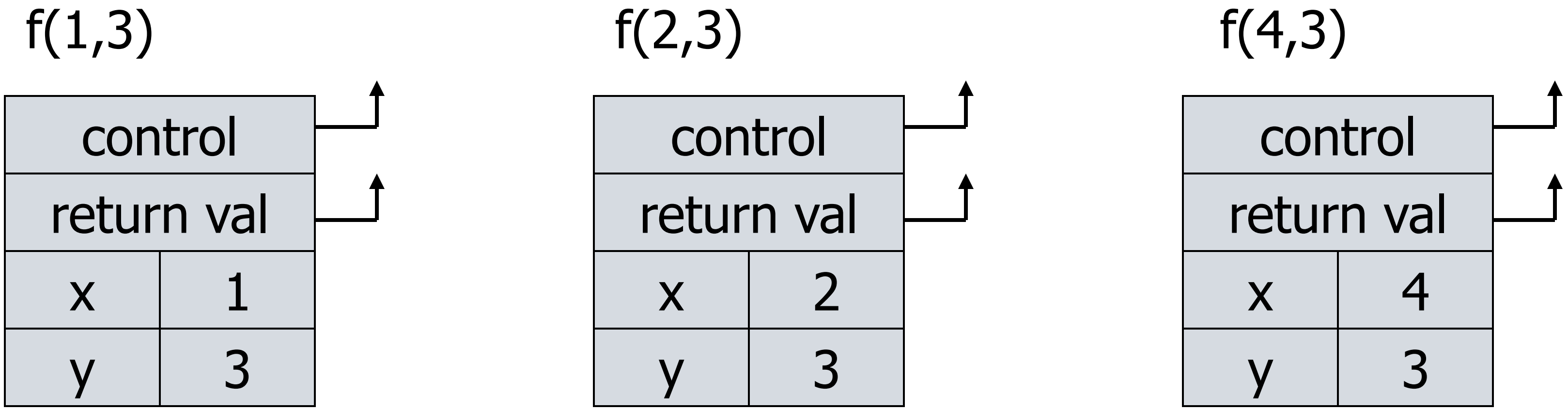
Question

- Can we do the same with control link?

Optimization

- avoid return to caller

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);
```

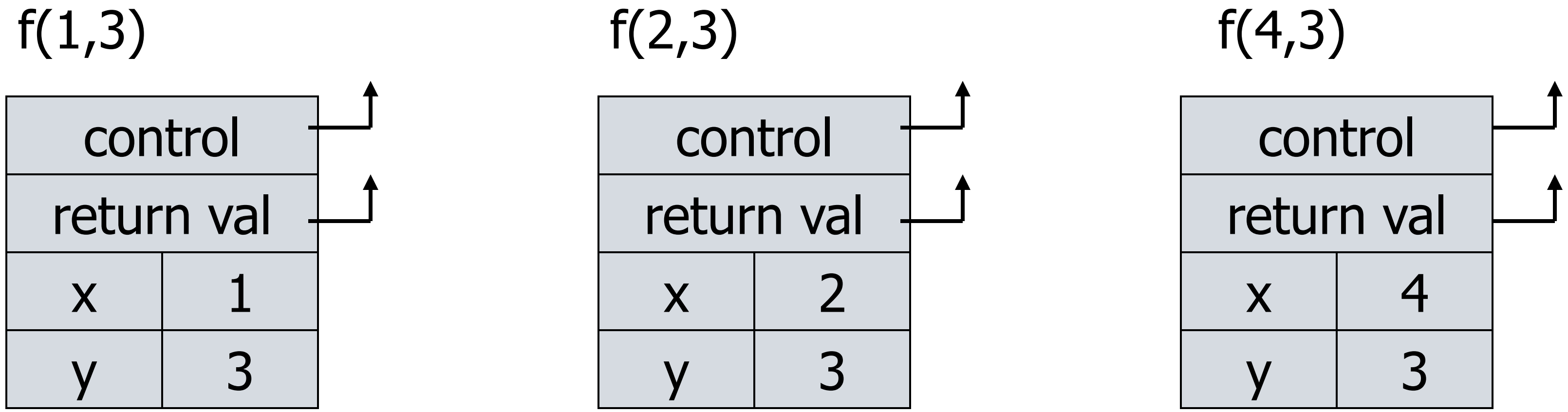
Optimization

- pop followed by push = reuse activation record in place

Conclusion

- Tail recursive function equiv to iterative loop

Tail recursion and iteration



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,y);
```

test

loop body

initial value

```
function g(y) {
   var x = 1;
   while (!x>y)
      x = 2*x;
   return x;
}
```

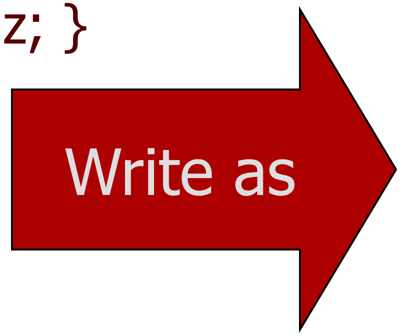
Not essential to understand the ML code here.

Higher-Order Functions

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

Complex nesting structure

```
function m(…) {
   var x=1;
   …
   function n( … ){
     function g(z) { return x+z; }
     …
     { …
          function f(y) {
             var x = y+1;
             return g(y*x); }
        …
        f(3);  … }
     … n( … ) …}
  … m(…)
```



Write as

```
var x=1;
   function g(z) { return x+z; }
        function f(y)
          { var x = y+1;
             return g(y*x); }
        f(3);
```

Simplified code has same block nesting, if we follow convention that each declaration begins a new block.

RECAP

JavaScript blocks and scopes

- { } groups JavaScript statements
  - Does not provide a separate scope
- Blocks w/scope can be expressed using *function*
  - (function(){ … })() - create function of no args and call
  - Example

```
var y=0;
(function () {     // begin block
    var x=2;     // local variable x
    y = y+x;
 }) ();            // end block
```

Translating examples to JS

```
var x = 5;
  function f(y) {return (x+y)-2};
    function g(h){var x = 7; return h(x)};
      {var x = 10; g(f)};
```

```
(function (){
    var x = 5;
    (function (){
        function f(y) {return (x+y)-2};
        (function (){
            function g(h){var x = 7; return h(x)};
            (function (){
                var x = 10; g(f);
            })()
        })()
    })()
})()
```

Example and HW convention:

Each new declaration begins a new scope

Pass function as argument

Haskell

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

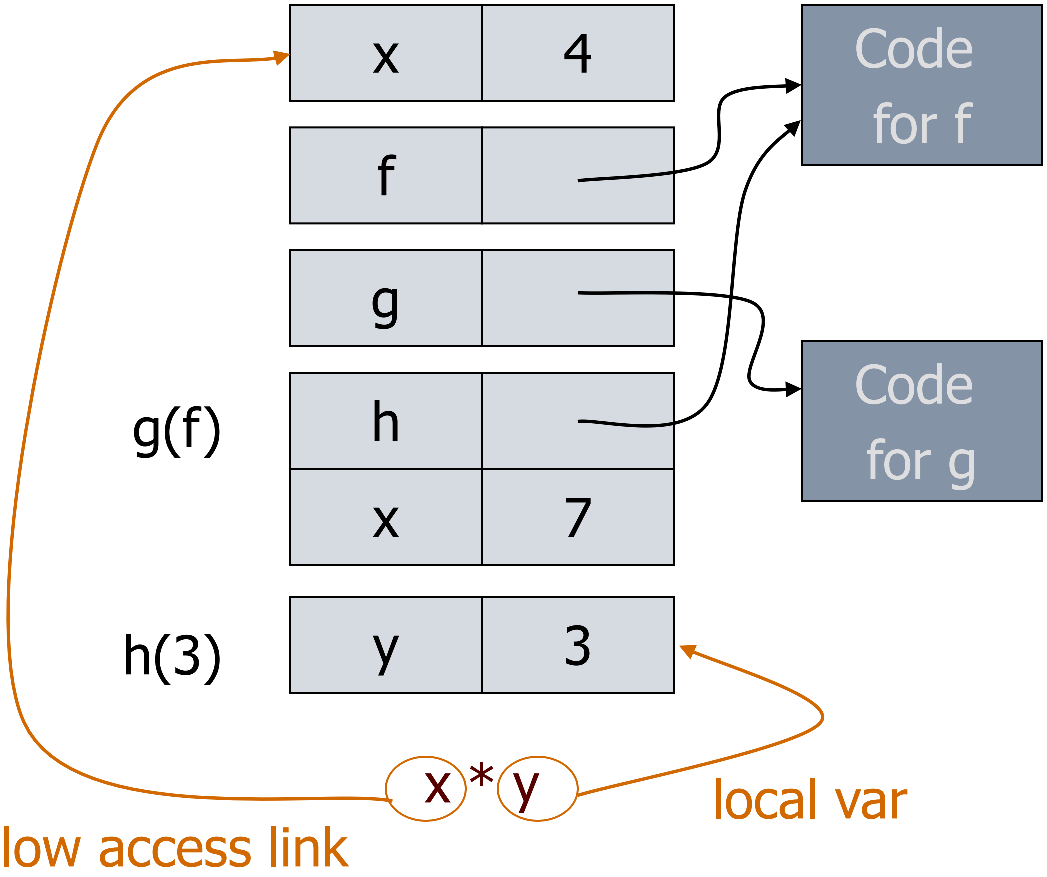
Pseudo-JavaScript

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

There are two declarations of x
Which one is used for each occurrence of x?

Static Scope for Function Argument

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



x	4
f	
g	

g(f)

h	
x	7

h(3)

y	3

Code for f

Code for g

x*y

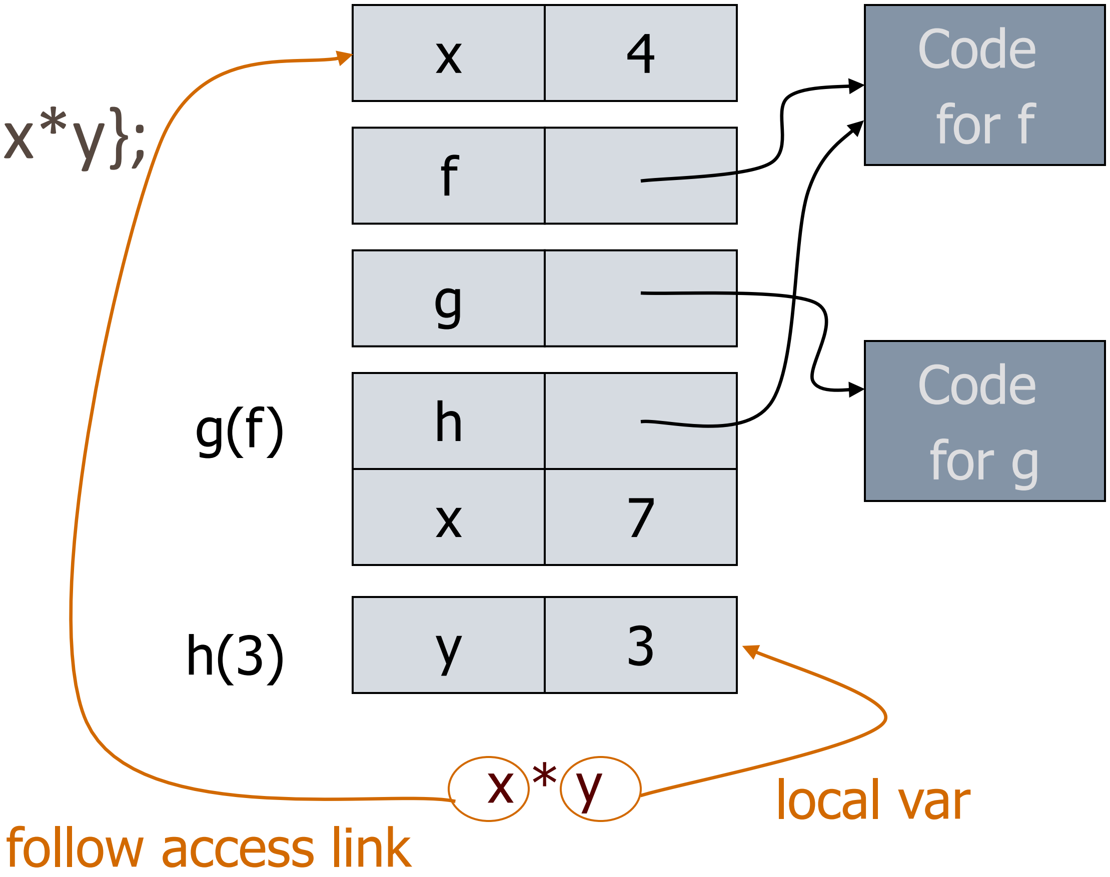
follow access link

local var

- How is access link for h(3) set?

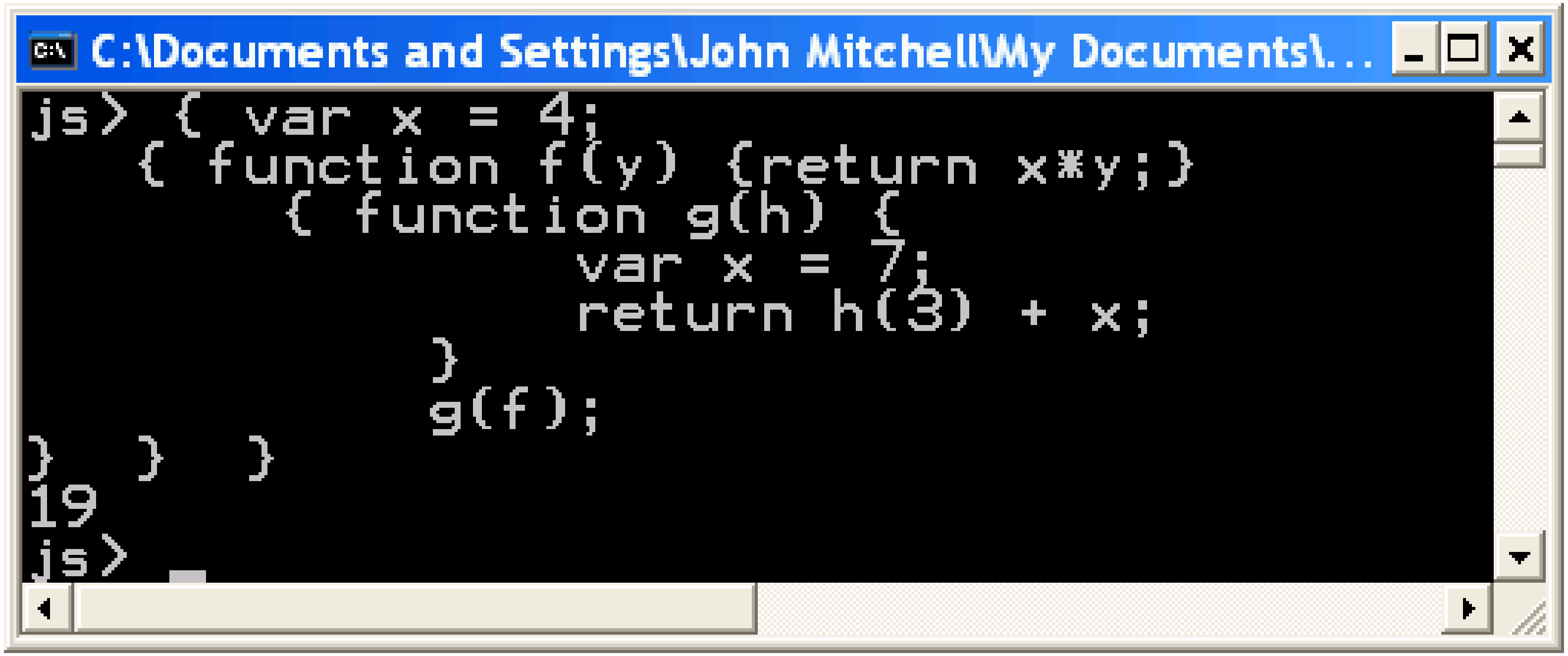
Static Scope for Function Argument

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



- How is access link for h(3) set?

Result of function call



```
js> { var x = 4;
   { function f(y) {return x*y;}
       { function g(h) {
               var x = 7;
               return h(3) + x;
           }
           g(f);
}  }  }
19
js>
```

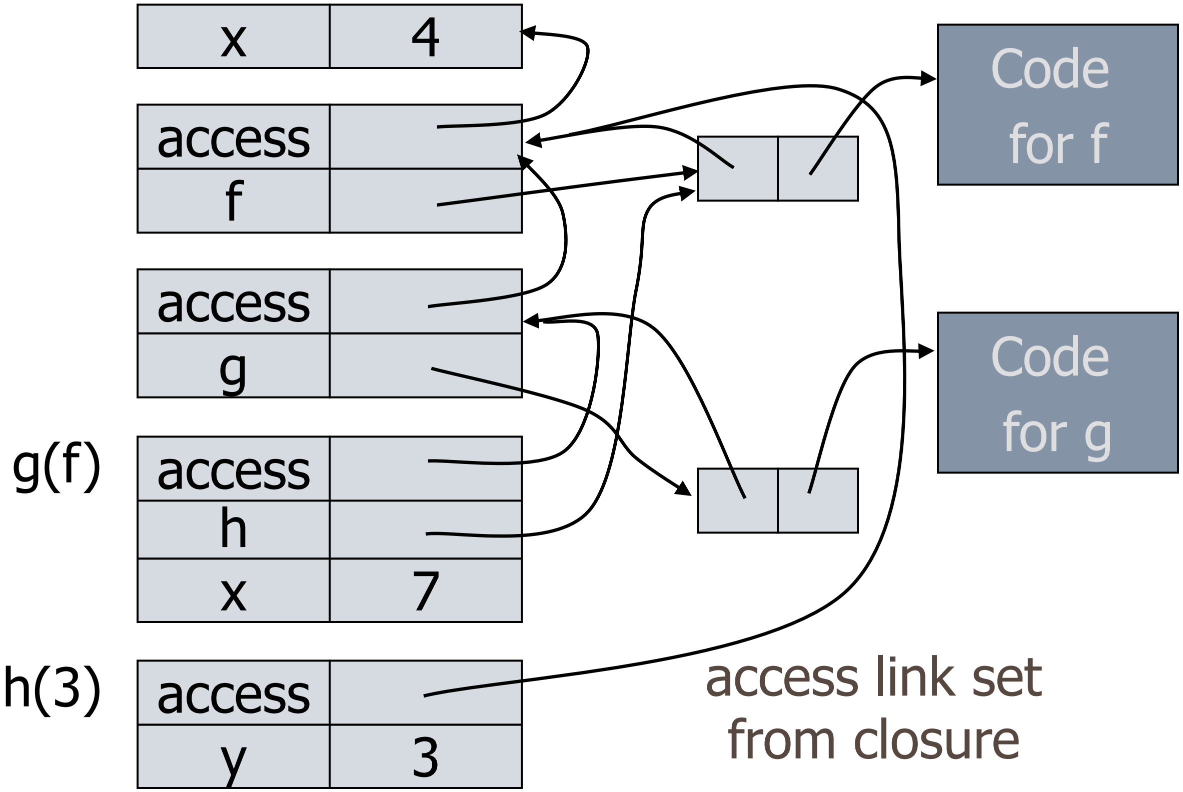
Closures

- 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

Run-time stack with access links

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

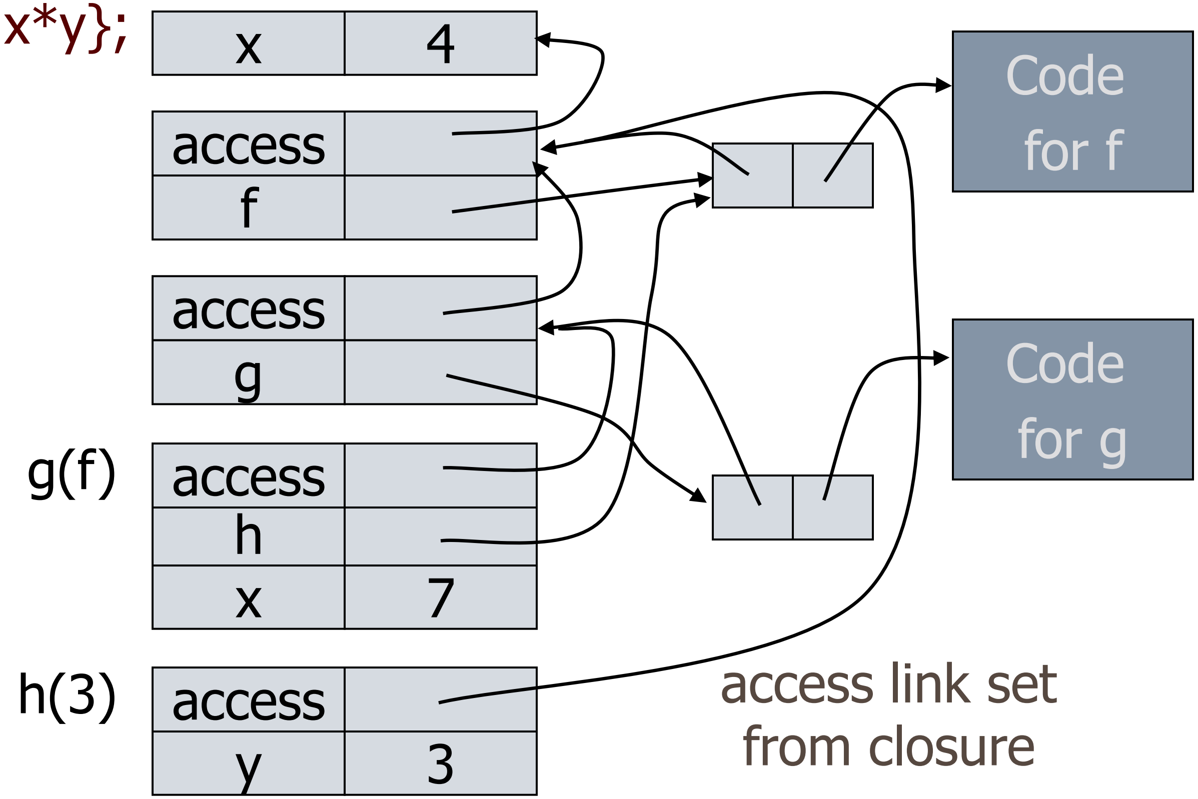


access link set from closure

Function Argument and Closures

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

Run-time stack with access links



access link set from closure

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 activ records to find global vars
  - Still deallocate activ 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

```
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);
```

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

Example: Return fctn with private state

JS

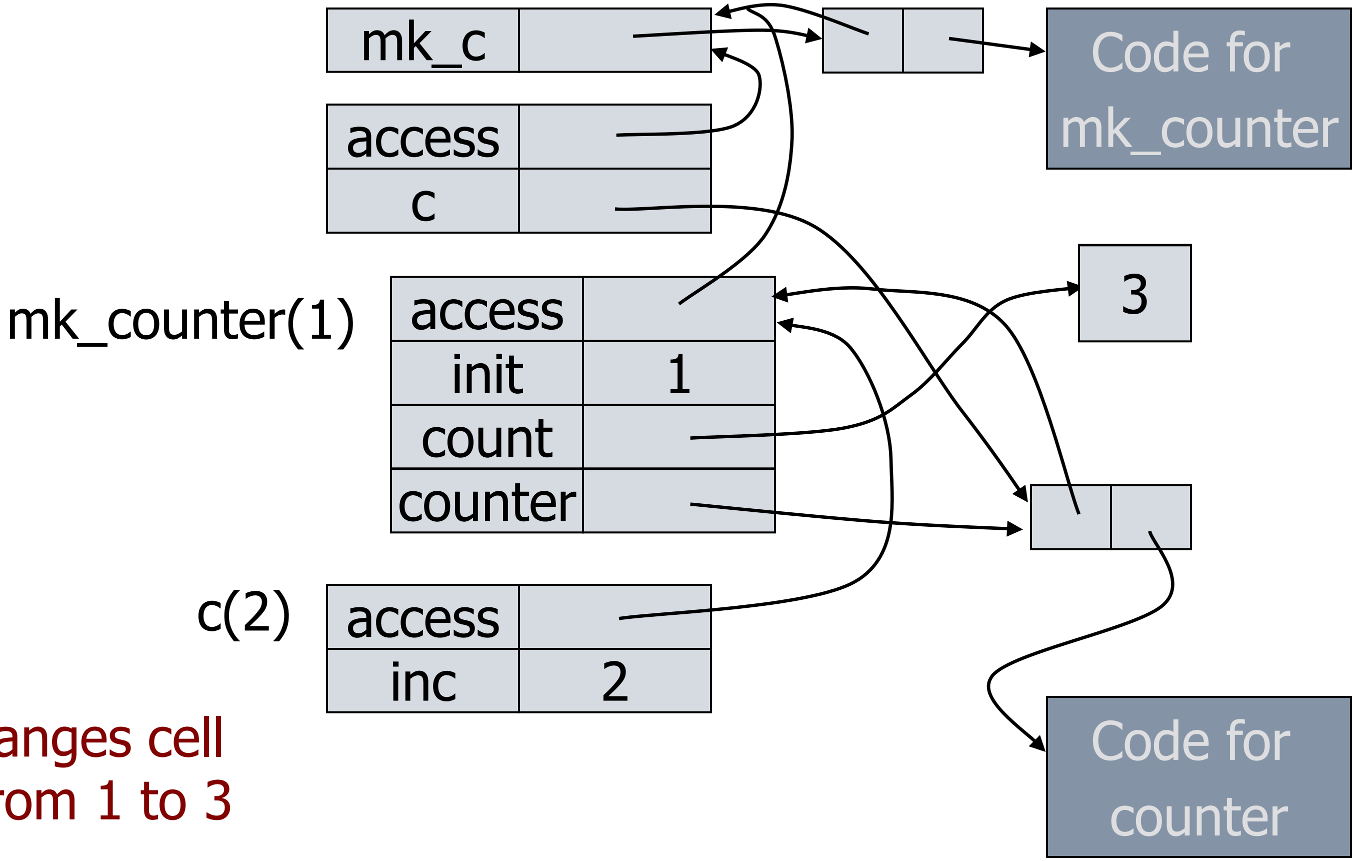
```
function mk_counter (init) {
   var count = init;
   function counter(inc) {count=count+inc; return
  count};
   return counter};
var c  = mk_counter(1);
c(2) + c(2);
```

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

ML

Function Results and Closures

```
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);
```



mk_c

access

c

Code for mk_counter

mk_counter(1)

access	
init	1
count	
counter	

3

c(2)

access	
inc	2

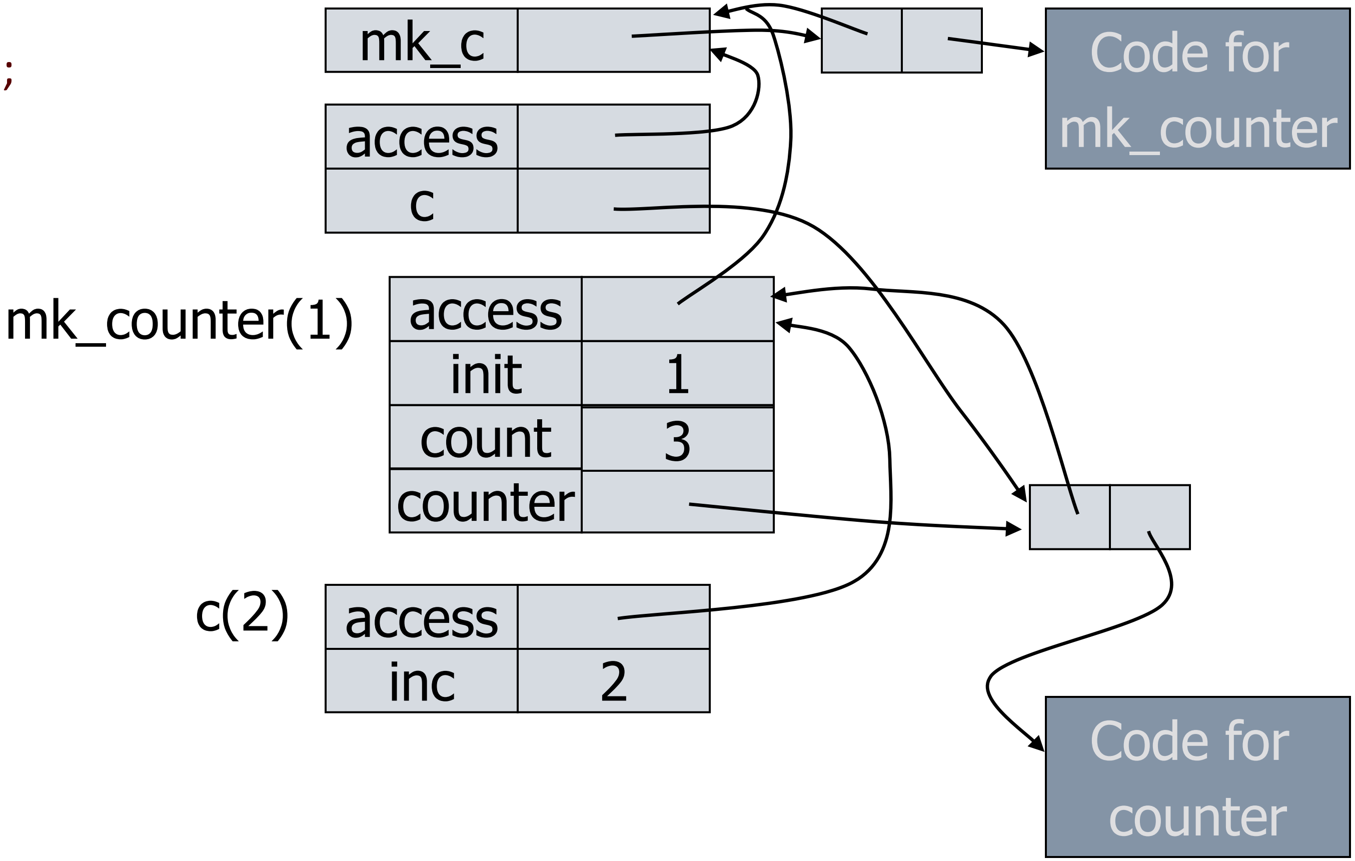
Code for counter

Call changes cell value from 1 to 3

JS

Function Results and Closures

```
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: Return Function Results

- 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

Summary of scope issues

- 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