Programming Languages
Lecture 5

Continuation of nested functions and why having no mutation is super cool and how dynamic typing is totally awesome (I'm really tired)

Review

- Use let for local variable definitions:

```
(let ((var1 value1)
      (var2 value2) ...)
     expression)
```

Review

- Use define for local function definitions:

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

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 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

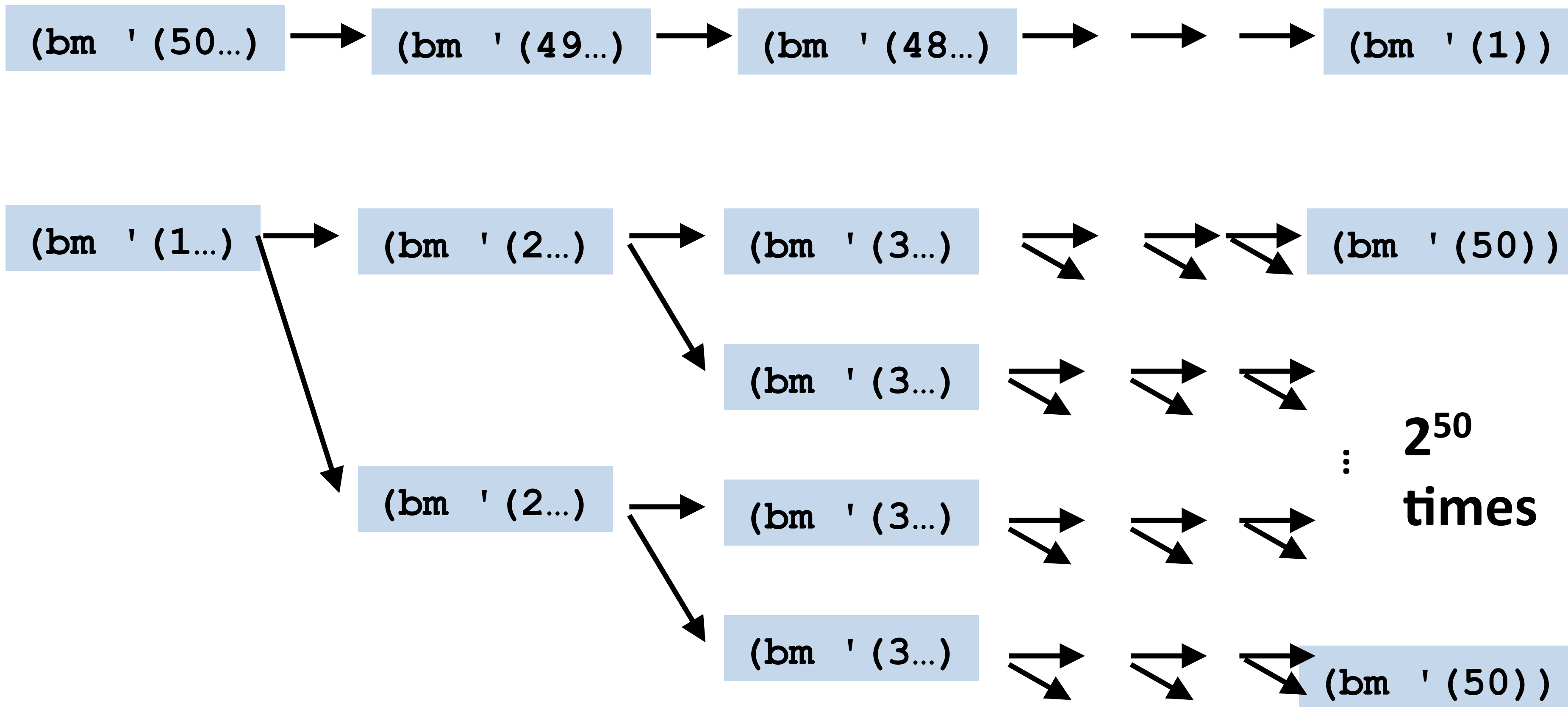
- Don't worry about calls to **`null?`**, **`car`**, and **`cdr`** because they do a small constant amount of work

```
(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…

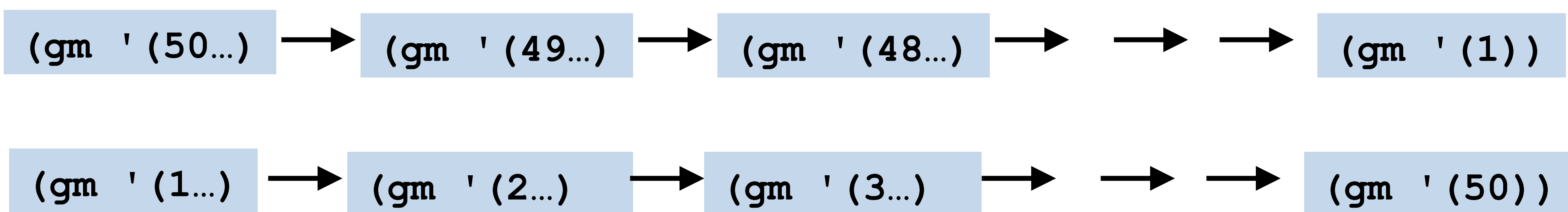
- Saving recursive results in local bindings is essential…

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 proj1

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

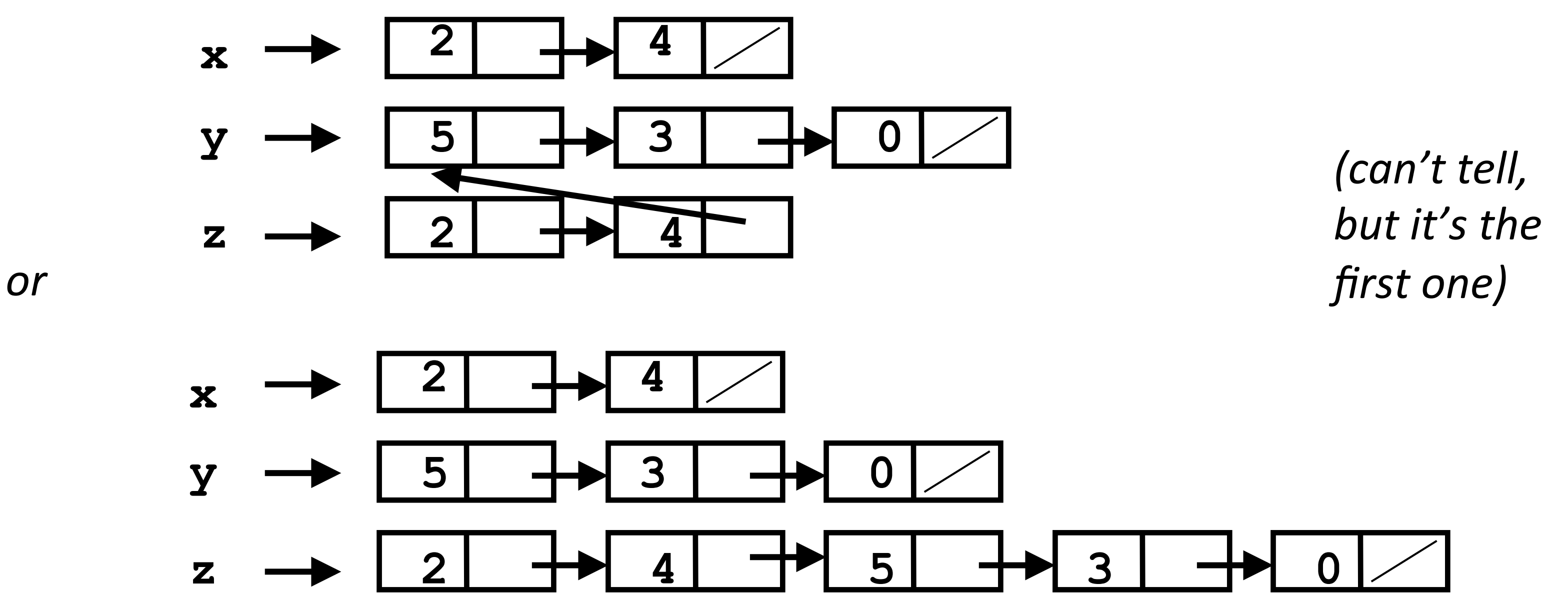
- But only because tuples are immutable
- The first is better style: simpler and avoids making a new pair in the then-branch

```
(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))
```



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++ (and sometimes Python), 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)

Dynamic typing vs static typing

Declaring functions in C++ vs Python

C++ uses ***static typing***: most code can be checked at compile-time to make sure rules involving types are not violated.

```
int double(int n) {
   return 2 * n;
}
```

Python uses ***dynamic typing***: most code cannot be checked for type errors at compile-time; this has be delayed until run-time.

```
def double(n):
   return 2 * n
```

Dynamic typing

- Racket (like most Scheme or Lisp dialects) is dynamically typed.
- Some characteristics of dynamic typing:
  - Values have types, but variables do not.
    - A variable can refer to different types during its lifetime.
  - Most type-error bugs cannot be found before the program is run, and not until the offending line of code is encountered.
    - Possible to write code with type errors that aren't discovered for a long time, if buried in code that isn't executed often.
  - Traditionally (but not always), dynamically-typed languages are interpreted, whereas statically-typed languages are compiled.

Some good things about dynamic typing

- Enables polymorphism (enabling code to handle any data type).
  - Example: Calculating the length of a list.

```
(define (length lst)
   (if (null? lst) 0 (+ 1 (length (cdr lst)))))
```

versus

```
int length_int_array(int_node* array) {
   if (array->next == NULL) return 0;
   else return 1 + length_int_array(array->next);
}
```

Easier to create flexible data structures

- In Racket, it's easy to create a list that can contain any other kind of data structure:
  - List of integers: '(1 2 3)
  - List of booleans: '(#f #f #t #f #t)
  - List of strings: '("a" "b" "c")
  - List of mixed types: '("a" 42 #f)
  - List of really mixed types: '(17 (3 #f) ("hi") -9 (1 (2 (3) 4 ())))
- Also, all of these lists will work with our length function!

- Mixing types in a single data structure is not easy in statically-typed languages.
- In C++, arrays or vectors must all hold the same type.

"Manual" type-checking

- Dynamically-typed languages often have some way for the programmer to discover the type of a variable.
- In Racket (all of these return #t or #f):
  - `number?`
    - `also integer?, rational?, real?`
  - `list?`
  - `pair?`
  - `string?`
  - `boolean?`
- Enables a single function to do different things depending on the type of an argument.

Length of a list vs length of nested lists

- For "regular" list
  - if empty list, return 0
  - else return 1 + length of the cdr of the list.

- For a list with possible nested lists…
  - if empty list, return 0
  - if the car of the list is a list…   do what?
  - else (car is not a list)…   do what?

Length of a list vs length of nested lists

- For "regular" list
  - if empty list, return 0
  - else return 1 + length of the cdr of the list.

- For a list with possible nested lists…
  - if empty list, return 0
  - if the car of the list is a list
    - return length of the car (which is a list) plus length of cdr
  - else (car is not a list)
    - return 1 + length of the cdr

Length of a list vs length of nested lists

```
(define (length-nested lst)
  (cond ((null? lst) 0)
        ((list? (car lst))
           (+ (length-nested (car lst))
              (length-nested (cdr lst))))
        (#t (+ 1 (length-nested (cdr lst))))))
```

Side effects

- In programming, a function has a side effect if it modifies some state or has an observable interaction with functions outside of itself (other functions or the outside world).
- Mutation is an example of a side effect.
  - Also: printing to the screen, modifying files, etc
- Functional programming (in Racket, Scheme, LISP) traditionally avoids side effects as much as possible.
  - Makes it much simpler to reason about how a program works.
  - Without side effects, calling a function with a fixed set of arguments is guaranteed to always return the same value.

Side effects

- In Racket, function bodies may contain more than one expression, if the extra expressions ***come first and are evaluated only for their side effects.***
  - In "pure" functional programming, you don't have side effects.
  - But it's nice to have this facility at times.
  - For debugging, can use (display <whatever>) and (newline)
- Example:

```
(define (length lst)
   (display lst)
   (newline)
   (if (null? lst) 0 (+ 1 (length (cdr lst)))))
```