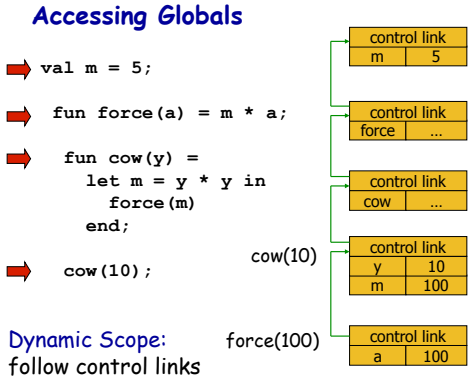
Scope and Memory Management (part 2)

CSCI 334

Stephen Freund

Accessing Globals

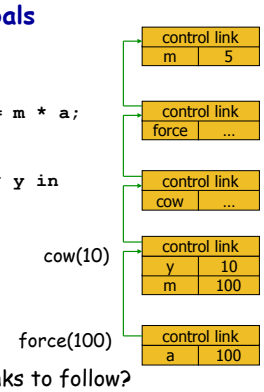
```
val m = 5;

fun force(a) = m * a;

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

cow(10);
```



Dynamic Scope:
follow control links

Accessing Globals

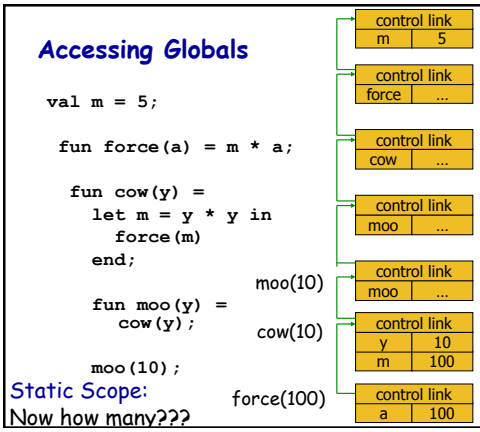
```
val m = 5;

fun force(a) = m * a;

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

cow(10);
```



Static Scope:
how to find m? # links to follow?

Accessing Globals

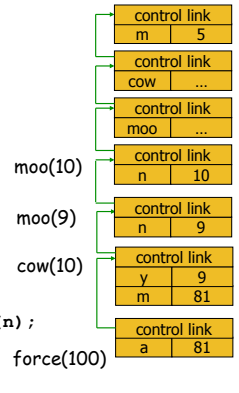
```
val m = 5;

fun force(a) = m * a;

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

fun moo(y) =
  cow(y);

moo(10);
```



Static Scope:
Now how many???

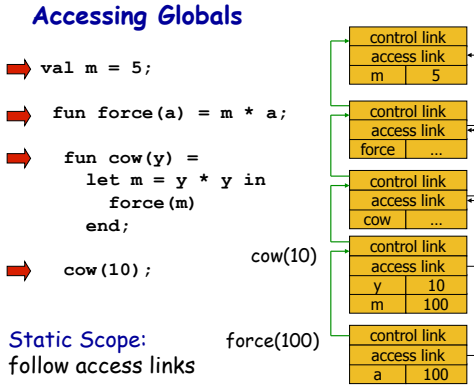
Even Worse...

```
val m = 5;

fun force(a) = m * a;

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

fun moo(0) = 0
  | moo(n) =
      moo(n-1) + cow(n);

moo(10);
```



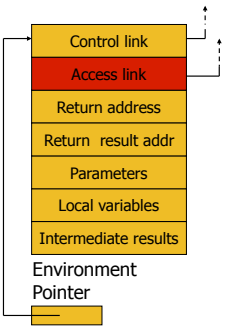
Accessing Globals

```
val m = 5;

fun force(a) = m * a;

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

cow(10);
```



Static Scope:
follow access links

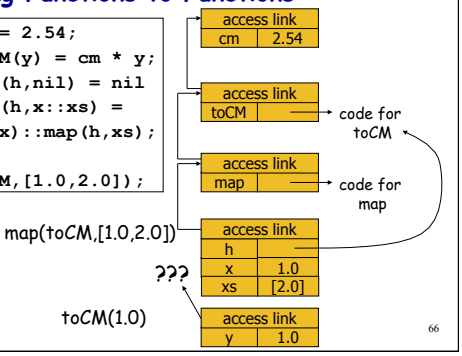
Activation record for static scope



- 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

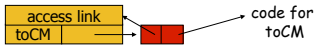
Passing Functions to Functions

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

map(toCM,[1.0,2.0]);
```



Function Values are Closures

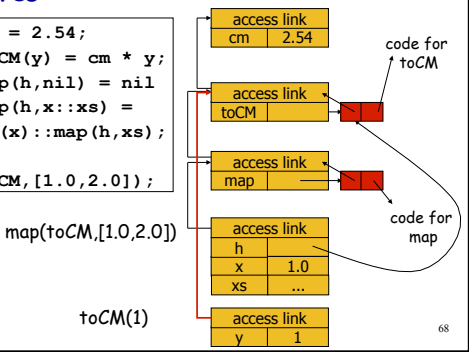
- *closure* = ⟨*env*, *code*⟩
  - env is pointer to activation record for scope in which function is declared
  - code is pointer to start of instructions



- When function is called:
  set the access link using the environment pointer from the closure

Closures

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

map(toCM,[1.0,2.0]);
```



Summary of Function Arguments

- Closure maintains pointer to static environment of a function body
- When called, access link set from closure
- All access links point "up" in the stack
  - can still deallocate activation records in lifo order

makeRand

```
fun makeRand(seed1, seed2) =
  let val generator = Random.rand(seed1,seed2)
  in
    fn (x,y) =>
      Random.randRange(x,y)(generator)
  end;
```

Returning Functions From Functions

```
fun make(seed) =
  let val count = ref seed;
      fun next(max) =
        (count := !count + 1;
         !count mod max)
  in
    next
  end;

val gen = make(0);
gen(5) -> returns 1
gen(3) -> returns 2
gen(2) -> returns 1
```

71

Returning Functions From Functions

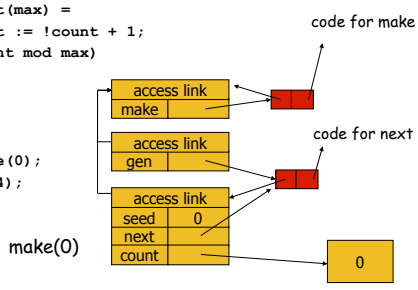
```
fun make(seed) =
  let val count = ref seed;
      fun next(max) =
        (count := !count + 1;
         !count mod max)
  in
    next
  end;

val gen = make(0);
gen(5) + gen(4);
```

count is global var in next

72

Function Results and Closures

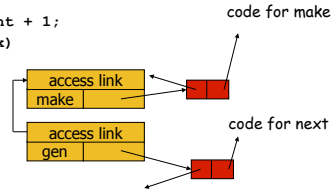
```
fun make(seed) =
  let val count = ref seed;
      fun next(max) =
        (count := !count + 1;
         !count mod max)
  in
    next
  end;

val gen = make(0);
gen(5) + gen(4);
```



73

Function Results and Closures

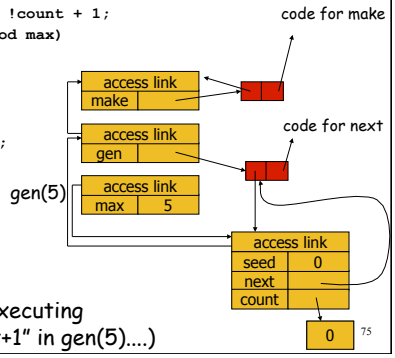
```
fun make(seed) =
  let val count = ref seed;
      fun next(max) =
        (count := !count + 1;
         !count mod max)
  in
    next
  end;

val gen = make(0);
gen(5) + gen(4);
```



Deallocating AR is bad...

74

```
fun make(seed) =
  let val count = ref seed;
      fun next(max) =
        (count := !count + 1;
         !count mod max)
  in
    next
  end;

val gen = make(0);
gen(5) + gen(4);
```



(Right before executing "count := !count+1" in gen(5)....)

75

Summary of Returning Functions

- Closure maintains 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 it sounds...

76