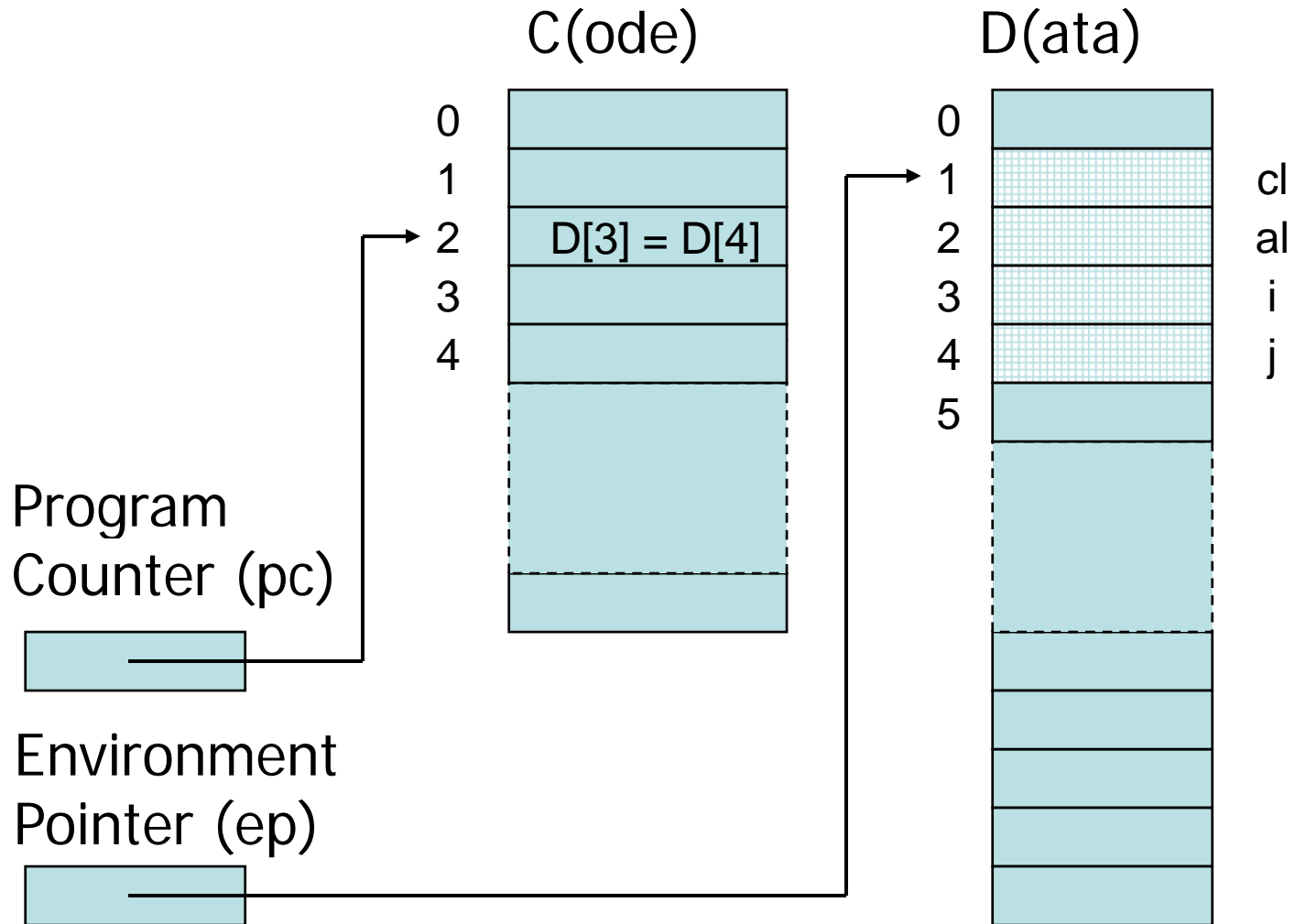


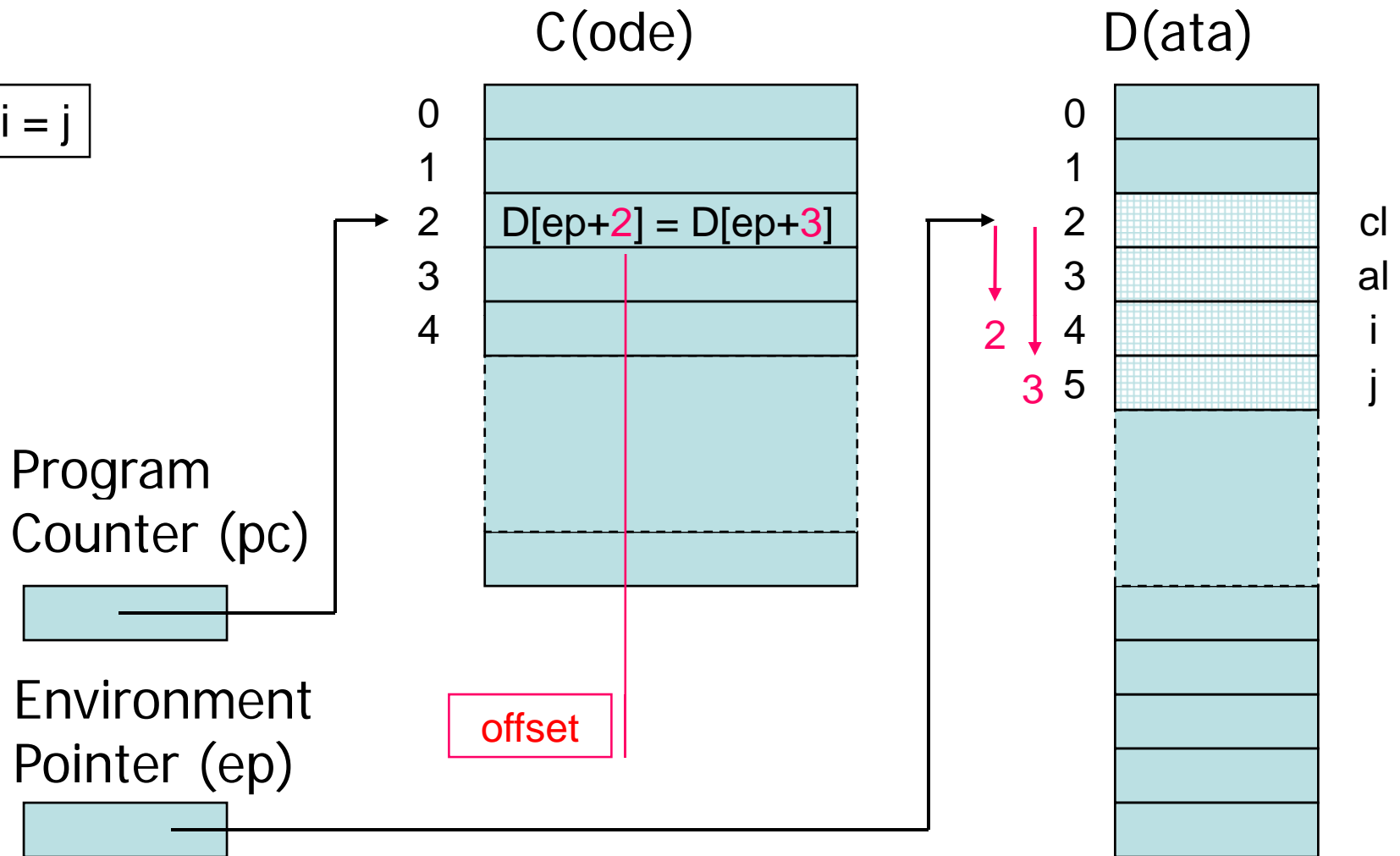
# Details of virtual machine operation

$i = j$



# Offsets

$i = j$

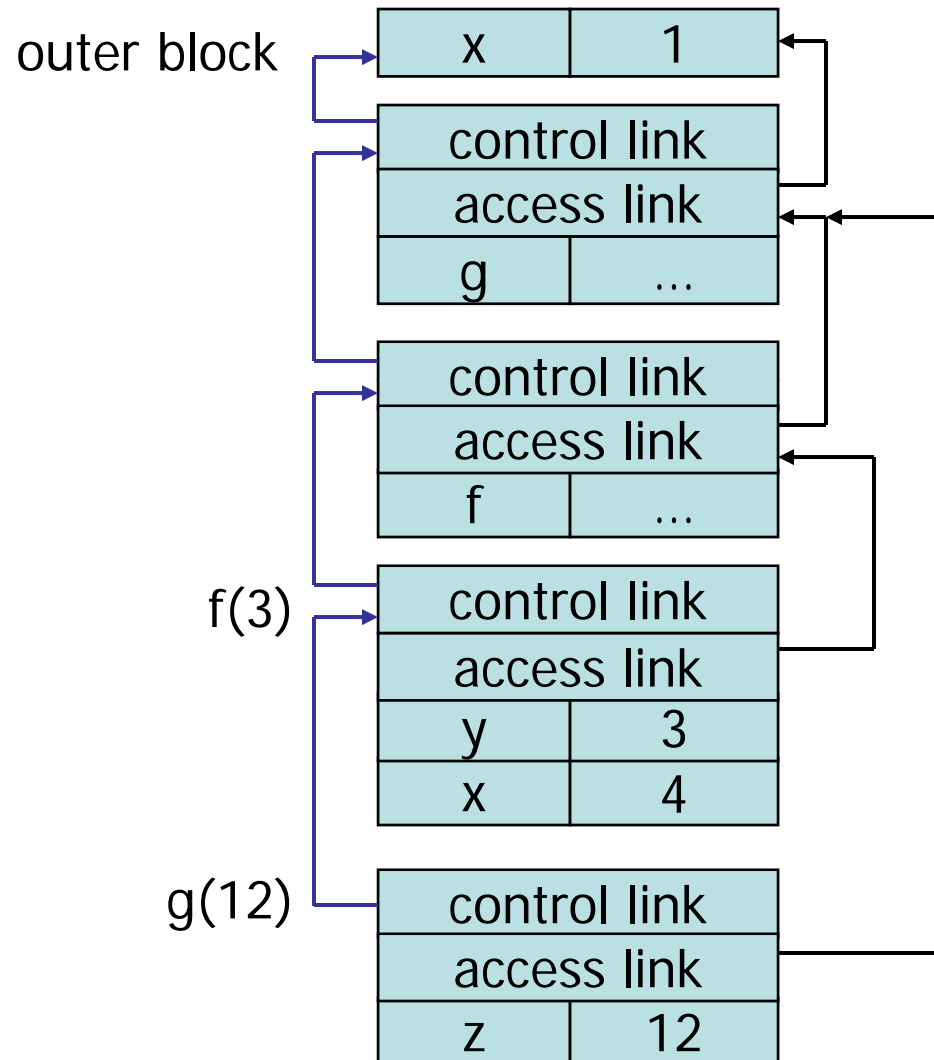


# Static scope with access links

```
int x=1;
function g(z) = x+z;
function f(y) =
  { int 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



# Access link

- A variable will always be reached in a certain distance from the actual block
  - A local variable has distance = 0.
  - A local variable declared in immediate enclosing block has distance = 1.
  - ...
- How to find variables
  - Each application of a variable is turned into a pair <distance, offset >
  - Follow access link distance times, and then add offset.

# Block Level (BL) and Context Vector (CV)

Blocks

```
{ int x=1 ...
  { ...
    { ...;
      X; ...
    }
  }
  { ...
    { ...;
      X;
    }
  }
}
```

BL

1  
2  
3  
  
2  
3

```
void makeContextVector() {
  int i;
  CV[ep.BL] = ep;
  for (i = ep.BL; i >= 2; i--) {
    CV[i-1] = CV[i].AL;
  }
}
```

where AL is the Access Link.

CV[1] always denotes the main program block, and is thereby constant.

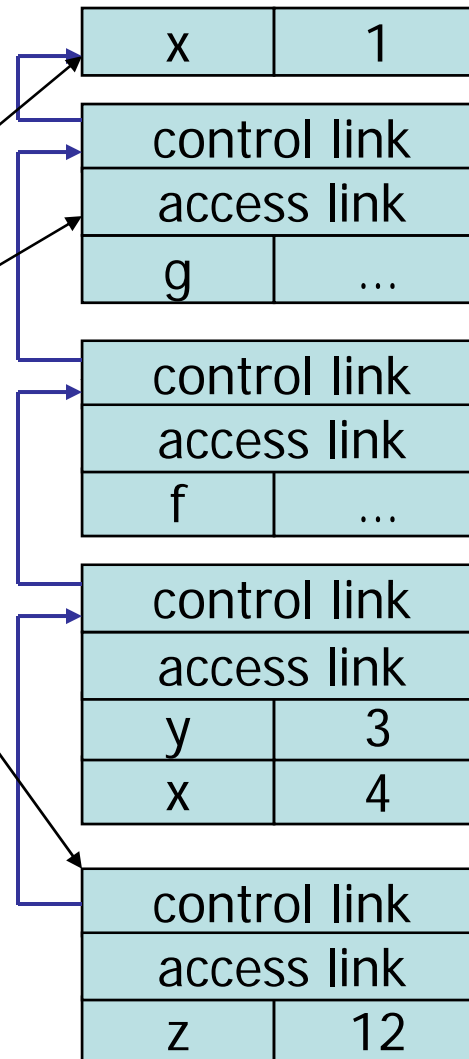
# Context vector

- A vector pointing to the activation records that are currently visible

```

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

1	
2	
3	



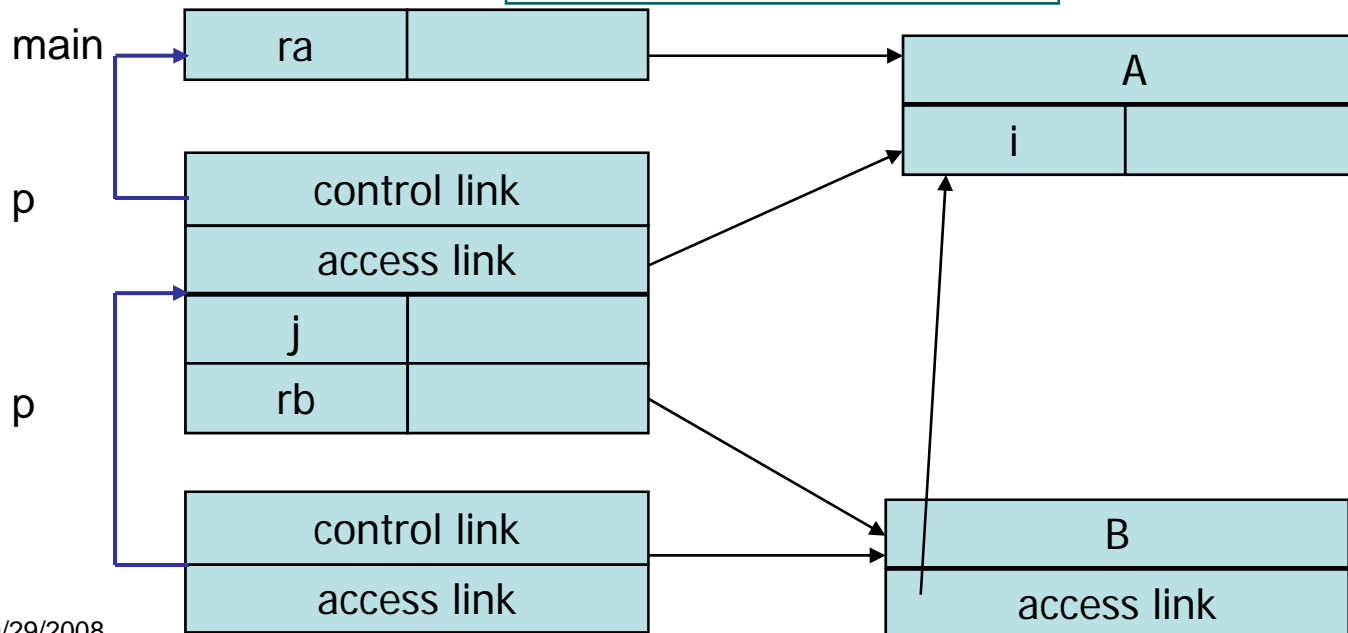
- + variable access faster
- - More work at entry/exit

# Access links for object oriented languages

```
class Program {
  public static void
  main(String[] args) {
    A ra = new A();
    ra.p();
  };
};
```

```
class A {
  int i = 1;
  class B {
    void p(){ i = 2; };
  };
  void p() {
    B rb = new B();
    int j=i;
    rb.p();
  };
};
```

- Access links for method activations may be objects
- Objects may also have access links (classes in classes, or classes in methods)



# By-name parameters

```
begin integer i;
  integer procedure sum(i, j);
    integer i, j;
  begin
    integer sm; sm:= 0;
    for i = 1 step 1 until 100 do
      sm := sm + j;
    sum:= sm
  end;
  print(sum(i,i*10))
end;
```

```
swap(int a, b) {
  int temp;
  temp = a;
  a = b;
  b = temp;
};
```

```
i=3;
a[3]=6;
swap(i, a[i]);
```

```
-- i = 6
```

```
-- a[3] = 6
-- a[6] = 3
```

```
temp = i;
i = a[i];
a[i] = temp;
```



# by name

“4.7.3.2. **Name replacement (call by name)**. Any formal parameter not quoted in the value list is replaced, throughout the procedure body, by the corresponding actual parameter, after enclosing this latter in parentheses wherever syntactically possible. Possible conflicts between identifiers inserted through this process and other identifiers already present within the procedure body will be avoided by suitable systematic changes of the formal or local identifiers involved. “

What is the difference between this and macro expansion?

```
int i; int a[];
swap(int a, b) {
  int i;
  i = a;
  a = b;
  b = i;
};

swap(i, a[i]);
```

```
i=3;
a[3]=6;
swap(i, a[i]);
```

## By name

```
-- i = 6
-- a[3] = 6
-- a[6] = 3
```

## By macro expansion

```
i = i;      -- local i = 0
i = a[i];   -- local i = a[0]
a[i] = i;   -- a[0] = 0
```

# Correspondence Principle

<pre>... int i; {   i = j;   print(i); } ...</pre>	<pre>... void p(int i) {   print(i); }; p(j); ...</pre>
--	---

- To each kind of parameter there is a corresponding kind of declaration
  - by value and variables (above)
  - by reference ?
  - by name?
  - Functions?

# 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

# Why functions as parameters?

```
procedure fsum(f, a, l, u);  
  value l, u; integer array a;  
  integer procedure f;  
begin  
  integer sum:= 0;  
  for i:= l step 1 until u do sum:= sum + f(a[i]);  
  fsum:= sum  
end
```

```
integer array aa[5:100];  
integer procedure ip1(i); value i; integer i; begin ... end;  
integer procedure ip2(i); value i; integer i; begin ... end;
```

```
fsum(ip1, aa, 5, 100)  
fsum(ip2, aa, 5, 100)
```

# Pass function as argument

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

There are two declarations of x

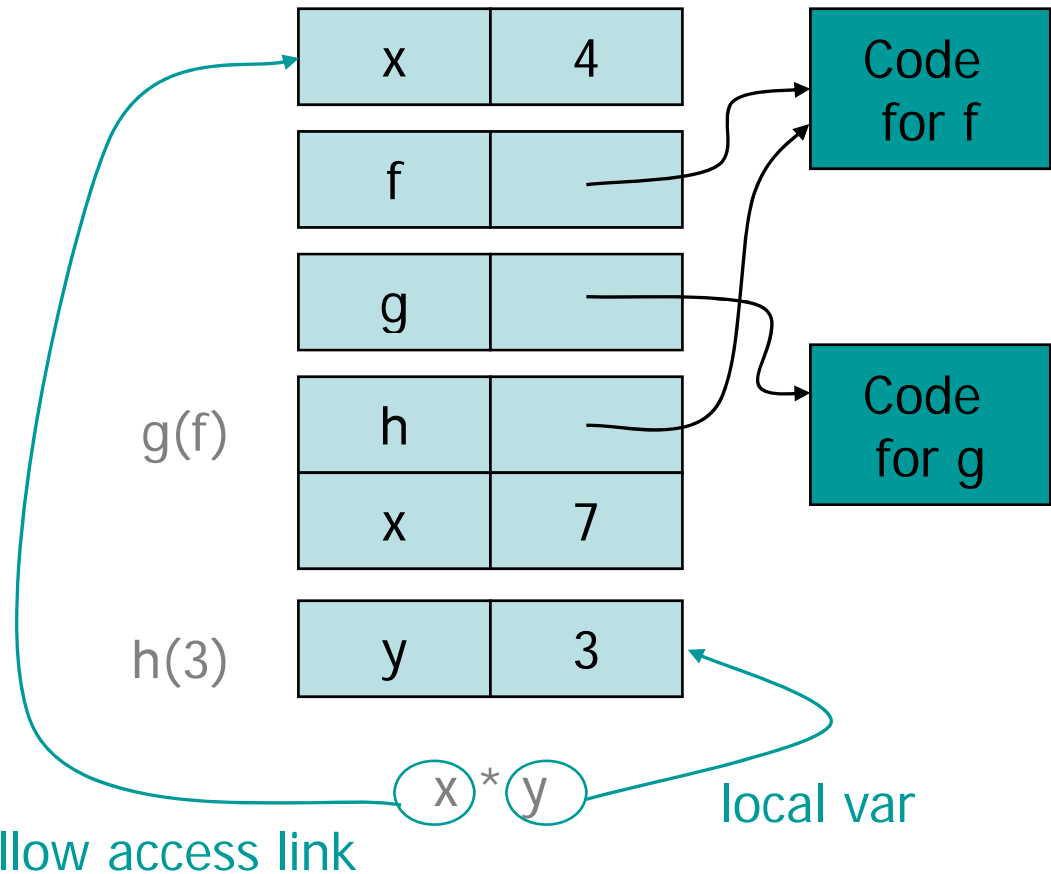
Which one is used for each occurrence of x?

# Static Scope for Function Argument

```

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

```



How is access link for h(3) set?

# 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

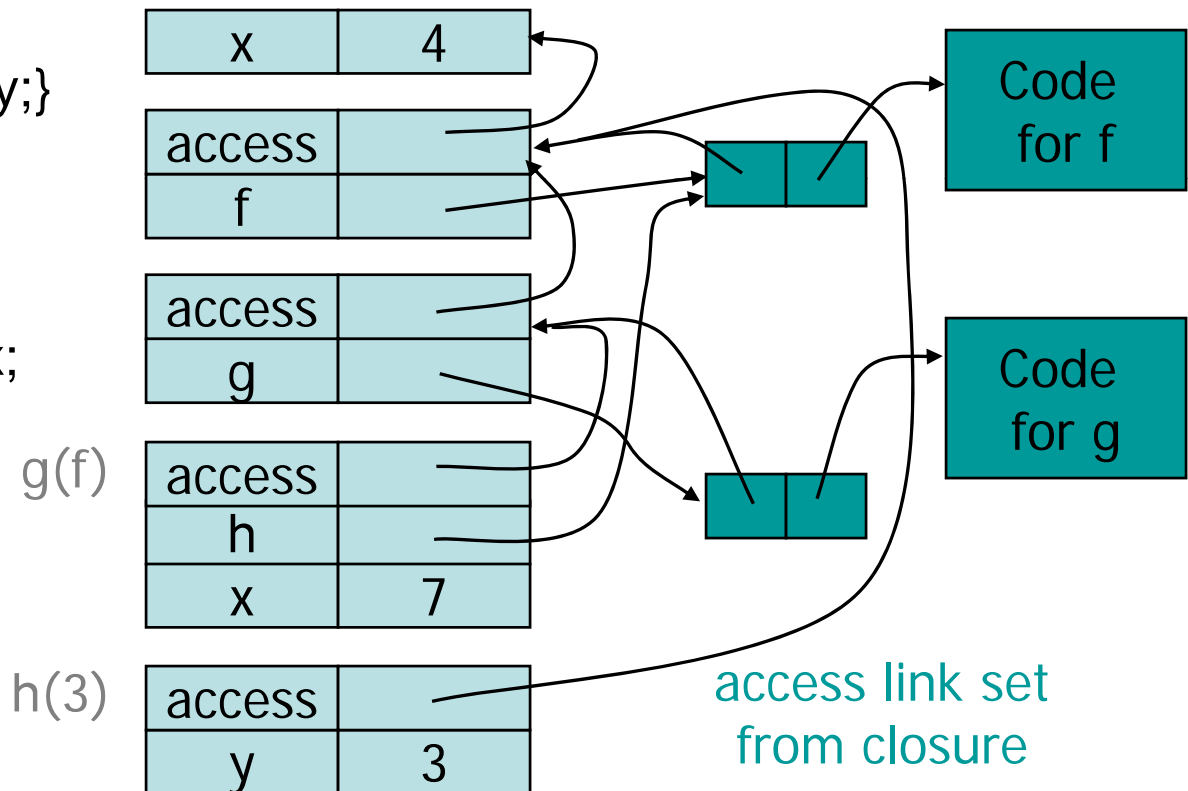
# Function Argument and Closures

Run-time stack with access links

```

{ int x = 4;
  { int f(int y){return x*y;}
    { int g(int→int 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 active records to find global variables
  - Still de-allocate active records using stack (LIFO) order

# Return Function as Result

- Language feature
  - Functions that return “new” functions
  - Need to maintain environment of function
- Function “created” dynamically
  - function value is closure =  $\langle \text{env}, \text{code} \rangle$
  - code *not* compiled dynamically (in most languages)

## Example: Return function with private state

```
{ int→int mk_counter (int init) {  
    int count = init;  
    int counter(int inc)  
        { return count += inc;}  
    return counter  
}  
int→int c = mk_counter(1);  
print c(2) + c(2);  
}
```

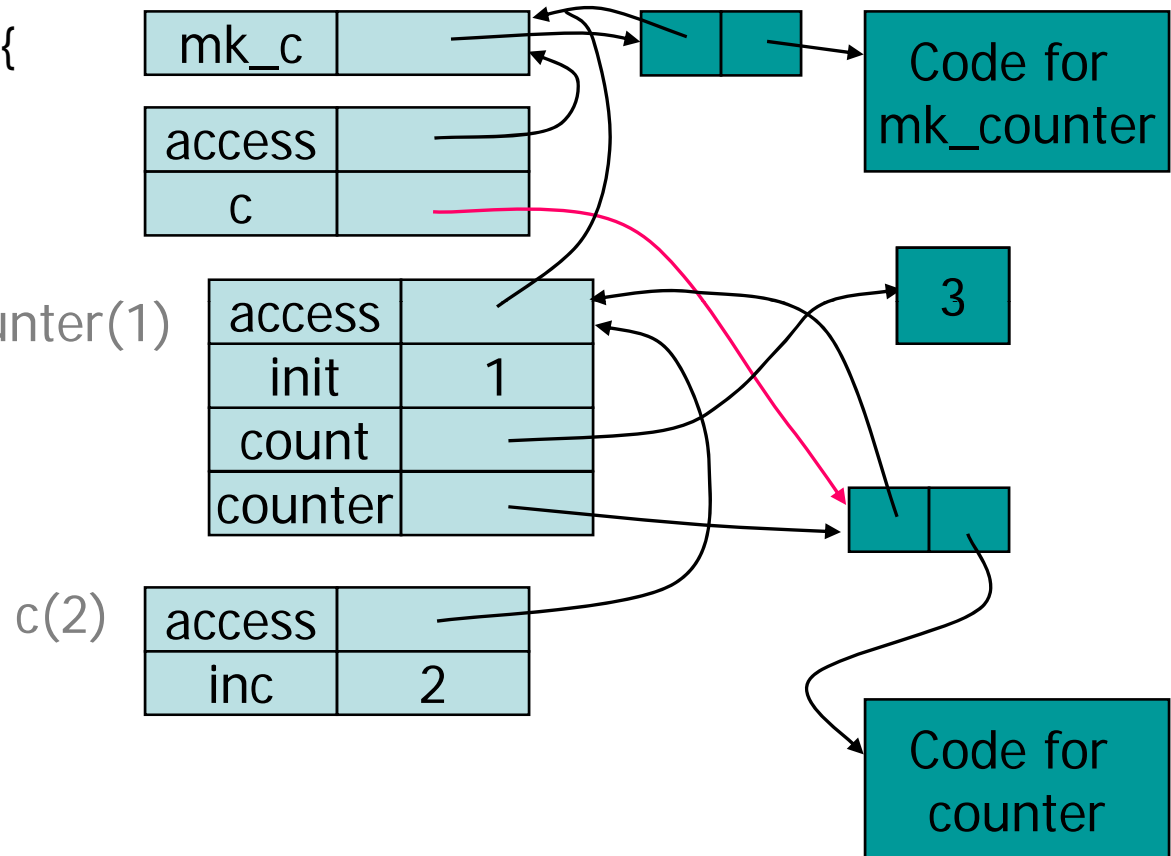
Function to “make counter” returns a closure

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

# Function Results and Closures

```

{int→int mk_counter (int init) {
  int count = init;
  int counter(int inc)
    { return count+=inc;}
  return counter  mk_counter(1)
}
int→int c = mk_counter(1);
print c(2) + c(2);
}
    
```



Call changes cell value from 1 to 3

# 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 de-allocation
  - Put activation records on heap
  - Invoke garbage collector as needed
- How to achieve the effect of function parameters in e.g. Java??

```

class CwF {int f(int i) {...} };

fsum(CwF ref, int[] a) {
    int sum= 0;
    for (i= 1, i<aa.length, i++)
        sum= sum + ref.f(a[i]);
    return sum
}

int aa[] = new aa[95];
int ip1(int i); {...};
int ip2(int i); {...};

```

```

class CwFip1 extends CwF {
    int f(int i) {return ip1(i)}
};
class CwFip2 extends CwF {
    int f(int i) {return ip2(i)}
};

CwFip1 reflp1 = new CwFip1();
CwFip1 reflp2 = new CwFip2();

fsum(reflp1, aa)
fsum(reflp2, aa)

```

# Dynamic languages – I

```
{ int n;  
  n= ...;  
  { int a[n];  
    };  
};
```

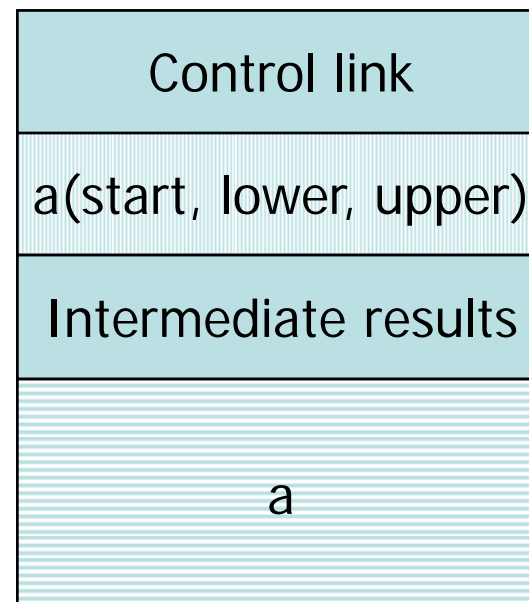
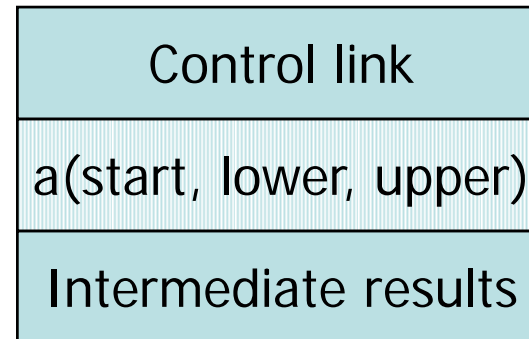
## At compilation

Each dynamic array gets a *descriptor* (start of array, lower, upper).

Array-access via this descriptor.

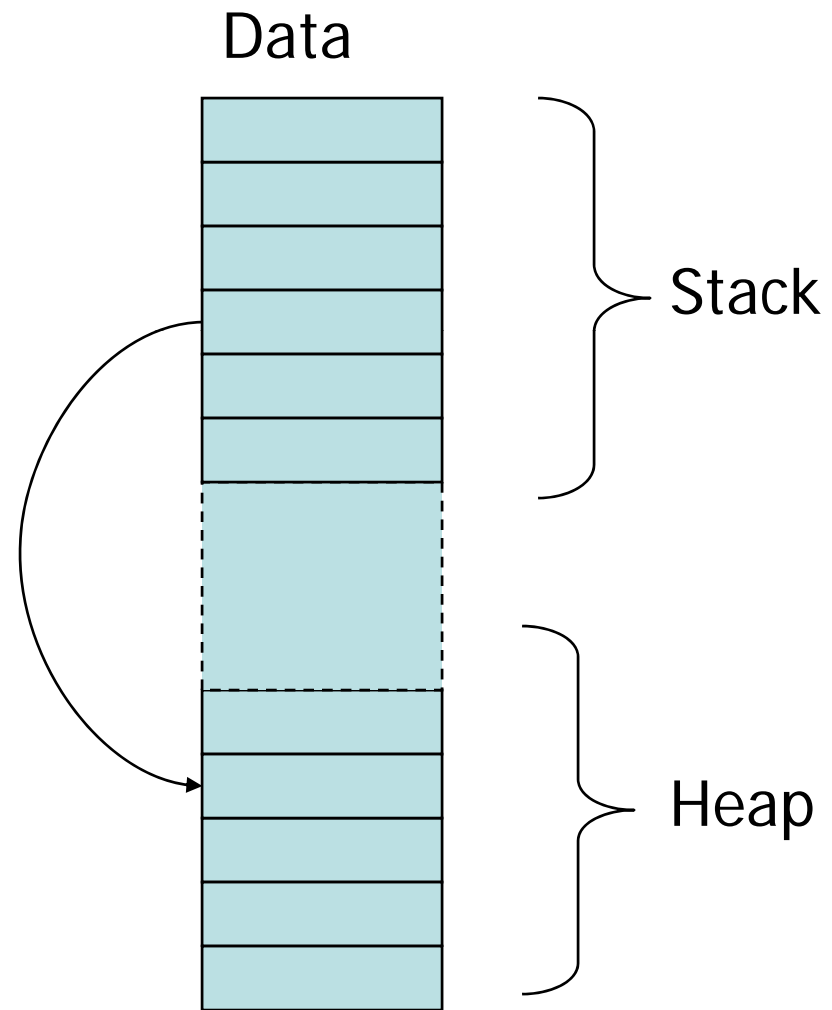
## At execution

Dynamic array-declaration: extend the activation record with necessary space and update the values of the descriptor.



# Dynamic languages - II

```
{  
  class Node {  
    Object contents  
    Node left, right;  
  };  
  {  
    Node n;  
    ...  
    n = new Node();  
    ...  
  }  
};
```





# Garbage Collection

- If not automatic
  - Explicit delete
  - Programmers responsibility, dangling references
- If automatic
  - Simple reference counting
  - When needed versus real-time
  - One-sweep versus generation-based

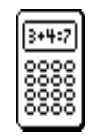
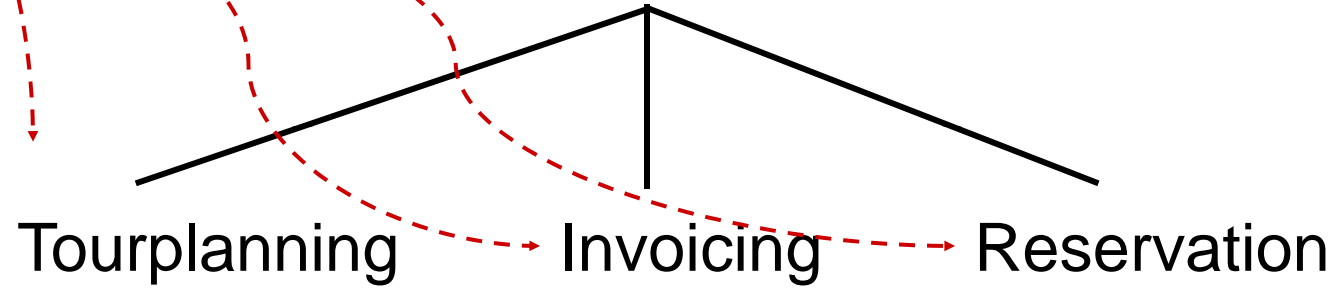
# Stack organization in perspective



**concurrent** objects

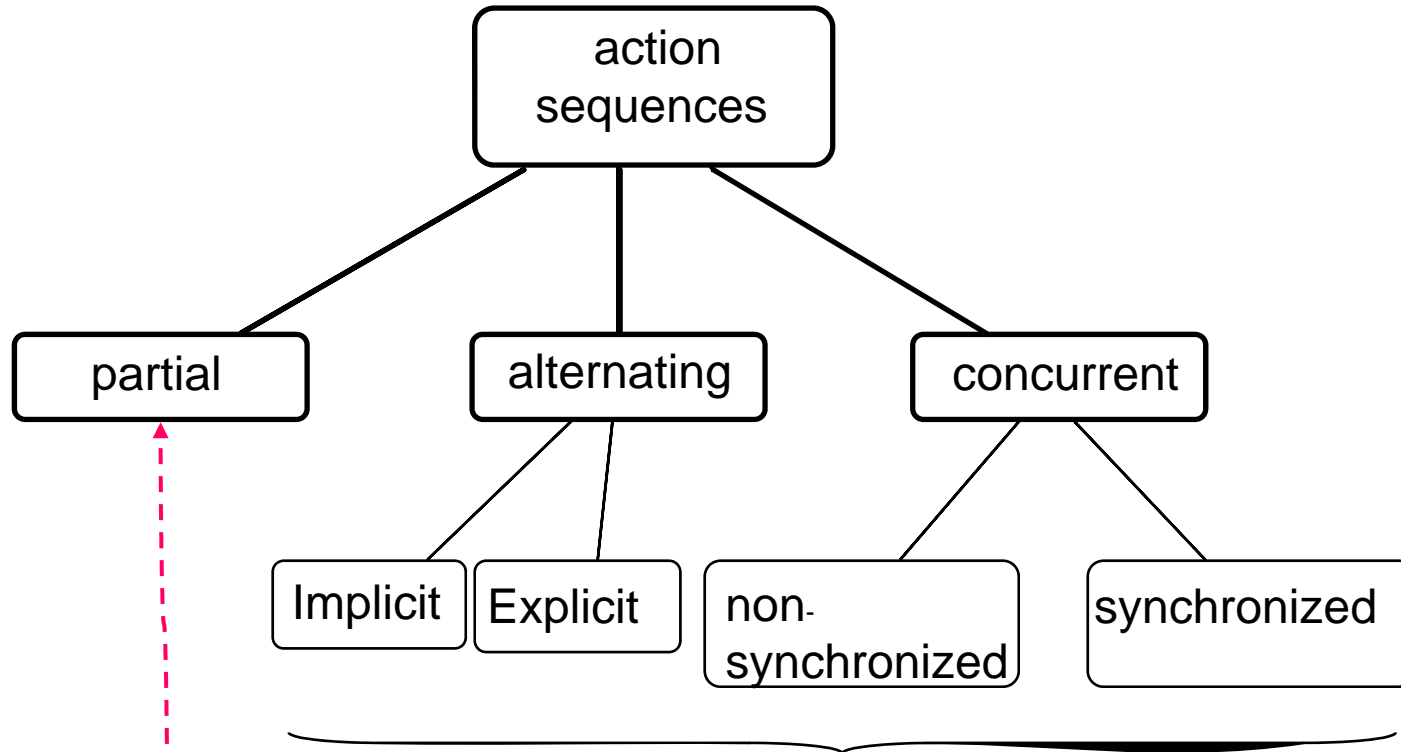


objects with alternating actions



partial actions of object (or of actions)





Activation records  
Stack organization

Objects - heap organization  
Each object may have a stack

Functions within functions  
Methods within methods

Coroutines: suspend, resume, ...  
Threads: run(), runnable()