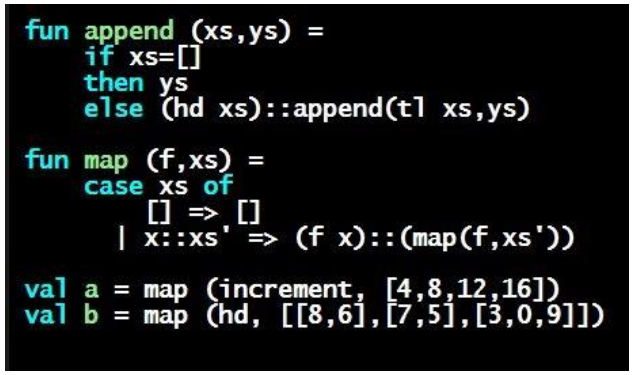


Programming Languages

Dan Grossman

Introduction to First-Class Functions

What is functional programming?

"*Functional programming*" can mean a few different things:

1. Avoiding mutation in most/all cases (done and ongoing)
2. Using functions as values (this section)

…

- Style encouraging recursion and recursive data structures
- Style closer to mathematical definitions
- Programming idioms using *laziness* (later topic, briefly)
- Anything not OOP or C? (not a good definition)

Not sure a definition of "*functional language*" exists beyond "makes functional programming easy / the default / required"

  – No clear yes/no for a particular language

First-class functions

- *First-class functions*: Can use them *wherever* we use values
  - Functions are values too
  - Arguments, results, parts of tuples, bound to variables, carried by datatype constructors or exceptions, …

```
fun double x = 2*x
fun incr x = x+1
val a_tuple = (double, incr, double(incr 7))
```

- Most common use is as an argument / result of another function
  - Other function is called a *higher-order function*
  - Powerful way to *factor out* common functionality

Function Closures

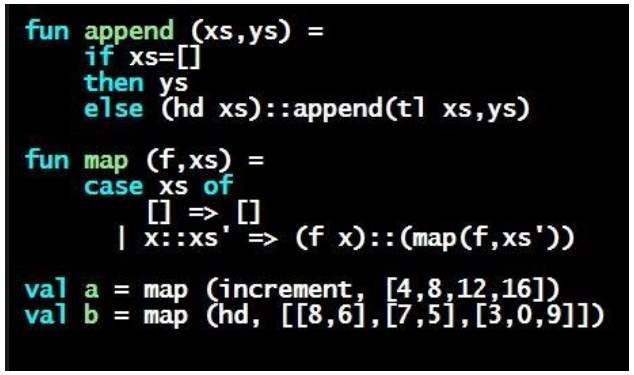
- *Function closure*: Functions can use bindings from outside the function definition (in scope where function is defined)
  - Makes first-class functions *much* more powerful
  - Will get to this feature in a bit, after simpler examples

- Distinction between terms *first-class functions* and *function closures* is not universally understood
  - Important conceptual distinction even if terms get muddled

Onward

Most of this section of course:

- How to use first-class functions and closures
- The precise semantics
- Multiple powerful idioms



Programming Languages

Dan Grossman

Functions As Arguments

Functions as arguments

- We can pass one function as an argument to another function
  - Not a new feature, just never thought to do it before

```
fun f (g,…) = … g (…) …
fun h1 … = …
fun h2 … = …
…   f(h1,…) … f(h2,…) …
```

- Elegant strategy for factoring out common code
  - Replace *N* similar functions with calls to 1 function where you pass in *N* different (short) functions as arguments

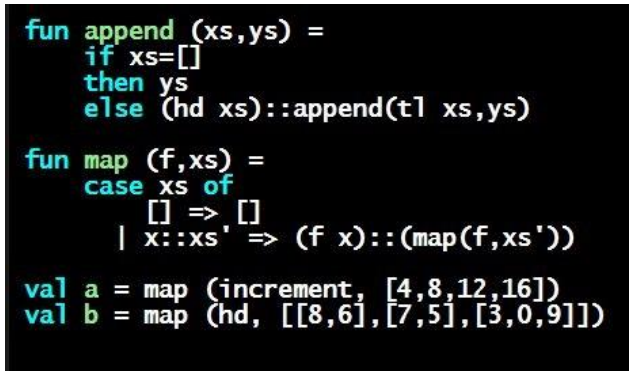
[See the code file for this segment]

Example

Can reuse **`n_times`** rather than defining many similar functions

- Computes **`f(f(…f(x)))`** where number of calls is **`n`**

```
fun n_times (f,n,x) =
   if n=0
   then x
   else f (n_times(f,n-1,x))

fun double x = x + x
fun increment x = x + 1
val x1 = n_times(double,4,7)
val x2 = n_times(increment,4,7)
val x3 = n_times(tl,2,[4,8,12,16])

fun double_n_times (n,x) = n_times(double,n,x)
fun nth_tail (n,x) = n_times(tl,n,x)
```



```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Polymorphic Types and
Functions As Arguments

The key point

- Higher-order functions are often so “generic” and “reusable” that they have polymorphic types, i.e., types with type variables
- But there are higher-order functions that are not polymorphic
- And there are non-higher-order (first-order) functions that are polymorphic
- Always a good idea to understand the type of a function, especially a higher-order function

Types

```
fun n_times (f,n,x) =
    if n=0
    then x
    else f (n_times(f,n-1,x))
```

- **`val n_times : ('a -> 'a) * int * 'a -> 'a`**
  - Simpler but less useful: **`(int -> int) * int * int -> int`**

- Two of our examples *instantiated* **`'a`** with **`int`**
- One of our examples *instantiated* **`'a`** with **`int list`**
- This *polymorphism* makes **`n_times`** more useful

- Type is *inferred* based on how arguments are used (later lecture)
  - Describes which types must be exactly something (e.g., **`int`**) and which can be anything but the same (e.g., **`'a`**)

Polymorphism and higher-order functions

- Many higher-order functions are polymorphic because they are so reusable that some types, "can be anything"

- But some polymorphic functions are not higher-order
  - Example: **`len : 'a list -> int`**

- And some higher-order functions are not polymorphic
  - Example: **`times_until_0 : (int -> int) * int -> int`**

```
fun times_until_0 (f,x) =
   if x=0 then 0 else 1 + times_until_0(f, f x)
```

Note: Would be better with tail-recursion

```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Anonymous Functions

Toward anonymous functions

- Definitions unnecessarily at top-level are still poor style:

```
fun triple x = 3*x
fun triple_n_times (f,x) = n_times(triple,n,x)
```

- So this is better (but not the best):

```
fun triple_n_times (f,x) =
  let fun trip y = 3*y
  in
     n_times(trip,n,x)
  end
```

- And this is even smaller scope
  - It makes sense but looks weird (poor style; see next slide)

```
fun triple_n_times (f,x) =
  n_times(let fun trip y = 3*y in trip end, n, x)
```

Anonymous functions

- This does not work: A function *binding* is not an *expression*

```
fun triple_n_times (f,x) =
  n_times((fun trip y = 3*y), n, x)
```

- This is the best way we were building up to: an expression form for *anonymous functions*

```
fun triple_n_times (f,x) =
  n_times((fn y => 3*y), n, x)
```

  - Like all expression forms, can appear anywhere
  - Syntax:
    - **`fn`** not **`fun`**
    - **`=>`** not **`=`**
    - no function name, just an argument pattern

Using anonymous functions

- Most common use: Argument to a higher-order function
  - Don't need a name just to pass a function

- But: Cannot use an anonymous function for a recursive function
  - Because there is no name for making recursive calls
  - If not for recursion, **`fun`** bindings would be syntactic sugar for **`val`** bindings and anonymous functions

```
fun triple x = 3*x

val triple = fn y => 3*y
```

```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Unnecessary Function Wrapping

A style point

Compare:

```
if x then true else false
```

With:

```
(fn x => f x)
```

So don't do this:

```
n_times((fn y => tl y),3,xs)
```

When you can do this:

```
n_times(tl,3,xs)
```

```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Map and Filter

Map (see the course logo)

```
fun map (f,xs) =
    case xs of
      [] => []
    | x::xs' => (f x)::(map(f,xs'))
```

```
val map : ('a -> 'b) * 'a list -> 'b list
```

Map is, without doubt, in the "higher-order function hall-of-fame"

- The name is standard (for any data structure)
- You use it *all the time* once you know it: saves a little space, but more importantly, *communicates what you are doing*
- Similar predefined function: **`List.map`**
  - But it uses currying (coming soon)

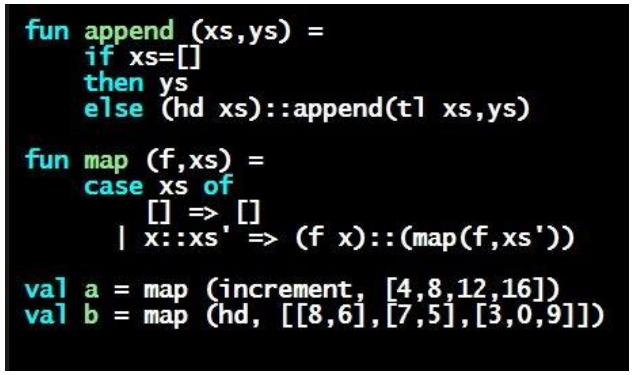
Filter

```
fun filter (f,xs) =
    case xs of
      [] => []
    | x::xs' => if f x
                then x::(filter(f,xs'))
                else filter(f,xs')
```

```
val filter : ('a -> bool) * 'a list -> 'a list
```

Filter is also in the hall-of-fame

- So use it whenever your computation is a filter
- Similar predefined function: **`List.filter`**
  - But it uses currying (coming soon)



```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Generalizing Prior Topics

Generalizing

Our examples of first-class functions so far have all:

- Taken one function as an argument to another function
- Processed a number or a list

But first-class functions are useful anywhere for any kind of data

- Can pass several functions as arguments
- Can put functions in data structures (tuples, lists, etc.)
- Can return functions as results
- Can write higher-order functions that traverse your own data structures

Useful whenever you want to abstract over "what to compute with"

- No new language features

Returning functions

- Remember: Functions are first-class values
  - For example, can return them from functions

- Silly example:

```
fun double_or_triple f =
    if f 7
    then fn x => 2*x
    else fn x => 3*x
```

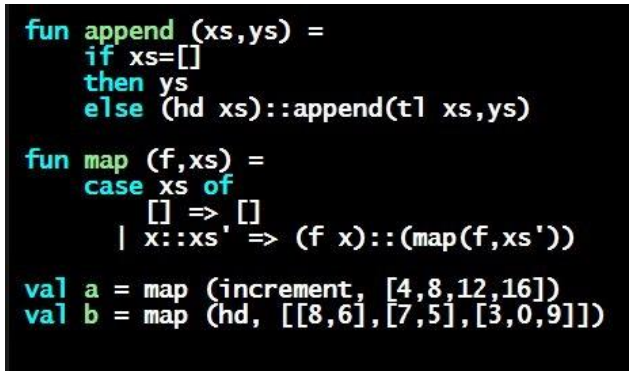
Has type **`(int -> bool) -> (int -> int)`**

But the REPL prints **`(int -> bool) -> int -> int`**
because it never prints unnecessary parentheses and
`t1 -> t2 -> t3 -> t4` means **`t1->(t2->(t3->t4))`**

Other data structures

- Higher-order functions are not just for numbers and lists
- They work great for common recursive traversals over your own data structures (datatype bindings) too
- Example of a higher-order *predicate*:
  - Are all constants in an arithmetic expression even numbers?
  - Use a more general function of type

    `(int -> bool) * exp -> bool`
  - And call it with **`(fn x => x mod 2 = 0)`**



Programming Languages

Dan Grossman

Definition of Lexical Scope

Very important concept

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

Where the function was defined
(not where it was called)

- This semantics is called *lexical scope*
- There are lots of good reasons for this semantics (why)
  - Discussed after explaining what the semantics is (what)
  - Later in course: implementing it (how)
- Must “get this” for homework, exams, and competent programming

Example

Demonstrates lexical scope even without higher-order functions:

```
(* 1 *) val x = 1
(* 2 *) fun f y = x + y
(* 3 *) val x = 2
(* 4 *) val y = 3
(* 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 **`5`**
  - Calls the function with **`5`**, which evaluates the body in the old environment, producing **`6`**

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

- Line 2 creates a closure and binds **`f`** to it:
  - Code: "take **`y`** and have body **`x+y`**"
  - Environment: "**`x`** maps to **`1`**"
    - (Plus whatever else is in scope, including **`f`** for recursion)

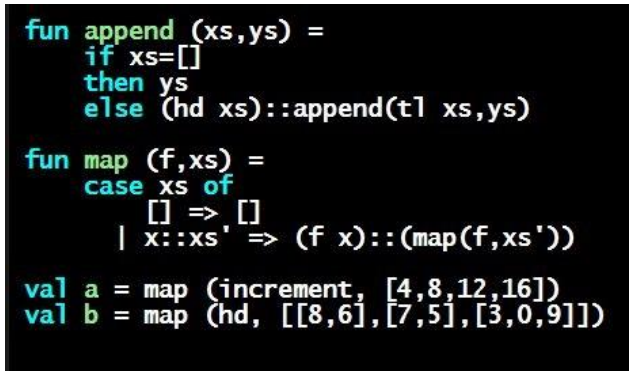
- Line 5 calls the closure defined in line 2 with **`5`**
  - So body evaluated in environment "**`x`** maps to **`1`**" extended with "**`y`** maps to **`5`**"

Coming up:

Now you know the rule: *lexical scope*.

Next steps (rest of section):

- (Silly) examples to demonstrate how the rule works with higher-order functions
- Why the other natural rule, *dynamic scope*, is a bad idea
- Powerful *idioms* with higher-order functions that use this rule
  - Passing functions to iterators like **`filter`**
  - Several more idioms



Programming Languages

Dan Grossman

Lexical Scope and Higher-Order Functions

The rule stays the same

A function body is evaluated in the environment where the function was defined (created)

- Extended with the function argument

Nothing changes to this rule when we take and return functions

- But “the environment” may involve nested let-expressions, not just the top-level sequence of bindings

Makes first-class functions much more powerful

- Even if may seem counterintuitive at first

Example: Returning a function

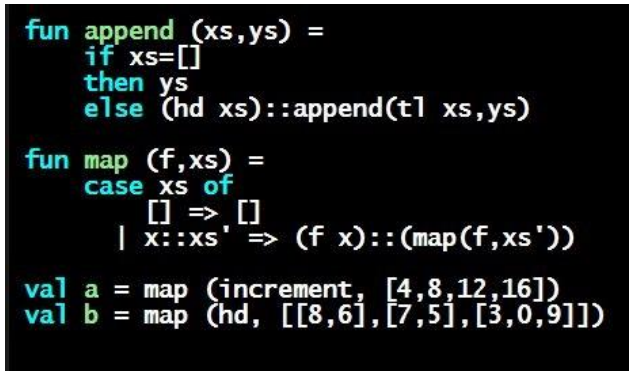
```
(* 1 *) val x = 1
(* 2 *) fun f y =
(* 2a *)     let val x = y+1
(* 2b *)     in fn z => x+y+z end
(* 3 *) val x = 3
(* 4 *) val g = f 4
(* 5 *) val y = 5
(* 6 *) val z = g 6
```

- Trust the rule: Evaluating line 4 binds to **`g`** to a closure:
  - Code: "take **`z`** and have 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: Passing a function

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

- Trust the rule: Evaluating line 3 binds **`h`** to a closure:
  - Code: "take **`y`** and have body **`x+y`**"
  - Environment: "**`x`** maps to **`4`**, **`f`** maps to a closure, …"
  - So this closure will always add **`4`** to its argument
- So line 4 binds **`6`** to **`z`**
  - Line 1a is as stupid and irrelevant as it should be



```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Why Lexical Scope

Why lexical scope

- *Lexical scope*: use environment where function is defined

- *Dynamic scope*: use environment where function is called

Decades ago, both might have been considered reasonable, but now we know lexical scope makes much more sense

Here are three precise, technical reasons

  – Not a matter of opinion

Why lexical scope?

1. Function meaning does not depend on variable names used

Example: Can change body of **`f`** to use **`q`** everywhere instead of **`x`**

- Lexical scope: it cannot matter
- Dynamic scope: depends how result is used

```
fun f y =
    let val x = y+1
    in fn z => x+y+z end
```

Example: Can remove unused variables

- Dynamic scope: but maybe some **`g`** uses it (weird)

```
fun f g =
    let val x = 3
    in g 2 end
```

Why lexical scope?

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

Example: Dynamic scope tries to add a string and an unbound variable to **6**

```
val x = 1
fun f y =
    let val x = y+1
    in fn z => x+y+z end
val x = "hi"
val g = f 7
val z = g 4
```

Why lexical scope?

3. Closures can easily store the data they need
   - Many more examples and idioms to come

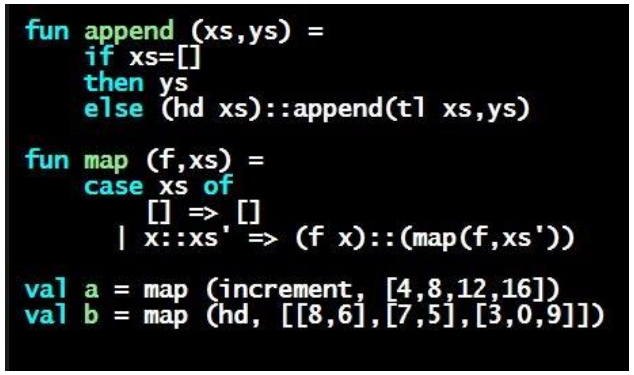
```
fun greaterThanX x = fn y => y > x

fun filter (f,xs) =
    case xs of
      [] => []
    | x::xs => if f x
               then x::(filter(f,xs))
               else filter(f,xs)

fun noNegatives xs = filter(greaterThanX ~1, xs)
fun allGreater (xs,n) = filter(fn x => x > n, 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., Racket) have special ways to do it
  - But most do not bother

- If you squint some, exception handling is more like dynamic scope:
  - **`raise e`** transfers control to the current innermost handler
  - Does not have to be syntactically inside a handle expression (and usually is not)



Programming Languages

Dan Grossman

Closures and Recomputation

When Things Evaluate

Things we know:

- A function body is not evaluated until the function is called
- A function body is evaluated every time the function is called
- A variable binding evaluates its expression when the binding is evaluated, not every time the variable is used

With closures, this means we can avoid repeating computations that do not depend on function arguments

- Not so worried about performance, but good example to emphasize the semantics of functions

Recomputation

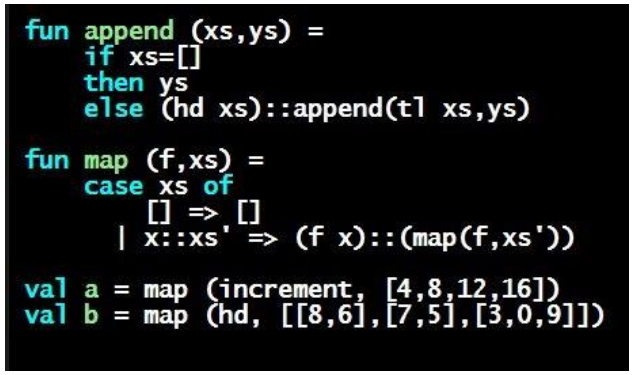
These both work and rely on using variables in the environment

```
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*



```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Fold and More Closures

Another famous function: Fold

`fold` (and synonyms / close relatives reduce, inject, etc.) is another very famous iterator over recursive structures

Accumulates an answer by repeatedly applying **`f`** to answer so far

- **`fold(f,acc,[x1,x2,x3,x4])`** computes **`f(f(f(f(acc,x1),x2),x3),x4)`**

```
fun fold (f,acc,xs) =
    case xs of
      []    => acc
    | x::xs => fold(f, f(acc,x), xs)
```

- This version "folds left"; another version "folds right"
- Whether the direction matters depends on **`f`** (often not)

```
val fold = fn : ('a * 'b -> 'a) * 'a * 'b list -> 'a
```

Why iterators again?

- These “iterator-like” functions are not built into the language
  - Just a programming pattern
  - Though many languages have built-in support, which often allows stopping early without resorting to exceptions

- This pattern separates recursive traversal from data processing
  - Can reuse same traversal for different data processing
  - Can reuse same data processing for different data structures
  - In both cases, using common vocabulary concisely communicates intent

Examples with fold

These are useful and do not use "private data"

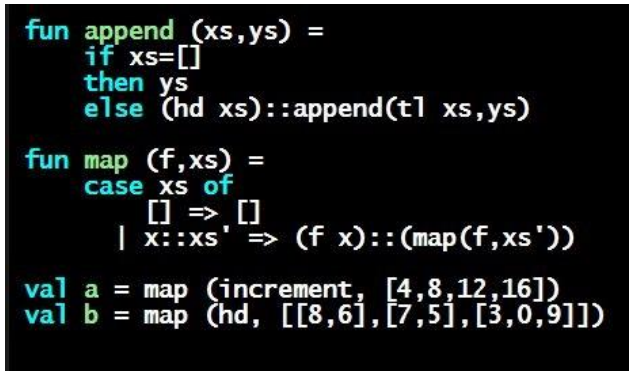
```
fun f1 xs = fold((fn (x,y) => x+y), 0, xs)
fun f2 xs = fold((fn (x,y) => x andalso y>=0),
                 true, xs)
```

These are useful and do use "private data"

```
fun f3 (xs,hi,lo) =
    fold(fn (x,y) =>
            x + (if y >= lo andalso y <= hi
                 then 1
                 else 0)),
         0, xs)
fun f4 (g,xs) = fold(fn (x,y) => x andalso g y),
                     true, xs)
```

Iterators made better

- Functions like **`map`**, **`filter`**, and **`fold`** are *much* more powerful thanks to closures and lexical scope
- Function passed in can use any “private” data in its environment
- Iterator “doesn’t even know the data is there” or what type it has



```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Another Closure Idiom: Combining Functions

More idioms

- We know the rule for lexical scope and function closures
  - Now what is it good for

A partial but wide-ranging list:

- Pass functions with private data to iterators: Done
- Combine functions (e.g., composition)
- Currying (multi-arg functions and partial application)
- Callbacks (e.g., in reactive programming)
- Implementing an ADT with a record of functions

Combine functions

Canonical example is function composition:

```
fun compose (f,g) = fn x => f (g x)
```

- Creates a closure that "remembers" what **`f`** and **`g`** are bound to
- Type **`('b -> 'c) * ('a -> 'b) -> ('a -> 'c)`**
  but the REPL prints something *equivalent*

- ML standard library provides this as infix operator **`o`**
- Example (third version best):

```
fun sqrt_of_abs i = Math.sqrt(Real.fromInt(abs i))
fun sqrt_of_abs i = (Math.sqrt o Real.fromInt o abs) i
val sqrt_of_abs = Math.sqrt o Real.fromInt o abs
```

Left-to-right or right-to-left

```
val sqrt_of_abs = Math.sqrt o Real.fromInt o abs
```

As in math, function composition is "right to left"

- "take absolute value, convert to real, and take square root"
- "square root of the conversion to real of absolute value"

"Pipelines" of functions are common in functional programming and many programmers prefer left-to-right

- Can define our own infix operator
- This one is very popular (and predefined) in F#

```
infix |>
fun x |> f = f x

fun sqrt_of_abs i =
   i |> abs |> Real.fromInt |> Math.sqrt
```

Another example

- "Backup function"

```
fun backup1 (f,g) =
    fn x => case f x of
              NONE => g x
            | SOME y => y
```

- As is often the case with higher-order functions, the types hint at what the function does

```
('a -> 'b option) * ('a -> 'b) -> 'a -> 'b
```

```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Another Closure Idiom: Currying

Currying

- Recall every ML function takes exactly one argument
- Previously encoded *n* arguments via one *n*-tuple
- Another way: Take one argument and return a function that takes another argument and…
  - Called “currying” after famous logician Haskell Curry

Example

```
val sorted3 = fn x => fn y => fn z =>
                  z >= y andalso y >= x

val t1 = ((sorted3 7) 9) 11
```

- Calling `(sorted3 7)` returns a closure with:
  - Code `fn y => fn z => z >= y andalso y >= x`
  - Environment maps `x` to `7`
- Calling *that* closure with `9` returns a closure with:
  - Code `fn z => z >= y andalso y >= x`
  - Environment maps `x` to `7`, `y` to `9`
- Calling *that* closure with `11` returns `true`

Syntactic sugar, part 1

```
val sorted3 = fn x => fn y => fn z =>
                  z >= y andalso y >= x

val t1 = ((sorted3 7) 9) 11
```

- In general, **`e1 e2 e3 e4 …`**,
  means **`(…((e1 e2) e3) e4)`**
- So instead of **`((sorted3 7) 9) 11`**,
  can just write **`sorted3 7 9 11`**
- Callers can just think "multi-argument function with spaces instead of a tuple expression"
  - Different than tupling; caller and callee must use same technique

Syntactic sugar, part 2

```
val sorted3 = fn x => fn y => fn z =>
                  z >= y andalso y >= x

val t1 = ((sorted3 7) 9) 11
```

- In general, **`fun f p1 p2 p3 … = e`**,
  means **`fun f p1 = fn p2 => fn p3 => … => e`**
- So instead of **`val sorted3 = fn x => fn y => fn z => …`**
  or **`fun sorted3 x = fn y => fn z => …`**,
  can just write **`fun sorted3 x y z = x >=y andalso y >= x`**
- Callees can just think "multi-argument function with spaces instead of a tuple pattern"
  - Different than tupling; caller and callee must use same technique

Final version

```
fun sorted3 x y z = z >= y andalso y >= x

val t1 = sorted3 7 9 11
```

As elegant syntactic sugar (even fewer characters than tupling) for:

```
val sorted3 = fn x => fn y => fn z =>
                  z >= y andalso y >= x

val t1 = ((sorted3 7) 9) 11
```

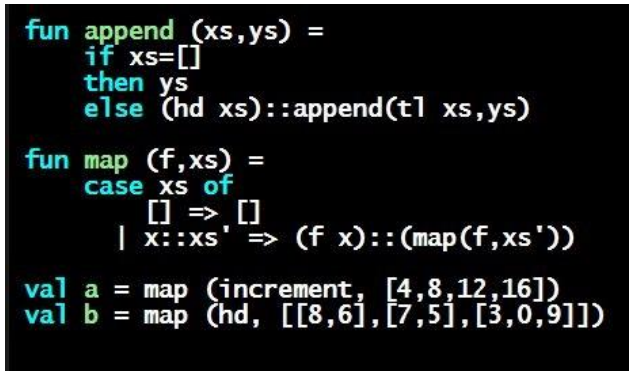
Curried fold

A more useful example and a call to it

- Will improve call next

```
fun fold f acc xs =
    case xs of
      []      => acc
    | x::xs' => fold f (f(acc,x)) xs'

fun sum xs = fold (fn (x,y) => x+y) 0 xs
```

Note: **`foldl`** in ML standard-library has **`f`** take arguments in opposite order



Programming Languages

Dan Grossman

Partial Application

"Too Few Arguments"

- Previously used currying to simulate multiple arguments
- But if caller provides "too few" arguments, we get back a closure "waiting for the remaining arguments"
  - Called partial application
  - Convenient and useful
  - Can be done with any curried function
- No new semantics here: a pleasant idiom

Example

```
fun fold f acc xs =
    case xs of
      []     => acc
    | x::xs' => fold f (f(acc,x)) xs'

fun sum_inferior xs = fold (fn (x,y) => x+y) 0 xs

val sum = fold (fn (x,y) => x+y) 0
```

As we already know, **`fold (fn (x,y) => x+y) 0`**
evaluates to a closure that given **`xs`**, evaluates the case-expression with **`f`** bound to **`fold (fn (x,y) => x+y)`** and **`acc`** bound to **`0`**

Unnecessary function wrapping

```
fun sum_inferior xs = fold (fn (x,y) => x+y) 0 xs

val sum = fold (fn (x,y) => x+y) 0
```

- Previously learned not to write `fun f x = g x` when we can write `val f = g`

- This is the same thing, with `fold (fn (x,y) => x+y) 0` in place of `g`

Iterators

- Partial application is particularly nice for iterator-like functions
- Example:

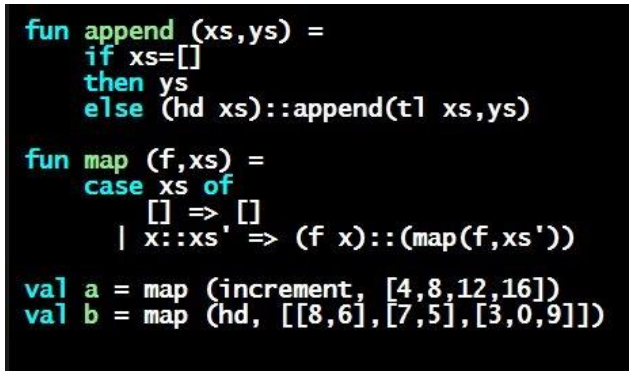
```
fun exists predicate xs =
    case xs of
      []     => false
    | x::xs' => predicate x
                orelse exists predicate xs'

val no = exists (fn x => x=7) [4,11,23]
val hasZero = exists (fn x => x=0)
```

- For this reason, ML library functions of this form usually curried
  - Examples: **`List.map`**, **`List.filter`**, **`List.foldl`**

The Value Restriction Appears ☹

If you use partial application to *create a polymorphic function*, it may not work due to the value restriction

- Warning about "type vars not generalized"
  - And won't let you call the function
- This should surprise you; you did nothing wrong ☺ but you still must change your code
- See the code for workarounds
- Can discuss a bit more when discussing type inference



Programming Languages

Dan Grossman

Currying Wrapup

More combining functions

- What if you want to curry a tupled function or vice-versa?
- What if a function's arguments are in the wrong order for the partial application you want?

Naturally, it is easy to write higher-order wrapper functions

  – And their types are neat logical formulas

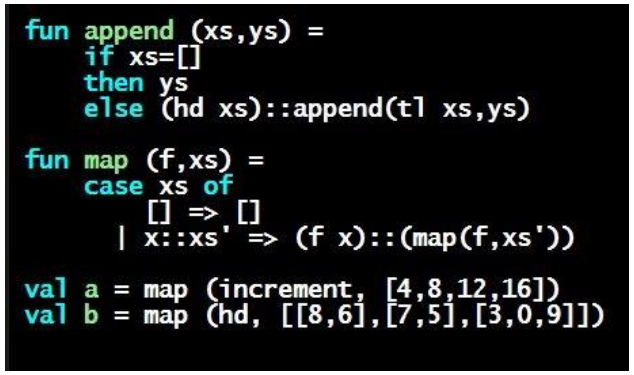
```
fun other_curry1 f = fn x => fn y => f y x
fun other_curry2 f x y = f y x
fun curry f x y = f (x,y)
fun uncurry f (x,y) = f x y
```

Efficiency

So which is faster: tupling or currying multiple-arguments?

- They are both constant-time operations, so it doesn't matter in most of your code – "plenty fast"
  - Don't program against an *implementation* until it matters!

- For the small (zero?) part where efficiency matters:
  - It turns out SML/NJ compiles tuples more efficiently
  - But many other functional-language implementations do better with currying (OCaml, F#, Haskell)
    - So currying is the "normal thing" and programmers read `t1 -> t2 -> t3 -> t4` as a 3-argument function that also allows partial application



Programming Languages

Dan Grossman

Mutable References

ML has (separate) mutation

- Mutable data structures are okay in some situations
  - When “update to state of world” is appropriate model
  - But want most language constructs truly immutable
- ML does this with a separate construct: references
- Introducing now because will use them for next closure idiom
- Do not use references on your homework
  - You need practice with mutation-free programming
  - They will lead to less elegant solutions

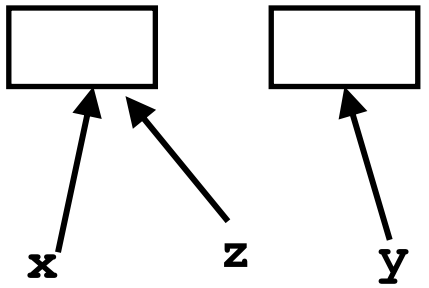
References

- New types: **`t ref`** where **`t`** is a type

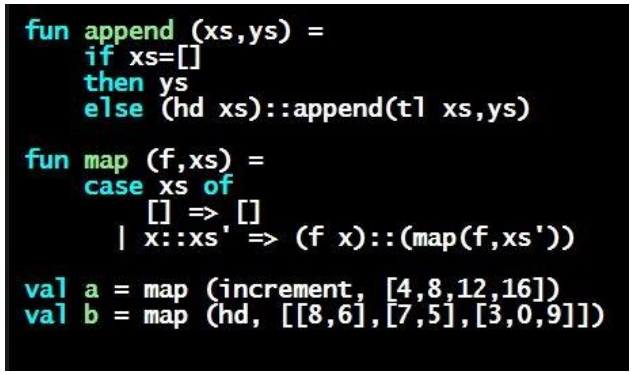
- New expressions:
  - **`ref e`** to create a reference with initial contents **`e`**
  - **`e1 := e2`** to update contents
  - **`!e`** to retrieve contents (not negation)

References example

```
val x = ref 42
val y = ref 42
val z = x
val _ = x := 43
val w = (!y) + (!z) (* 85 *)
(* x + 1 does not type-check *)
```



- A variable bound to a reference (e.g., `x`) is still immutable: it will always refer to the same reference
- But the contents of the reference may change via `:=`
- And there may be aliases to the reference, which matter a lot
- References are first-class values
- Like a one-field mutable object, so `:=` and `!` don't specify the field



Programming Languages

Dan Grossman

Another Closure Idiom: Callbacks

Callbacks

A common idiom: Library takes functions to apply later, when an *event* occurs – examples:

- When a key is pressed, mouse moves, data arrives
- When the program enters some state (e.g., turns in a game)

A library may accept multiple callbacks

- Different callbacks may need different private data with different types
- Fortunately, a function's type does not include the types of bindings in its environment
- (In OOP, objects and private fields are used similarly, e.g., Java Swing's event-listeners)

Mutable state

While it's not absolutely necessary, mutable state is reasonably appropriate here

- We really do want the "callbacks registered" to *change* when a function to register a callback is called

Example call-back library

Library maintains mutable state for "what callbacks are there" and provides a function for accepting new ones

- A real library would all support removing them, etc.
- In example, callbacks have type **`int->unit`**

So the entire public library interface would be the function for registering new callbacks:

```
val onKeyEvent : (int -> unit) -> unit
```

(Because callbacks are executed for side-effect, they may also need mutable state)

Library implementation

```
val cbs : (int -> unit) list ref = ref []

fun onKeyEvent f =  cbs := f :: (!cbs)

fun onEvent i =
    let fun loop fs =
            case fs of
              []     => ()
            | f::fs' => (f i; loop fs')
    in loop (!cbs) end
```

Clients

Can only register an **`int -> unit`**, so if any other data is needed, must be in closure's environment

- And if need to "remember" something, need mutable state

Examples:

```
val timesPressed = ref 0
val _ = onKeyEvent (fn _ =>
            timesPressed := (!timesPressed) + 1)

fun printIfPressed i =
   onKeyEvent (fn j =>
       if i=j
       then print ("pressed " ^ Int.toString i)
       else ())
```

```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Standard-Library Documentation

One last thing

This topic is not particularly related to the rest of the section, but we made it a small part of Homework 3

ML, like many languages, has a standard library

- For things you could not implement on your own
  - Examples: Opening a file, setting a timer
- For things so common, a standard definition is appropriate
  - Examples: **`List.map`**, string concatentation

You should get comfortable seeking out documentation and gaining intuition on where to look

- Rather than always being told exactly what functions do

Where to look

`http://www.standardml.org/Basis/manpages.html`

Organized into structures, which have signatures

- Define our own structures and signatures next section

Homework 3: Find-and-use or read-about-and-use a few functions under **`STRING`**, **`Char`**, **`List`**, and **`ListPair`**

To use a binding: **`StructureName.functionName`**

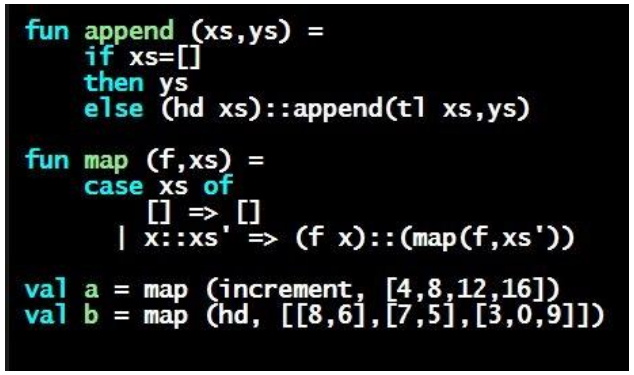
- Examples: **`List.map`**, **`String.isSubstring`**

REPL trick

- I often forget the order of function arguments

- While no substitute for full documentation, you can use the REPL for a quick reminder
  - Just type in the function name for its type
  - Can also guess function names or print whole structure
  - No special support: this is just what the REPL does

- Some REPLs (for other languages) have special support for printing documentation



```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Optional: Abstract Data Types with Closures

Implementing an ADT

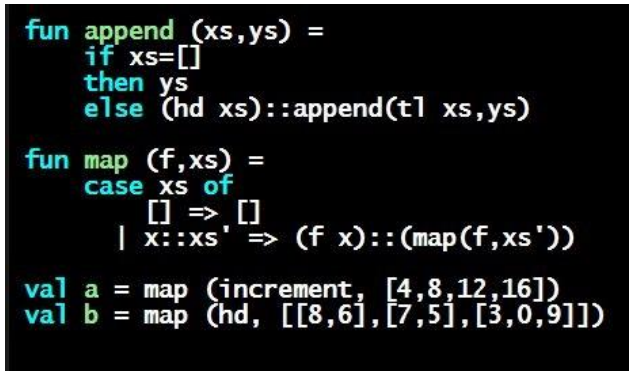
As our last idiom, closures can implement abstract data types

- Can put multiple functions in a record
- The functions can share the same private data
- Private data can be mutable or immutable
- Feels a lot like objects, emphasizing that OOP and functional programming have some deep similarities

See code for an implementation of immutable integer sets with operations *insert*, *member*, and *size*

The actual code is advanced/clever/tricky, but has no new features

- Combines lexical scope, datatypes, records, closures, etc.
- Client use is not so tricky



Programming Languages

Dan Grossman

Optional: Closure Idioms Without Closures

Higher-order programming

- 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

Outline

This segment:

- Just the code we will “port” to Java and/or C
- Not using standard library to provide fuller comparison

Next segments:

- The code in Java and/or C
- What works well and what is painful

```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Optional: Closure Idioms Without Closures in Java

Java

- Java 8 scheduled to have closures (like C#, Scala, Ruby, …)
  - Write like
```
xs.map((x) => x.age)
  .filter((x) => x > 21)
  .length()
```
  - Make parallelism and collections much easier
  - Encourage less mutation

- But how could we program in an ML style without help
  - Will not look like the code above
  - Was even more painful before Java had generics

One-method interfaces

```
interface Func<B,A> {  B m(A x); }

interface Pred<A> { boolean m(A x); }
```

- An interface is a named [polymorphic] type
- 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

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

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

```
static <A,B> List<B> map(Func<B,A> 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(Pred<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){ … }
```

Why not instance methods?

A more OO approach would be instance methods:

```
class List<T> {
  <B> List<B> map(Func<B,T> f){…}
  List<T> filter(Pred<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!

Clients

- To use **`map`** method to make a **`List<Bar>`** from a **`List<Foo>`**:
  - Define a class **`C`** that implements **`Func<Bar,Foo>`**
    - 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?

```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Optional: Closure Idioms Without Closures in C

Now C

- Closures and OOP objects can have "parts" that do not show up in their types

- In C, a *function pointer* is only a code pointer
  - So without extra thought, functions taking function-pointer arguments will not 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

Don't do this:

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

Do this to support clients that need private data:

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

List libraries like this are not common in C, *but callbacks are!*

- *Always define callback interfaces to pass an extra* **`void*`**
- Lack of generics means lots of type casts in clients ☹