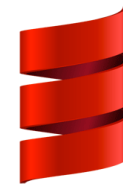
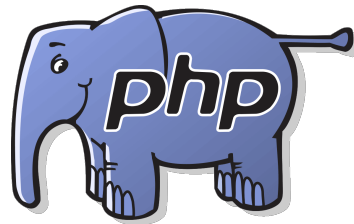


CS 360

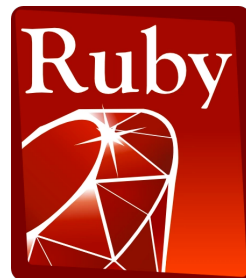
Programming Languages Day 13 – Dynamic Scope, Closure Idioms



Scala



Swift



Lexical scoping vs dynamic scoping

- The alternative to lexical scoping is called ***dynamic scoping***.
- In lexical (static) scoping, if a function *f* references a non-local variable *x*, the language will look for *x* in the environment where *f* was ***defined***.
- In dynamic scoping, if a function *f* references a non-local variable *x*, the language will look for *x* in the environment where *f* was ***called***.
 - If it's not found, will look in the environment that called the function that called *f* (and so on).

Example

- Assume we have a Python/C++-style language.
- What does this program print under lexical scoping?
 - 5, 5
- What does this program print under dynamic scoping?
 - 5, 10

```
x = 5

def foo():
    print(x)

def bar():
    x = 10
    foo()

foo()
bar()
```

Why do we prefer lexical over dynamic scope?

1. Function meaning does not depend on variable names used.

Example: Can rename variables at will, as long as you are consistent.

– Lexical scope: guaranteed to have no effects.

Dynamic scope: might change the function meaning.

```
(define (f x)
  (lambda (y) (+ x y)))
```

When the anonymous function that `f` returns is called, in lexical scoping, we always know where the values of `x` and `y` will be (what frames they're in). With dynamic scoping, `x` will be searched for in the functions that called the anonymous function, so who knows what frames they'll be in.

Why do we prefer lexical over dynamic scope?

1. Function meaning does not depend on variable names used.

Example: Can remove unused variables in lexical scoping.

- Dynamic scope: May change meaning of a function (weird)

```
(define (f g)
  (let ((x 3))
    (g 2)))
```

- You would never write this in a lexically-scoped language, because the binding of `x` to 3 is never used.
 - (No way for `g` to access this particular binding of `x`.)
- In a dynamically-scoped language, function `g` might refer to a non-local variable `x`, and this binding might be necessary.

Why do we prefer lexical over dynamic scope?

2. Easy to reason about functions where they're defined.

```
(define x 1)

(define (f y)
  (+ x y))

(define (g)
  (let ((x "hello"))
    (f 4)))
```

Example: Dynamic scope tries to add a string to a number
(b/c in the call to (+ x y), x will be "hello")

In lexical scope, we always know what function f does even before the program is compiled or run.

Why do we prefer lexical over dynamic scope?

3. Closures can easily store the data they need.

- Many more examples and idioms to come.

```
(define (gteq x) (lambda (y) (>= y x)))  
(define (no-negs lst) (filter (gteq 0) lst))
```

- The anonymous function returned by `gteq` references a non-local variable `x`.
- In lexical scoping, the closure created for the anonymous function will point to `gteq`'s frame so `x` can be found.
- In dynamic scoping, who knows what `x` would be. Makes it impossible to use this functionality.

Why 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 (e.g., exception handling).
 - So some languages (e.g., Racket) have special ways to do it.
 - But most don't bother.
- Historically, dynamic scoping was used more frequently in older languages because it's easier to implement than lexical scoping.
 - Strategy: Just search through the call stack until variable is found. No closures needed.
 - Call stack maintains list of functions that are currently being called, so might as well use it to find non-local variables.

Iterators made better

- Functions like **map** and **filter** are *much* more powerful thanks to closures and lexical scope
- Function passed in can use any “private” data in its environment
- Iterator (e.g., map or filter) “doesn’t even know the data is there”
 - It just calls the function that it's passed, and that function will take care of everything.

```
(define (gteq x) (lambda (y) (>= y x)))  
(define (no-negs lst) (filter (gteq 0) lst))
```

More idioms

- We know the rules for lexical scope and function closures.
 - Now we'll see what it's good for.

A partial but wide-ranging list:

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

Currying and Partial Application

- Currying is the idea of calling a function with an incomplete set of arguments.
- When you "curry" a function, you get a function back that accepts the remaining arguments.
- Named after Haskell Curry, who studied related ideas in logic.
 - PL Haskell is named after him.



 **Haskell**

Currying and Partial Application: Example

- We know **(expt x y)** raises **x** to the **y**'th power.
- We could define a curried version of **expt** like this:
- **(define (expt-curried x)
 (lambda (y) (expt x y)))**
- We can call this function like this:
((expt-curried 4) 2)
- This is useful because **expt-curried** is now a function of a single argument that can make a family of "raise-this-to-some-power" functions.
- This is critical in some other functional languages (though not Racket or Scheme) where functions may have at most one argument.

Currying and Partial Application

- Currying is still useful in Racket with the **curry** function:
 - Takes a function **f** and (optionally) some arguments **a1, a2, ...**
 - Returns an anonymous function **g** that accumulates arguments to **f** until there are enough to call **f**.
- **(curry expt 4)** returns a function that raises 4 to its argument.
 - **(curry expt 4) == expt-curried**
 - **((curry expt 4) 2) == ((expt-curried 4) 2)**
- **(curry * 2)** returns a function that doubles its argument.
- These can be useful in definitions themselves:
 - **(define (double x) (* 2 x))**
 - **(define double (curry * 2))**

Currying and Partial Application

- Currying is also useful to shorten longish lambda expressions:
- Old way: `(map (lambda (x) (+ x 1)) '(1 2 3))`
- New way: `(map (curry + 1) '(1 2 3))`
- Great for encapsulating private data: (*below, list-ref is the built-in get-nth.*)

```
(define get-month  
  (curry list-ref '(Jan Feb Mar Apr May Jun  
                  Jul Aug Sep Oct Nov Dec)))
```

Currying and Partial Application

- But this gives zero-based months:

- ```
(define get-month
 (curry list-ref
 '(Jan Feb Mar Apr May Jun
 Jul Aug Sep Oct Nov Dec)))
```

- Let's subtract one from the argument first:

```
(define get-month
 (compose
 (curry list-ref
 '(Jan Feb Mar Apr May Jun
 Jul Aug Sep Oct Nov Dec))
 (curryr - 1)))
```

**curryr** curries  
from right to left,  
rather than left to  
right.

# Currying and Partial Application

- A few more examples:
- `(map (compose (curry + 2) (curry * 4)) '(1 2 3))`
  - quadruples then adds two to the list '(1 2 3)
- `(filter (curry < 10) '(6 8 10 12))`
  - Careful! `curry` works from the left, so `(curry < 10)` is equivalent to `(lambda (x) (< 10 x))` so this filter keeps numbers that are greater than 10.
- Probably clearer to do:  
`(filter (curryr > 10) '(6 8 10 12))`
- (In this case, the confusion is because we are used to "<" being an infix operator).



## *Return to the foldr* 😊

Currying becomes really powerful when you curry higher-order functions.

Recall `(foldr f init (x1 x2 ... xn))` returns

```
(f x1 (f x2 ... (f xn-2 (f xn-1 (f xn init))))
```

```
(define (sum-list-ok lst) (foldr + 0 lst))
```

```
(define sum-list-super-cool (curry foldr + 0))
```

## *Another example*

- Scheme and Racket have **andmap** and **ormap**.
- **(andmap f (x1 x2...))** returns **(and (f x1) (f x2) ...)**
- **(ormap f (x1 x2...))** returns **(or (f x1) (f x2) ...)**

```
(andmap (curryr > 7) '(8 9 10)) → #t
```

```
(ormap (curryr > 7) '(4 5 6 7 8)) → #t
```

```
(ormap (curryr > 7) '(4 5 6)) → #f
```

```
(define contains7 (curry ormap (curry = 7)))
```

```
(define all-are7 (curry andmap (curry = 7)))
```

## Another example

Currying and partial application can be convenient even without higher-order functions.

*Note: (**range a b**) returns a list of integers from a to b-1, inclusive.*

```
(define (zip lst1 lst2)
 (if (null? lst1) '()
 (cons (list (car lst1) (car lst2))
 (zip (cdr lst1) (cdr lst2)))))

(define countup (curry range 1))

(define (add-numbers lst)
 (zip (countup (length lst)) lst))
```

## *When to use currying*

- When you write a lambda function of the form
  - **(lambda (y1 y2 ...) (f x1 x2 ... y1 y2...))**
- You can replace that with
  - **(curry f x1 x2 ...)**
  
- Similarly, replace
  - **(lambda (y1 y2 ...) (f y1 y2 ... x1 x2...))**
- with
  - **(curryr f x1 x2 ...)**

# *When to use currying*

- Try these:
  - Assuming **lst** is a list of numbers, write a call to **filter** that keeps all numbers greater than 4.
  - Assuming **lst** is a **list of lists of numbers**, write a call to **map** that adds a 1 to the front of each sublist.
  - Assuming **lst** is a list of numbers, write a call to **map** that turns each number (in **lst**) into the list (1 number).
  - Assuming **lst** is a list of numbers, write a call to **map** that squares each number (you should curry **expt**).
  - Define a function **dist-from-origin** in terms of currying a function (**dist x1 y1 x2 y2**) [assume **dist** is already defined elsewhere]
- Hint: Write each without currying, then replace the lambda with a curry.

# 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
- (Can accomplish this in C++ with objects that contain private fields.)

# *Mutable state*

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

- We really do want the “callbacks registered” and “events that have been delivered” to *change* due to function calls

In "pure" functional programming, there is no mutation.

- Therefore, it is **guaranteed** that calling a function with certain arguments will always return the same value, no matter how many times it's called.
- Not guaranteed once mutation is introduced.
- This is why global variables are considered "bad" in languages like C or C++ (global constants OK).

## *Mutable state: Example in C++*

```
times_called = 0

int function() {
 times_called++;
 return times_called;
}
```

This is useful, but can cause big problems if somebody else modifies **times\_called** from elsewhere in the program.



# *Mutable state*

- Scheme and Racket's variables are mutable.
- The name of any function which does mutation contains a "!"
- Mutate a variable with **set!**
  - Only works after the variable has been placed into an environment with **define**, **let**, or as an argument to a function.
  - **set!** does not return a value.

```
(define times-called 0)
```

```
(define (function)
```

```
 (set! times-called (+ 1 times-called))
```

```
 times-called)
```

- Notice that functions that have side-effects or use mutation are the only functions that need to have bodies with more than one expression in them.

## *Example Racket GUI with callback*

```
; Make a frame by instantiating the frame% class
(define frame (new frame% (label "Example")))
; Make a static text message in the frame
(define msg (new message% (parent frame)
 (label "No events so far...")))
; Make a button in the frame
(new button% (parent frame)
 (label "Click Me")
 (callback (lambda (button event)
 (send msg set-label
 (number->string (function))))))
; Show the frame by calling its show method
(send frame show #t)
```

## *Example Racket GUI with callback*

Key code:

```
(new button% (parent frame)
 (label "Click Me")
 (callback (lambda (button event)
 (send msg set-label
 (number->string (function))))))
```

```
(define times-called 0)
(define (function)
 (set! times-called (+ 1 times-called))
 times-called)
```

## *Avoid cluttering the global frame*

Key code:

```
(new button% (parent frame2)
 (label "Click Me")
 (callback (let ((count-clicks 0))
 (lambda (button event)
 (set! count-clicks (+ 1 count-clicks))
 (send msg2 set-label
 (number->string count-clicks))))))
```

## *How does that work?*

- What does the environment diagram for these look like?

```
(define (f x)
 (let ((y 1))
 (lambda (y) (+ x y z))))
```

```
(define g
 (let ((x 1))
 (lambda (y) (+ x y))))
```

- This idea is called let-over-lambda. Used to make local variables in a function that persist between function calls.

# *Implementing an ADT*

As our last pattern, closures can implement abstract data types

- They can share the same private data
- Private data can be mutable or immutable
- Feels quite a bit like objects, emphasizing that OOP and functional programming have similarities

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

- Combines lexical scope, closures, and higher-level functions
- Client use is not so tricky

```

(define (new-stack)
 (let ((the-stack '()))
 (define (dispatch method-name)
 (cond ((eq? method-name 'empty?) empty?)
 ((eq? method-name 'push) push)
 ((eq? method-name 'pop) pop)
 (#t (error "Bad method name"))))
 (define (empty?) (null? the-stack))
 (define (push item) (set! the-stack (cons item the-
stack)))
 (define (pop)
 (if (null? the-stack) (error "Can't pop an empty
stack")
 (let ((top-item (car the-stack)))
 (set! the-stack (cdr the-stack))
 top-item)))
 dispatch)) ; this last line is the return value
 ; of the let statement at the top.

```