Review
- Features learned: functions, tuples, lists, let expressions, options, records, datatypes, case expressions, pattern matching, exceptions, type variables, higher-order and anonymous functions, infix operators, type constructors, currying
- Today:
  - Lexical scope
  - Dynamic scope
  - Function closures

Example
A brain-teaser for scope and shadowing:

```
(* 1 *) val x = 1
(* 2 *) fun f y = x + y
(* 3 *) val x = 3
(* 4 *) val y = 4
(* 5 *) val z = f (x + y)
```

- Line 2 defines a function that, when called, evaluates body `x+y` in environment where `x` maps to 1 and `y` maps to the argument
- Call on line 5:
  - Looks up `f` to get the function defined on line 2
  - Evaluates `x+y` in current environment, producing 7
  - Calls the function, which evaluates the body in the old environment, producing 8

Scoping
- We know function bodies can use any bindings in scope
- But now that functions can be passed around: In scope where?

  When the function was defined
  (not when it was called)

- There are lots of good reasons for this semantics
  - Discussed after explaining what the semantics is
- For HW, exams, and competent programming, you must "get this"
- This semantics is called lexical scope

Lexical vs. Dynamic Scope

Rule of lexical scope: The body of a function is evaluated in the old dynamic environment that existed at the time the function was defined, not the current environment when the function is called.

Rule of dynamic scope: The body of a function is evaluated in the current dynamic environment at the time the function is called, not the old dynamic environment that existed at the time the function was defined.

(In both, environment is extended to map function argument to passed value.)

Closures

How can functions be evaluated in old environments that aren’t around anymore?
- The language implementation keeps them around as necessary

Can define the semantics of functions as follows:
- A function value has two parts
  - The code (obviously)
  - The environment that was current when the function was defined
- This is a “pair” but unlike ML pairs, you cannot access the pieces
- All you can do is call this “pair”
- This pair is called a function closure
- A call evaluates the code part in the environment part (extended with the function argument)
Example 1 (Lexical scope)

```ml
(* 1 *) val x = 1
(* 2 *) fun f y = x + y
(* 3 *) val x = 3
(* 4 *) val y = 4
(* 5 *) val z = f (x + y)
```

- Line 2 creates a closure and binds `f` to it:
  - Code: "argument `y` and body `x+y`"
  - Environment: "`x` maps to 1"
  - (Plus whatever else is in scope, including `f` for recursion)
- Line 5 calls that closure with `x` as argument.
- In function body, `x` maps to 2 and `y` maps to 7.
- So `z` is bound to 8.

Example 2 (Lexical scope)

```ml
(* 1 *) val x = 1
(* 2 *) fun f y = (* 6 *) val z = g 6
(* 2a *) let val x = y+1
(* 2b *) in fn z => x+y+z and
(* 3 *) val x = 3
(* 4 *) val g = f 4
(* 5 *) val y = 5
```

- Evaluating line 4 binds `g` to a closure:
  - Code: "argument `y` and body `x+y+z`"
  - Environment: "`y` maps to 4, `x` maps to 5 (shadowing), ..."
  - So this closure will always add 9 to its argument.
- So line 6 binds 15 to `z`.

Example 3 (Lexical scope)

```ml
(* 1 *) fun f g = (* 1a *) let val x = 3
(* 1b *) in g 2 end
(* 2 *) val x = 4
(* 3 *) fun h y = x + y
(* 4 *) val z = f (x + y)
```

- Evaluating line 3 binds `h` to a closure:
  - Code: "argument `y` and body `x+y`"
  - Environment: "`x` maps to 4, `f` maps to another closure, ..."
  - So `h`'s closure will always add 4 to its argument.
- So line 4 binds 6 to `z`.
- Line 1a is irrelevant (and bad style).

Example 1 (dynamic scope)

```ml
(* 1 *) val x = 1
(* 2 *) fun f y = (* 2a *) let val x = y+1
(* 2b *) in fn z => x+y+z and
(* 3 *) val x = 3
(* 4 *) val g = f 4
(* 5 *) val y = 5
(* 6 *) val z = g 6
```

- Evaluating line 4 binds `g` to a closure:
  - Code: "argument `y` and body `x+y+z`"
  - Environment: "`y` maps to 4, `x` maps to 5 (shadowing), ..."
  - So this closure will always add 9 to its argument.
- So line 6 binds 15 to `z`.

Example 2 (dynamic scope)

```ml
(* 1 *) val x = 1
(* 2 *) fun f y = (* 6 *) val z = g 6
(* 2a *) let val x = y+1
(* 2b *) in fn z => x+y+z and
(* 3 *) val x = 3
(* 4 *) val g = f 4
(* 5 *) val y = 5
```

- Evaluating line 4 binds `g` to a closure:
  - Code: "argument `y` and body `x+y+z`"
  - Environment: "`y` maps to 4, `x` maps to 5 (shadowing), ..."
  - So this closure will always add 9 to its argument.
- So line 6 binds 15 to `z`.

Example 2 (dynamic scope)

```ml
(* 1 *) val x = 1
(* 2 *) fun f y = (* 2a *) let val x = y+1
(* 2b *) in fn z => x+y+z end
(* 3 *) val x = 3
(* 4 *) val g = f 4
(* 5 *) val y = 5
(* 6 *) val z = g 6
```

- Evaluating line 4 binds `g` to a closure:
  - Code: "argument `y` and body `x+y+z`"
  - Environment: "`y` maps to 4, `x` maps to 5 (shadowing), ..."
  - So this closure will always add 9 to its argument.
- So line 6 binds 15 to `z`.
Example 3 (dynamic scope)

• Line 4 calls f with argument h
  – Body of f is evaluated in current dynamic environment, which maps x to 4.
  – let binding shadows x, makes it bind to 3.
  – Body of let calls g with argument 2
  – But g is bound to h, so h is called with argument 2

OLD semantics of function calls

Evaluation:
1. (Under current dynamic environment,) evaluate e0 to a function
   \( \text{fun } x_0(\ldots, e_n) = e \)
   – Since call type-checked, result is guaranteed to be a function
2. (Under current dynamic environment,) evaluate arguments to values e1, ..., en
3. Result is evaluation of \( e \) in an environment extended to map \( x_0 \) to \( e_0 \), ..., \( x_n \) to \( e_n \)
   – ("An environment" is actually the environment where the function was defined, and includes e0 for recursion. We'll give a careful definition later in the course.)

Why lexical scope?

1. Function meaning does not depend on variable names used

Example:
  - Lexical scope: Can change body to use q instead of x without changing final value of z

Semantics of function calls

Evaluation rule:
1. In current dynamic environment, evaluate e0 to a function closure.
2. In current dynamic environment, evaluate argument e1 to value v.
3. Take environment part of closure. Extend it to map x to v. Call the resulting dynamic environment E.
4. Evaluate code part of closure in dynamic environment E, resulting in a value v'.
5. Return v' as the result of the function call.
Why lexical scope?

1. Function meaning does not depend on variable names used

Example:
- Lexical scope: Can change body to use q instead of x without changing final value of z
- Dynamic scope: Run-time error; q isn’t in scope at line 6!

```
(* 1 *) val x = 1
(* 2 *) fun f y =
  (* 2a *) let val q = y+1
  (* 2b *) in fn z => q+y+z end
(* 3 *) val x = 3
(* 4 *) val g = f 4
(* 5 *) val y = 5
(* 6 *) val z = g 6 (* error! *)
```

Why lexical scope?

2. Functions can be type-checked & reasoned about where defined

Example:
- Lexical scope: Could change line 3 to `val x = "hi"` without changing result of computation
- Dynamic scope: Run-time error; can’t add string to int!

```
(* 1 *) val x = 1
(* 2 *) fun f y =
  (* 2a *) let val x = y+1
  (* 2b *) in fn z => x+y+z end
(* 3 *) val x = 3
(* 4 *) val g = f 4
(* 5 *) val y = 5
(* 6 *) val z = g 6 (* Error! *)
```

Why lexical scope?

3. Unused bindings can be eliminated

(see Example 3, earlier)

Why lexical scope?

4. Closures can easily store the data they need

```
fun greaterThanX x = fn y => y > x
fun noNegatives xs = filter(greaterThanX ~1, xs)
```
Does dynamic scope exist?

- Lexical scope for variables is definitely the right default
  - Very common across languages
- Dynamic scope is occasionally convenient in some situations
  - So some languages (e.g., Perl, Racket) have special ways to do it
  - In some languages, it’s the norm (e.g., Emacs LISP, LaTeX)
  - But most languages just don’t have it
- If you squint some, exception handling is like dynamic scope:
  - `raise e` transfers control to the current innermost handler
  - Does not have to be syntactically inside a `handle` expression (and usually isn’t)

A note on closures and performance: Recomputation

These both work and rely on using variables in the environment

```ml
fun allShorterThan1 (xs, s) = filter(fn x => String.size x < String.size s, xs)
fun allShorterThan2 (xs, s) = let val i = String.size s
  in filter(fn x => String.size x < i, xs) end
```

The first one computes `String.size` once per element of `xs`
The second one computes `String.size` s once per list
- Nothing new here: let-bindings are evaluated when encountered and function bodies evaluated when called

Higher-order programming in other languages

- Higher-order programming, e.g., with `map` and `filter`, is great
- Language support for closures makes it very pleasant
- Without closures, we can still do it more manually / clumsily
  - In OOP (e.g., Java) with one-method interfaces
  - In procedural (e.g., C) with explicit environment arguments
- Working through this:
  - Shows connections between languages and features
  - Can help you understand closures and objects

Example in ML

```ml
datatype 'a mylist = Cons of 'a * ('a mylist) | Nil
(* ('a -> 'b) -> 'a mylist -> 'b mylist *)
fun map f xs = case xs of ...
(* ('a -> bool) -> 'a mylist -> 'a mylist *)
fun filter f xs = case xs of ...
(* 'a mylist -> int *)
fun length xs = case xs of ...
val doubleAll = map (fn x => x*2)
val countNs xs n = length (filter (fn x => x=n) xs)
```

Java

- Java likely to have closures (like C#, Scala, Ruby, …)
  - Write like `list.map((x) => x.age)`
  - Make parallelism and collections much easier
  - Encourage less mutation
  - Hard parts for language designers:
    - Implementation with other features and VM
    - Evolving current standard library (else not worth it?)
- But how could we program in an ML style in Java today…
  - Won’t look like pseudocode above
  - Was even more painful before Java had generics

One-method interfaces

```java
interface FuncA,B> {  B m(A x); }
interface PredC> { boolean m(A x); }
interface Foo { String m(int x, int y); }
```

- An interface is a named type (constructor)
- An object with one method can serve as a closure
  - Different instances can have different fields (possibly different types)
  - Like different closures can have different environments (possibly different types)
- So an interface with one method can serve as a function type
List types

Creating a generic list class works fine
- Assuming null for empty list here, a choice we may regret
- null makes every type an option type with implicit `valOf`

```java
class List<T> {
    T head;
    List<T> tail;
    List(T x, List<T> xs) {
        head = x;
        tail = xs;
    }
    ...
}
```

Why not instance methods?

A more OO approach would be instance methods:

```java
class List<T> {
    <B> List<B> map(Func<B,T> f) {...}
    List<T> filter(Func<T> f) {...}
    int length() {...}
}
```

Can work, but interacts poorly with null for empty list
- Cannot call a method on null
- So leads to extra cases in all clients of these methods if a list might be empty

An even more OO alternative uses a subclass of List for empty-lists rather than null
- Then instance methods work fine!

Higher-order functions

- Let’s use static methods for `map`, `filter`, `length`
- Use our earlier generic interfaces for “function arguments”
- These methods are recursive
  - Less efficient in Java
  - Much simpler than common previous-pointer acrobatics

```java
static <A,B> List<B> map(Func<A,B> f, List<A> xs) {
    if (xs == null) return null;
    return new List<B>(f.m(xs.head), map(f, xs.tail));
}
static <A> List<A> filter(Func<A> f, List<A> xs) {
    if (xs == null) return null;
    if (f.m(xs.head))
        return new List<A>(xs.head, filter(f, xs.tail));
    return filter(f, xs.tail);
}
static <A> length(List<A> xs) { ... }
```

Clients

- To use `map` method to make a `List<Bar>` from a `List<Foo>`:
  - Define a class `C` that implements `Func<Foo,Bar>`
  - Use fields to hold any “private data”
  - Make an object of class `C`, passing private data to constructor
  - Pass the object to `map`
- As a convenience, can combine all 3 steps with anonymous inner classes
  - Mostly just syntactic sugar
  - But can directly access enclosing fields and `final` variables
  - Added to language to better support callbacks
  - Syntax an acquired taste? See Java code for lecture.

Closures in Java used in real world

- Used everywhere for Swing GUI!
  - You might even have done it yourself already.
  - (not higher order)
  - [http://docs.oracle.com/javase/tutorial/uiswing/events/ generalAbout.html#FrameClasses](http://docs.oracle.com/javase/tutorial/uiswing/events/generalAbout.html#FrameClasses)

Now C [for C experts]

- In Java, objects, like closures, can have “parts” that do not show up in their types (interfaces)
- In C, a function pointer is just a code pointer, period
  - So without extra thought, functions taking function pointer arguments won’t be as useful as functions taking closures
- A common technique:
  - Always define function pointers and higher-order functions to take an extra, explicit environment argument
  - But without generics, no good choice for type of list elements or the environment
  - Use `void*` and various type casts...
The C trick

[ignore if not (yet) a C wizard; full implementation in C code for lecture]

Don’t do this:

```c
list_t* map(void* (*f)(void*), list_t xs){
  ... f(xs->head) ...
}
```

Do this to support clients that need private data:

```c
list_t* map(void* (*f)(void*,void*)
void* env, list_t xs) {
  ... f(env,xs->head) ...
}
```

List libraries like this aren’t common in C, but callbacks are!
- Lack of generics means lots of type casts in clients

Closures in C used in real world

- [http://lxr.linux.no/linux+v2.6.34/kernel/workqueue.c#L1149](http://lxr.linux.no/linux+v2.6.34/kernel/workqueue.c#L1149)
- [http://lxr.linux.no/linux+v2.6.34/drivers/usb/storage/usb.c#L960](http://lxr.linux.no/linux+v2.6.34/drivers/usb/storage/usb.c#L960)