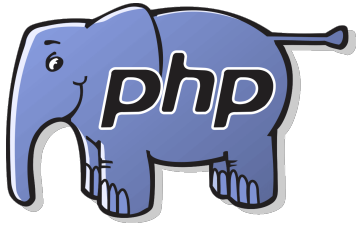
CS 360
Programming Languages
Day 14 – Closure Idioms









Why lexical scope rocks

- Last time: currying

- Today: implementing callbacks and object-oriented programming.

Review: mutable state

- Racket's variables are mutable.

- The name of any function which mutates something contains a "!"

- Mutate a variable with **`set!`**
  - **`(set! variable new-value)`**

  - 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.

Review: mutable state

```
(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.

- Wouldn't it be nice to not have the times-called variable cluttering up the global frame?

Lexical scope to the rescue!

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

- Why does this not work?
  - The let is executed when the function is *called*. We want it to be executed when the function is *defined*.

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

Example use: callbacks

- A common idiom: Library takes functions to apply later, when an *event* occurs:
  - When a key is pressed, mouse moves, data arrives.
  - When the program enters or leaves some state (e.g., a turn in a game begins or ends).

- Most callback libraries use a higher-order function to setup a callback.

Example Racket GUI with callback 1

```
; Make a frame by instantiating the frame% class
(define frame (new frame% (label "Example")))

; Make a button in the frame
(define btn (new button% (parent frame)
  (label "Click Me")
  (callback (lambda (button event)
              (send btn set-label "Hello!")))))

; Show the frame by calling its show method
(send frame show #t)
```

Example Racket GUI with callback 2

```
; Make a frame by instantiating the frame% class
(define frame (new frame% (label "Example")))

; Make a button in the frame
(define btn (new button% (parent frame)
  (label "Click Me")
  (callback (lambda (button event)
              (send btn set-label
                 (number->string (function)))))))

; Show the frame by calling its show method
(send frame show #t)
```

Example Racket GUI with callback 3

```
; Make a frame by instantiating the frame% class
(define frame (new frame% (label "Example")))

; Make a button in the frame
(define btn (new button% (parent frame)
  (label "Click Me")
  (callback (let ((count-clicks 0))
              (lambda (button event)
                (set! count-clicks (+ 1 count-clicks))
                (send btn set-label
                   (number->string count-clicks)))))))

; Show the frame by calling its show method
(send frame show #t)
```

Implementing an ADT

As our last pattern, closures can implement psuedo-classes in an object-oriented style.

- "Pseudo" because you don't have things like polymorphism, public/private variables, etc.
- Good illustration of the power of closures.

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.
```

New stuff!

- A little more about set! and mutation.

- Delayed evaluation.

Set!

- Yes, Racket really has assignment statements
  - But used *only-when-really-appropriate!*

```
(set! x e)
```

- For the **x** in the current environment, subsequent lookups of **x** get the result of evaluating expression **e**
  - Any code using this **x** will be affected
  - Like C++/Python's **x = e**

- Once you have side-effects, sequences are useful:

```
(begin e1 e2 … en)
```

Example

Example uses **`set!`** at top-level; mutating local variables is similar

```
(define b 3)
(define f (lambda (x) (* 2 (+ x b))))
(define c (+ b 4))
(set! b 5)
(define z (f 4))
(define w c)
```

Not much new here:

- Environment for closure determined when function is defined, but body is evaluated when function is called

Top-level

- Mutating top-level definitions is particularly problematic
  - What if any code could do **`set!`** on anything?
  - How could we defend against this?

- A general principle: If something you need not to change might change, make a local copy of it. Example:

```
(define b 3)
(define f
  (let ((b b))
    (lambda (x) (* 2 (+ x b)))))
```

- Could use a different name for local copy but do not need to.
- Called defensive copying --- used often in languages like C++ and Java.

cons cells are immutable

What if you wanted to mutate the *contents* of a cons cell?

- In Racket you can't (major change from Scheme)
- This is good
  - List-aliasing irrelevant
  - Implementation can make a fast **`list?`** since listness is determined when cons cell is created

This does *not* mutate the contents:

```
(define x (cons 14 '()))
(define y x)
(set! x (cons 42 '()))
(define fourteen (car y))
```

- Like C++: **`x = Cons(42,null)`**, not **`x.car = 42`**

mcons cells are mutable

Since mutable pairs are sometimes useful (will use them later in class), Racket provides them too:

- **`mcons`**
- **`mcar`**
- **`mcdr`**
- **`mpair?`**
- **`set-mcar!`**
- **`set-mcdr!`**

Run-time error to use **`mcar`** on a cons cell or **`car`** on a mcons cell