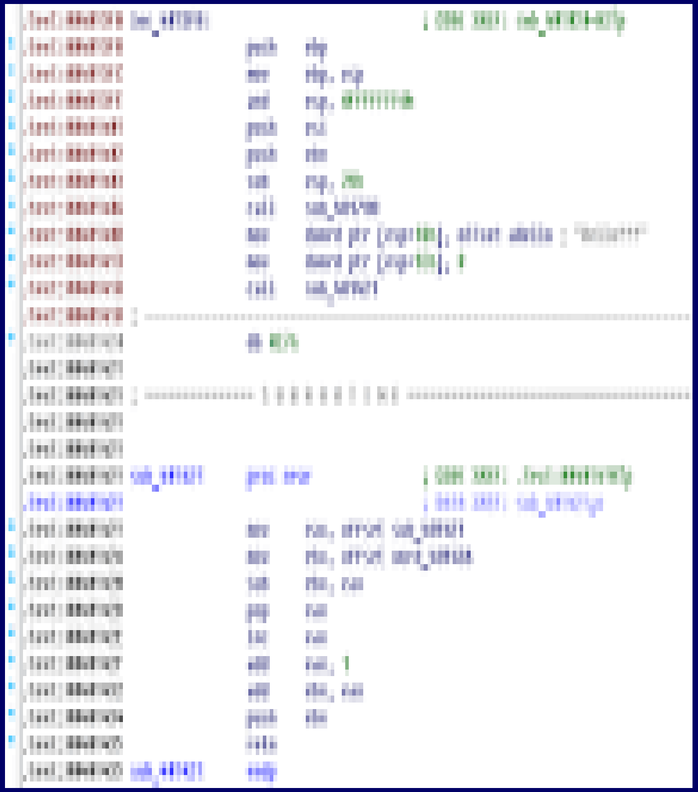
Storage Management for Programming Languages

John Mitchell

Adapted by Mooly Sagiv

High level Programs vs. Executable Programs

Executable Code



Abstractions for productive programming

First order order functions

Higher order functions

Polymorphism

Abstract Data Types

Algebraic Data Types

Modules

Control abstractions

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
- Garbage Collection

Block-Structured Languages

- Nested blocks, local variables
  - Example

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

new variables declared in nested blocks

outer block

inner block

local variable

global variable

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

Examples

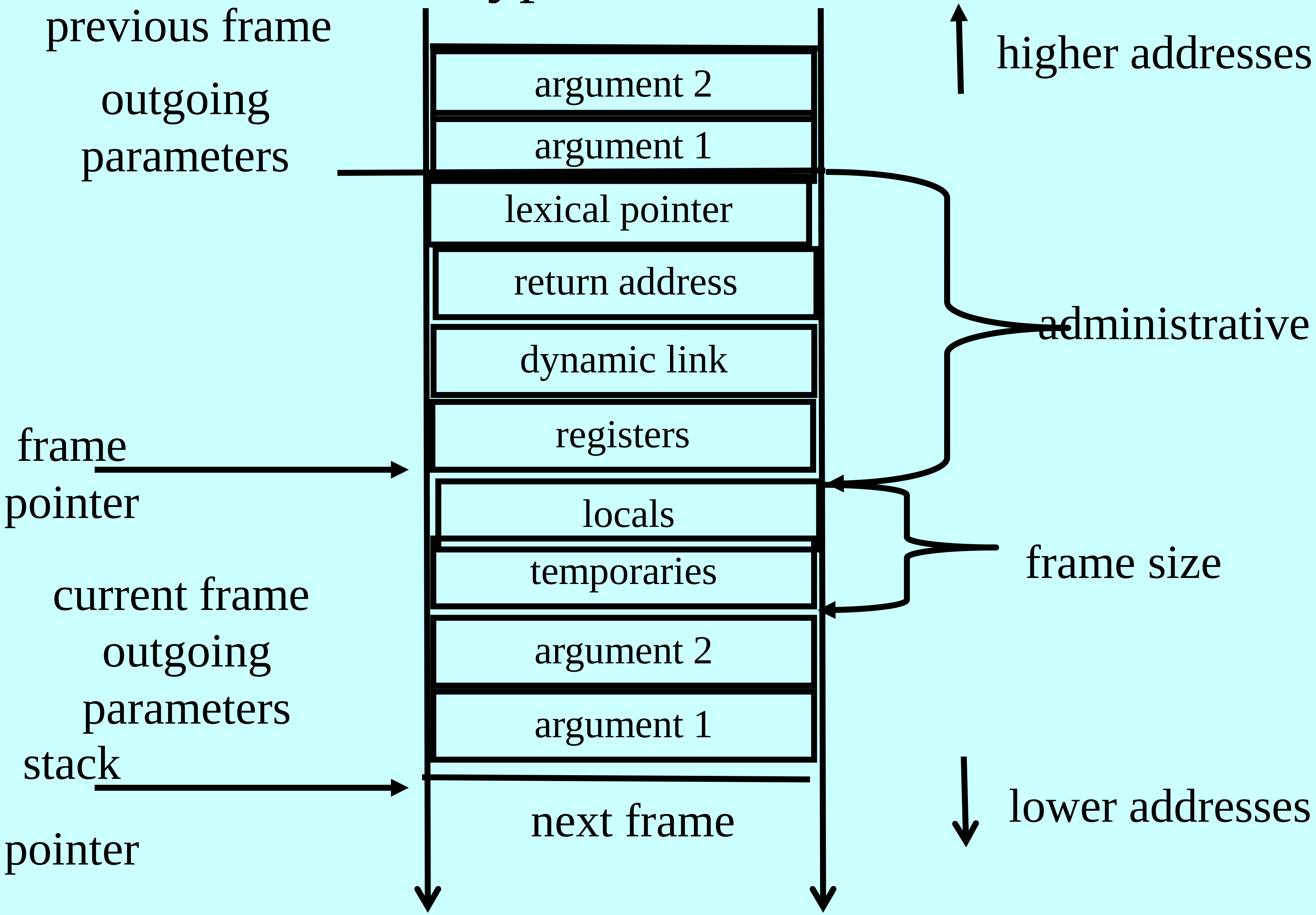
- Blocks in common languages
  - C, ~~JavaScript~~ * { … }
  - Algol begin … end
  - ML let … in …
- Two forms of blocks
  - In-line blocks
  - Blocks associated with functions or procedures
- Topic: block-based memory management, access to local variables, parameters, global variables

* JavaScript functions provide blocks

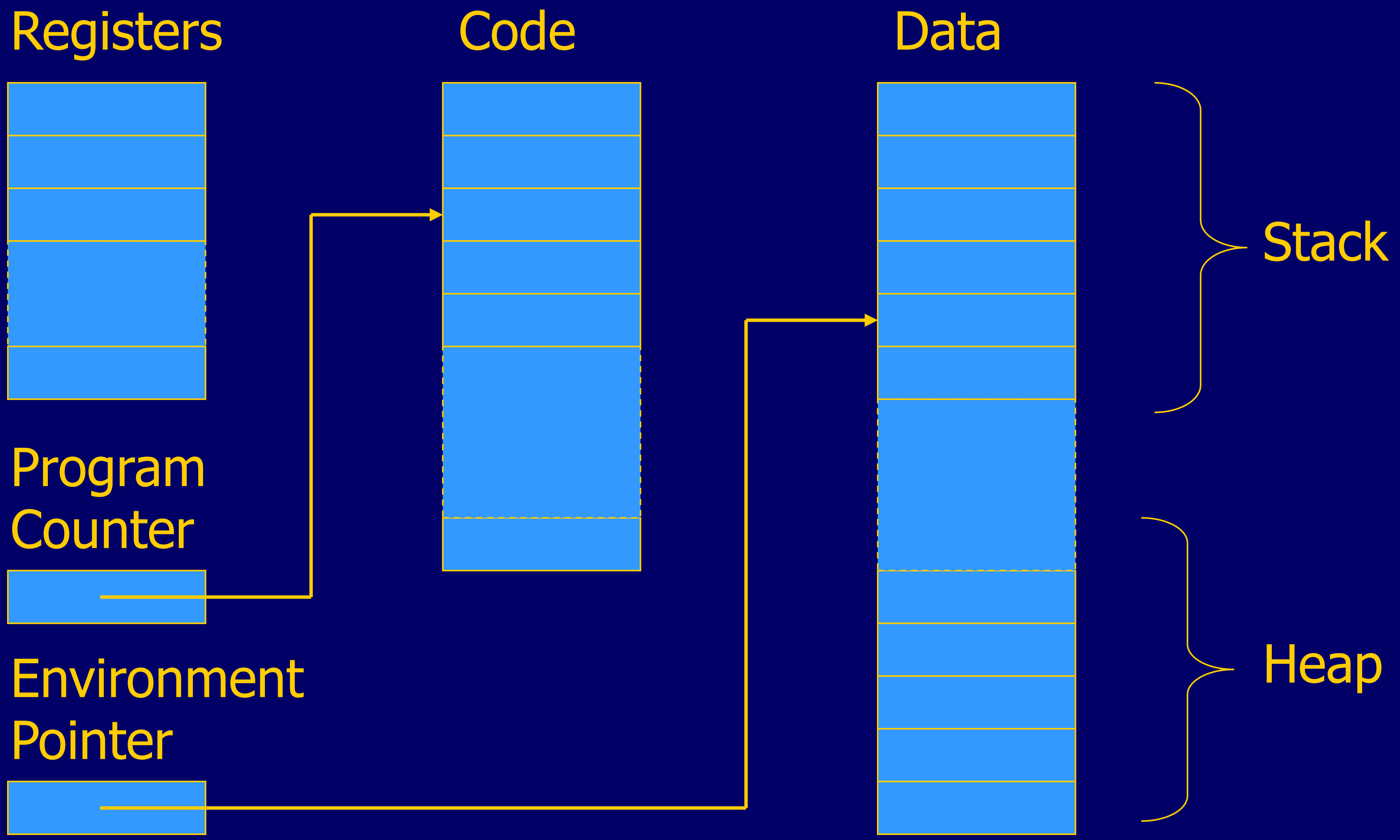
Stack Frames

- Allocate a separate space for every procedure incarnation
- Relative addresses
- Provide a simple mean to achieve modularity
- Supports separate code generation of procedures
- Naturally supports recursion
- Efficient memory allocation policy
  - Low overhead
  - Hardware support may be available
- LIFO policy
- Not a pure stack
  - Non local references
  - Updated using arithmetic

A Typical Stack Frame



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 (Duration)
  - 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

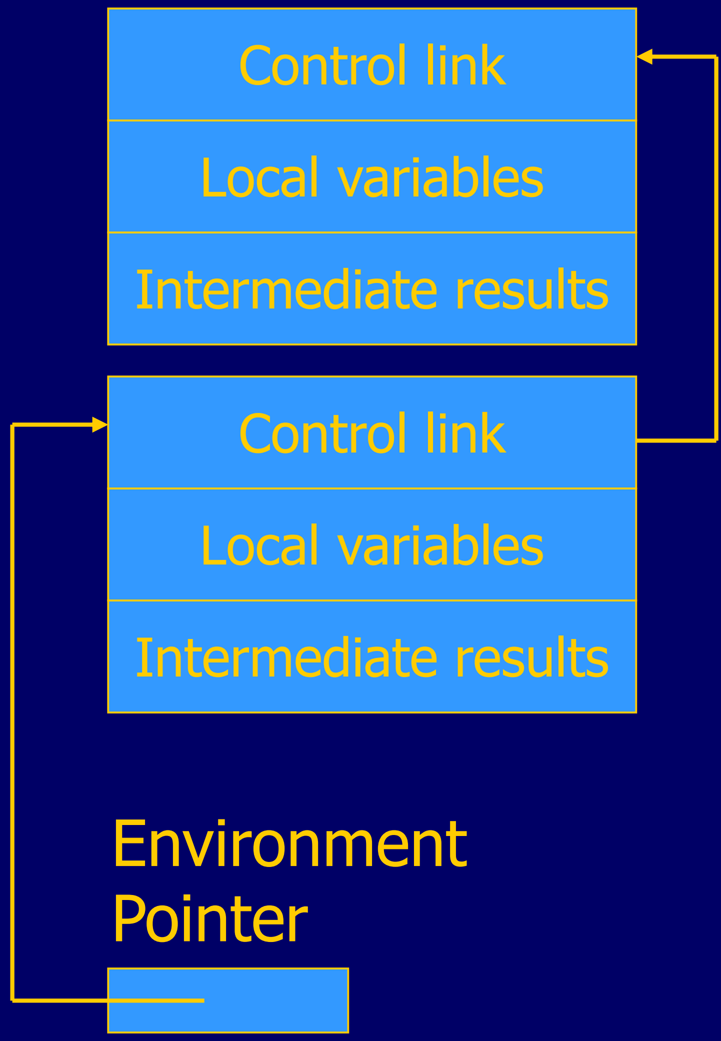
- Activation record
  - Data structure stored on run-time stack
  - Contains space for local variables
- Example

```
{ 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
  - 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

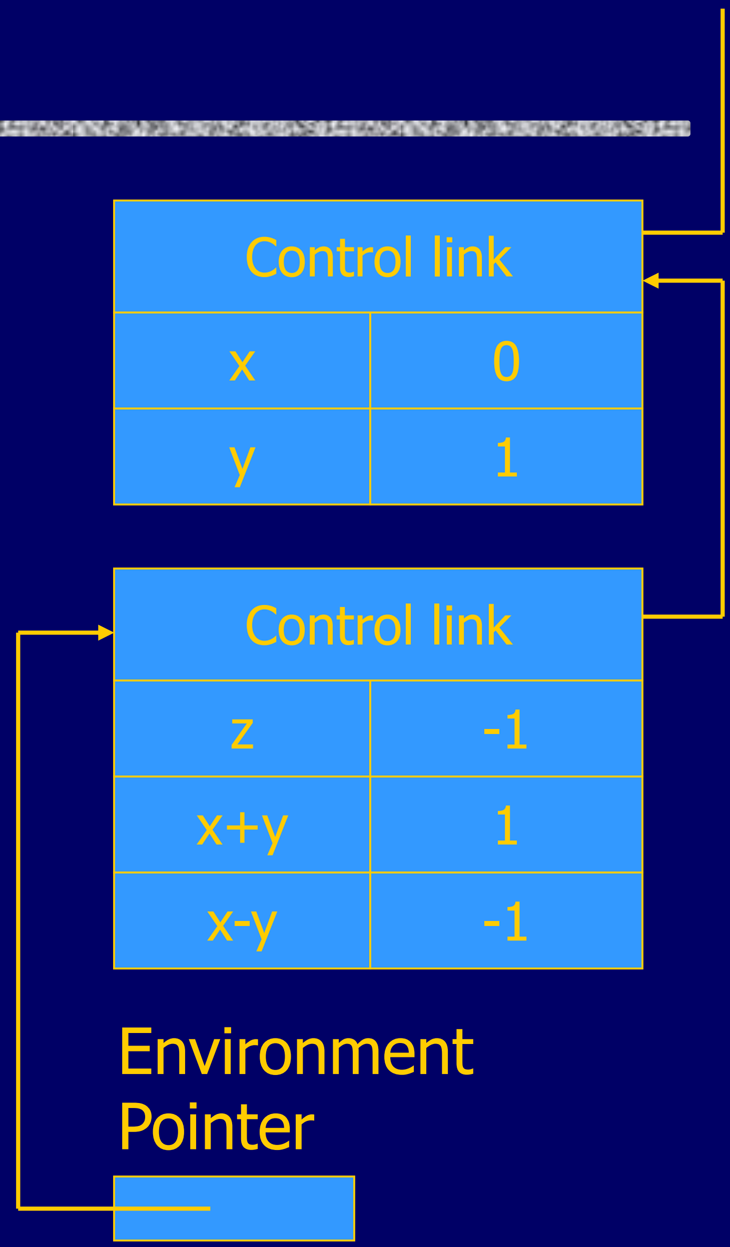
```
{ int x=0;
   int y=x+1;
      {  int z=(x+y)*(x-y);
       };
};
```

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



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.

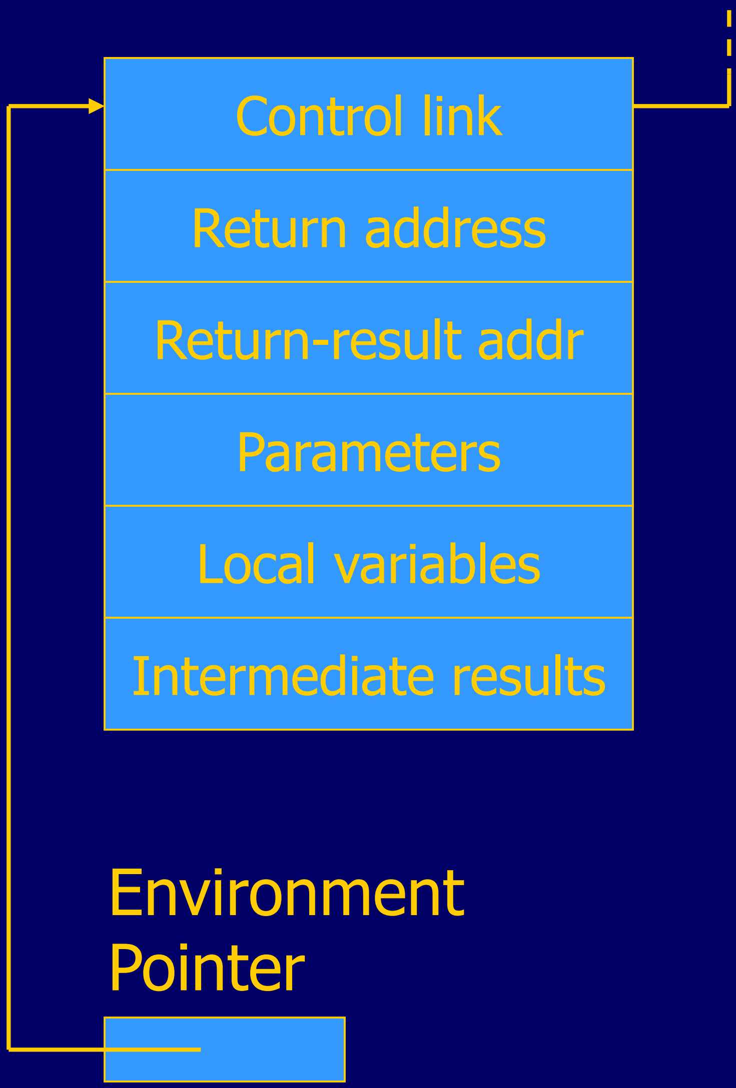
Functions and procedures

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

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

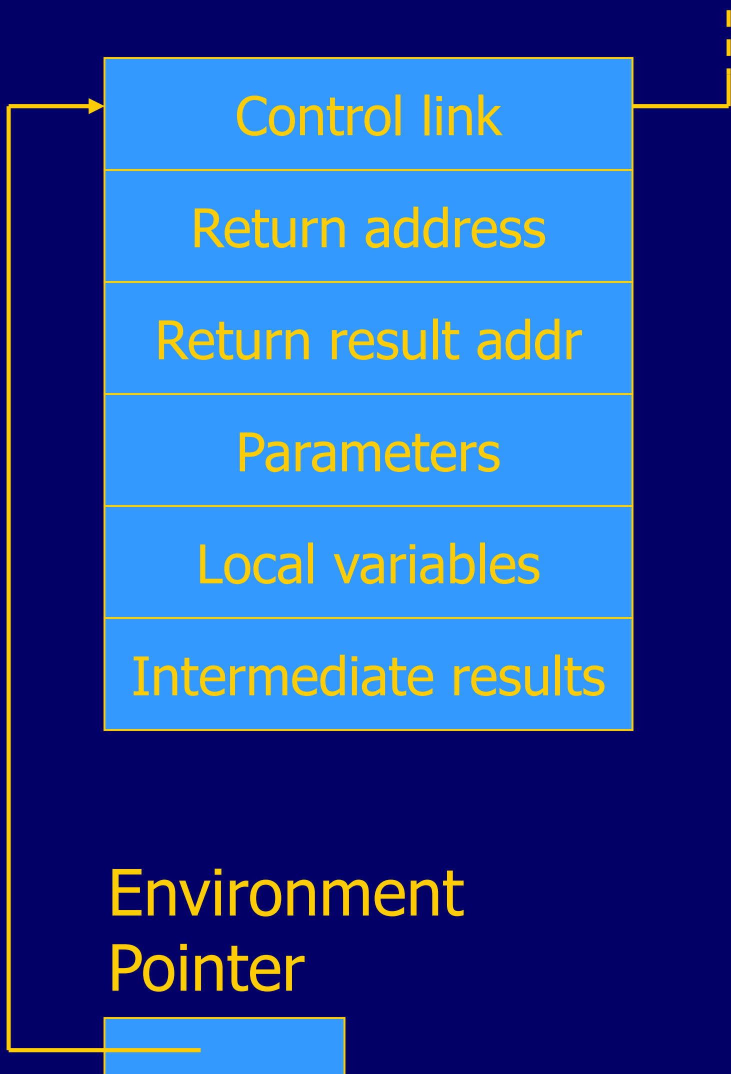
- Activation record must include space for
  - parameters
  - return address
  - local variables, intermediate results
  - return value (an intermediate result)
  - location to put return value on function exit

Activation record for function



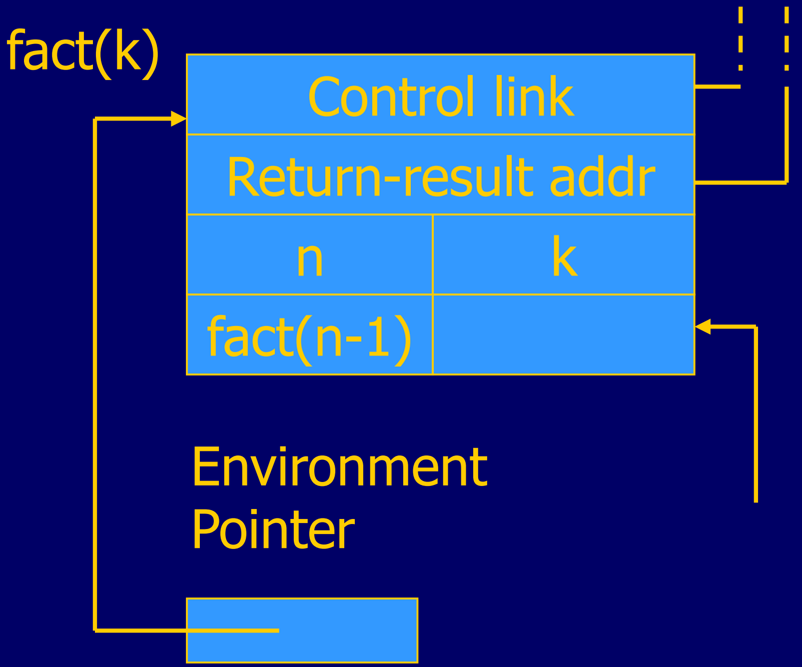
- Return address
  - Location of code to execute on function return
- Return-result address
  - Address in activation record of calling block to receive return address
- Parameters
  - Locations to contain data from calling block

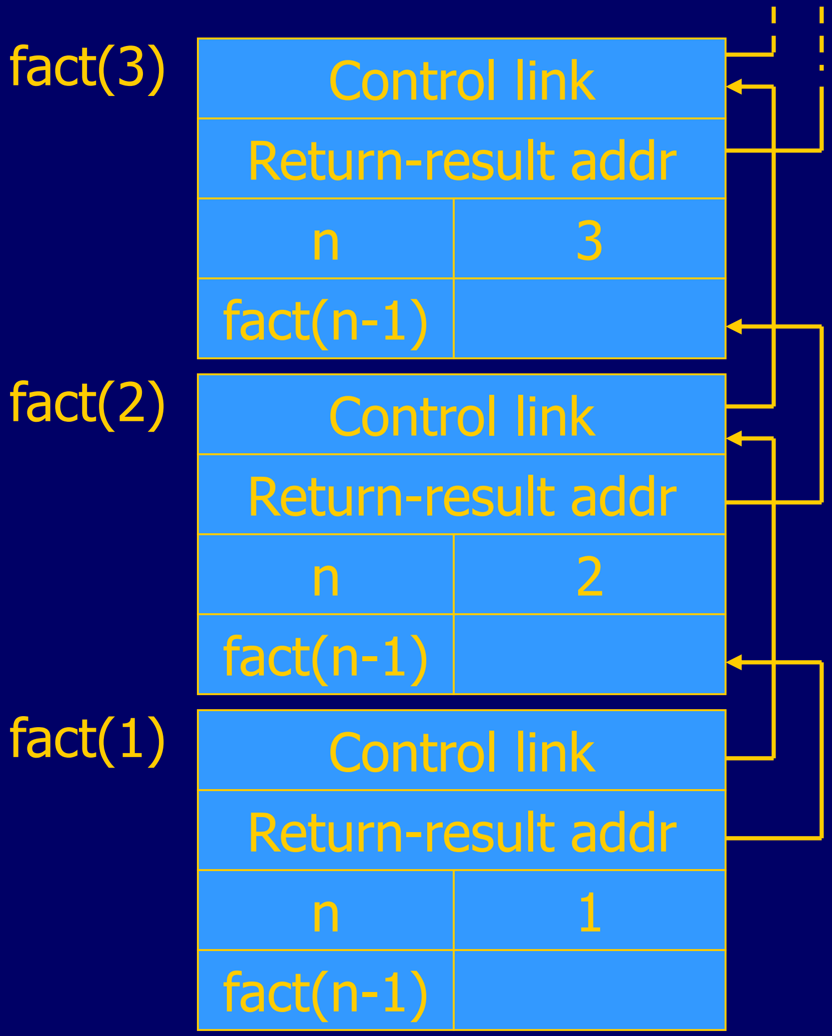
Example



- **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



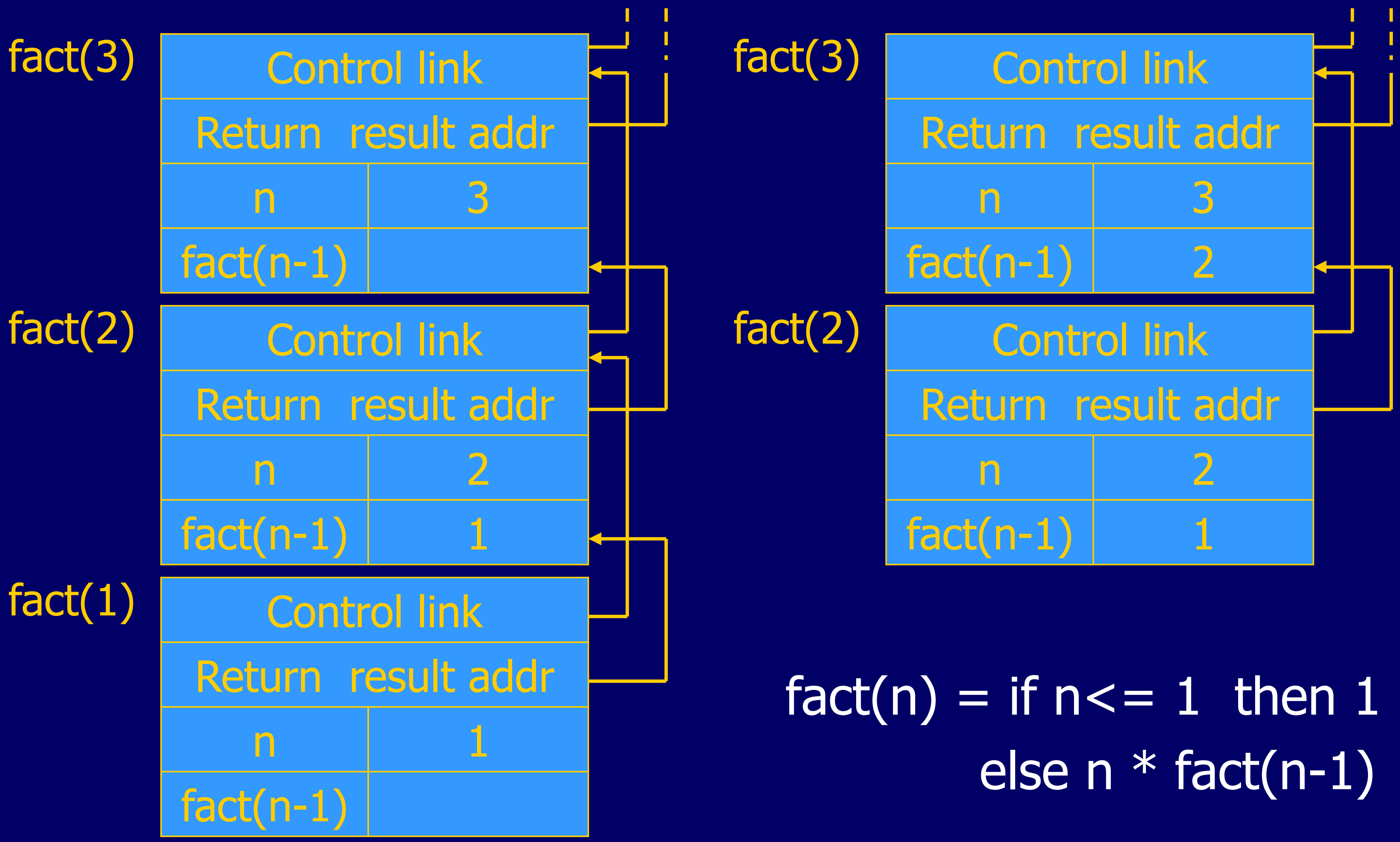


fact(n) = if n<= 1 then 1
else n * fact(n-1)

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

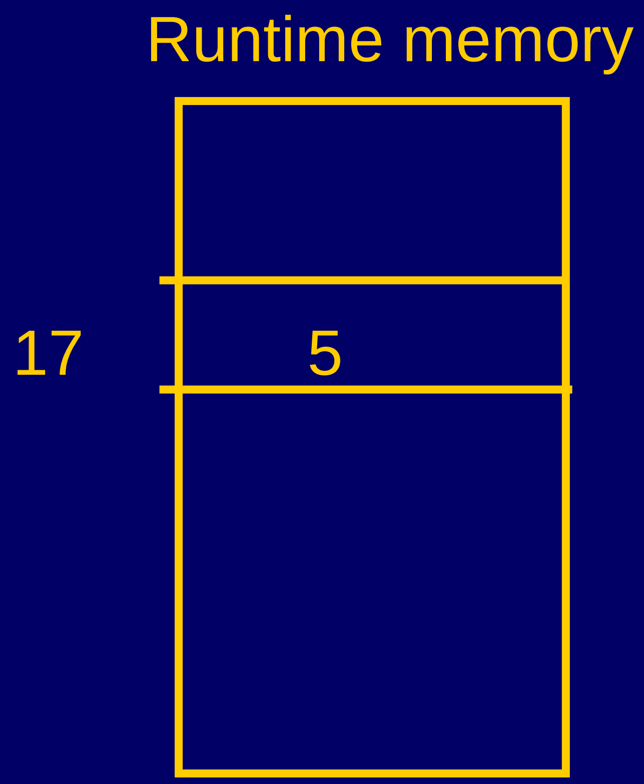
L-values vs. R-values

- Assignment x := exp is compiled into:
  - Compute the **address** of x
  - Compute the **value** of exp
  - Store the value of exp into the address of x
- Generalization
  - R-value
    - Maps program expressions into Context values
  - L-value
    - Maps program expressions into locations
    - Not always defined
  - Java has no small L-values

A Simple Example

int x = 5;

x = x + 1;



A Simple Example

int x = 5;

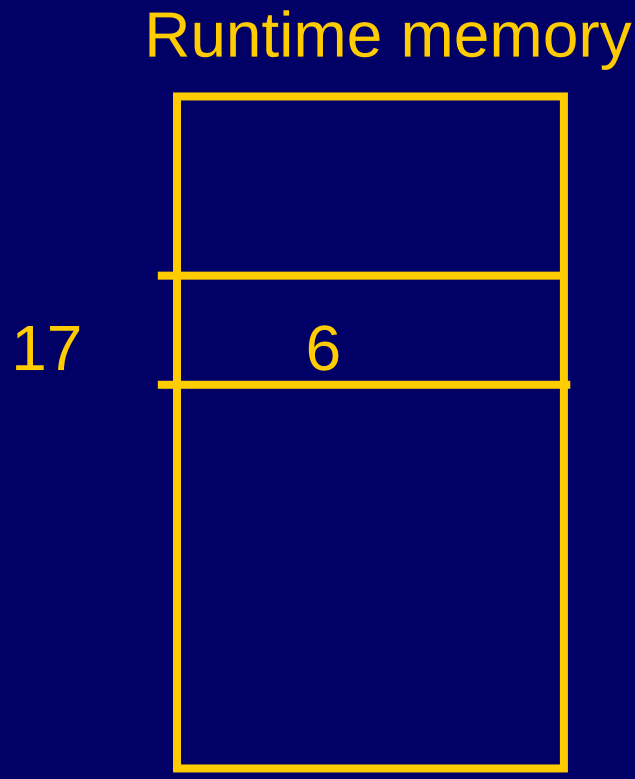
lvalue(x)=17, rvalue(x) =5

lvalue(5)=$\perp$, rvalue(5)=5

x = x + 1;

lvalue(x)=17, rvalue(x) =6

lvalue(5)=$\perp$, rvalue(5)=5



Partial rules for Lvalue in C

- Type of e is pointer to T
- Type of e1 is integer
- lvalue(e2) $\neq$ undefined

```
{ int a[100];
*(a + 5) = 8;
}
```

exp	lvalue	rvalue
id	location(id)	content(location(id))
const	undefined	value(const)
***e**	rvalue(e)	content(rvalue(e))
&e2	undefined	lvalue(e2)
e + e1	undefined	rvalue(e)+sizeof(T)*rvalue(e1)

Parameter passing

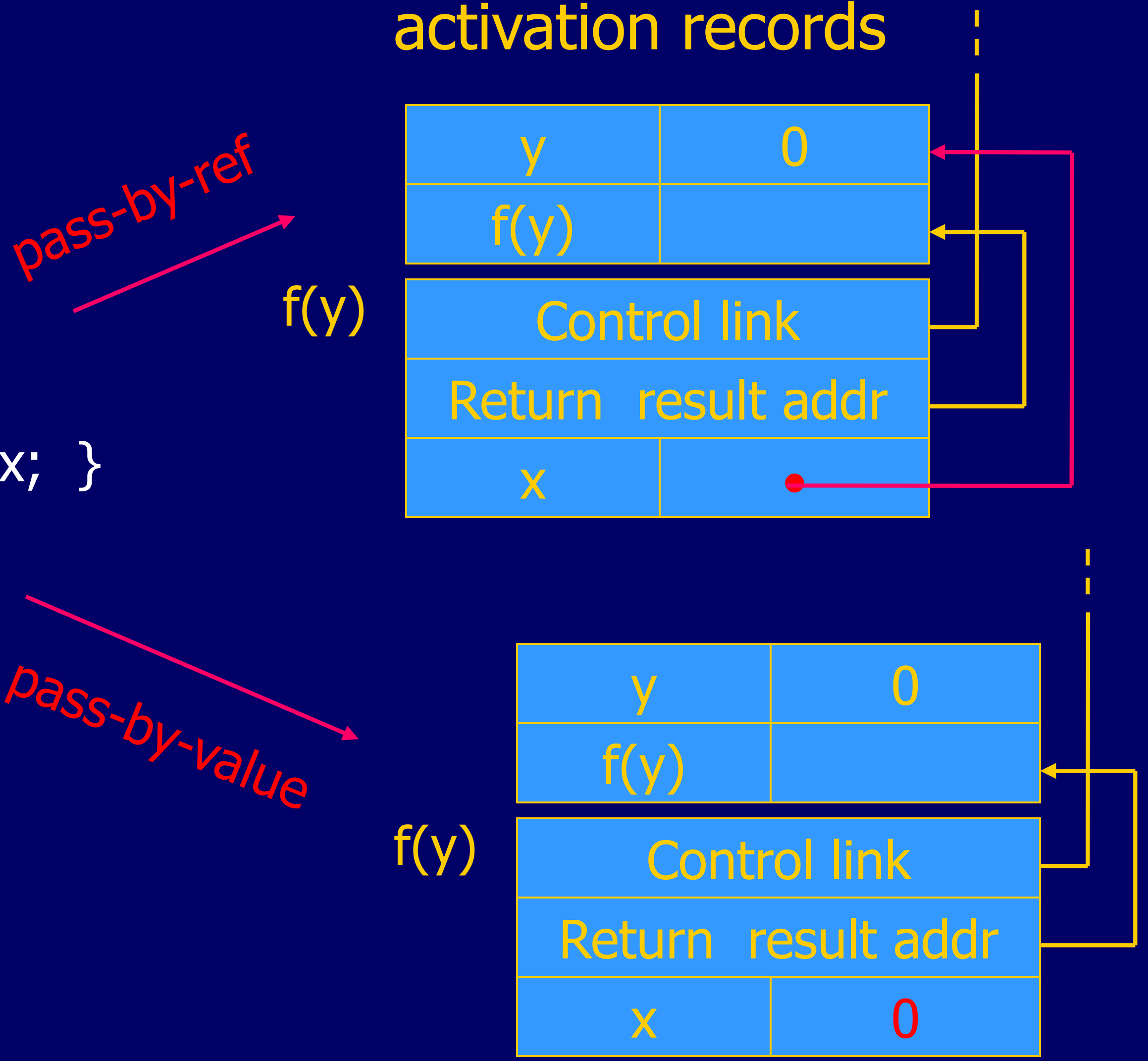
- Pass-by-reference
  - Place L-value (address) in activation record
  - Function can assign to variable that is passed
- Pass-by-value
  - Place R-value (contents) in activation record
  - Function cannot change value of caller’s variable
  - Reduces aliasing (alias: two names refer to same loc)

Example

pseudo-code

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

activation records



Access to global variables

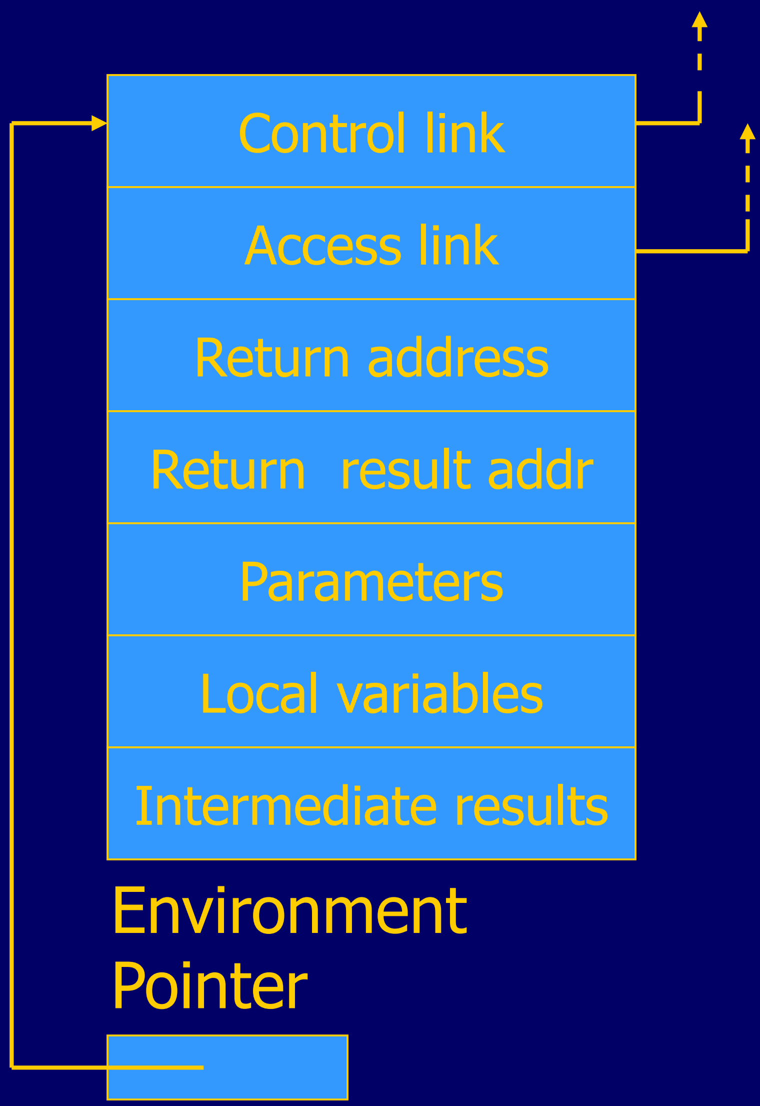
- Two possible scoping conventions
  - Static scope: refer to closest enclosing block
  - Dynamic scope: most recent activation record on stack
- Example

```
var x=1;
function g(z) { return x+z; }
function f(y) {
    var x = y+1;
    return g(y*x);
}
f(3);
g(4)
```

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

Which x is used for expression x+z ?

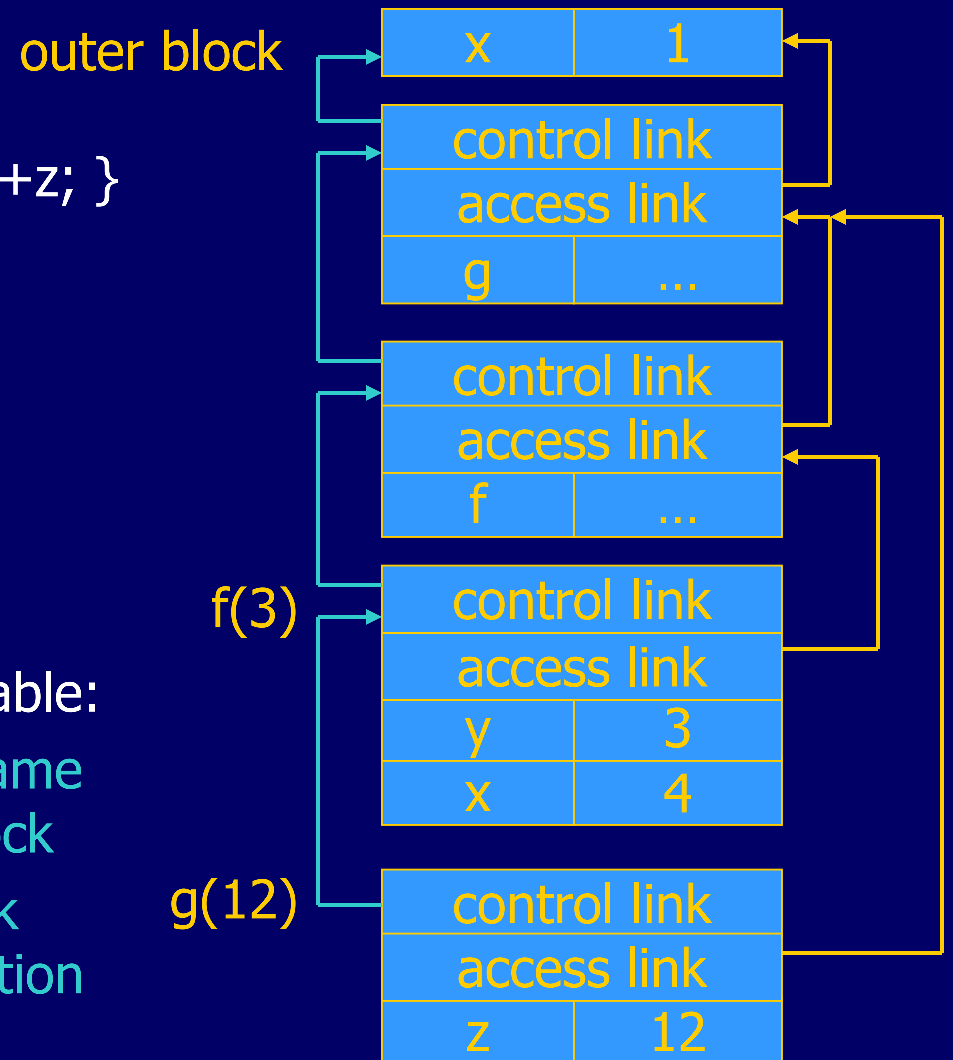
Activation record for static scope



- **Control link**
  - 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);
```



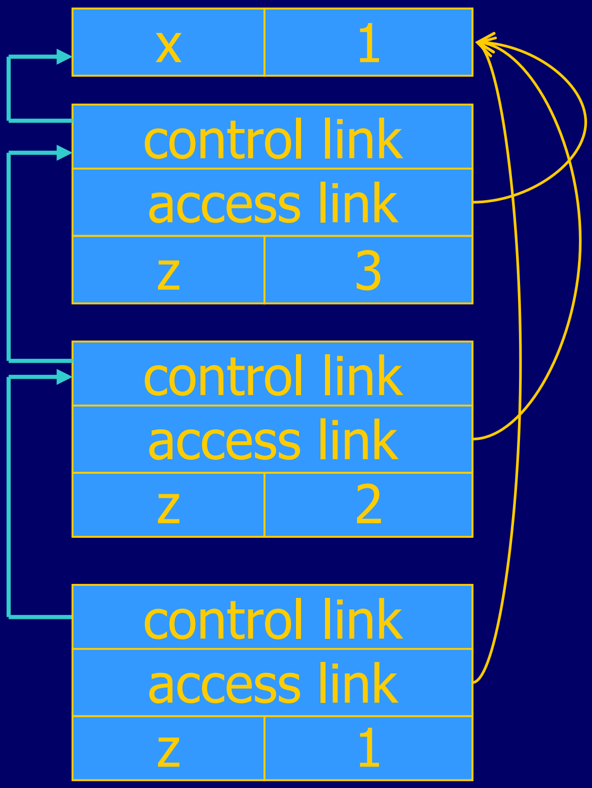
Use access link to find global variable:

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

Static scope with access links & recursion

```
var x=1;
  function fac(z) = {
    if z = 1 then return x;
              else return z * fac(z-1);
  }

fac(3)
```



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

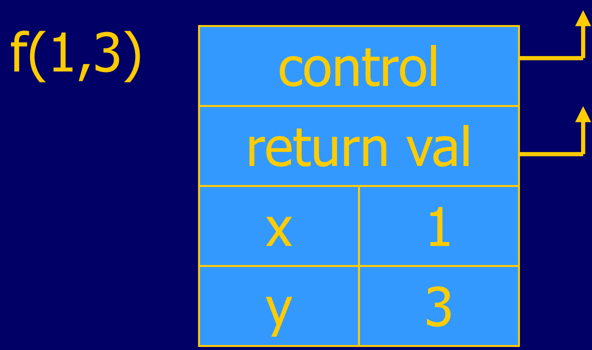
tail call — not a tail call

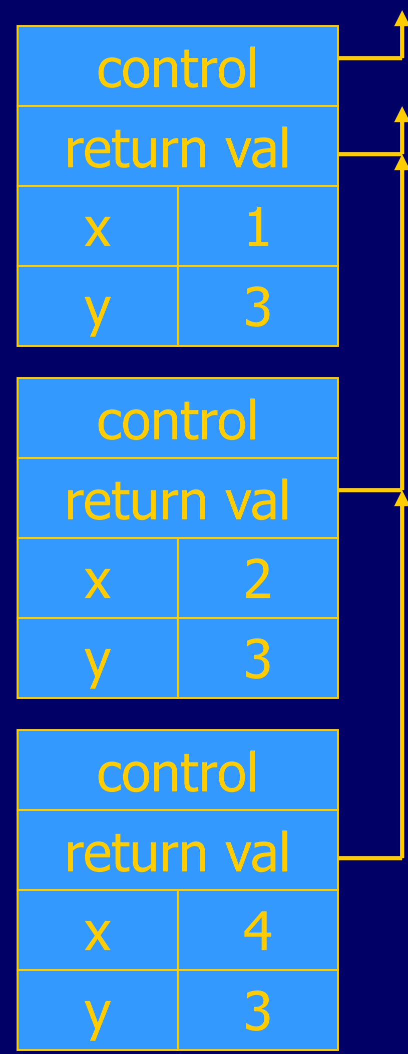
```
fun g(x) = if x>0 then f(x) else f(x)*2
```

- 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

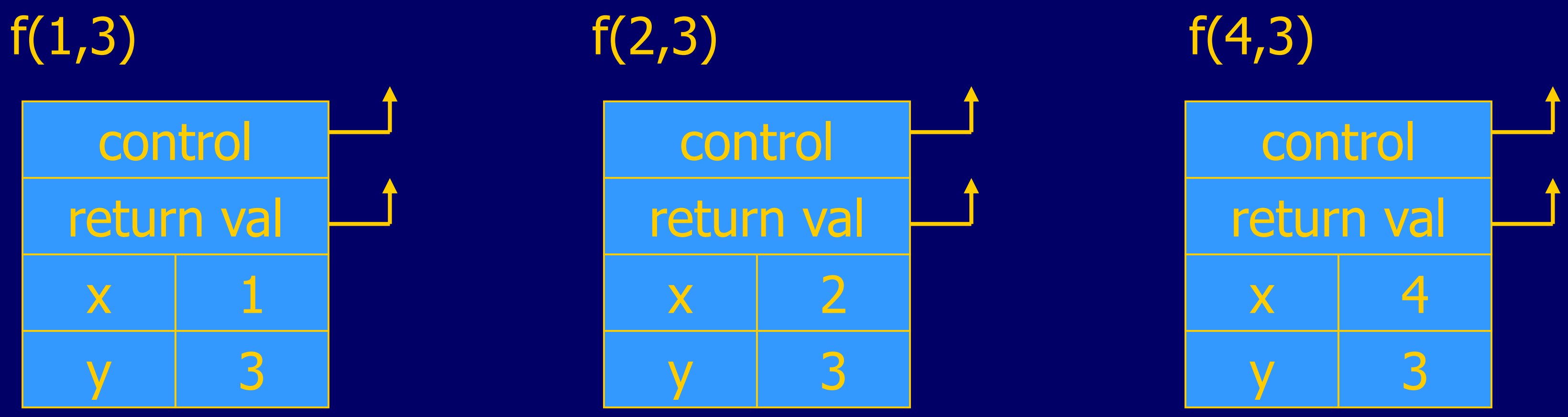
Question

- Can we do the same with control link?

Optimization

- avoid return to caller

Tail recursion elimination




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

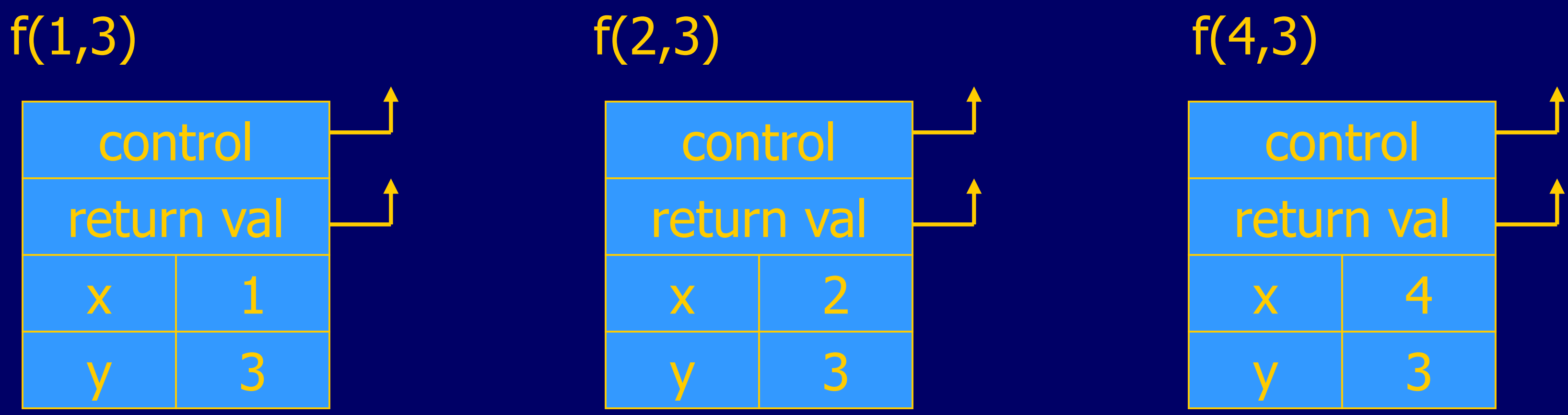
Optimization

- pop followed by push = reuse activation record in place

Conclusion

- Tail recursive function equiv to iterative loop

Tail recursion and iteration




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

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

test

loop body

initial value

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

Pass function as argument

OCaml

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

Pseudo-JavaScript

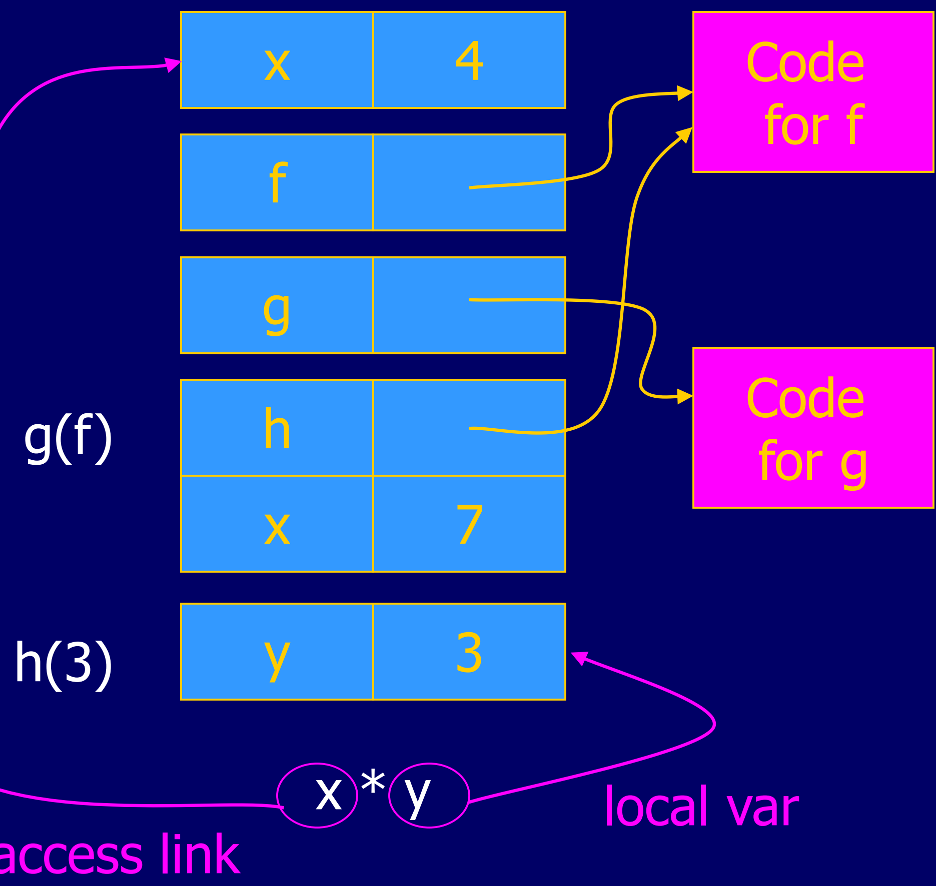
```
{ var x = 4;
  { function f(y) {return x*y};
    { function g(h) {
          var x = 7;
          return h(3) + x;
      };
      g(f);
} } }
```

There are two declarations of x

Which one is used for each occurrence of x?

Static Scope for Function Argument

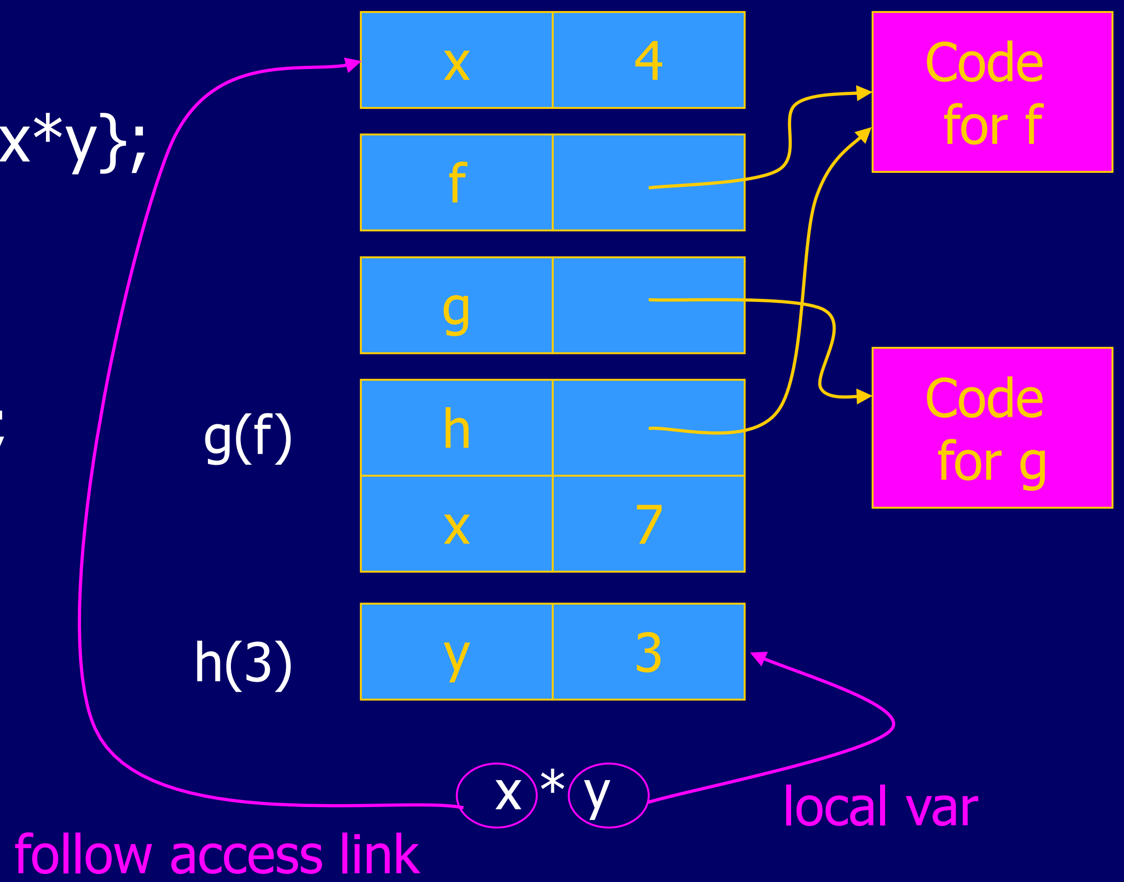
```
let x = 4 in
   let f = fun -> x*y in
      let g = fun h ->
         let
            int x=7
         in
            h(3) + x
      in
      g(f)
```



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

C:\Documents and Settings\John Mitchell\My Documents\...

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

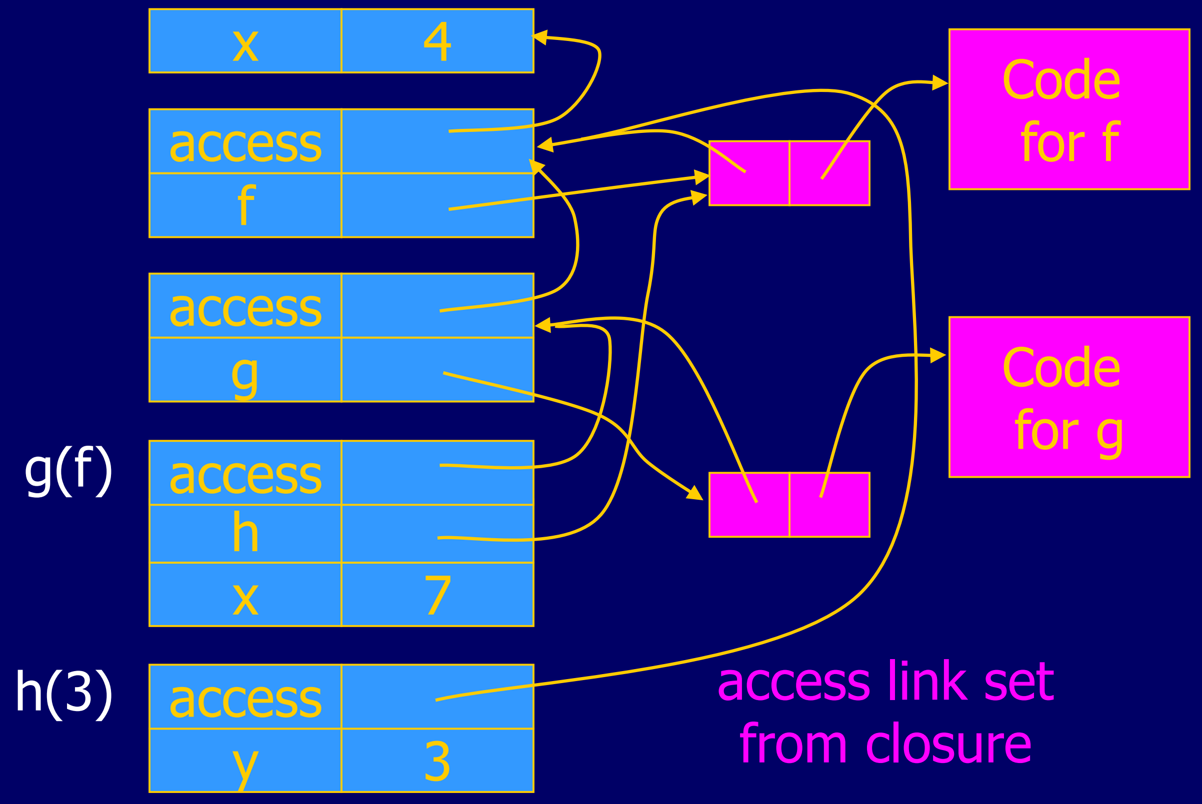
Closures

- 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

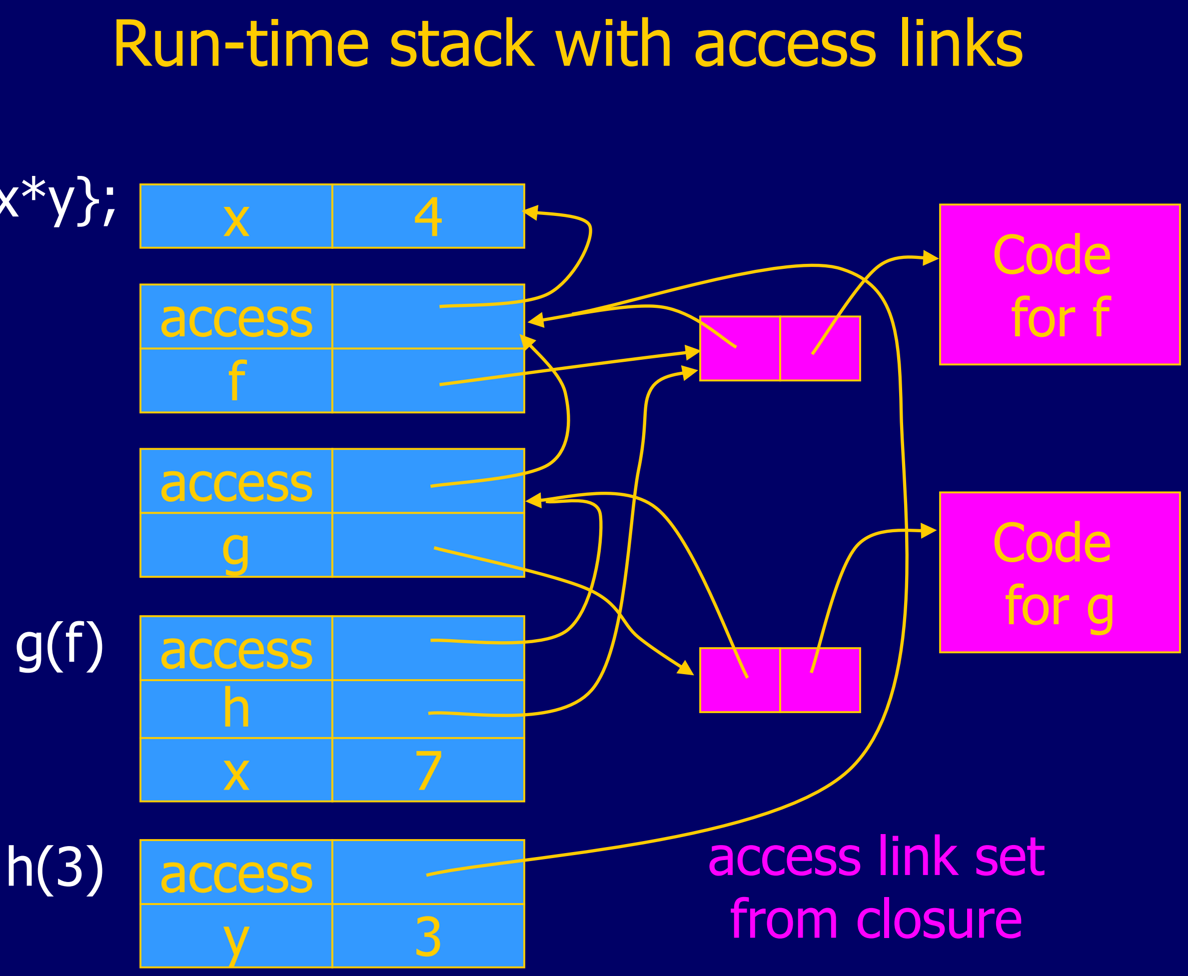
Run-time stack with access links

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



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);
}}}
```



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 = $\langle$env, code$\rangle$
  - code *not* compiled dynamically (in most languages)

Example: Return fctn with private state

OCaml

```
let mk_counter = fun init  ->
    let count = ref init in
      let counter = fun inc ->
              (count := !count + inc; !count)
      in
         counter
in
   let c = mk_counter 1
     in
   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

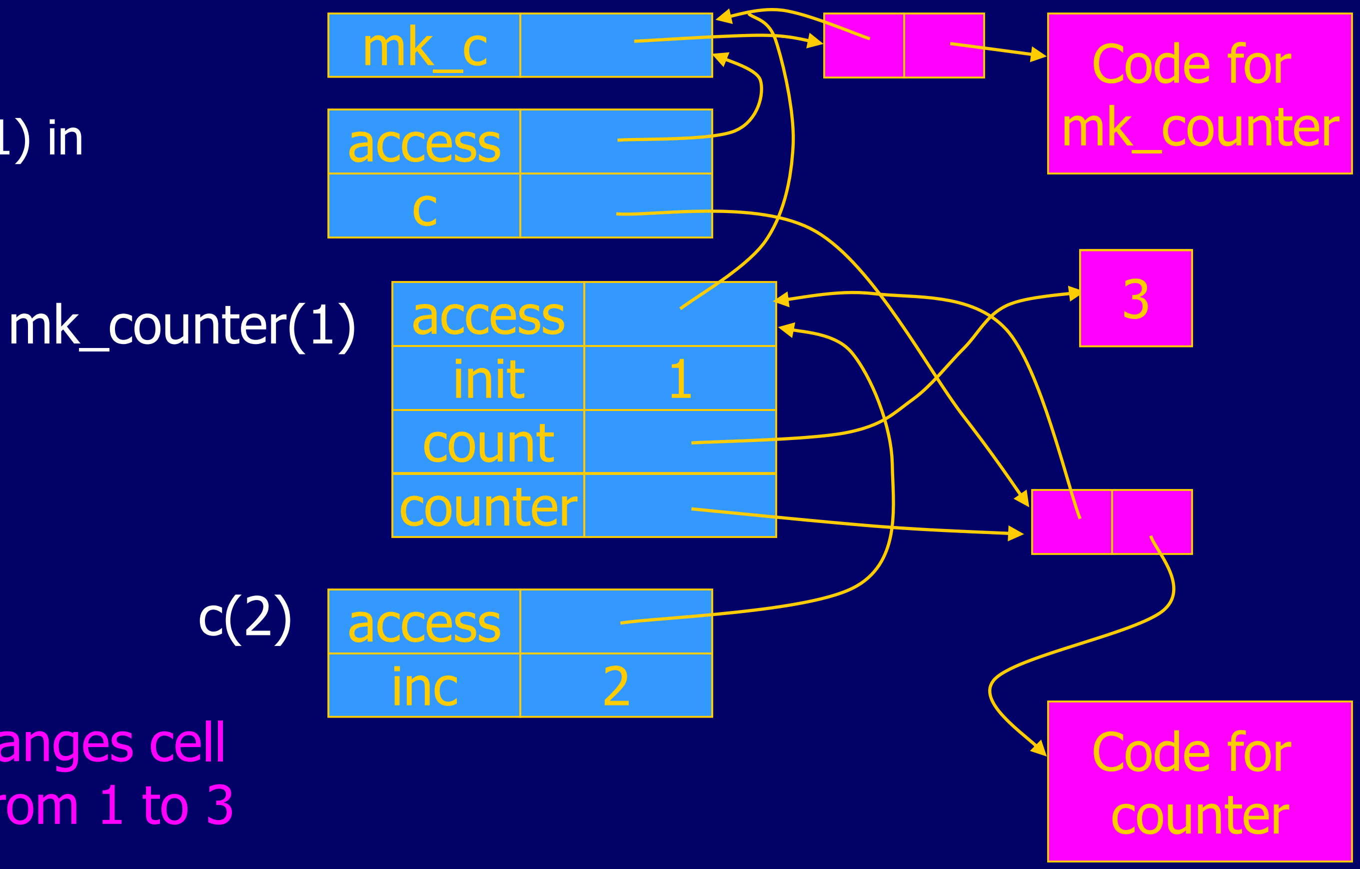
```
function mk_counter (init) {
    var count = init;
    function counter(inc) {count=count+inc; return count};
    return counter};
var c  = mk_counter(1);
c(2) + c(2);
```

Function to “make counter” returns a closure

How is correct value of count determined in call c(2) ?

Function Results and Closures

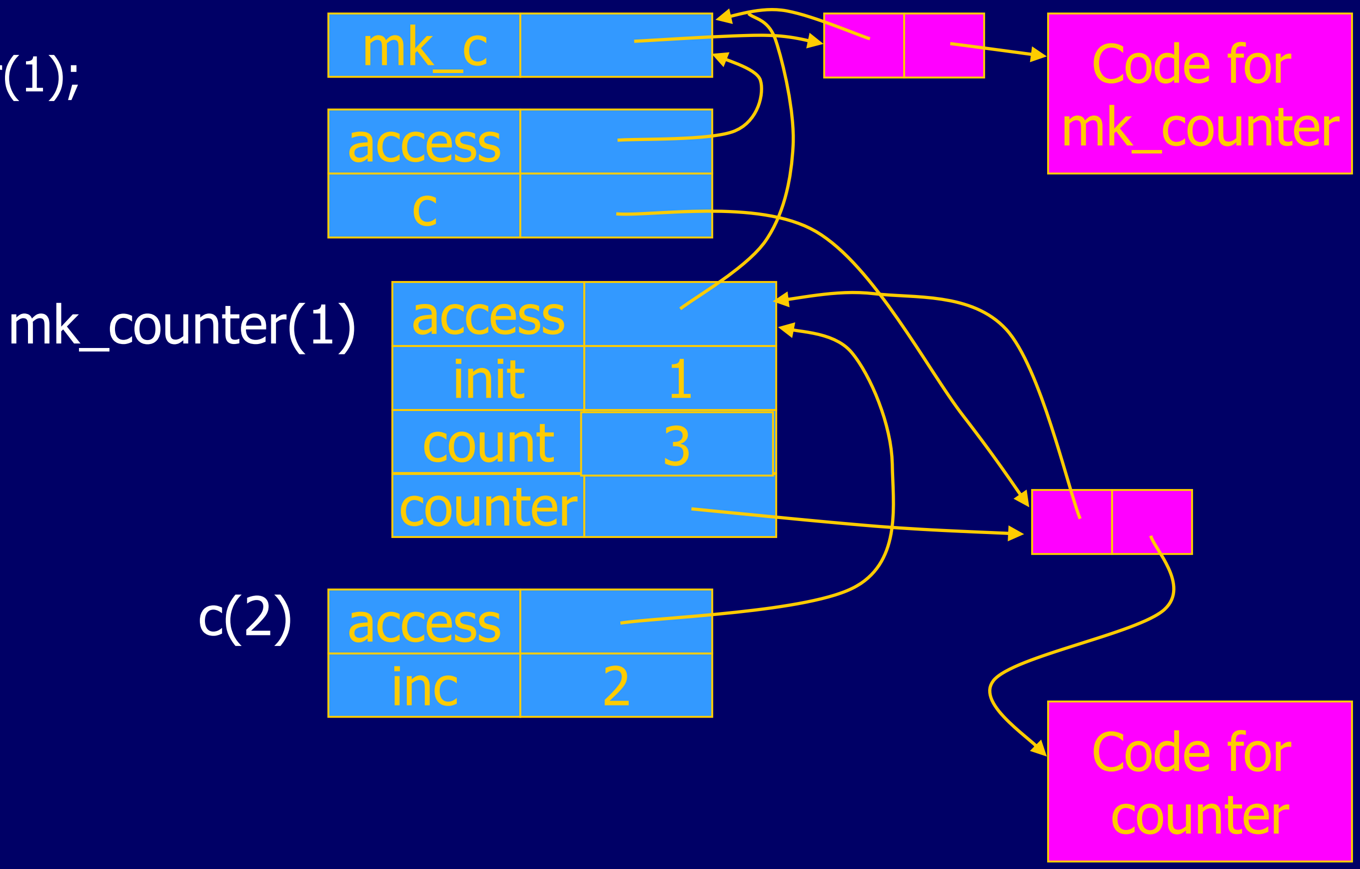
```
let mk_counter =fun init ->
   let count = ref init  in
        let counter = fun inc -> (count := !count + inc; !count)
        in counter
in
let c = mk_counter(1) in
c(2) + c(2)
```



Call changes cell value from 1 to 3

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);
```



Closures in Web programming

- Useful for event handlers in Web programming:

```
function AppendButton(container, name, message) {
    var btn = document.createElement('button');
    btn.innerHTML = name;
    btn.onclick = function (evt) { alert(message); }
    container.appendChild(btn);
}
```

- Environment pointer lets the button’s click handler find the message to display

Simple C Program

```
foo (int y) {
   int x = y ;
   if  (x > 8) {
     int x = y + 1 ;
     x = x  + 1 ;
     }
```

foo(9)

control	
return val	
x	9
y	9
x	10

The C Programming Language

- Designed to allow stack allocation
- Local variables are flattened
- No need for control link
- Permit
  - Nested blocks
  - Passing functions as parameters and return values
- Forbid
  - Nested functions

Summary: Return Function Results

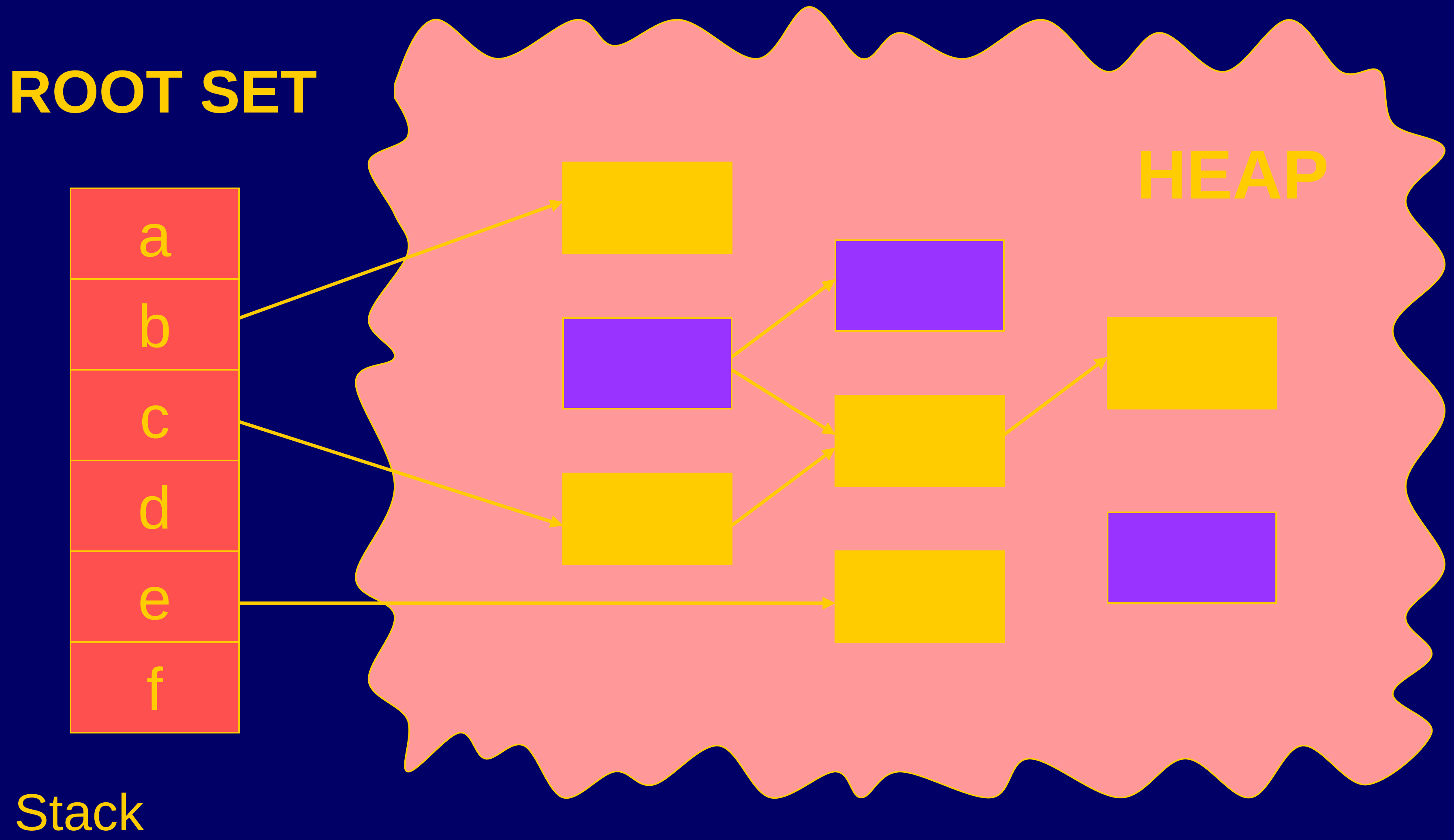
- Use closure to maintain static environment
- May need to keep activation records after return
  - Stack (lifo) order fails!
- Possible "stack" implementation
  - Forget about explicit deallocation
  - Put activation records on heap
  - Invoke garbage collector as needed
  - Not as totally crazy as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

    May only need to search reachable data

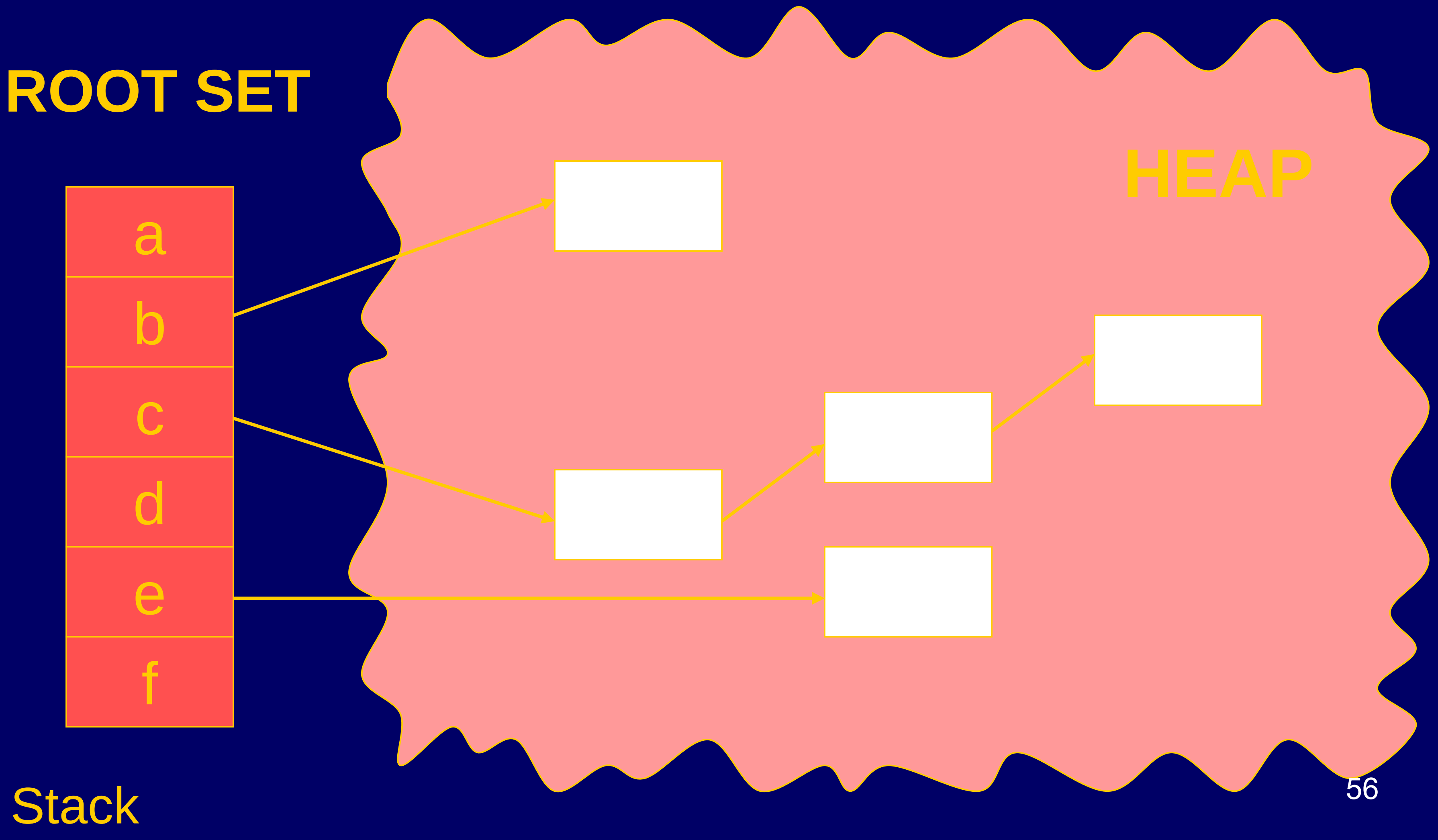
Summary of scope issues

- 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

Garbage Collection



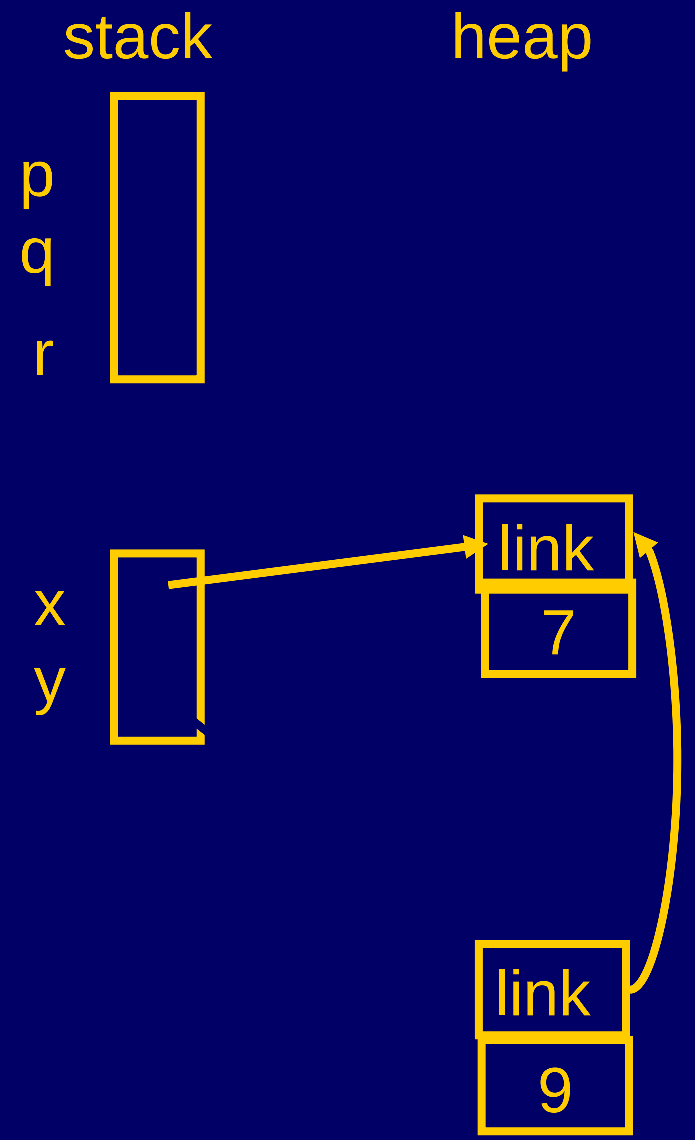
Garbage Collection



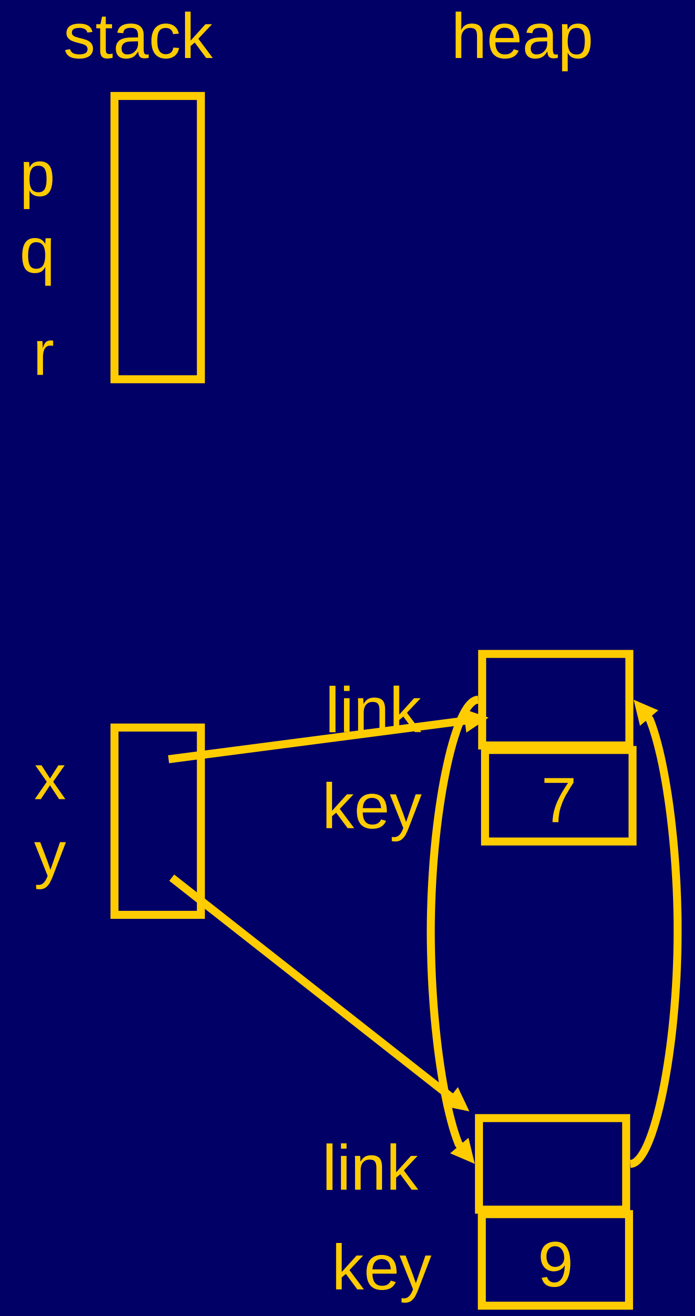
What is garbage collection

- The runtime environment reuse chunks that were allocated but are not subsequently used garbage chunks
  - not live
- It is undecidable to find the garbage chunks:
  - Decidability of liveness
  - Decidability of type information
- Conservative collection
  - every live chunk is identified
  - some garbage runtime chunk are not identified
- Find the reachable chunks via pointer chains
- Often done in the allocation function

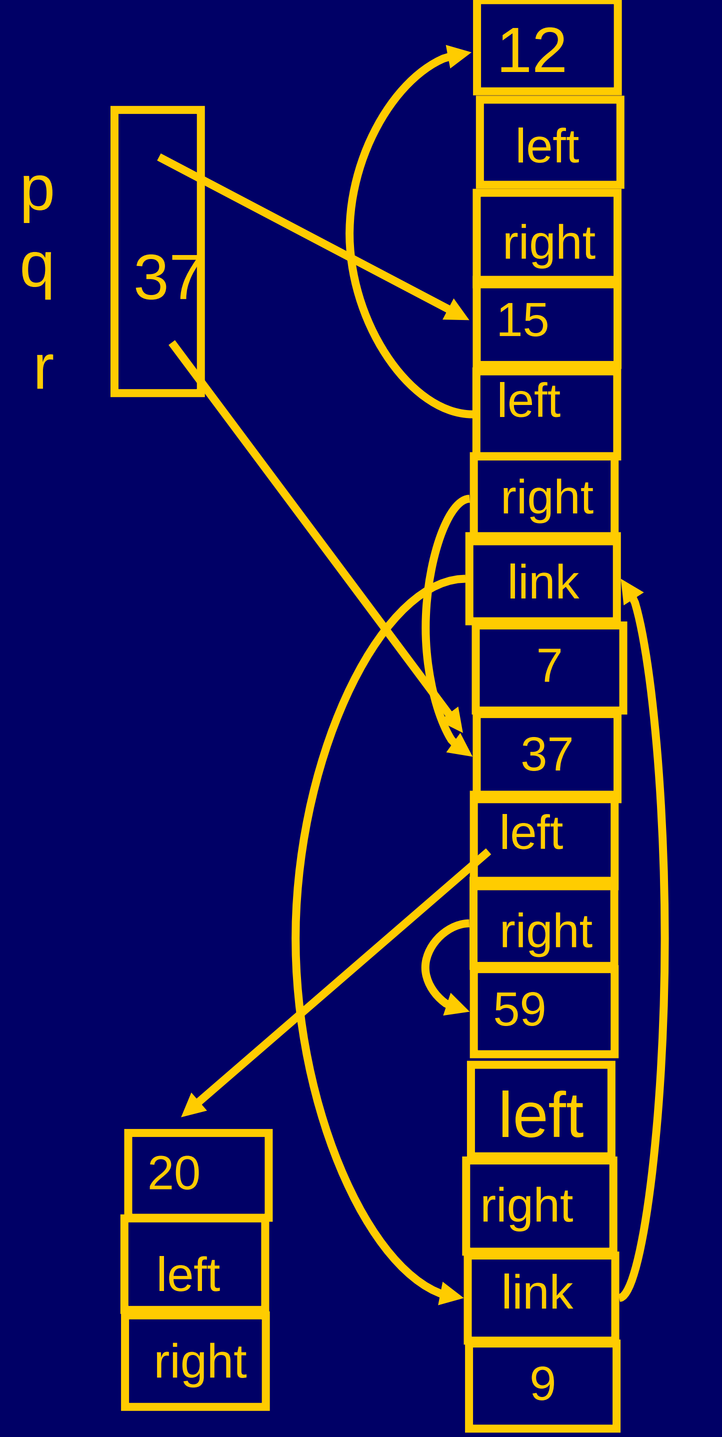
```
typedef struct list  {struct list *link; int key} *List;
typedef struct tree {int key;
                              struct tree *left:
                              struct tree *right} *Tree;
foo() {    List x = cons(NULL, 7);
    List y = cons(x, 9);
   x->link = y;
     }
void main() {
  Tree p, r; int q;
  foo();
  p = maketree();   r = p->right;
  q= r->key;
  showtree(r);}
```



```
typedef struct list  {struct list *link; int key} *List;
typedef struct tree {int key;
                              struct tree *left:
                              struct tree *right} *Tree;
foo() {    List x = cons(NULL, 7);
    List y = cons(x, 9);
   x->link = y;
     }
void main() {
  Tree p, r; int q;
  foo();
  p = maketree();   r = p->right;
  q= r->key;
  showtree(r);}
```



```
typedef struct list  {struct list *link; int key} *List;
typedef struct tree {int key;
                                struct tree *left:
                                struct tree *right} *Tree;
foo() {     List x = create_list(NULL, 7);
    List y = create_list(x, 9);
   x->link = y;
     }
void main() {
  Tree p, r; int q;
  foo();
  p = maketree();   r = p->right;
  q= r->key;
  showtree(r);}
```



Garbage Collection Techniques

- Tracing
  - Scan the reachable heaps from the root
  - Release unreachable elements
  - Cost proportional to reachable heap
- Reference Counting
  - Maintain a counter of references to each chunk of memory
  - The compiler generates the update code for references when pointers are manipulated
  - Release objects with zero reference counter
  - Constant cost

Mark-and-Sweep(Scan) Collection

- Mark the chunks reachable from the roots (stack, static variables and machine registers)
- Sweep the heap space by moving unreachable chunks to the freelist (Scan)

The Mark Phase

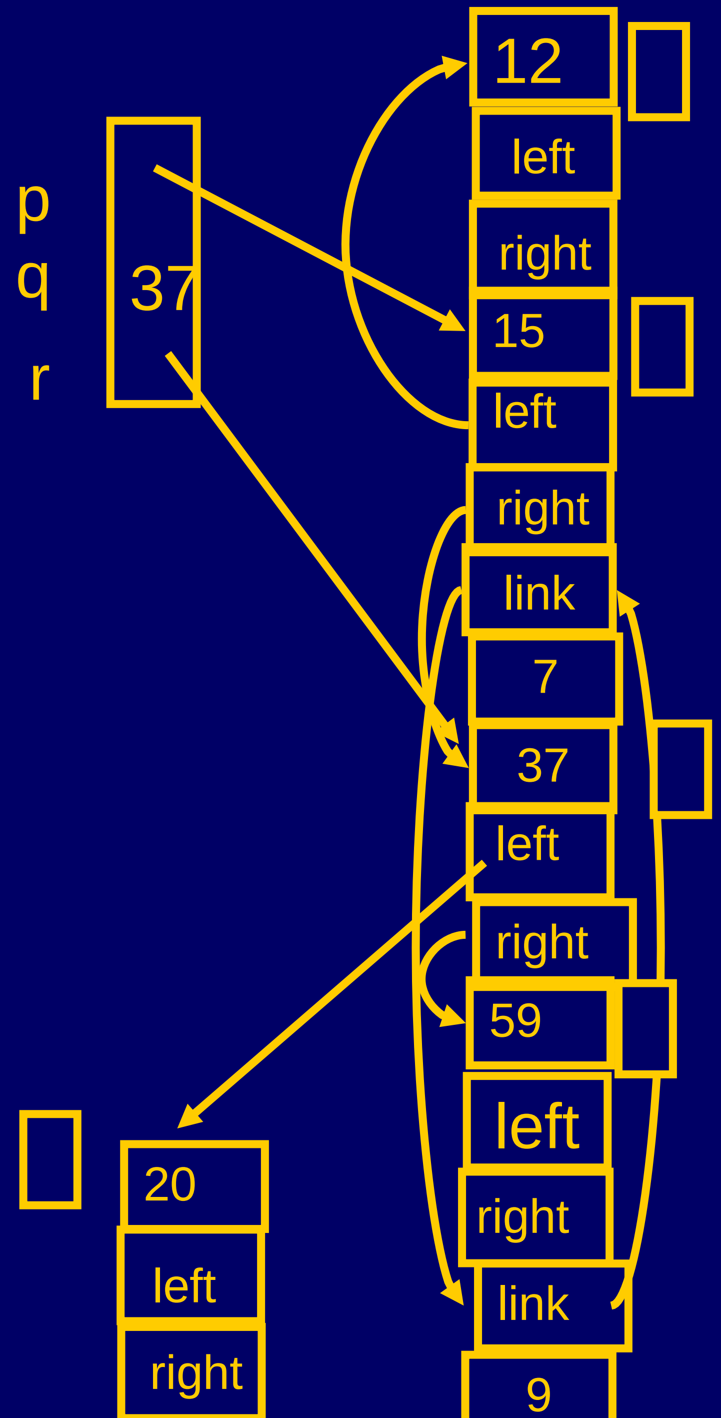
```
for each root v
   DFS(v)

function DFS(x)
   if x is a pointer and chunk x is not marked
         mark x
         for each reference field f_i of chunk x
             DFS(x.f_i)
```

The Sweep Phase

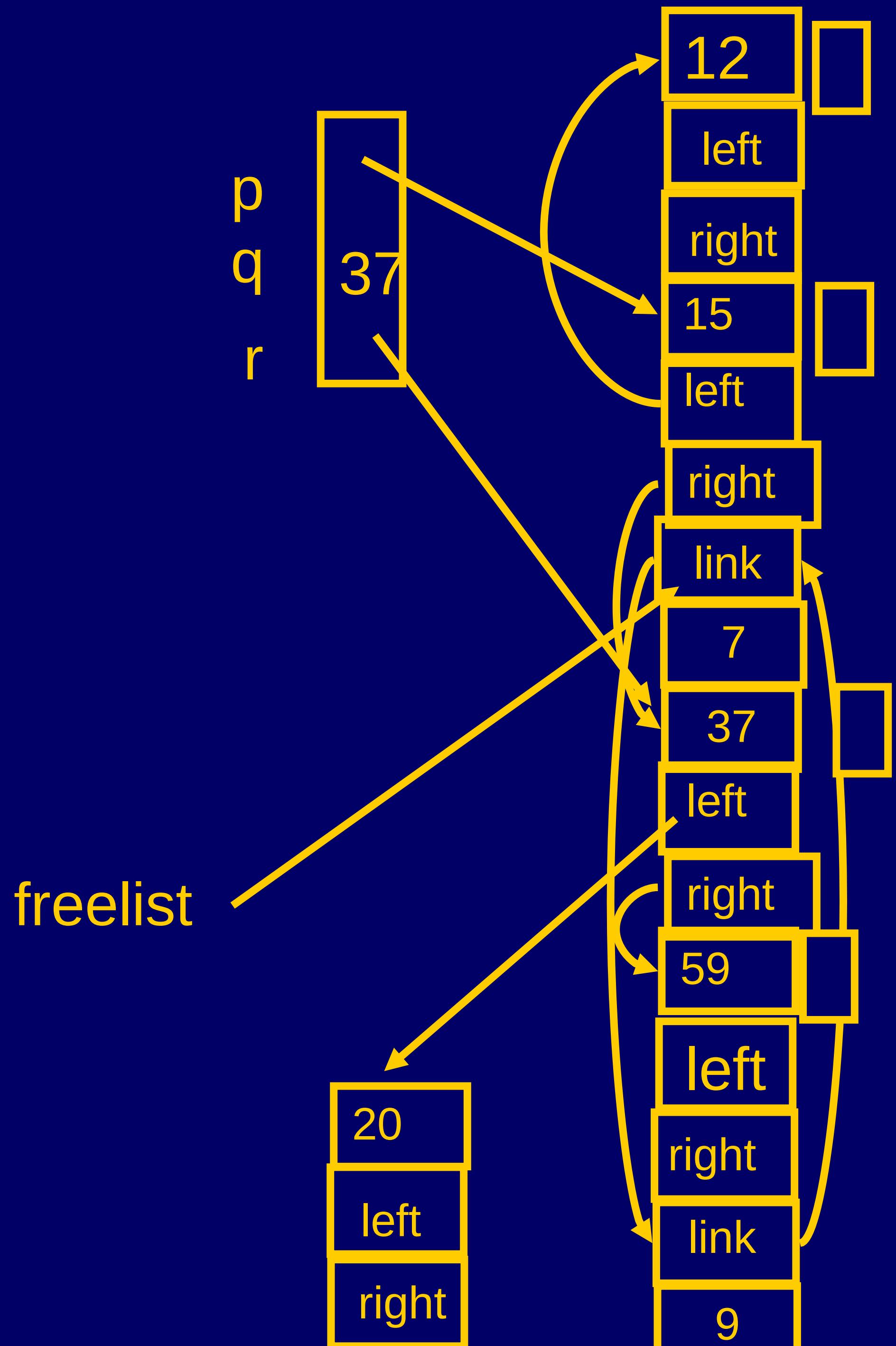
```
p := first address in heap
while p < last address in the heap
        if chunk p is marked
                unmark p
        else let f1 be the first pointer reference field in
p
                p.f1 := freelist
                freelist := p
        p := p + size of chunk p
```

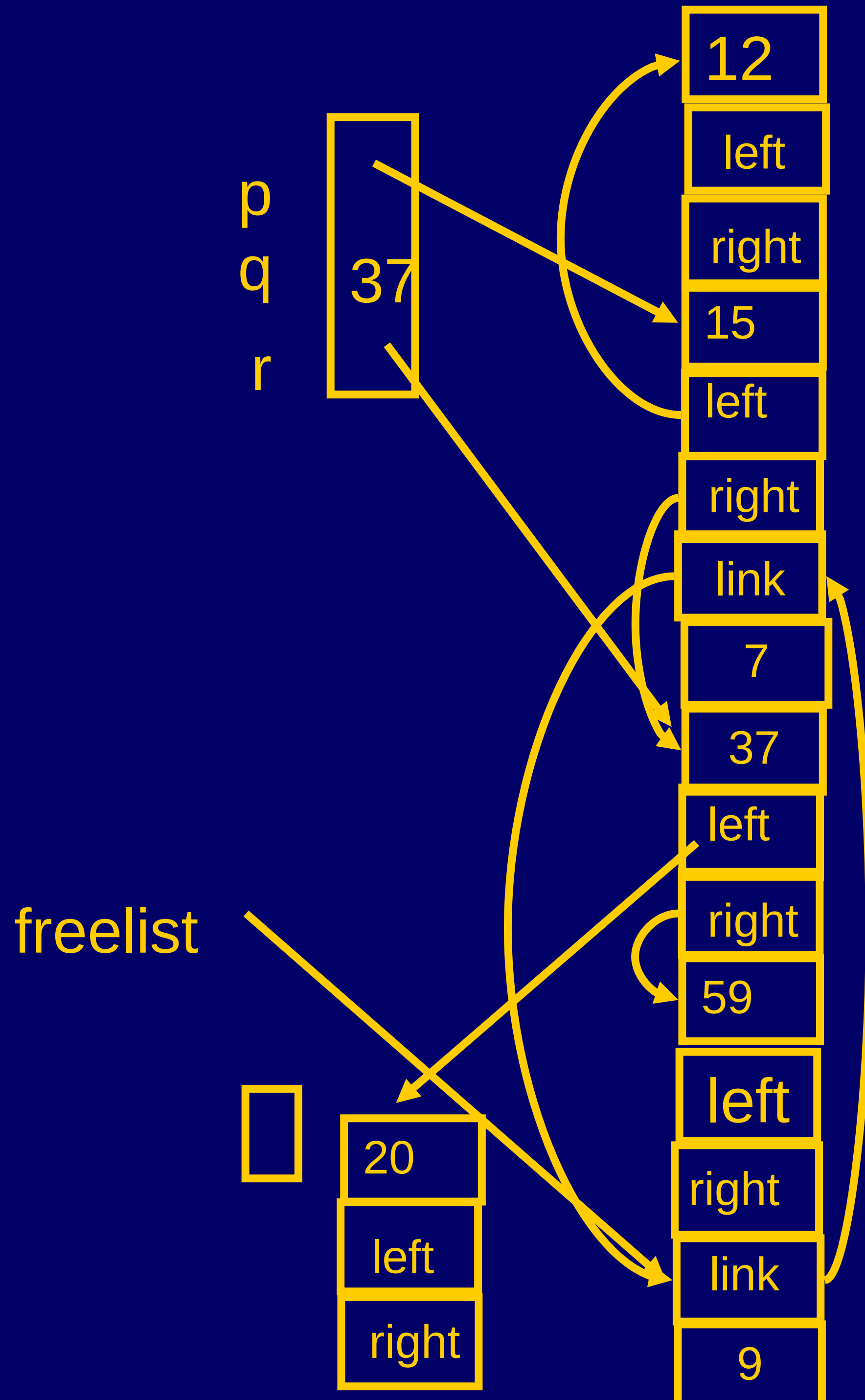
Mark



p
q
37
r
12
left
right
15
left
right
link
7
37
left
right
59
left
right
link
9
20
left
right

Sweep


p
q
r
37
12
left
right
15
left
right
link
7
37
left
right
59
left
right
link
9
freelist
20
left
right



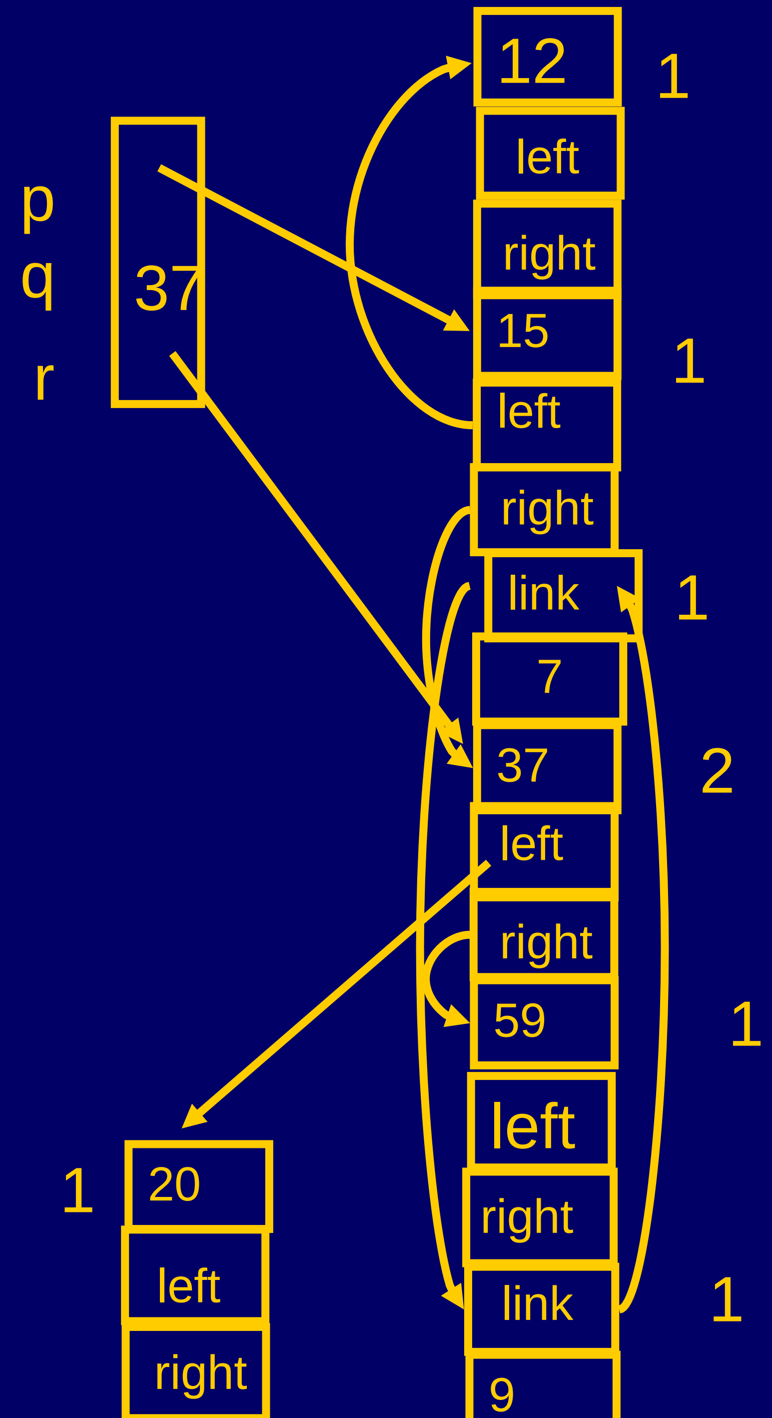
12
left
right
p
q
37
r
15
left
right
link
7
37
left
freelist
right
59
left
20
right
left
link
right
9

Cost of GC

- The cost of a single garbage collection can be linear in the size of the store
  - may cause quadratic program slowdown
- Amortized cost
  - collection-time/storage reclaimed
  - Cost of one garbage collection
    - $c_1 R + c_2 H$
  - H - R Reclaimed chunks
  - Cost per reclaimed chunk
    - $(c_1 R + c_2 H)/ (H - R)$
  - If $R/H > 0.5$
    - increase H
  - if $R/H < 0.5$
    - cost per reclaimed word is $c_1 + 2c_2 \sim 16$
  - There is no lower bound

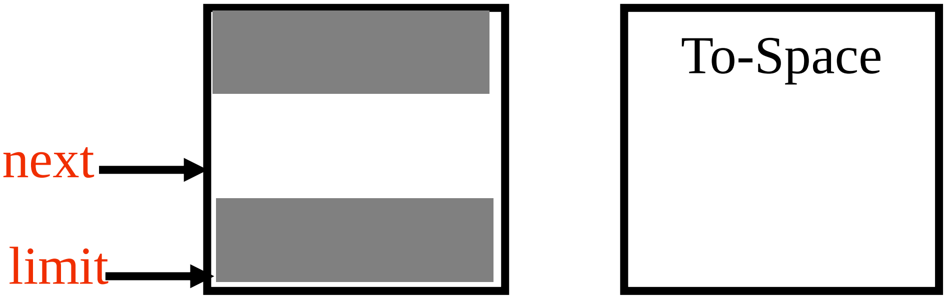
Reference Counting

- Maintain a counter per object
- The compiler generates updates for counters
- Release object with zero counters
- Cannot reclaim cyclic objects


p
q
r
37
12
left
right
15
left
right
link
7
37
left
right
59
left
right
link
9
20
left
right
1
1
1
2
1
1
1

Copying Collection

- Maintains two separate heaps
  - from-space
  - to-space
- pointer next to the next free chunk in from-space
- A pointer limit to the last chunk in from-space
- If next = limit copy the reachable chunks from from-space into to-space
  - set next and limit
  - Switch from-space and to-space
- Requires type information



Generational Garbage Collection

- Newly created objects contain higher percentage of garbage
- Partition the heap into generations G_1 and G_2
- First garbage collect the G_1 heap
  - chunks which are reachable
- After two or three collections chunks are promoted to G_2
- Once a while garbage collect G_2
- Can be generalized to more than two heaps
- But how can we garbage collect in G_1?

Scanning roots from older generations

- remembered list
  - The compiler generates code after each destructive update $b.f_i := a$
    to put b into a vector of updated objects scanned by the garbage collector
- remembered set
  - remembered-list + “set-bit”
- Card marking
  - Divide the memory into 2^k cards
- Page marking
  - k = page size
  - virtual memory system catches updates to old-generations using the dirty-bit

Incremental Collection

- Even the most efficient garbage collection can interrupt the program for quite a while
- Under certain conditions the collector can run concurrently with the program (mutator)
- Need to guarantee that mutator leaves the chunks in consistent state, e.g., may need to restart collection
- Two solutions
  - compile-time
    - Generate extra instructions at store/load
  - virtual-memory
    - Mark certain pages as read(write)-only
    - a write into (read from) this page by the program restart mutator

Garbage Collection vs. Explicit Memory Deallocation

- Faster program development
- Less error prone
- Can lead to faster programs
  - Can improve locality of references
- Support very general programming styles, e.g. higher order and OO programming, Closure
- Standard in ML, Java, C#, Javascript
- Supported in C and C++ via separate libraries

- May require more space
- Needs a large memory
- Can lead to long pauses
- Can change locality of references
- Effectiveness depends on programming language and style
- Hides documentation
- More trusted code

Summary

- Runtime memory management is crucial for functionality and correctness
- Lexical scope is natural
  - Becomes tricky with higher order functions
  - Closures
- Garbage Collection permits general programming style