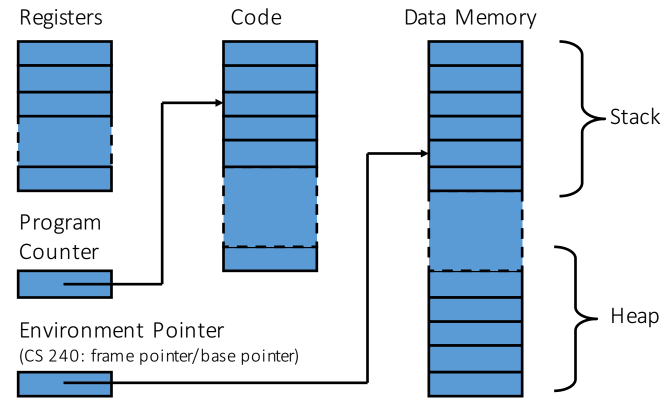
Implementing Control Flow and Scope

reconciling "the call stack" with "the environment" under the hood

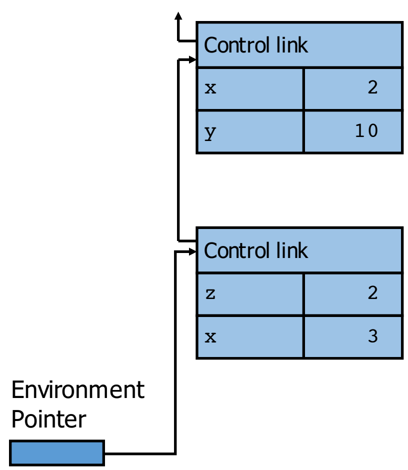
Slides adapted from Steve Freund, Williams College CS334

Simplified Machine Model (take CS 240)



Inline Blocks

```
{
  int x = 2;
  int y = 10
  {
    int z = 2;
    int x = 3;
    x = z + y;
  }
  print x;
}
```

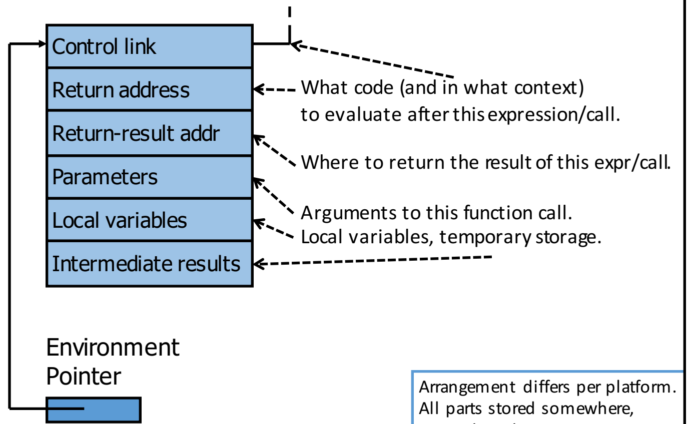


Function Calls

```
int squm(int n) {
  int i, sum = 0;
  for (i = 0; i < n; i++)
    sum = sum + i * i;
  return sum;
}

void main() {
  int x = squm(15);
  print x;
}
```

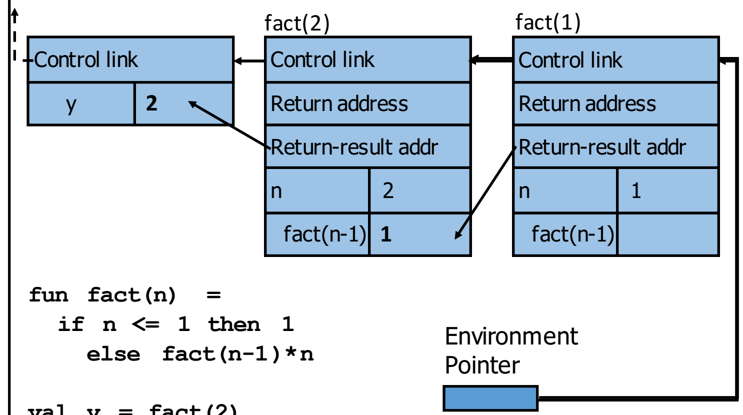
Activation Record stores evaluation context

(CS 240: a.k.a. call frame, stack frame, etc.)



Arrangement differs per platform.
All parts stored somewhere,
may mix registers, memory..

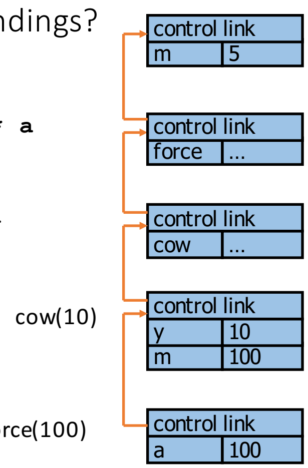
Activation Records




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

val y = fact(2)
```

Accessing outer bindings?

```
val m = 5

fun force(a) = m * a

fun cow(y) =
  let m = y * y in
    force(m)
  end

val _ = cow(10)
```



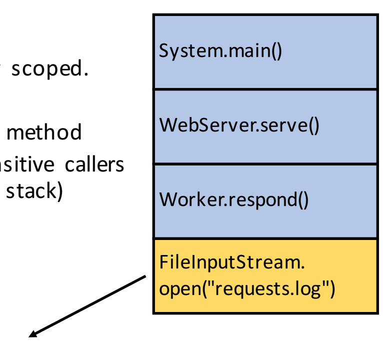
Dynamic Scope = follow control links

Stack Inspection (Java/JVM)

Permission is dynamically scoped.

Depends on:

- permission of calling method
- permission of all transitive callers (methods deeper on stack)



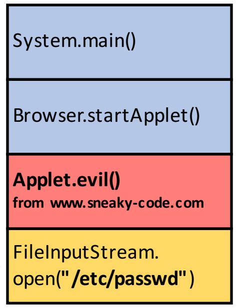

```
void open(String s) {
  SecurityManager.checkRead();
  ...
}
```

Stack Inspection (Java/JVM)

Permission is dynamically scoped.

Depends on:

- permission of calling method
- permission of all transitive callers (methods deeper on stack)

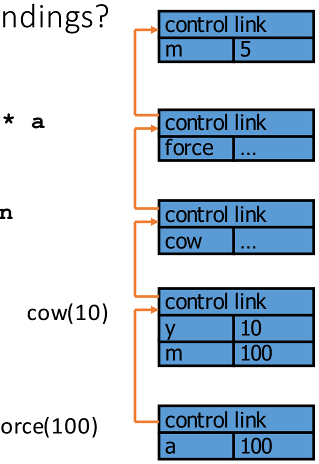



```
void open(String s) {
  SecurityManager.checkRead();
  ...
}
```

Fails if Applet code is not trusted

Accessing outer bindings?

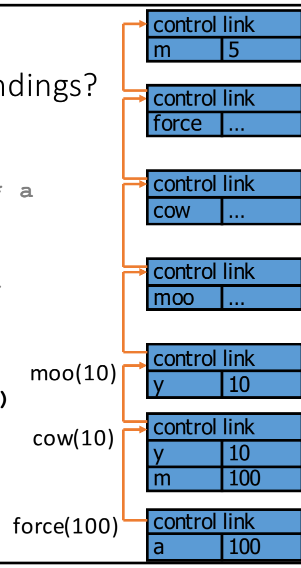
```
val m = 5

fun force(a) = m * a

fun cow(y) =
  let m = y * y in
    force(m)
  end

val _ = cow(10)
```



Lexical Scope = ???
links to follow?

Accessing outer bindings?

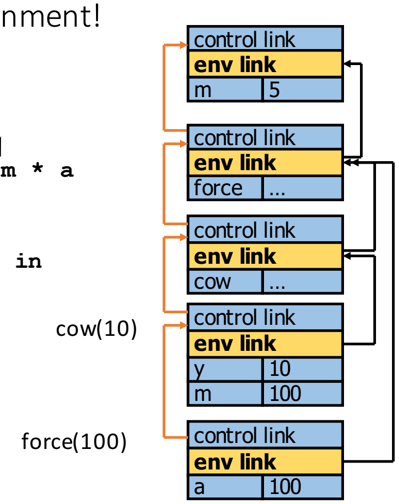
```
val m = 5

fun force(a) = m * a

fun cow(y) =
  let m = y * y in
    force(m)
  end
val _ = cow(10)
fun moo(y) = cow(y)
val _ = moo(y)
```



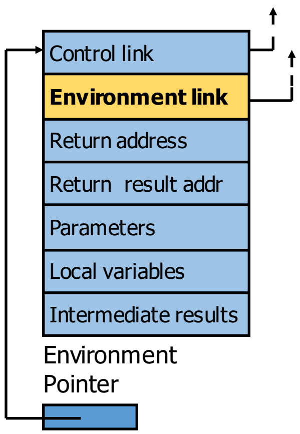
Lexical Scope = ???
links to follow?

Control ≠ Environment!
Separate link.

```
val m = 5

fun force(a) = m * a

fun cow(y) =
  let m = y * y in
    force(m)
  end

cow(10)
```



Activation Record for Lexical Scope
(static)



Control link
- to activation record of caller

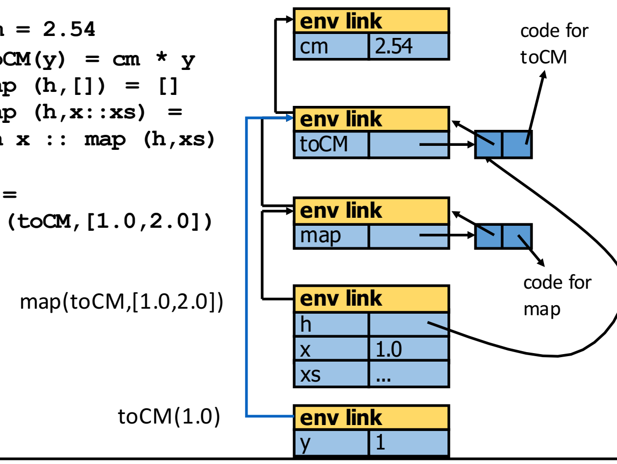
Environment link
- to activation record of closest lexically enclosing scope in program
- a.k.a. access link, scope link

Difference
- **Control** structure depends on **dynamic** behavior of program.
- **Environment** structure depends on **static** (lexical) form of program text.

Closures

(Control link omitted to save slide space. Vertical stack alignment indicates control.)

```
val cm = 2.54
fun toCM(y) = cm * y
fun map (h,[]) = []
  | map (h,x::xs) =
      h x :: map (h,xs)

val _ =
  map (toCM,[1.0,2.0])
```

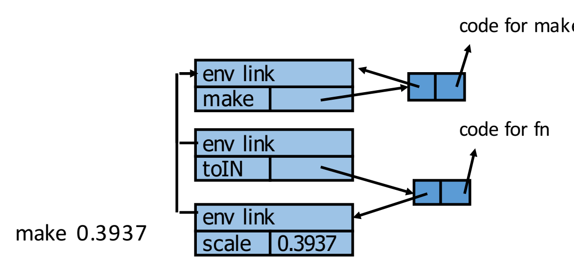


Implementation So Far

- Activation records track *separate*:
  - **Control link:** what code called this code/should continue executing next?
  - **Environment link:** what environment does this activation record extend?
- Closures:
  - Environment reference: to activation record where defined.
  - Code reference: to code
- On function call, new activiation record with:
  - Control link set to caller's acvtivation record.
  - Environment link set to closure's environment.
- **SO FAR:** all control/environment links point "back" (deeper) in the stack
  - Can still deallocate activation records in LIFO order.
- **But** what about returning functions...?

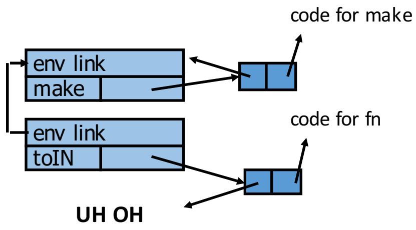
Returning a closure...

```
(* make a unit converter *)
fun make scale = fn x => x * scale
val toIN      = make (1.0 / 2.54)
val inches    = (toIN 6.0) + (toIN 20.0)
```



Returning a closure: *broken*

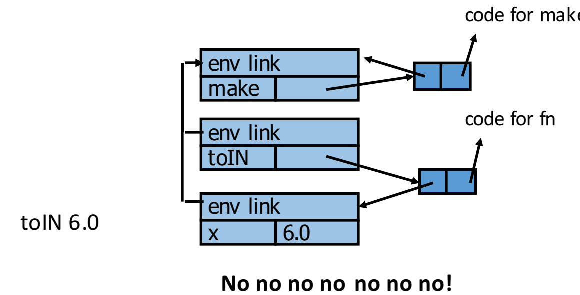
```
(* make a unit converter *)
fun make scale = fn x => x * scale
val toIN       = make (1.0 / 2.54)
val inches     = (toIN 6.0) + (toIN 20.0)
```



LIFO stack of activation records will not work!

Returning a closure: *even more broken*

```
(* make a unit converter *)
fun make scale = fn x => x * scale
val toIN       = make (1.0 / 2.54)
val inches     = (toIN 6.0) + (toIN 20.0)
```



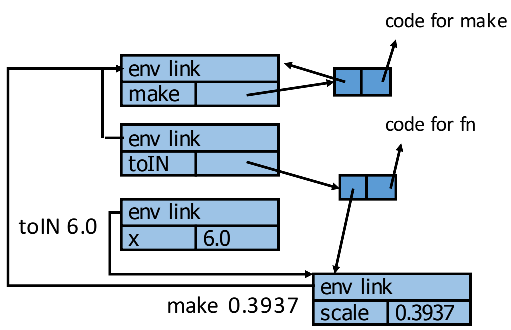
LIFO stack of activation records will not work!

One solution: *the "heap-allocated stack"*

```
(* make a unit converter *)
fun make scale = fn x => x * scale
val toIN       = make (1.0 / 2.54)
val inches     = (toIN 6.0) + (toIN 20.0)
```

Give up on stack.
Heap-allocate + GC activation records.

Contributes to generational hypothesis.



Free variables: when scope matters

```
(* xs is a long list *)
fun make x xs =
  let val temp1 = map (fn y => ...) xs
      val temp2 = filter (fn y => ...) temp1
      ... bind temp3 – temp17 to lists ...
      val (a::rest) = map (fn y => ...) temp17
  in
    fn z => x + a + z
  end
val f = make 31 [...]
val n = f 57
```

a, x

x, xs, map, filter

FV(*expr*) = variables used where not bound within *expr*.

Recursive definition:

FV(x) = {x}

FV(e1 + e2) = FV(e1) ∪ FV(e2)

FV(fn x => e) = FV(e) - {x}

...

Inefficiencies of Basic Heap-Allocated Stack

```
(* xs is a long list *)
fun make x xs =
  let val temp1 = map (fn y => ...) xs
      val temp2 = filter (fn y => ...) temp1
      ...
      val (a::rest) = map (fn y => ...) temp17
  in
     fn z => x + a + z
  end
val f = make 31 [...]
val n = f 57
```

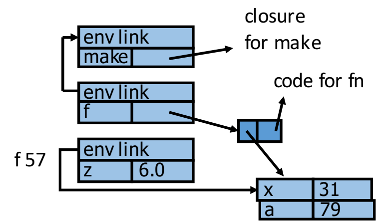
Lots of garbage reachable from closure. (2)

(1) Closure creation: **O(1)**
Variable lookup: **O(|env|)**

Alternative: ***save only free-variable bindings***

```
(* xs is a long list *)
fun make x xs =
  let val temp1 = map (fn y => ...) xs
      val temp2 = filter (fn y => ...) temp1
      ...
      val (a::rest) = map (fn y => ...) temp17
  in
     fn z => x + a + z
  end
val f = make 31 [...]
val n = f 57
```

Closure creation: **O(|env|)**
Variable lookup: **O(1)**



Even better:
http://users-cs.au.dk/danvy/sfp12/papers/keep-hearn-dybvig-paper-sfp12.pdf

Summary: Implementing Control and Scope

- Activation records track :
  - **Control link:** what code called this code/should continue executing next?
  - **Environment link:** what environment does this activation record extend?
- Closures:
  - Environment reference: to activation record where defined (or copy of free vars)
  - Code reference: to code
- On function call, new activiation record with:
  - Control link set to caller's activation record.
  - Environment link set to closure's environment.
- **Cannot manage activation records with stack discipline alone, but:**
  - Heap-allocate the stack or at least the copied closure environments.
  - Either way: Generational GC useful!