

PL Summary

Oded Padon

Mooly Sagiv

Concepts & Techniques

- Mathematical Induction
- Syntax
 - Context free grammar
 - Ambiguous grammars
 - Syntax vs. semantics
- Static semantics
 - Scope rules
 - Type Rules
- Semantics
 - Small vs. big step
- Runtime management
 - Activation records
 - GC
- Functional programming
 - Lambda calculus
 - Recursion
 - Higher order programming
 - Lazy vs. Eager evaluation
 - Pattern matching
 - Closure
- Types
 - Type safety
 - Static vs. dynamic
 - Type checking vs. type inference
 - Most general type
 - Polymorphism
 - Type inference algorithm

Languages

- Ocaml
- Javascript

Types of Context Free Grammars

- Linear Grammars
- *Non-Ambiguous Grammars*

Example Languages

- $a^* (a \mid b)^*$
- $a^n b^n$
- Regular expressions over letters, +, ., *, (,)
- $w c w$
- $w c w^r$

What do I have to know about grammars

- Define programming language constructs
- Ambiguity

Static Semantics

Context Analysis

Semantic Analysis

- Properties which are not enforced by the grammar
- Every used identifier is previously defined
- The type of the arguments match the type of the function prototype

Static Scope Rules

- Modern programming languages provide two ways to open new scopes
 - Inline blocks
 - Functions
- C like syntax
 - { }
- Ocaml like syntax
 - let x = 5 in x + 7

C like

```
void main {  
    int x=1;  
    int g(z) { return x+z; }  
    int f(y) {  
        int x = y+1;  
        return g(y*x);  
    }  
    f(3);  
    g(4);
```

Javascript

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

Ocaml

```
let x = 1 in
  let rec g z = x + z
  and
        f y = let x = y + 1 in g (y * x)
  in
    f 3 , g 4
```

Ocaml Emphasis

- Algebraic data types
- Higher order functions and recursion
- Closures
- Modules

Type Checking and Type Inference

- Statically typed languages permit effective compile time checking of types
 - Proves type-safety
 - Identify subtle interface mismatch
- Type inference
 - Automatically infer the most general type for a given expression

Most General Type Examples

```
fun f x =  
  let  
    val a = x * 2  
    and  
    val b = x * 3  
  in  
    [0, x, a, b, x * 4]
```

```
rec fun f x =  
  match x with  
    [] → 0  
    |  
    h :: t → 1 + f t
```

```
rec fun f x =  
  match x with  
    [] → 0  
    |  
    h :: t → 1
```

```
rec fun f x =  
  match x with  
    [] → [0]  
    |  
    h :: t → 1
```

Type Inference Algorithm

- Parse program to build parse tree
- Assign type variables to nodes in tree
- Generate constraints:
 - From environment: literals (2), built-in operators (+), known functions (tail)
 - From form of parse tree: e.g., application and abstraction nodes
- Solve constraints using *unification*
- Determine types of top-level declarations

Example

```
fun f x =  
  let  
    val a = x  
    and  
    val b = x  
  in  
    (a, [b])
```

Alternative Formal Semantics

- **Operational Semantics**
 - The meaning of the program is described “operationally”
 - Natural Operational Semantics
 - Structural Operational Semantics
- **Denotational Semantics**
 - The meaning of the program is an input/output relation
 - Mathematically challenging but complicated
- **Axiomatic Semantics**
 - The meaning of the program are observed properties

Natural Semantics for While

$$[\text{ass}_{\text{ns}}] \langle x := a, s \rangle \rightarrow s[x \mapsto \mathbf{A}[[a]]s]$$

axioms

$$[\text{skip}_{\text{ns}}] \langle \mathbf{skip}, s \rangle \rightarrow s$$

$$[\text{comp}_{\text{ns}}] \frac{\langle S_1, s \rangle \rightarrow s', \langle S_2, s' \rangle \rightarrow s''}{\langle S_1; S_2, s \rangle \rightarrow s''}$$

rules

$$[\text{if}^{\text{tt}}_{\text{ns}}] \frac{\langle S_1, s \rangle \rightarrow s'}{\langle \text{if } b \text{ then } S_1 \text{ else } S_2, s \rangle \rightarrow s'}$$

if $\mathbf{B}[[b]]s = \text{tt}$

$$[\text{if}^{\text{ff}}_{\text{ns}}] \frac{\langle S_2, s \rangle \rightarrow s'}{\langle \text{if } b \text{ then } S_1 \text{ else } S_2, s \rangle \rightarrow s'}$$

if $\mathbf{B}[[b]]s = \text{ff}$

Natural Semantics for While (More rules)

$$\frac{[\text{while}_{ns}^{\text{ff}}] \quad \langle \text{while } b \text{ do } S, s \rangle \rightarrow s}{\text{if } \mathbf{B}[[b]]s = \text{ff}}$$

$$\frac{[\text{while}_{ns}^{\text{tt}}] \quad \langle S, s \rangle \rightarrow s', \langle \text{while } b \text{ do } S, s' \rangle \rightarrow s''}{\langle \text{while } b \text{ do } S, s \rangle \rightarrow s''} \quad \text{if } \mathbf{B}[[b]]s = \text{tt}$$

Structural Semantics for While

$$[\text{ass}_{\text{sos}}] \langle x := a, s \rangle \Rightarrow s[x \mapsto \mathbf{A}[[a]]s]$$

axioms

$$[\text{skip}_{\text{sos}}] \langle \mathbf{skip}, s \rangle \Rightarrow s$$

$$[\text{comp}^1_{\text{sos}}] \langle S_1, s \rangle \Rightarrow \langle S'_1, s' \rangle$$

rules

$$\langle S_1; S_2, s \rangle \Rightarrow \langle S'_1; S_2, s' \rangle$$

$$[\text{comp}^2_{\text{sos}}] \langle S_1, s \rangle \Rightarrow s'$$

$$\langle S_1; S_2, s \rangle \Rightarrow \langle S_2, s' \rangle$$

Structural Semantics for While if construct

$[if_{sos}^{tt}] \langle \text{if } b \text{ then } S_1 \text{ else } S_2, s \rangle \Rightarrow \langle S_1, s \rangle$ if $\mathbf{B}[[b]]s = tt$

$[if_{os}^{ff}] \langle \text{if } b \text{ then } S_1 \text{ else } S_2, s \rangle \Rightarrow \langle S_2, s \rangle$ if $\mathbf{B}[[b]]s = ff$

Structural Semantics for While while construct

$[\text{while}_{\text{sos}}]$ $\langle \text{while } b \text{ do } S, s \rangle \Rightarrow$
 $\langle \text{if } b \text{ then } (S; \text{while } b \text{ do } S) \text{ else skip}, s \rangle$

Usages of semantics

- Prove properties of programs
 - Type safety = No undefined behavior
 - Equivalence
 - ...
- Define new constructs
- Generic interpreters

Example Question

- while b odd do S1 even do S2

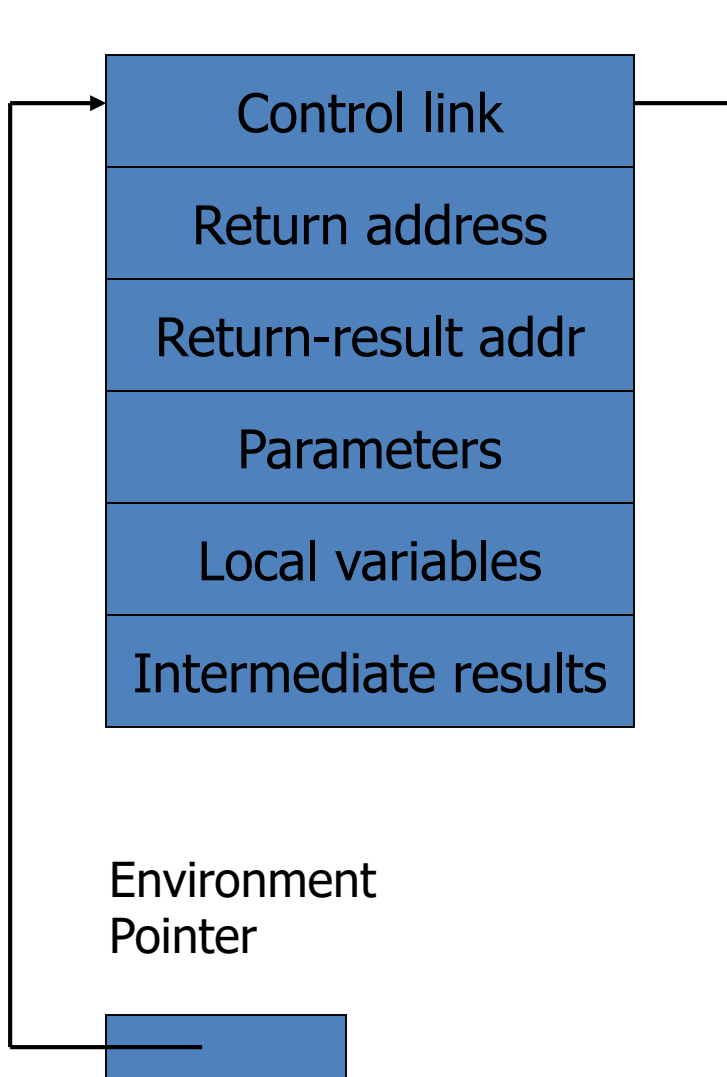
Semantic Equivalence

- Natural Operational Semantics
 - S_1 and S_2 are **semantically equivalent** ($S_1 \approx S_2$) if for all s and s'
 - $\langle S_1, s \rangle \rightarrow s'$ if and only if $\langle S_2, s \rangle \rightarrow s'$
- Structural Operational Semantics
 - S_1 and S_2 are **semantically equivalent** ($S_1 \approx S_2$) if:
 - for all s and γ which is either final or stuck $\langle S_1, s \rangle \Rightarrow^* \gamma$ if and only if $\langle S_2, s \rangle \Rightarrow^* \gamma$
 - there is an infinite derivation sequence starting at $\langle S_1, s \rangle$ if and only if there is an infinite derivation sequence starting at $\langle S_2, s \rangle$

Runtime Memory Management

Activation Records and Garbage
Collection

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

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)
```

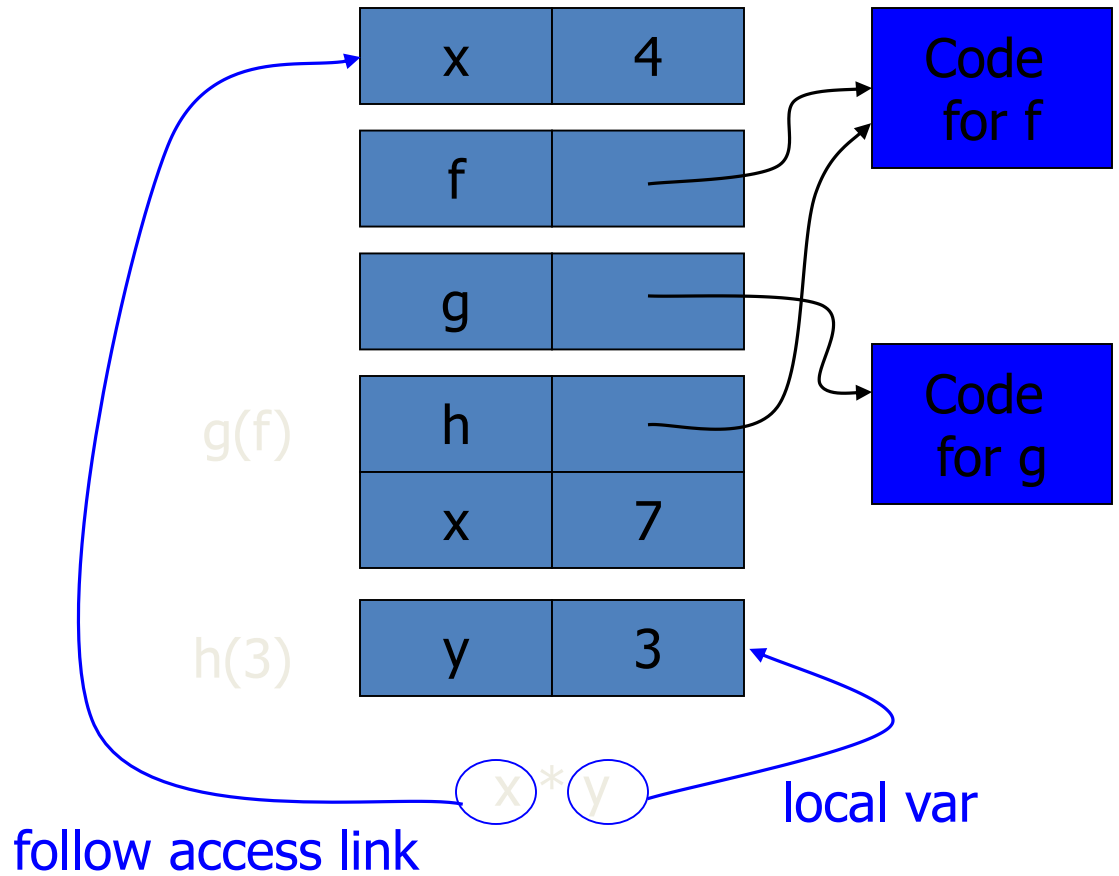
Pseudo-JavaScript

```
{ 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

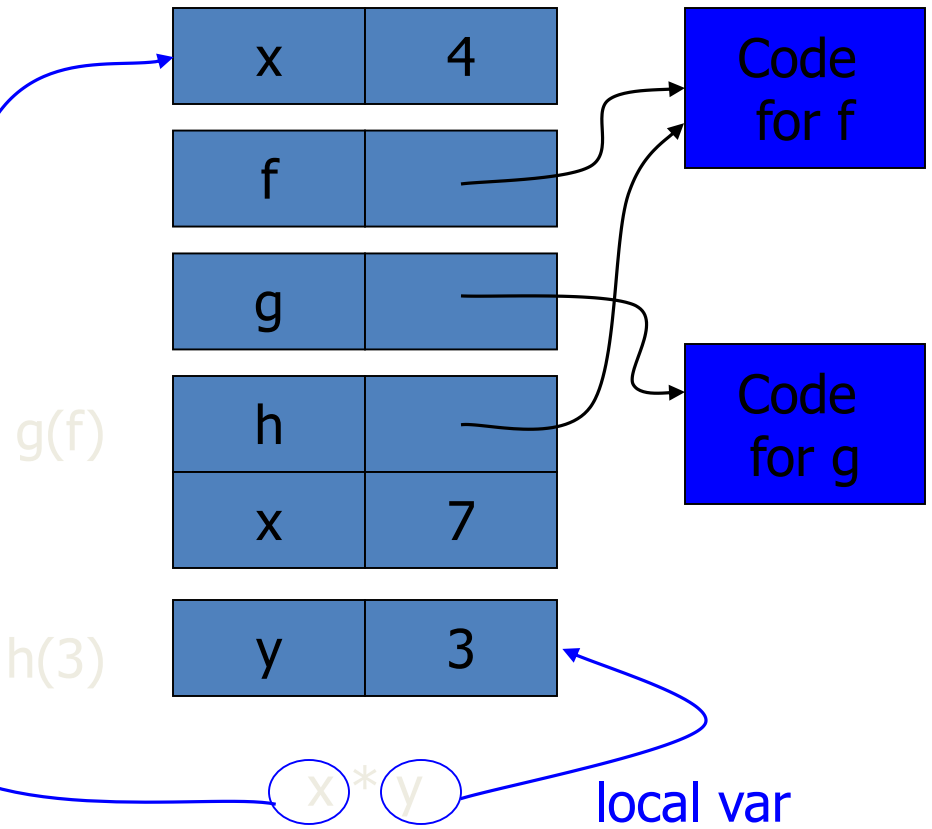
```
let x = 4 in
  let f = fun -> x*y in
    let g = fun h ->
      let
        int x=7
      in
        h(3) + x
    in
      g(f)
```



How is access link for h(3) set?

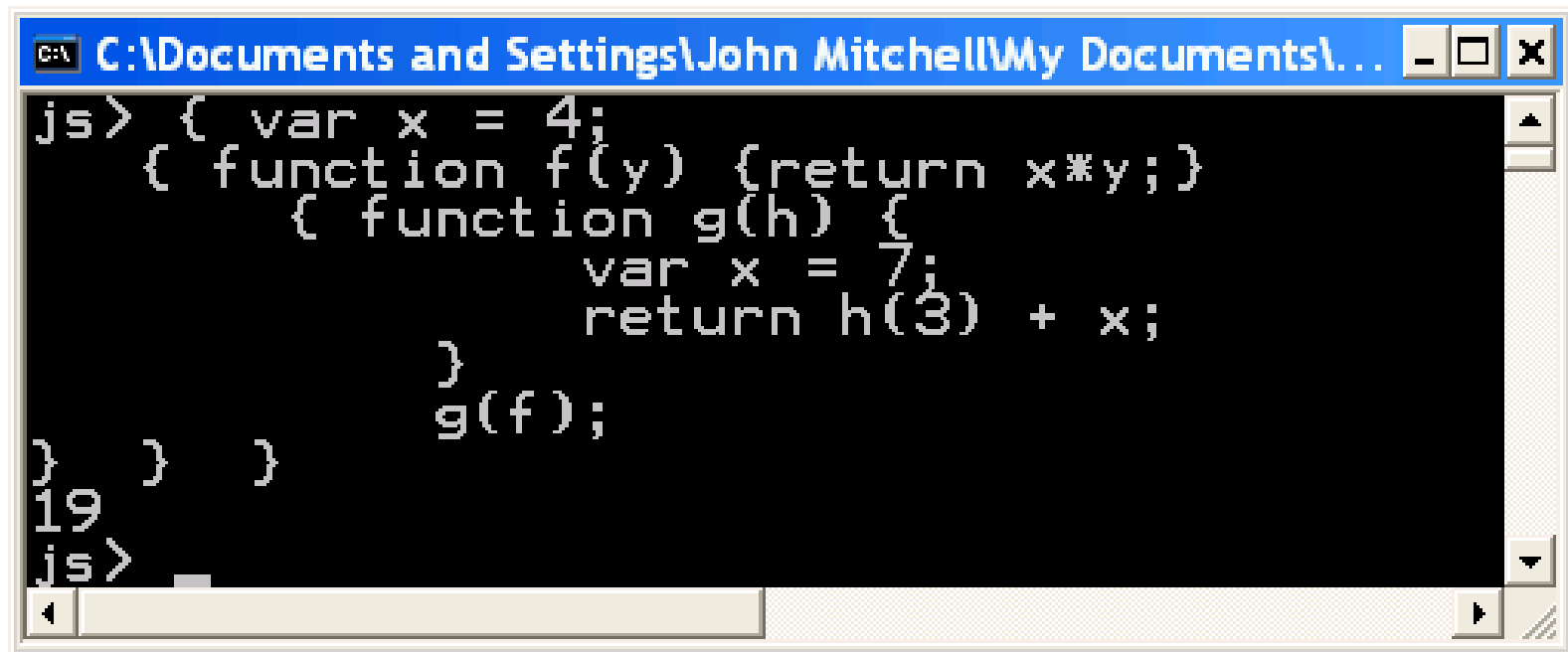
Static Scope for Function Argument

```
{ 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



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

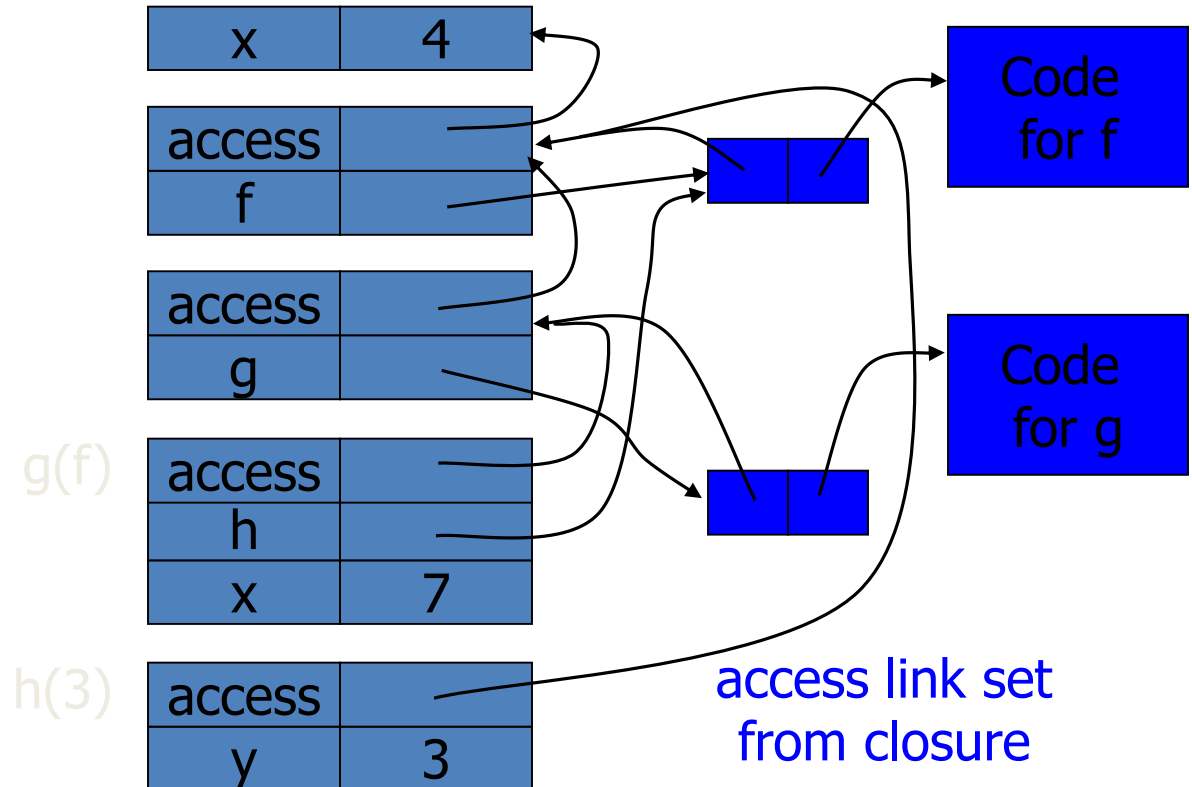
Closures

- Function value is pair *closure* = $\langle env, code \rangle$
- 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

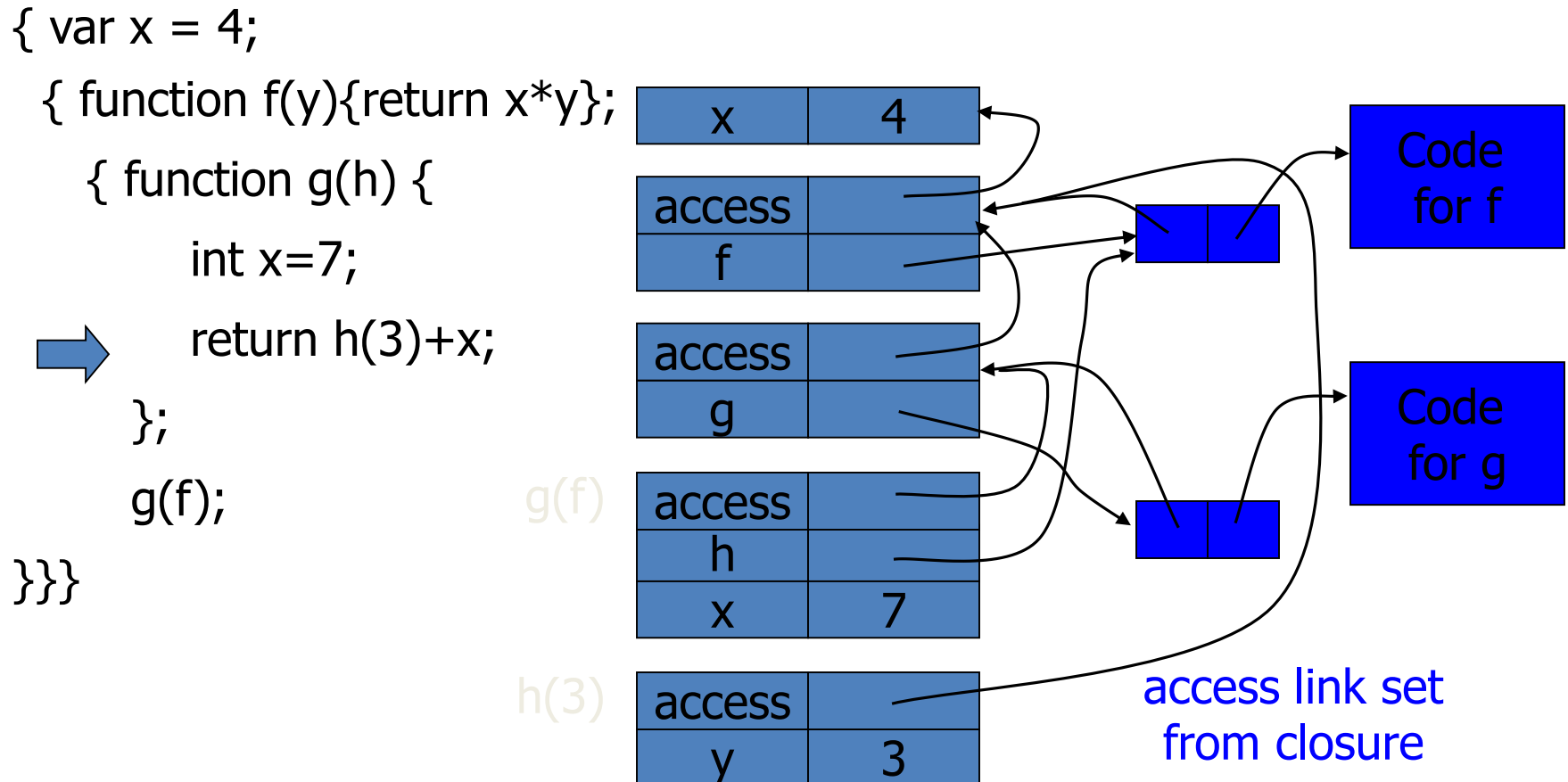
Run-time stack with access links

```
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

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 = $\langle \text{env}, \text{code} \rangle$
 - 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

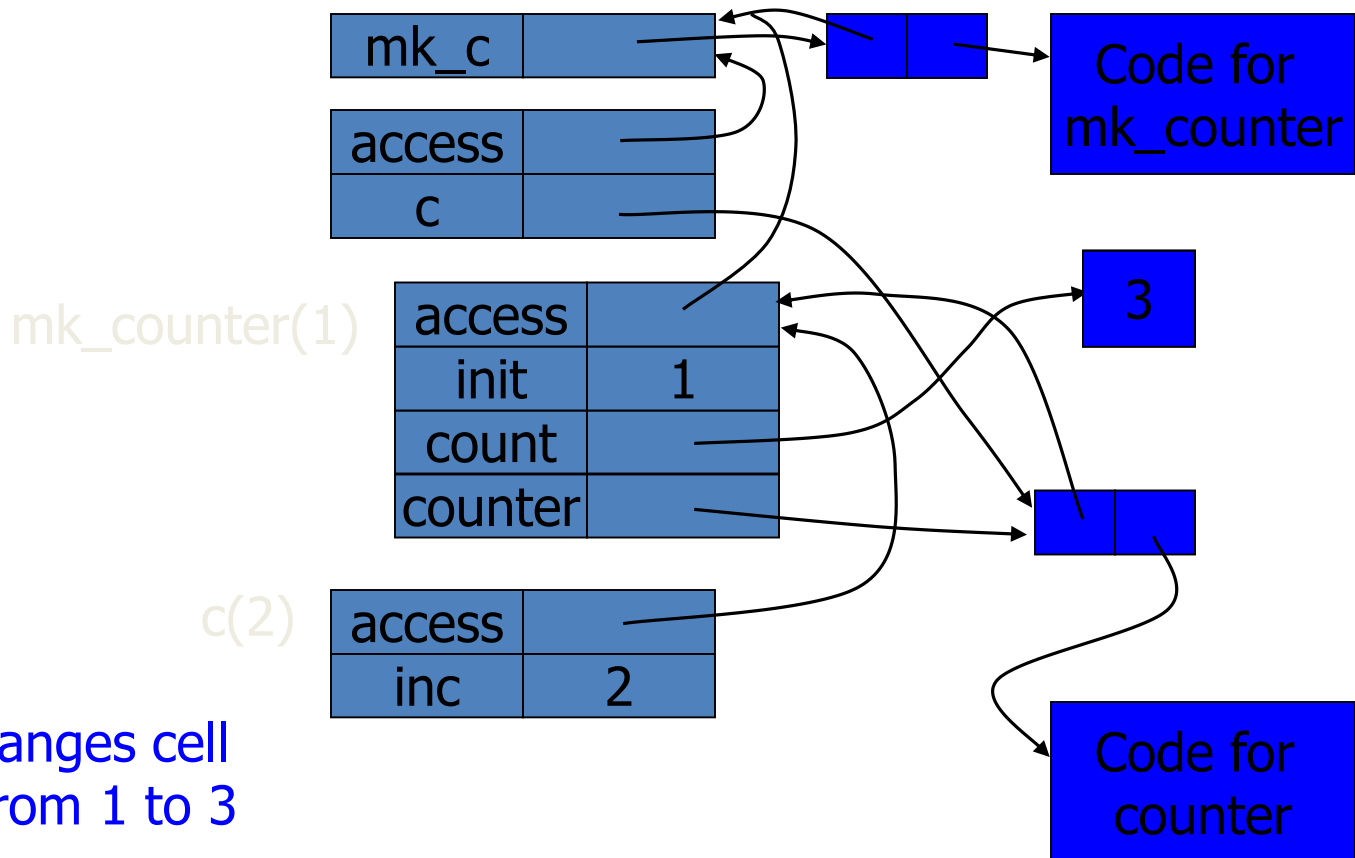
```
function mk_counter (init) {  
    var count = init;  
    function counter(inc) {count=count+inc; return count};  
    return counter};  
var c = mk_counter(1);  
c(2) + c(2);
```

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)
```

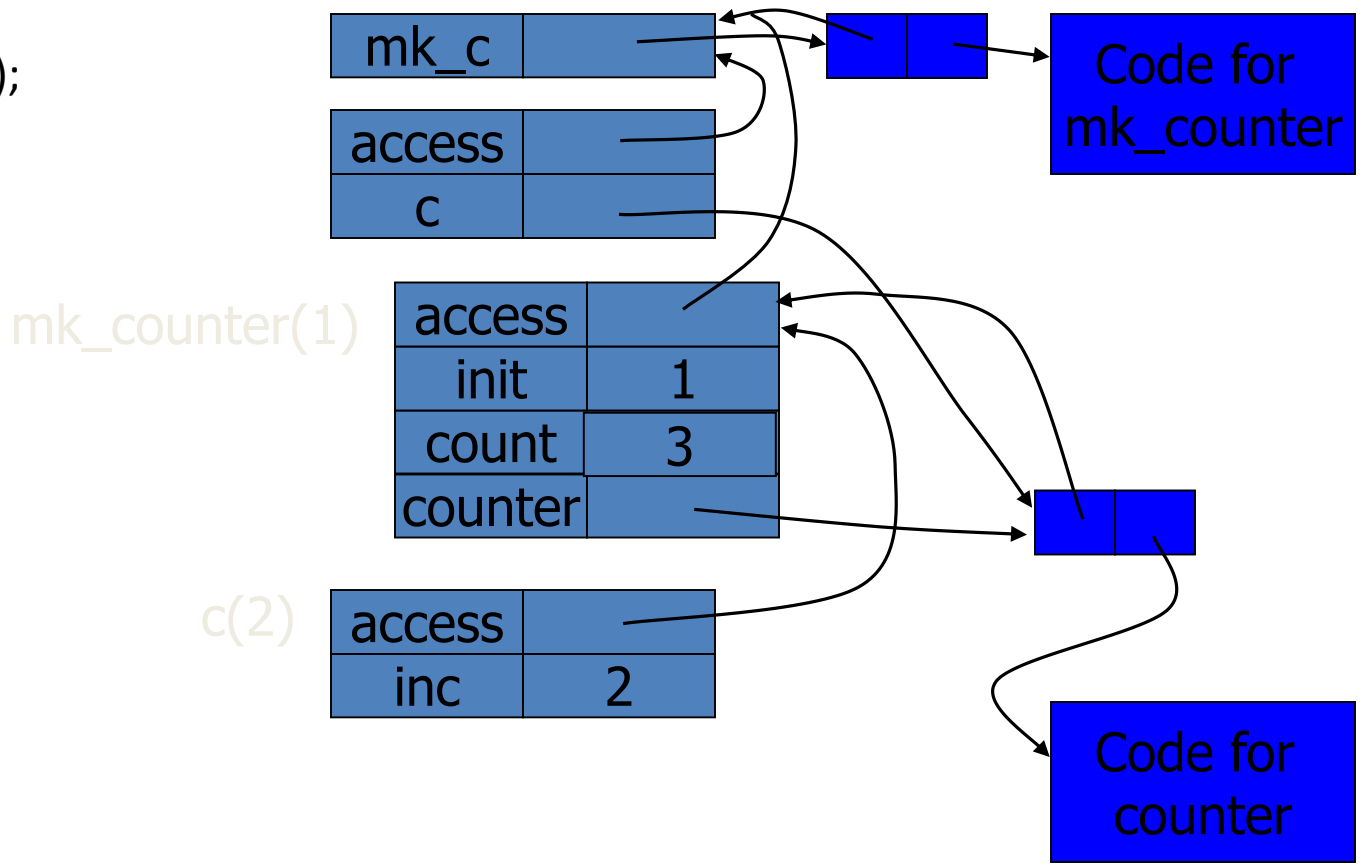


Call changes cell value from 1 to 3

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);
```



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

Recommendation

- Read material from recitations and lectures
- Solve exams from 2015,16,17 both Moed A and B
 - No parsing
- Relax