CSE 341
Lecture 11 b

closures; scoping rules

slides created by Marty Stepp
http://www.cs.washington.edu/341/
val x = 3;
fun f(n) = x * n;
f(8);
val x = 5;
f(8);

• The function produces 24 for both calls.
  • x's value of 3 is bound to f when f is defined.
  • A new definition of x later in the code doesn't affect f.
fun f(g) = 
  let val x = 3 
  in g(2) 
  end; 
val x = 4;
fun h(y) = x + y;
f(h);

• The call \( f(h) \) produces 6.
  - x's value of 4 is bound to h when h is defined.
  - A "later" definition of x in the let doesn't affect h.
fun multiplier(a) = 
  let fun f(b) = a * b 
  in f 
  end;
val m1 = multiplier(2); 
val m2 = multiplier(5); 
m1(10); 
m2(7);

• m1(10) produces 20, and m2(7) produces 35.
  ▪ On each call of multiplier, that call's a value becomes bound to inner function f as it is defined and returned.
  ▪ A later call to multiplier doesn't affect the past one's a.
The anatomy of functions

• A function really consists of a pair of things:
  ▪ some *code* to be evaluated
  ▪ an *environment* of variables and symbols used by the code

• This pair is also called a function *closure*. *
  ▪ Storing a function's environment with its code allows us to write some powerful code to utilize that environment.

* Many folks mistakenly refer to anonymous functions, or first-class functions, as "closures." This is a misuse of the term.
• **closure**: A first-class function that binds to free variables that are defined in its execution environment.

• **free variable**: A variable referred to by a function that is not one of its parameters or local variables.
  - **bound variable**: A free variable that is given a fixed value when "closed over" by a function's environment.

• A **closure** occurs when a function is defined and it attaches itself to the free variables from the surrounding environment to "close" up those stray references.
val x = 3;
fun f(n) = x * n;
f(8);
val x = 5;
f(8);

<table>
<thead>
<tr>
<th>symbol</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>3</td>
</tr>
<tr>
<td>n</td>
<td>(to be set on call)</td>
</tr>
</tbody>
</table>

**parent env.**

<table>
<thead>
<tr>
<th>symbol</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>5</td>
</tr>
<tr>
<td>f</td>
<td>(f's code/env.)</td>
</tr>
</tbody>
</table>

---

**global environment**
What's the result? (3)

```
fun multiplier(a) = 
  let fun f(b) = a * b 
  in f 
  end;
val m1 = multiplier(2); 
val m2 = multiplier(5); 
m1(10); 
m2(7);
```

<table>
<thead>
<tr>
<th>symbol</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>2</td>
</tr>
<tr>
<td>b</td>
<td>(set on call)</td>
</tr>
<tr>
<td>parent</td>
<td></td>
</tr>
</tbody>
</table>

f's environment (m1)

<table>
<thead>
<tr>
<th>symbol</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>5</td>
</tr>
<tr>
<td>b</td>
<td>(set on call)</td>
</tr>
<tr>
<td>parent</td>
<td></td>
</tr>
</tbody>
</table>

f's environment (m2)

<table>
<thead>
<tr>
<th>symbol</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>(f's code/env.)</td>
</tr>
<tr>
<td>a</td>
<td>(set on call)</td>
</tr>
<tr>
<td>parent</td>
<td></td>
</tr>
</tbody>
</table>

multiplier's environment

<table>
<thead>
<tr>
<th>symbol</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>m2</td>
<td>(m2's code/env.)</td>
</tr>
<tr>
<td>m1</td>
<td>(m1's code/env.)</td>
</tr>
<tr>
<td>multiplier</td>
<td>(multiplier's ...)</td>
</tr>
<tr>
<td>system libraries...</td>
<td>...</td>
</tr>
</tbody>
</table>

global environment
• **scope**: The enclosing context where values and expressions are associated.
  - essentially, the visibility of various identifiers in a program

• **lexical scope**: Scopes are nested via language syntax; a name refers to the *most local* definition of that symbol.
  - most modern languages (Java, C, ML, Scheme, JavaScript)

• **dynamic scope**: A name always refers to the *most recently executed* definition of that symbol.
  - Perl, Bash shell, Common Lisp (optionally), APL, Snobol
Lexical scope in Java

• In Java, every block ({} ) defines a scope.

```java
public class Scope {
    public static int x = 10;

    public static void main(String[] args) {
        System.out.println(x);
        if (x > 0) {
            int x = 20;
            System.out.println(x);
        }
        int x = 30;
        System.out.println(x);
    }
}
```
Lexical scope in ML

- In ML, a function, `let` expression, etc. defines a scope.

```ml
val y = 2;
fun f (n) = 
  let
    val x =
      let
        val n = 3
        in 10 * (n + y)
      end
    val y = 100 * n
    in
    x + y + n
  end;

f(6);
```
• What if Java used dynamic scoping?

```java
public class Scope2 {
    private static int x = 3;

    public static void one() {
        x *= 2;
        System.out.println(x); // could be any x!
    }

    public static void two() {
        int x = 5;
        one();
        System.out.println(x);
    }

    public static void main(String[] args) {
        one(); // program output: // 6
        two(); // 10
        int x = 2; // 10
        one();
        System.out.println(x); // 4
        System.out.println(x); // 4
    }
}
```
• **benefits of lexical scoping:**
  - functions can be reasoned about (defined, type-checked, etc.) where defined
  - function's meaning not related to choice of variable names
  - "Closing over" local variables creates "private" data; function definer knows function users cannot affect it

• **benefits of dynamic scoping:**
  - easier for compiler/interpreter author to implement!
  - useful for some domain-specific kinds of code (graphics, etc.); mixes the benefits of parameters with ease of globals
Closures in Java

- functions (methods) are not first-class citizens in Java

- but you can dynamically create an *inner or local class*
  - this class will exist inside of another (outer) class
  - it will have access to the outer class's local environment at the time of its creation
public class Outer { // note: n must be declared final
    public static Object foo(final int n) {
        class Inner {
            public String toString() {
                return "(My n is " + n + ")";
            }
        }
        return new Inner();
    }

    public static void main(String[] args) {
        Object o1 = foo(42);
        Object o2 = foo(17);
        System.out.println(o1 + " " + o2);
    }  // (My n is 42) (My n is 17)
public class Outer {
    public static Object foo(final int n) {
        return new Object() {
            public String toString() {
                return "(My n is " + n + ")";
            }
        };
    }
    public static void main(String[] args) {
        Object o1 = foo(42);
        Object o2 = foo(17);
        System.out.println(o1 + " " + o2);
        // (My n is 42) (My n is 17)
    }
}
Closure idioms

- You can use closures to:
  - create similar functions
  - combine functions
  - pass functions with private data to iterators (map, fold, ...)
  - provide an ADT
  - partially apply functions ("currying")
  - as a callback without the "wrong side" specifying the environment