CS 360
Programming Languages
Day 5

Today

- Local bindings
  - We will see these for variables and functions.

- Benefits of no mutation
  - No need for you to keep track of sharing/aliasing, which Java (and sometimes Python) 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.
- Key idea: a let-expression allows you to make local variables and evaluate an expression with those variables. The variables disappear outside of the let-expression.

Syntax

```
(let ((a 1) (b 2))
     (+ a b))

  ==> 3
```

*"Shadows" bindings from **`define`**s outside the let:*

```
(define a 10)
(define c 30)
(let ((a 1) (b 2))
     (+ a b c))

  ==> 33
```

However, much more common to use let inside of a function definition...

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))
```

- Normal `let` creates and assigns all the local variables "***simultaneously***," so they cannot reference each other.
- `let*` creates and assigns variables ***sequentially***, so they can "see" each other.

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 the 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). *[defines or other lets]*

- ***Nothing else is new!***

How do we do this with functions?

- 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…

Local/nested functions

- **`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

Solution: internal defines

```
(define (f (x1 x2 ... xn)
  (define (f1 (y1 y2 ... yn) f1-body-expr)
  (define (f2 (z1 z2 ... zn) f2-body-expr)
  f-body-expr)
```

- How does this not conflict with the idea of function bodies only having one expression?
- An additional define is ***not*** an expression.
  - Expressions can be evaluated to values.
  - Defines are not expressions, and have no values.

Without looking at the handout…

- Let's create a function that produces a list of increasing numbers:
- Ex: **`(count-up 1 5)`** produces the list **`'(1 2 3 4 5)`**
- **`(define (count-up from to)`**
  … **`what goes here?`** …
- Base case? Recursive case?

(Inferior) Example

```
(define (count-up-from-one end)
    (define (count-up start end)
        (if (= start end)
            (cons start '())
            (cons start (count-up (+ 1 start) end))))
    (count-up 1 end))
```

- 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 ”**`end"`** in the previous example

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

Avoid repeated recursion

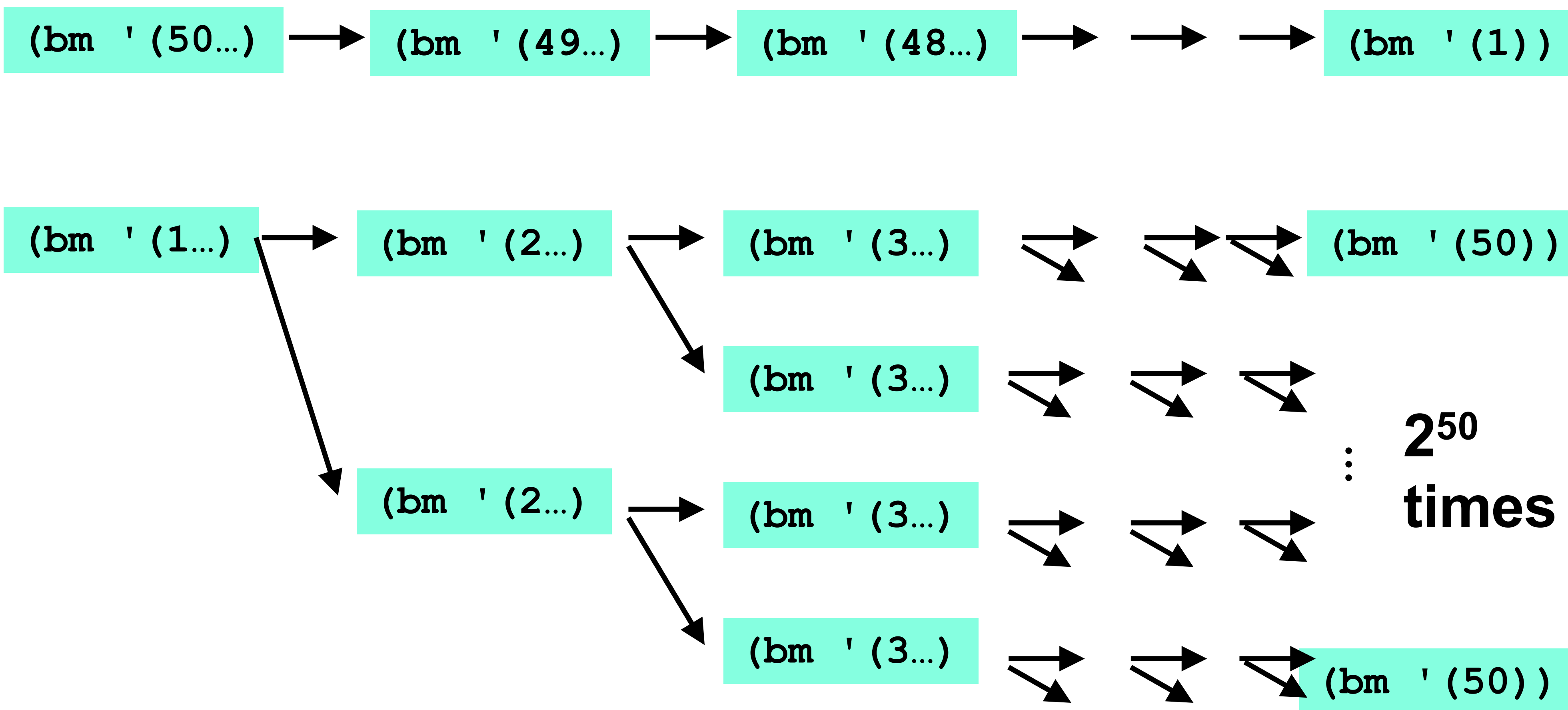
Consider this code and the recursive calls it makes

- 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 the **`cond`**, **`car`**, **`cdr`**, and **`null?`** parts of **`bad-max`** take 10^{-7} seconds total.

- 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…

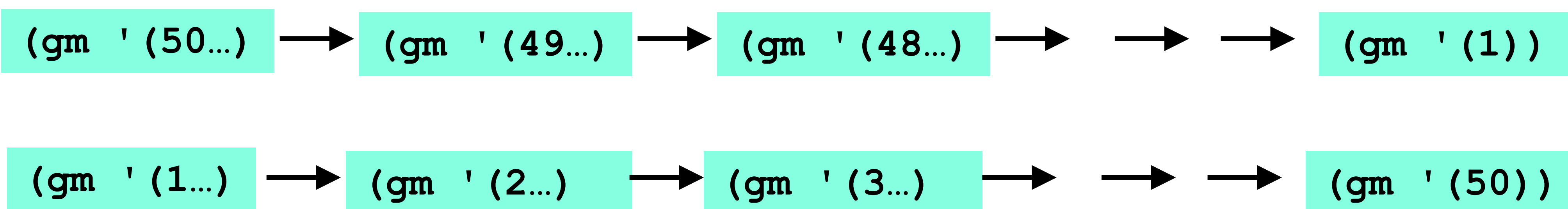
- Saving recursive results in local bindings is essential…

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

You now have all the features you need for project 1.

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…

```
; Recall that sort-pair takes a pair and returns
; an equivalent pair so that car > cdr.

(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

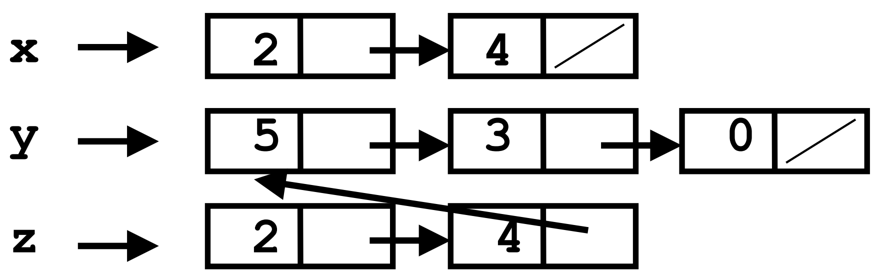
In Racket, these two implementations of **`sort-pair`** are indistinguishable

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

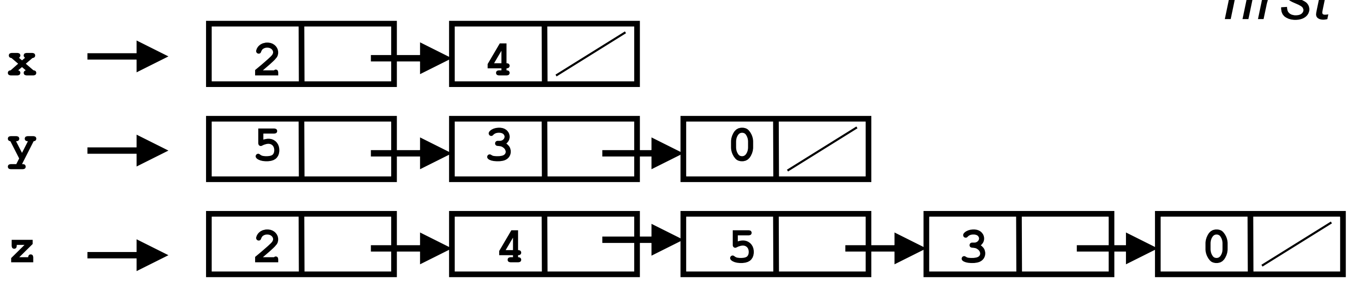
```
(define (sort-pair pair)
  (if (> (car pair) (cdr pair))
      pair
      (cons (cdr pair) (car pair))))

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

An even clearer example

```
(define (my-append lst1 lst2)
  (if (null? lst1)
      lst2
      (cons (car lst1) (my-append (cdr lst1) lst2))))
(define lst1 '(2 4))
(define lst2 '(5 3 0))
(define lst3 (my-append x y))
```



or



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

Racket vs. Python/Java 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 Python and Java, we have to think about the implications of mutability, which often forces us to copy manually.
  - E.g., passing references to objects around