Programming Languages

Lexical Scope, Function Closures, and Function Closure Idioms

Jiwon Seo

Today's Topic

- Lexical Scope
- Function Closures
- Function Closure Idioms (Currying or partial evaluation)

Sections in Textbook

- 5.4, 5.6 (higher-order functions)
- 5.5 (currying or partial evaluation)
- 3.3.3, 5.1.3 (anonymous functions)

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)
- 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:

- (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`**

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

classification

Title: generator expression implementation	
Type:	Stage:
Components: Interpreter Core	Versions:

msg45166 - (view) **Author: Armin Rigo (arigo) *** **Date: 2004-02-02 12:58**

```
Logged In: YES
user_id=4771

The behavior is indeed the one described by the PEP but it is still surprising.
As far as I'm concerned it is yet another reason why free variable bindings
("nested scopes") are done wrong in Python currently :-(
```

msg45159 - (view) **Author: Jiwon Seo (jiwon) *** **Date: 2004-01-27 07:44**

```
Logged In: YES
user_id=595483

I fixed the patch for the bug that arigo mentioned, and for
what perky mentioned - PyList_GetSlice error handling - .

now, I'll tackle python compiler package :)
```

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)

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*

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

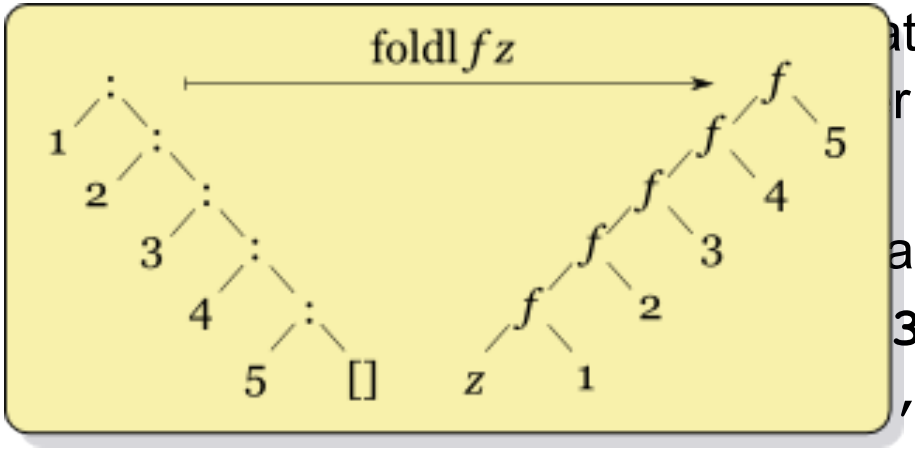
- **`fold(f,acc,[x1,x2,x3,x4])`** computes **`f(f(f(f(acc,x1),x2),x3),x4)`**

```
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
```

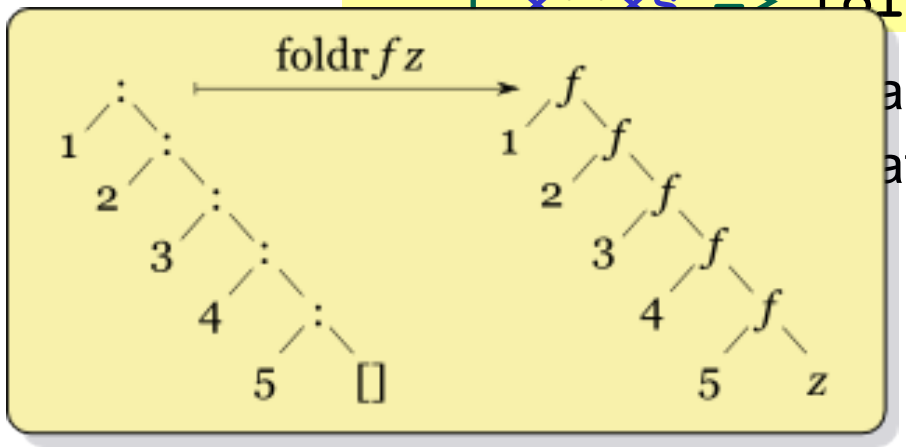
Another famous function: Fold



atives reduce, inject, etc.) is
r recursive structures

atedly applying **f** to answer so far
3,x4]) computes
,x3),x4)

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



another version “folds right”
atters depends on **f** (often not)

```
-> '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

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 *(of combining functions)*

- "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
```

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

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

"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 `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

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

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

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

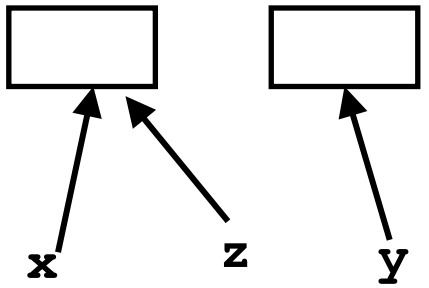
References

- 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

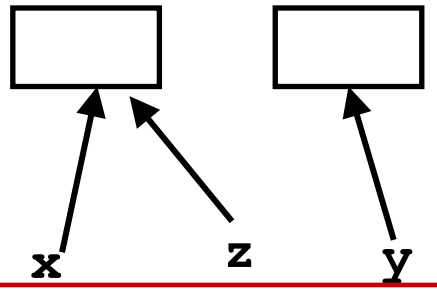
```
val x = ref 42
val y = ref 42
val z = x
val _ = x := 43
val w = (!y) + (!z) (* 85 *)
(* x + 1 does not type-check *)
```



- 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

References example

```
val x = ref 42
val y = ref 42
val z = x
val _ = x := 43
val w = (!x) + (!z) (* 85 *)
```



- *p = a + b;
  y = a + b;
- If *p aliases a or b, then second computation of a+b is not redundent

- x = 3; *p = 4; y = x;
- Is y constant? If *p and x do not alias each other, then yes. If *p and x always alias each other, then yes. If *p and x sometimes alias each other, then no.

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 ())
```

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

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