Programming Languages

Environment Diagrams Again, Mutation, Pairs, Thunks, Laziness, Streams, Memoization

Adapted from Dan Grossman's PL class, U. of Washington

Env. Diagram practice, now with mutation!

- Get into groups.
- Draw the environment diagram that would result from running the code on the next slide.

```
(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))
(define S (new-stack))
((S ‘push) 5)
```

Today

Primary focus: Powerful programming idioms related to:

- Delaying evaluation (using functions)
- Remembering previous results (using mutation)

Lazy evaluation, Streams, Memoization

But first need to discuss:

- Review of mutation in Racket
- mcons cells (mutable pairs)

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) (* 1 (+ x b))))
(define c (+ b 4)) ; 7
(set! b 5)
(define z (f 4))   ; 9
(define w c)       ; 7
```

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) (* 1 (+ x b)))))
```

Could use a different name for local copy but do not need to

But wait…

- Simple elegant language design:
  - Primitives like **+** and ***** are just predefined variables bound to functions
  - But maybe that means they are mutable
  - Example continued:

```
(define f
  (let ([b b]
        [+ +]
        [* +])
    (lambda (x) (* 1 (+ x b)))))
```

  - Even that won't work if **f** uses other functions that use things that might get mutated – all functions would need to copy everything mutable they used

No such madness

In Racket, *you do not have to program like this*

- Each file is a module
- *If* a module does not use **`set!`** on a top-level variable, then Racket makes it constant and forbids **`set!`** outside the module
- Primitives like **`+`**, **`*`**, and **`cons`** are in a module that does not mutate them

In Scheme, you really could do **`(set! + cons)`**

- Naturally, nobody defended against this in practice so it would just break the program

Showed you this for the *concept* of copying to defend against mutation

A bit about **`cons`**

`cons` just makes a pair

- By convention and standard library, lists are nested pairs that eventually end with **`` `() ``**

```
(define pr (cons 1 (cons #t "hi"))) ; '(1 #t . "hi")
(define hi (cdr (cdr pr)))
(define false (list? pr))
(define true (pair? pr))
(define lst (cons 1 (cons #t (cons "hi" `()))))
(define hi2 (car (cdr (cdr pr))))
```

Passing an improper list to functions like **`length`** is a run-time error

So why allow improper lists?

- Pairs are useful (can make another data structures)

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 lecture), 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

You've been lied to

- Everything that looks like a function call in Racket is not necessarily a function.
- Everything that looks like a function is either
  - A function call (as we thought)
  - Or a "special form"
- Special forms: define, let, lambda, if, cond, and, or, …
- Why can't these be functions?
- Recall the evaluation model for a function call:
  - (f e1 e2 e3…): evaluate e1 e2 … to obtain values v1 v2…, then evaluate f to get a closure, then evaluate the code of the closure with its arguments bound to v1 v2…
  - Why would this not work for defining if?

Delayed evaluation

In Racket, function arguments are eager (call by value)
Special form arguments are lazy (call by need)

- Delay evaluation of the argument until we really need its value

Why wouldn't these functions work?

```
(define (my-if-bad x y z)
  (if x y z))

(define (fact-wrong n)
    (my-if-bad (= n 0)
               1
               (* n (fact-wrong (- n 1)))))
```

Thunks

We know how to delay evaluation: put expression in a function definition!

- Because defining a function doesn't run the code until later.

A zero-argument function used to delay evaluation is called a *thunk*

- As a verb: *thunk the expression*

This works (though silly to re-define **`if`** like this):

```
(define (my-if x y z)
  (if x (y) (z)))

(define (fact n)
    (my-if (= n 0)
           (lambda() 1)
           (lambda() (* n (fact (- n 1))))))
```