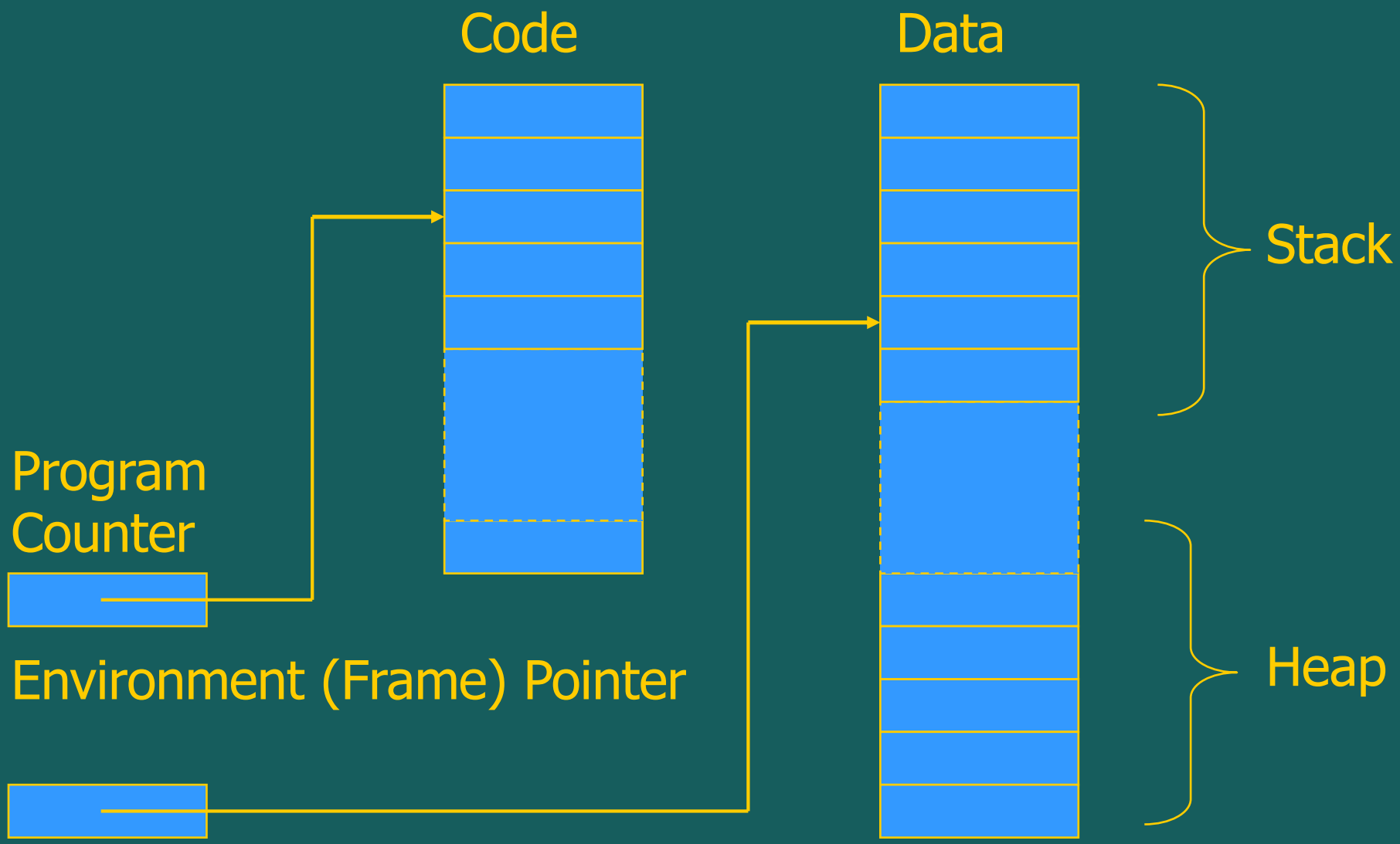
Scope, Function Calls and Storage Management

Lecture 10

Topics

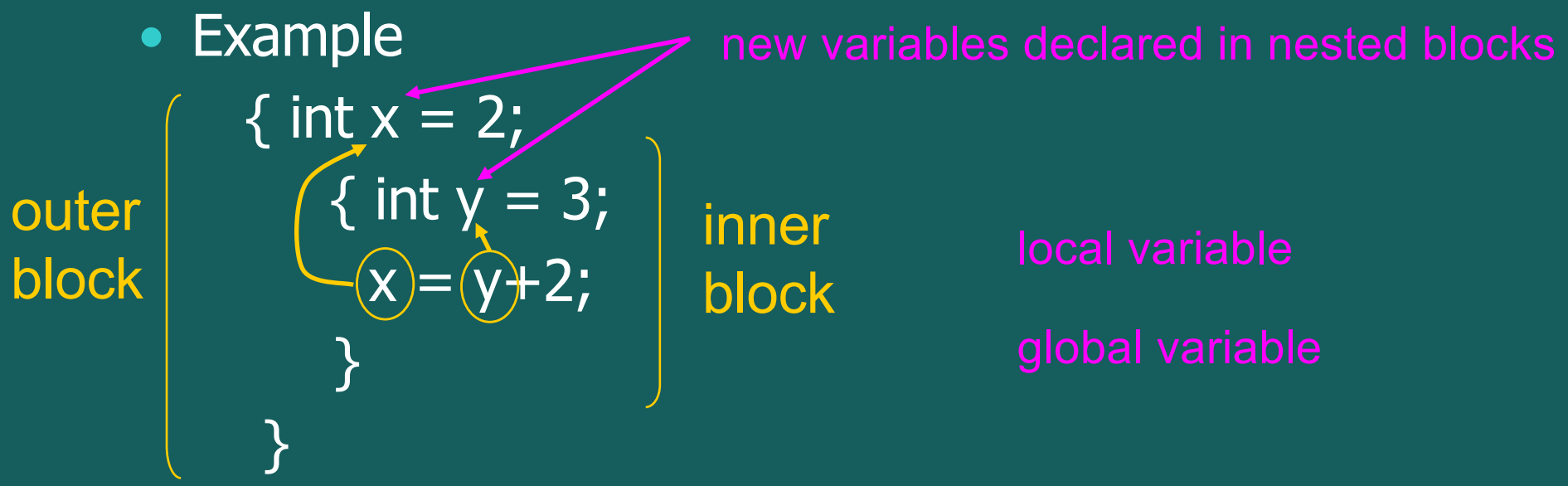
- Block-structured languages and stack storage
  - activation records
  - storage for local, global variables
- First-order functions
  - parameter passing (later)
  - tail recursion and iteration
- Higher-order functions
  - deviations from stack discipline
  - language expressiveness => implementation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

Simplified Machine Model



Block-Structured Languages

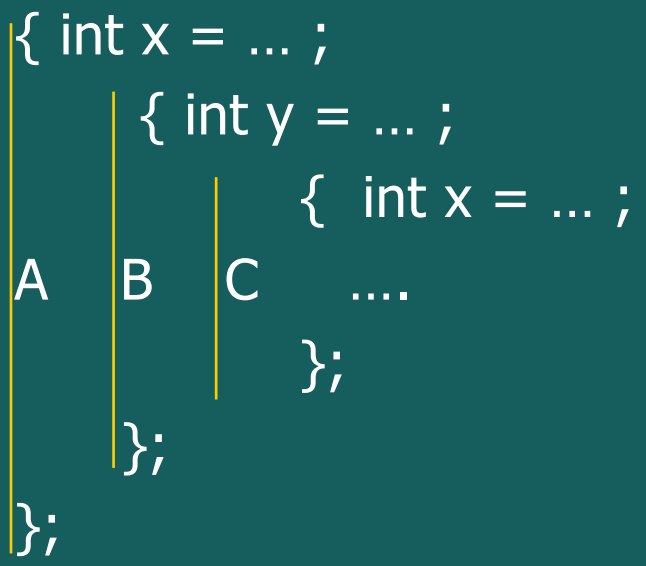
- Nested blocks, local variables
  - Example



  - Storage management
    - Enter block: allocate space for variables
    - Exits block: deallocate the space

Some basic concepts

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



- Inner declaration of x hides outer one.
- Called “hole in scope”
- Lifetime of outer x includes time when inner block is executed
- Lifetime ≠ scope
- Lines indicate “contour model” of scope.

Example

```
let
   val x = 0
   val y = x + 1
 in let val z = (x+y)*(x-y)
      in z
      end
end
```

In-line Blocks

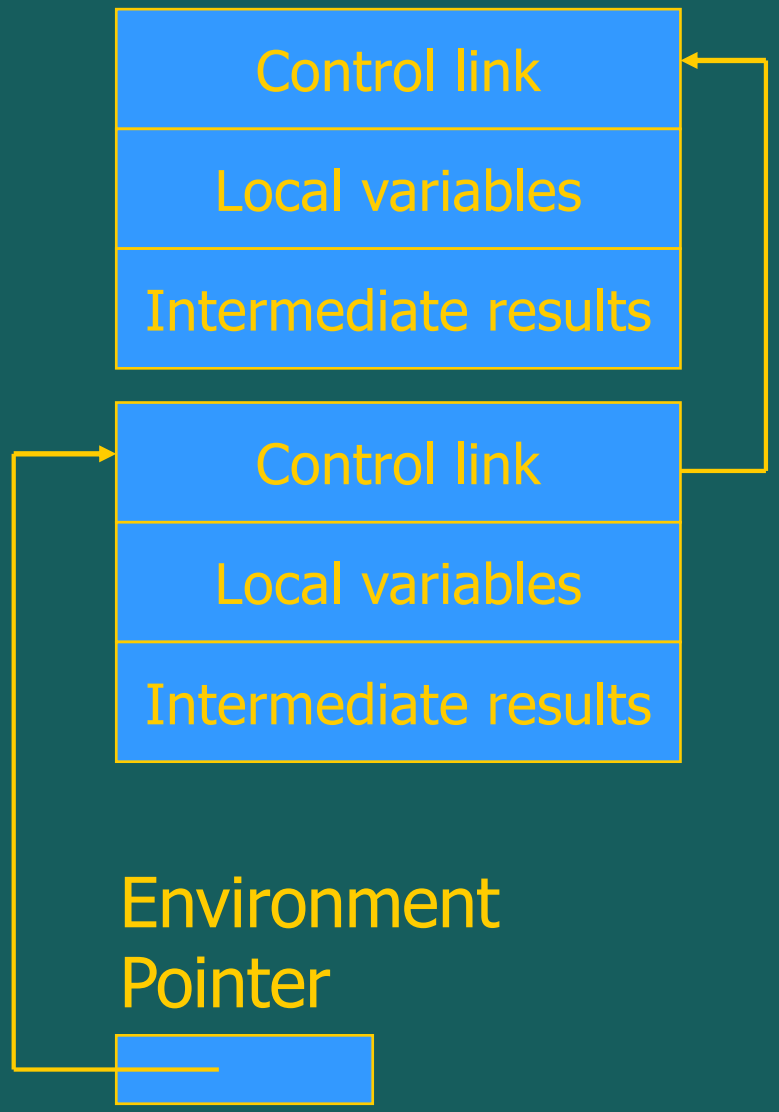
- Activation record
  - Data structure stored on run-time stack
  - Contains space for local variables
- Example

```
{ 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

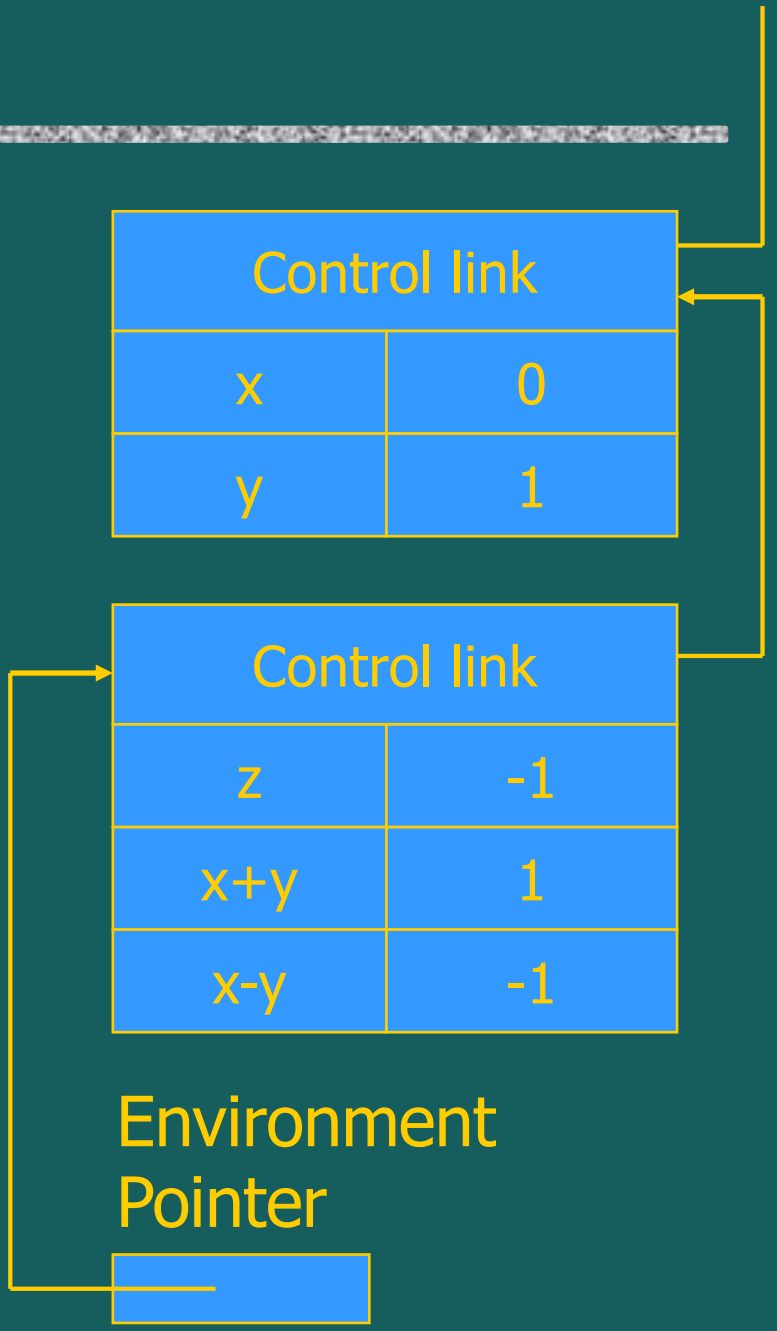


- Control link
  - pointer to previous record on stack
- Push record on stack:
  - Set new control link to point to old env ptr
  - Set env ptr to new record
- Pop record off stack
  - Follow control link of current record to reset environment pointer

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

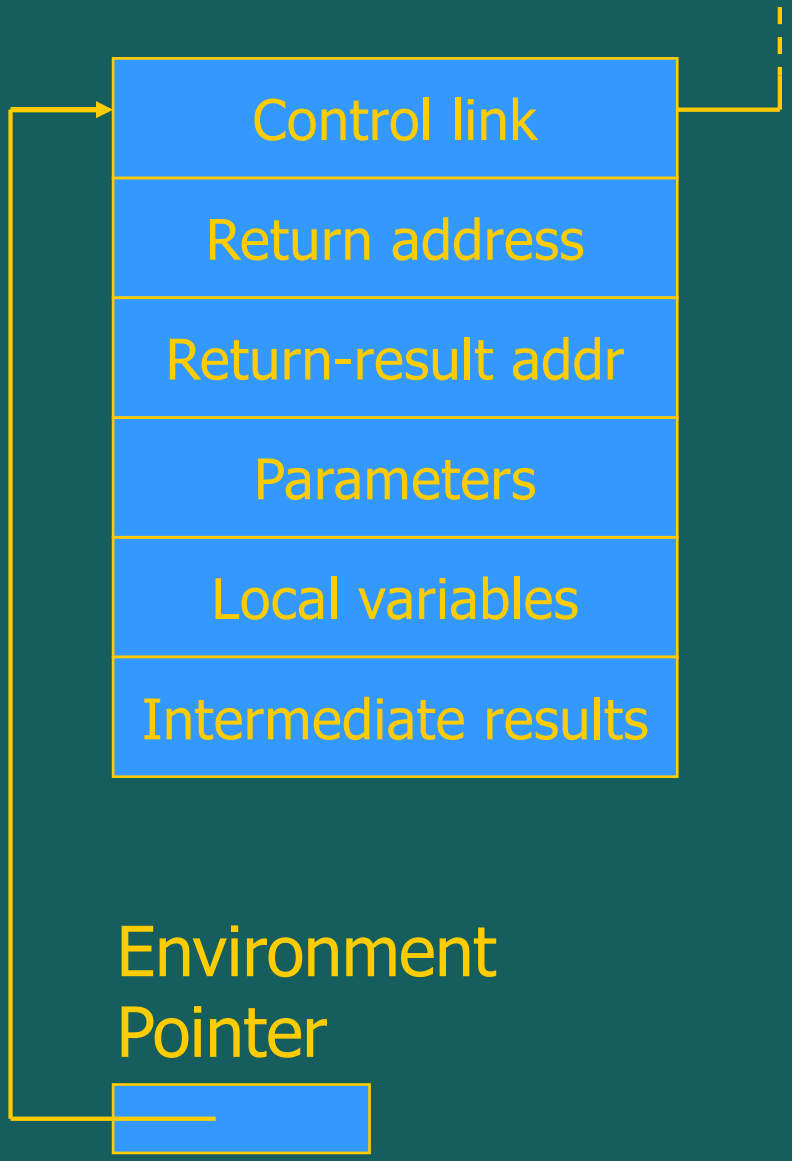
- Global and local variables
  - x, y are local to outer block
  - z is local to inner bock
  - x, y are global to inner block

```
{ 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

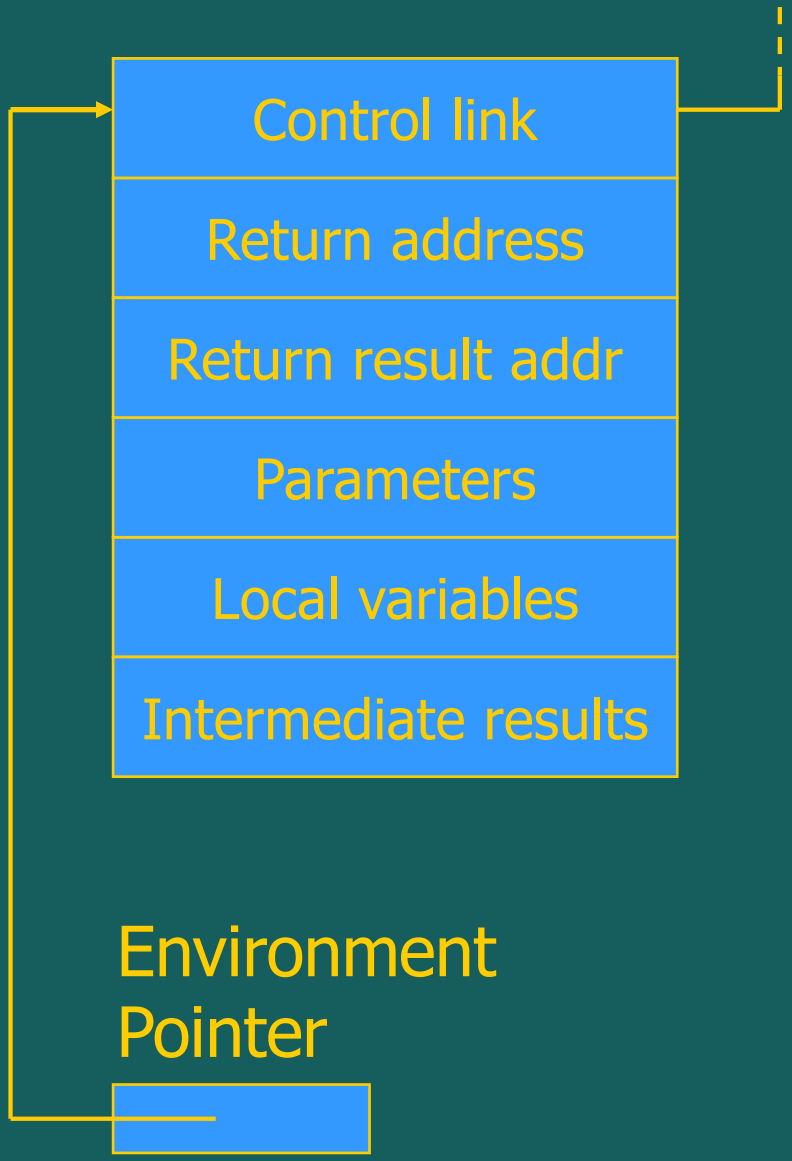
These are same until we consider function calls.

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

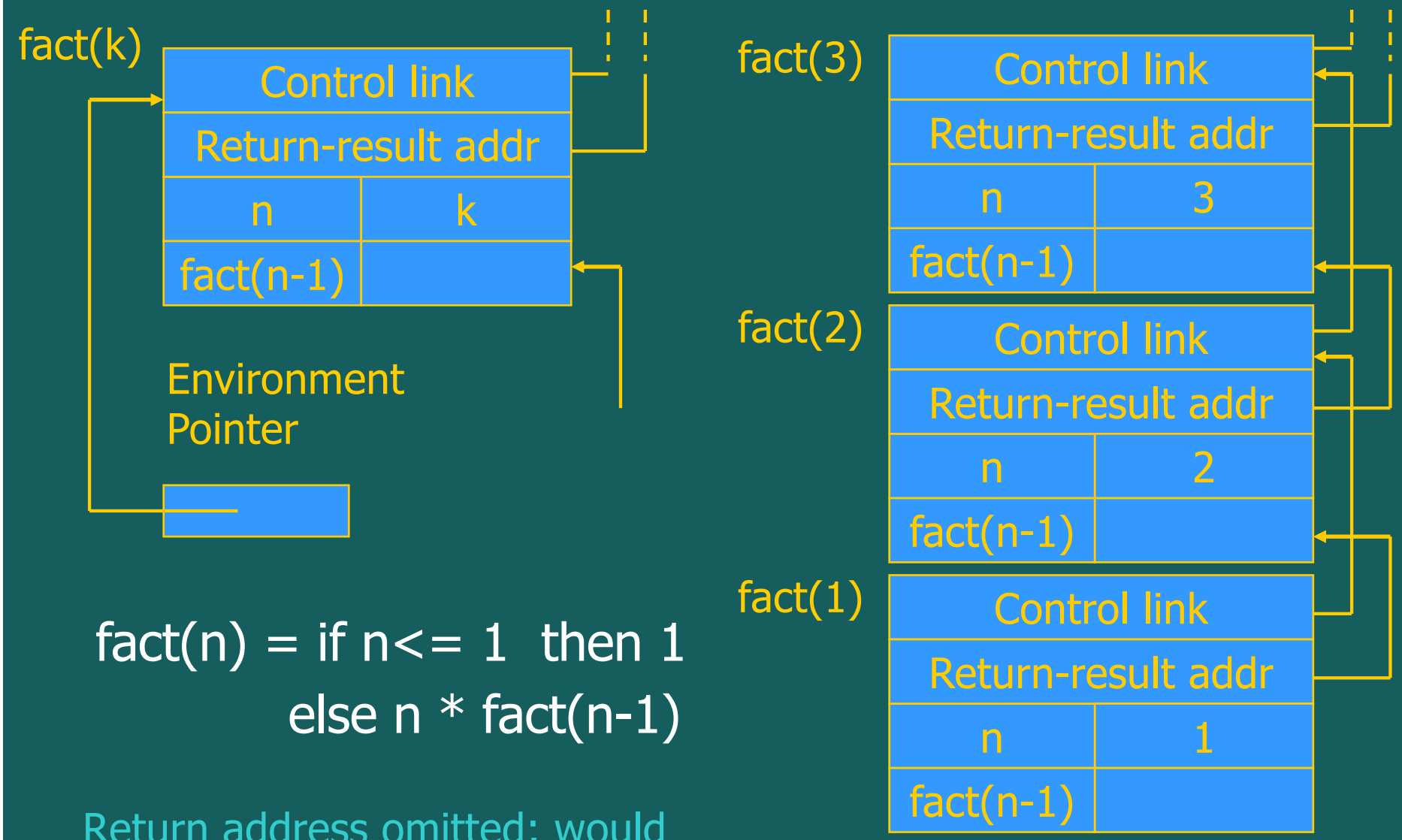
Example



- 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

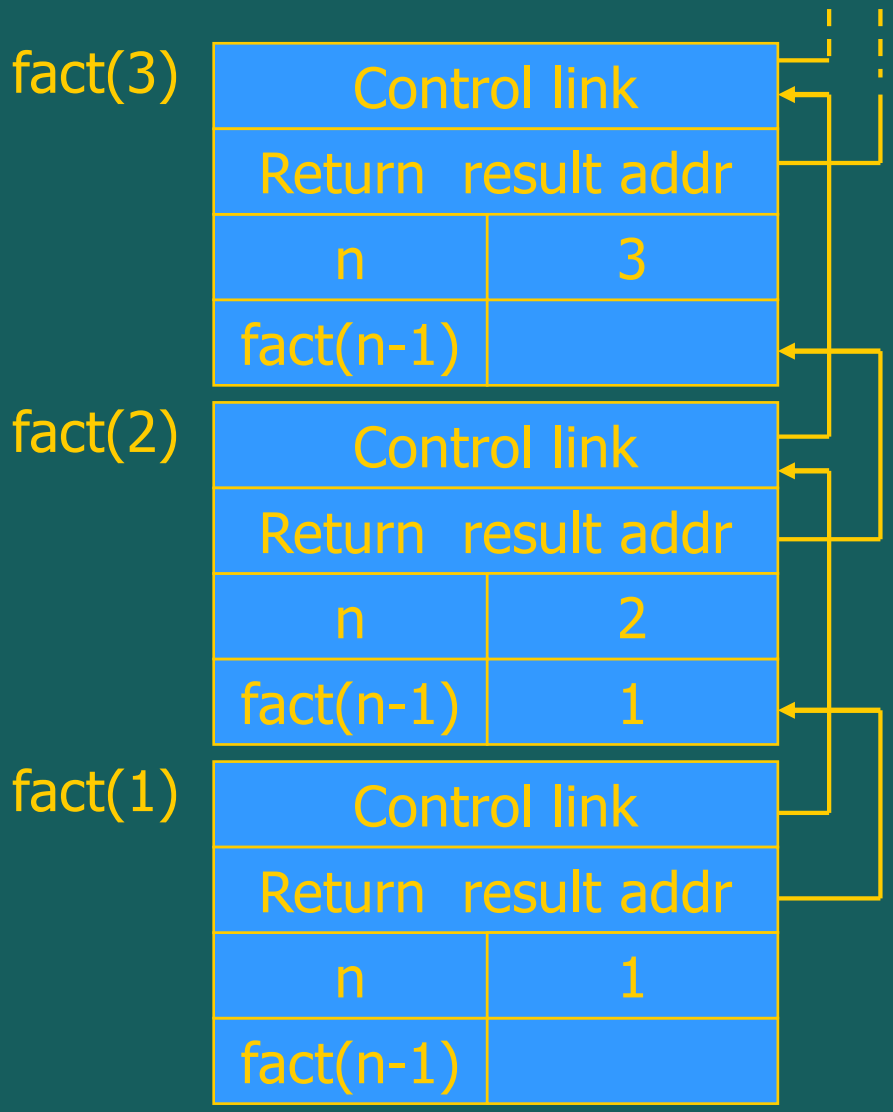


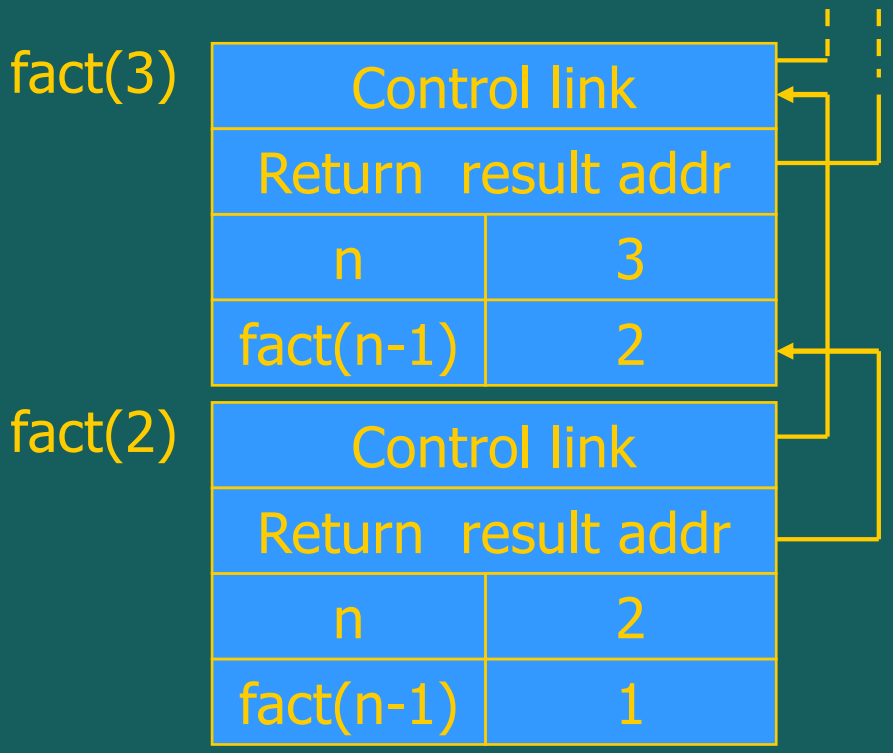
fact(n) = if n<= 1 then 1
else n * fact(n-1)

Return address omitted; would be ptr into code segment

Function return next slide →

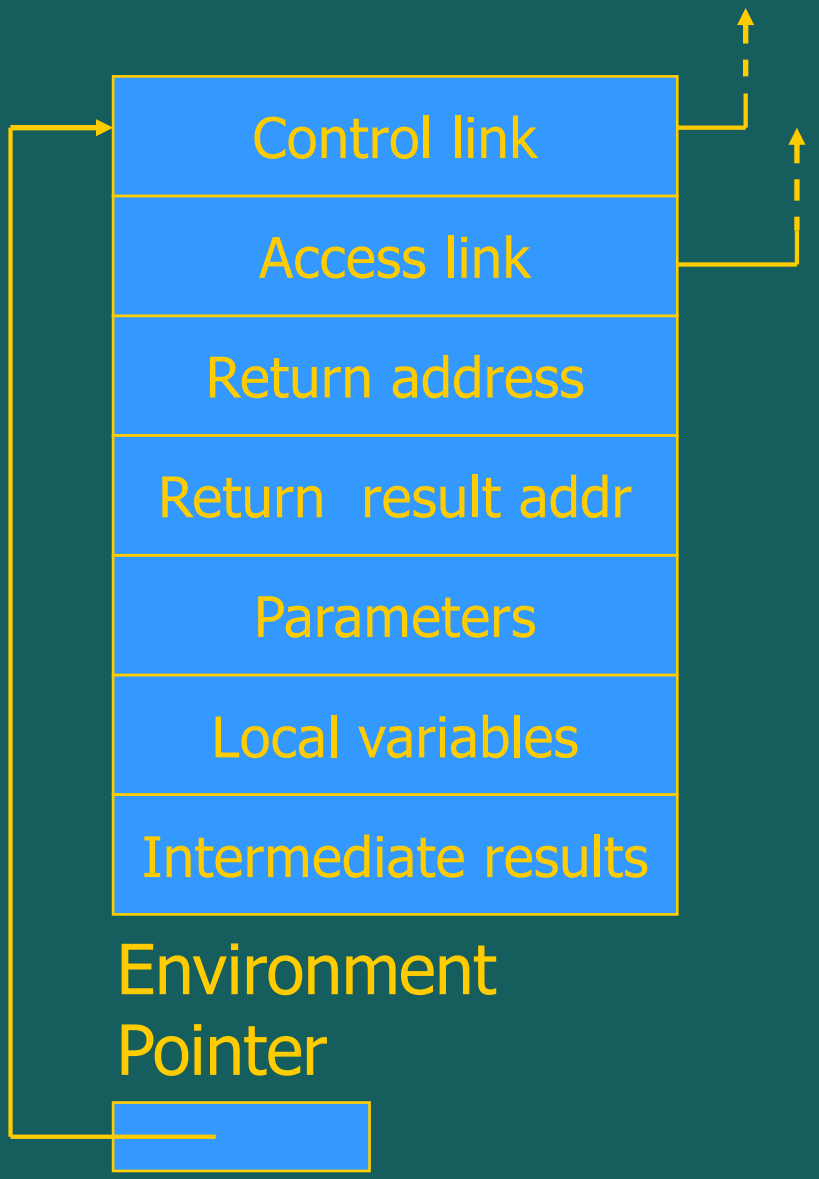
Function return





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

Activation record for static scope

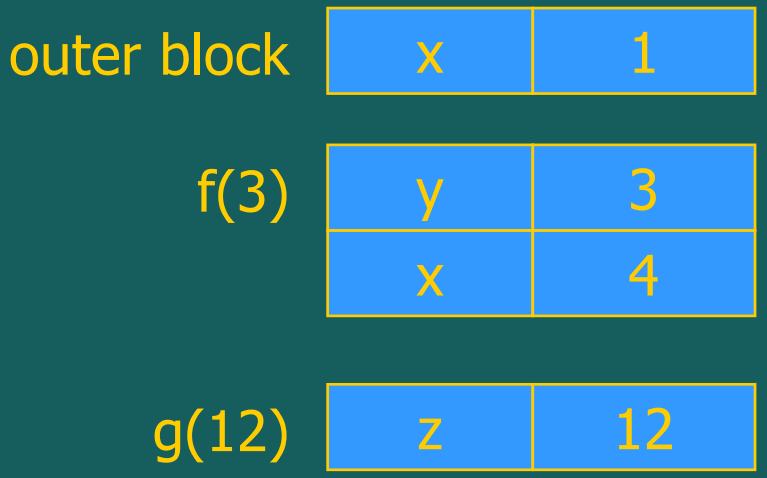


- Control link
  - Link to activation record of previous (calling) block
- Access link
  - Link to activation record of closest enclosing block in program text
- Difference
  - Control link depends on dynamic behavior of prog
  - Access link depends on static form of program text

Access to global variables

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

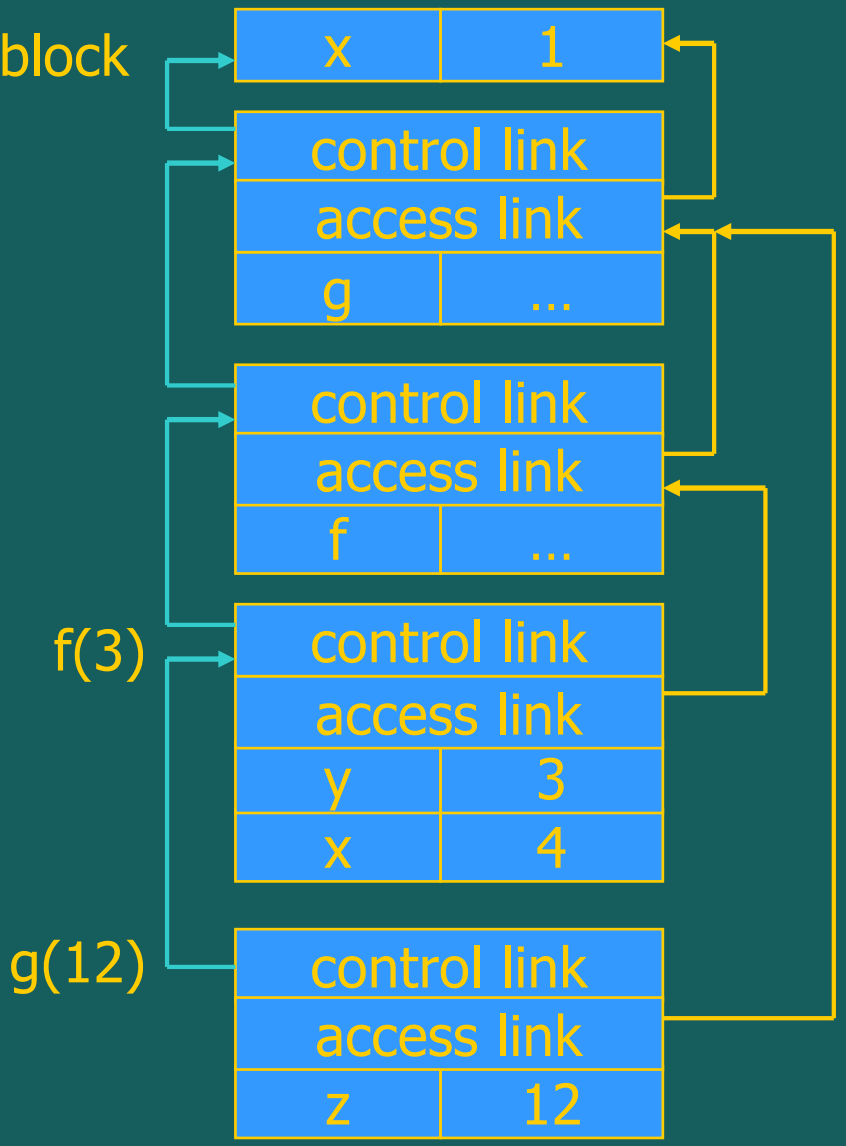
```
{let x=1;
function g(z) { return x+z; }
function f(y) {
    let  x = y+1;
    return g(y*x);
}
f(3);
```



Which x is used for expression x+z ?

Static scope with access links

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

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

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

JavaScript

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

There are two declarations of x

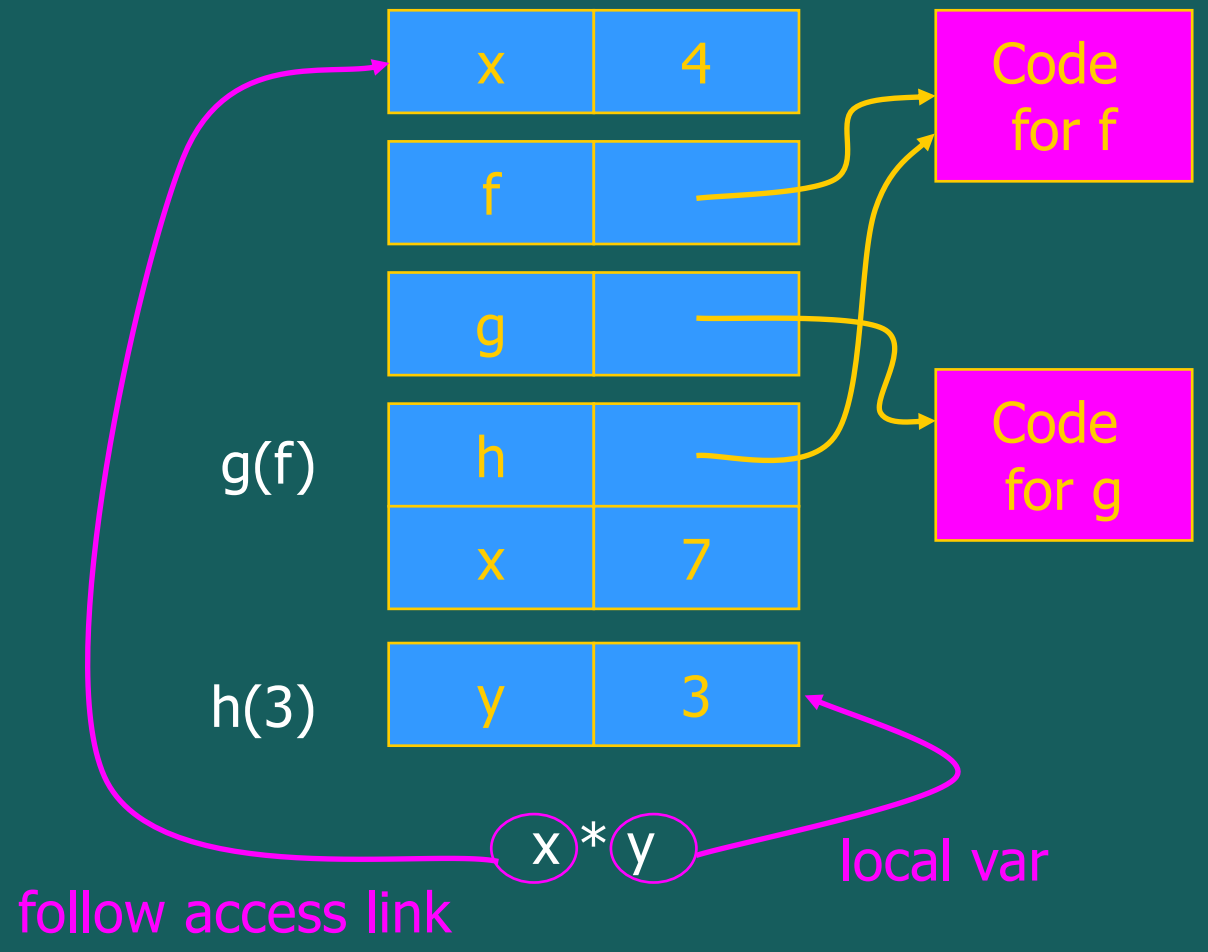
Which one is used for each occurrence of x?

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

Static Scope for Function Argument

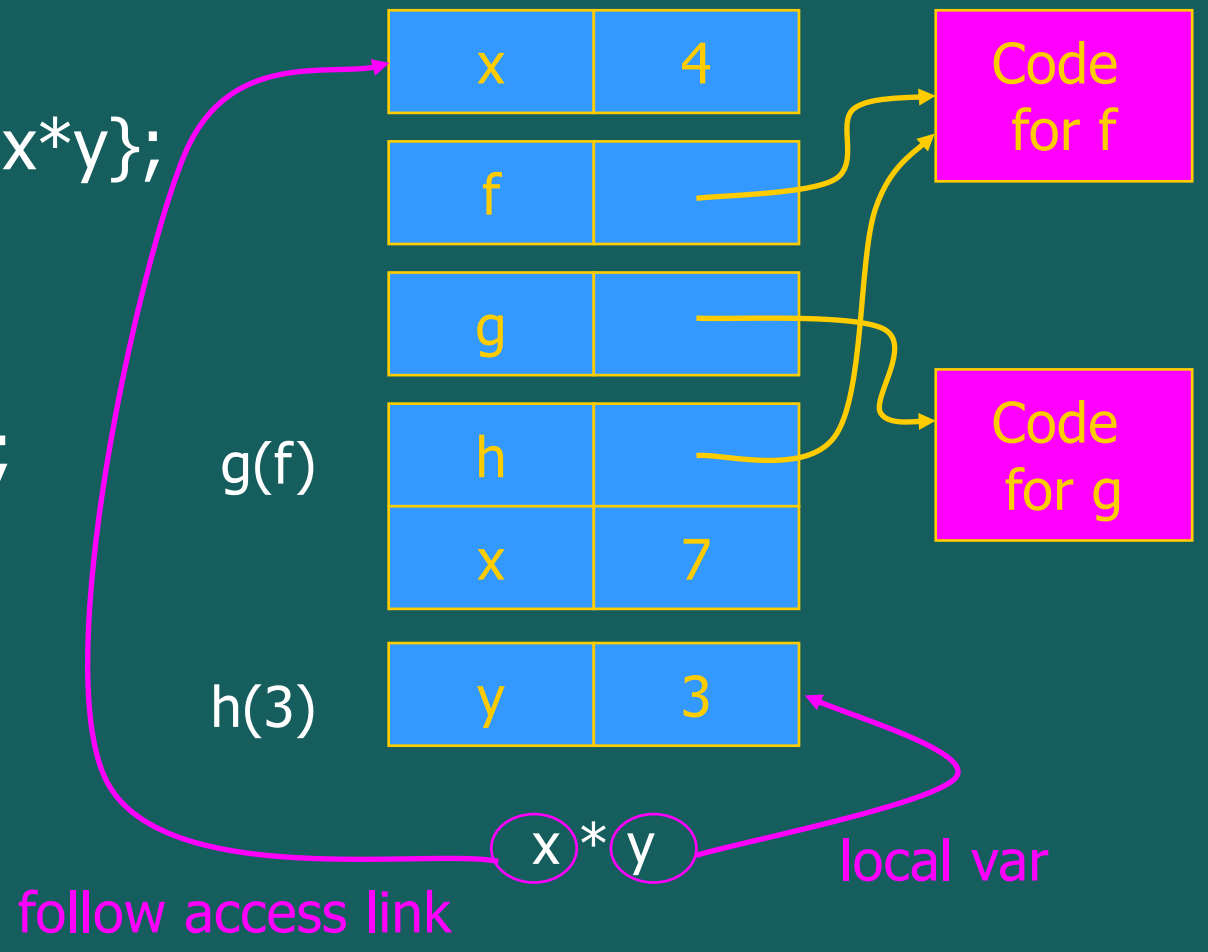
```
val x = 4;
fun f(y) = x*y;
    fun g(h) =
        let
            val x=7
        in
            h(3) + x;
        g(f);
```



How is access link for h(3) set?

Static Scope for Function Argument

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

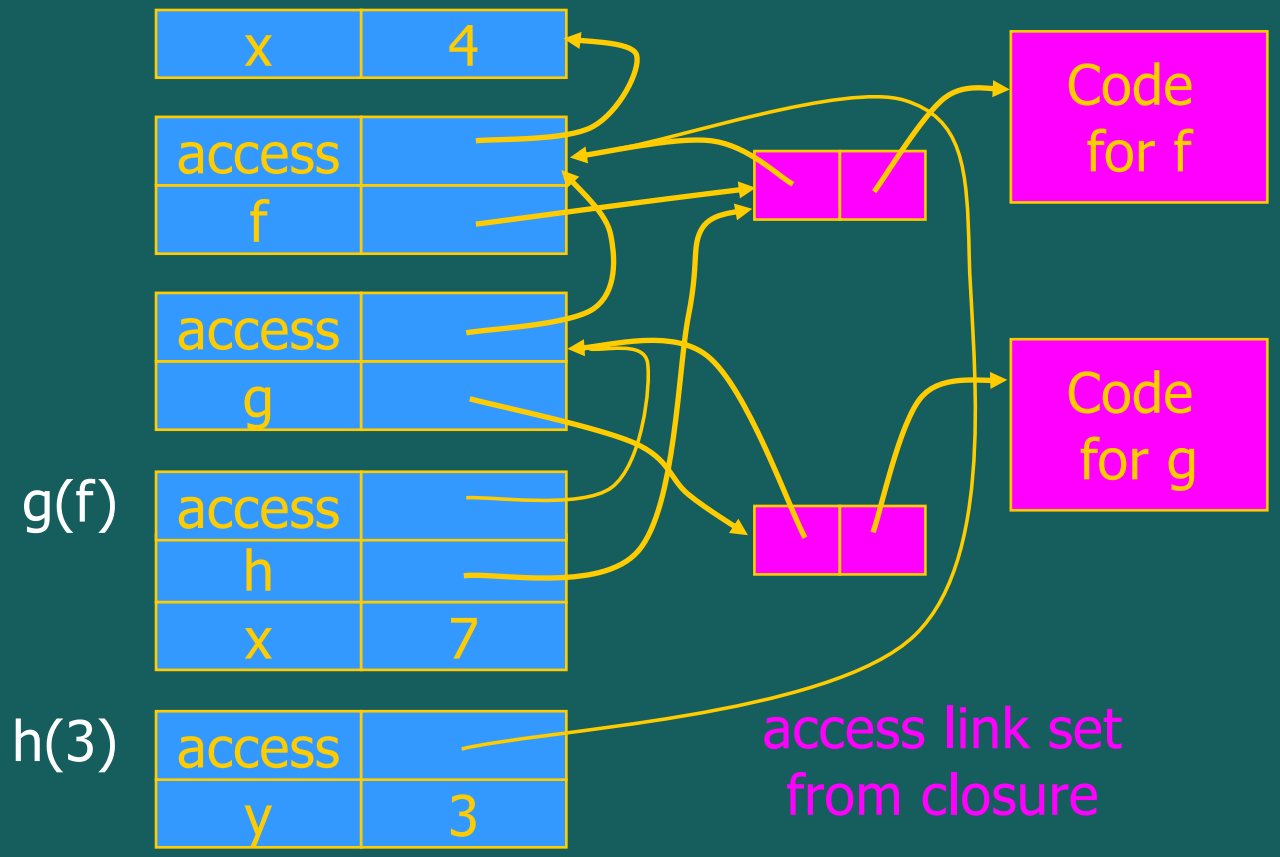


How is access link for h(3) set?

Function Argument and Closures

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

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 (later)
- 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