Scope and Memory Management (part 2)

CSCI 334
Stephen Freund

Inline Blocks

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

Accessing Globals

```
val m = 5;
fun force(a) = m * a;

fun cow(y) =
  let m = y * y in
  force(m)
end;
cow(10);

fun playGame() =
  ... if strategy(...) = goLeft then ...
end;
```

Examples of Dynamic Scoping

```
fun formatBuffer(buffer) =
  ... setColor(highlightColor) ...
  let highlightColor = Blue in
  formatBuffer(b);
```

Dynamic Scope:

```
fun playGame() =
  ... in playGame();
```
Stack Inspection

- Permission depends on:
  - permission of calling method
  - permission of all methods above it on stack

void open(String s) {
    SecurityManager.checkRead();
    FileInputStream.open("log");
}

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?

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 program
  - Access link depends on static form of program text

Fails if Applet code is not trusted
Another Example

```haskell
val cm = 2.54;
fun toCM(y) = cm * y;
...
toCM(5.0);
```

Passing Functions to Functions

```haskell
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]);
```

Closures

```haskell
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]);
```

makeRand

```haskell
fun makeRand(seed1, seed2) = 
  let val generator = Random.rand(seed1,seed2);
  fun rand(lo, hi) = Random.randRange(lo,hi)(generator)
  in
  rand
  end;

val gen = makeRand(10,12);
val x = gen(0,10);
```

make (not so random...)

```haskell
fun make(seed) = 
  let fun rand(lo) = lo + seed
      in 
      rand
      end;

val gen = make(0);
val x = gen(0,10);
```

Function Results and Closures

```haskell
fun make(seed) = 
  let fun rand(lo) = lo + seed
      in 
      rand
      end;

val gen = make(0);
gen(5) + gen(4);
```
Function Results and Closures

fun make(seed) =  
  let fun rand(lo) = lo + seed  
  in  
    rand  
  end;  
val gen = make(0);  
gen(5) + gen(4);

(Right before executing “lo + seed” in gen(5)....)