Programming Languages

Lecture 3
Local bindings and lambda, plus Benefits of No Mutation

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

Review

Huge progress in 2 lectures on the core pieces of Racket (Scheme):

- Variables and environments
  - **`(define variable expression)`**
- Functions
  - Build: **`(define (f x1 x2 …) e)`**
  - Use: **`(f e1 … en)`**
- Tuples
  - Build: **`(cons e1 e2)`** OR **`'(v1 . v2)`**
  - Use: **`(car e)`**, **`(cdr e)`**
- Lists
  - Build: **`'()`** **`(cons e1 e2)`** OR **`'(v1 v2 v3 …)`**
    `(list e1 e2 …)` **`(append e1 e2 …)`**
  - Use: **`(null? e)`** **`(car e)`** **`(cdr e)`**

Today

- The big thing we need: local bindings
  - For style and convenience
  - For efficiency (***not*** "just a little faster")
  - A big but natural idea: nested function bindings

- Why not having mutation (assignment statements) is a valuable language feature
  - No need for you to keep track of sharing/aliasing, which C++ programmers must obsess about
  - What makes global variables "bad" in most languages (languages that allow mutation)

Let-expressions

The construct for introducing local bindings is ***just an expression***, so we can use it anywhere we can use an expression

- Syntax: `(let ((var1 e1) (var2 e2) …) e)`
  - Each var_i is any *variable name,* each e_i is any *expression,* and **e** is also any *expression*.
- Evaluation: Evaluate each e_i, assign each e_i to var_i (all at once) in an environment that includes the bindings from the enclosing environment.
- Result of whole let-expression is result of evaluating **e** in the new environment.

Silly examples

```
(define (silly1 z)
  (let ((x 5))
       (+ x z)))

; this one won't work!
(define (silly2 z)
  (let ((x 5) (answer (+ x z)))
       answer))

(define (silly2-fixed z)
  (let* ((x 5) (answer (+ x z)))
       answer))
```

Silly examples

```
(define (silly3 z)
   (let* ((x (if (> z 0) z 4)) (y (+ x 1)))
        (if (> x y) (* 2 x) (* y y))))

(define (silly4)
   (let ((x 1))
        (+
               (let ((x 2)) (+ x 1))
               (let ((y (+ x 2))) (+ y 1)))))
```

`silly4` is poor style but shows let-expressions are expressions

- Could also use them in function-call arguments, parts of conditionals, etc.
- Also notice shadowing

What's new

- What's new is *scope*: contexts within a program where a variable has a value.
  - Variables bound using **`let`** can be used in the body of the let-expression.
  - Variables bound using **`let*`** can be used in the body of let-expression and in later bindings in the same **`let*`**.
  - Bindings in **`let`**/**`let`*** *shadow* bindings of the same variable name from the enclosing environment(s).

- *Nothing else is new:*
  - Can put any binding we want, even function bindings
  - Evaluation rules just like at "top-level" with (define…)

Nested functions, part 1

- Good style to define helper functions inside the functions they help if they are:
  - Unlikely to be useful elsewhere
  - Likely to be misused if available elsewhere
  - Likely to be changed or removed later

- A fundamental trade-off in code design: reusing code saves effort and avoids bugs, but makes the reused code harder to change later

- But we need some additional syntax…

Nested functions, part 1

- let and let* don't let you define function bindings using the same variations that define does:
  - **`(define var expr)`** OK
  - **`(define (func x1 x2…) body-expr)`** OK
  - **`(let ((var expr) (var expr)…) expr)`** OK
    - Can't do **`(let (((func x1 x2…) body-expr) …) expr)`** NO
  - Note that define statements are *not* expressions, so they don't evaluate to values.
  - Can't do **`(let ((func (define …`** NO

Nested functions, part 1

```
(let ((var1 e1) (var2 e2) …) e)
```

- We have expressions that evaluate to numbers: 34, (+ 4 5), (abs -9)
- We have expressions that evaluate to booleans: #t, #f, (> 4 5)
- Functions are first-class citizens in Racket (and Scheme), so we need an expression that evaluates to a function!

- Technically, we already have one: the name of a previously-defined function:

```
(define (silly5 n)
   (let ((my-function abs))
      (my-function n)))
```

  – But that's not particularly useful.

Lambda expressions

- Function to create functions: **`lambda`**
- Syntax:
  - **`(lambda (x1 x2 …) e)`**
- Evaluation:
  - Creates an *anonymous* (un-named) function that takes arguments ***`x1, x2, …`*** and whose body is ***`e`***.
  - This new function is a value, so **`(lambda …)`** is a value.
- For now, we will immediately bind these anonymous functions to names with either **`define`** or **`let`**/**`let*`**.
  - (This is not a rule of Racket or Scheme, though.)
  - (It is possible to call an anonymous function even if it has no name and has not been bound to a variable.) LATER

Lambda expressions

- The `define` variant for functions is "syntactic sugar" for lambda:

```
(define (double n)
    (* 2 n))

(define double
    (lambda (n) (* 2 n)))
```

- These are 100% equivalent!

Using lambda in a let expression

- Define will "handle" recursive anonymous functions:

```
(define count-up (lambda (from to)
   (if (= from to)
      (cons from '())
      (cons from (count-up (+ 1 from) to)))))
```

- But let/let* won't:

```
(define (count-up-from-one x)
   (let ((count-up (lambda (from to)
        (if (= from to)
           (cons from '())
           (cons from (count-up (+ 1 from) to))))))
     (count-up 1 x)))
```

Using lambda in a let expression

- When using **`let`** to define a recursive local function, use **`letrec`**:

```
(define (count-up-from-one x)
   (letrec ((count-up (lambda (from to)
        (if (= from to)
           (cons from '())
           (cons from (count-up (+ 1 from) to))))))
     (count-up 1 x)))
```

- Or nested defines:

```
(define (count-up-from-one x)
   (define (count-up from to)
     (if (= from to)
        (cons from '())
        (cons from (count-up (+ 1 from) to))))
   (count-up 1 x))
```

(Inferior) Example

```
(define (count-up-from-one x)
    (define (count-up from to)
        (if (= from to)
            (cons from '())
            (cons from (count-up (+ 1 from) to))))
    (count-up 1 x))
```

- This shows how to use a local function binding, but:
  - Will show a better version next
  - **`count-up`** might be useful elsewhere

Nested functions, better

- Functions can use any binding in the environment where they are defined:
  - Bindings from “outer” environments
    - Such as parameters to the outer function
  - Earlier bindings in let* (but not let)
- Usually bad style to have unnecessary parameters
  - Like **`to`** in the previous example

```
(define (count-up-from-one-better x)
  (define (count-up from)
      (if (= from x)
              (cons from '())
              (cons from (count-up (+ 1 from)))))
  (count-up 1))
```

Avoid repeated recursion

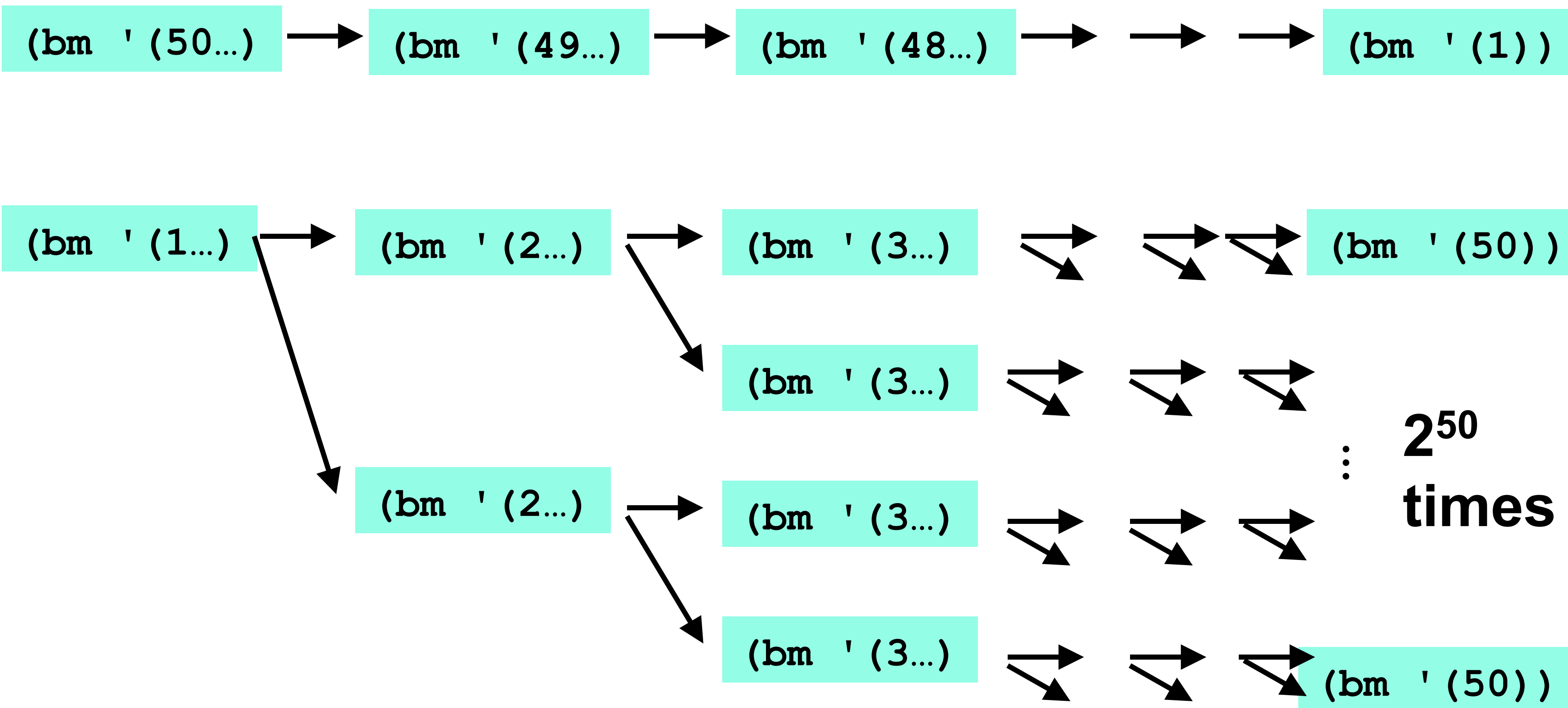
Consider this code and the recursive calls it makes

- Don't worry about calls to **`null?`**, **`car`**, and **`cdr`** because they do a small constant amount of work

```
(define (bad-max lst)
  (cond
      ((null? (cdr lst))
            (car lst))
      ((> (car lst) (bad-max (cdr lst)))
            (car lst))
      (#t
            (bad-max (cdr lst)))))

(define x (bad-max '(50 49 48 … 1)))
(define y (bad-max '(1 2 3 … 50)))
```

Fast vs. unusable

```
((> (car lst) (bad-max (cdr lst)))
    (car lst))
(#t (bad-max (cdr lst))))
```



Math never lies

Suppose one **`bad-max`** call's if-then-else logic and calls to **`car`**, **`cdr`**, and **`null?`** take 10^{-7} seconds

- Then **`(bad-max '(50 49 … 1))`** takes 50 x 10^{-7} seconds
- And **`(bad_max '(1 2 … 50))`** takes 2.25 x 10^{8} seconds
  - (over 7 years)
  - **`(bad-max '(55 54 … 1))`** takes over 2 centuries
  - Buying a faster computer won't help much ☺

The key is not to do repeated work that might do repeated work that might do…

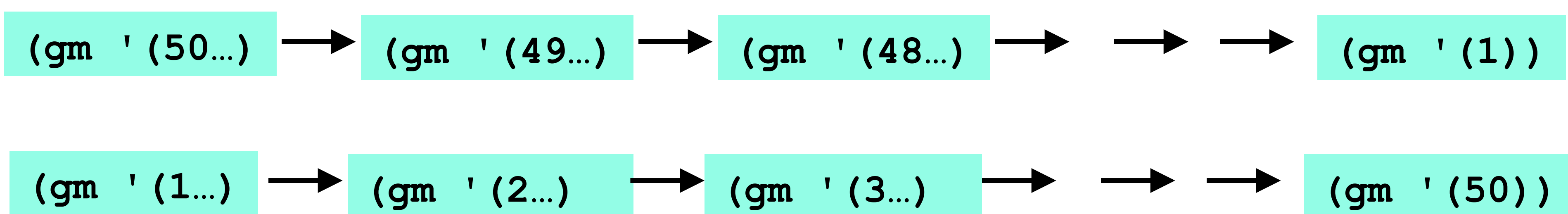
- Saving recursive results in local bindings is essential…

Efficient max

```
(define (good-max lst)
  (cond
    ((null? (cdr lst))
      (car lst))
    (#t
      (let ((max-of-cdr (good-max (cdr lst))))
        (if (> (car lst) max-of-cdr)
            (car lst)
           max-of-cdr)))))
```

Fast vs. fast

```
(let ((max-of-cdr (good-max (cdr lst))))
     (if (> (car lst) max-of-cdr)
       (car lst)
       max-of-cdr))
```



A valuable non-feature: no mutation

Those are all the features you need (and should use) on hw1

Now learn a very important non-feature

- Huh?? How could the *lack* of a feature be important?
- When it lets you know things *other* code will *not* do with your code and the results your code produces

A major aspect and contribution of functional programming:

Not being able to assign to (a.k.a. mutate) variables or parts of tuples and lists

Suppose we had mutation…

```
(define x '(4 . 3))
(define y (sort-pair x))

somehow mutate (car x) to hold 5

(define z (car y))
```

- What is **`z`**?
  - Would depend on how we implemented **`sort-pair`**
    - Would have to decide carefully and document **`sort-pair`**
  - But without mutation, we can implement "either way"
    - No code can ever distinguish aliasing vs. identical copies
    - No need to think about aliasing: focus on other things
    - Can use aliasing, which saves space, without danger

Interface vs. implementation

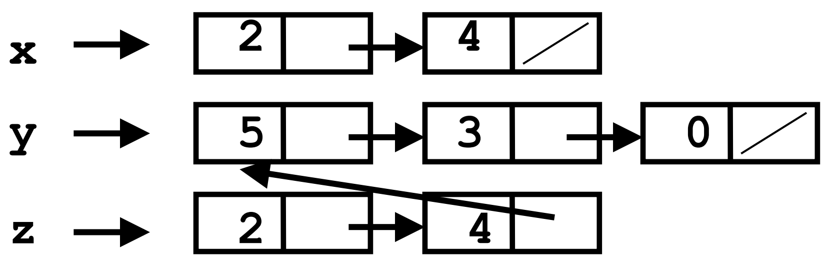
In Racket, these two implementations of **`sort-pair`** are indistinguishable

- But only because tuples are immutable
- The first is better style: simpler and avoids making a new pair in the then-branch

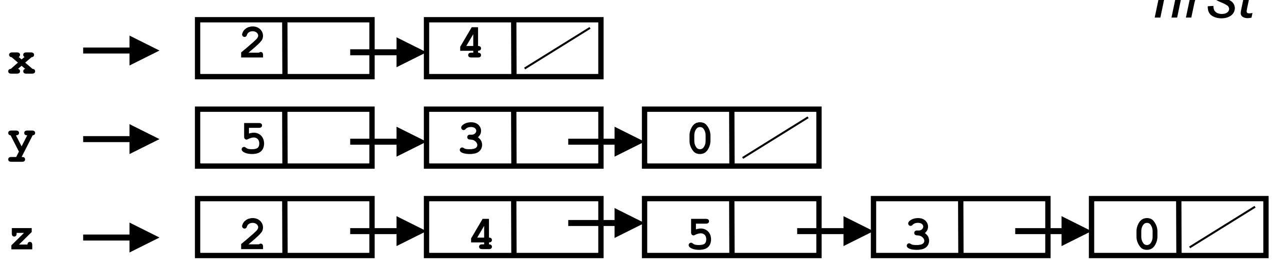
```
(define (sort-pair pair)
  (if (> (car pair) (cdr pair))
      pair
      (cons (cdr pair) (car pair))))

(define (sort-pair pair)
  (if (> (car pair) (cdr pair))
      (cons (car pair) (cdr pair))
      (cons (cdr pair) (car pair))))
```

An even better example

```
(define (my-append lst1 lst2)
  (if (null? lst1)
      lst2
      (cons (car lst1) (append (cdr lst1) lst2))))
(define x '(2 4))
(define y '(5 3 0))
(define z (append x y))
```



or



(can't tell, but it's the first one)

Racket vs. C++ on mutable data

- In Racket, we create aliases all the time without thinking about it because it is *impossible* to tell where there is aliasing
  - Example: **`cdr`** is constant time; does not copy rest of the list
  - So don't worry and focus on your algorithm

- In C++, we have to think about the implications of mutability, which often forces us to copy manually.
  - Hence why we have pass by reference **and** pass by value
  - And then you have pass by const reference to simulate pass by value but not waste time copying…
    - e.g., compare(const string& s1, const string& s2)