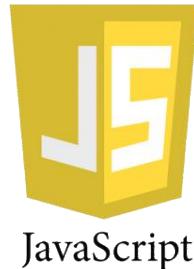
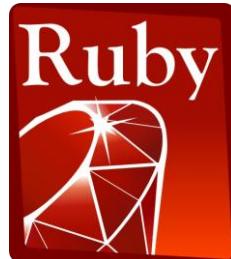


# CS 360

## Programming Languages

### Day 6



# *Today*

- Local bindings
  - We will see these for variables and functions.
- Slow Recursion
- Mutation
- First-Class Functions
  - What all is a function allowed to do or be?
- Anonymous Functions

# 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.

# 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  $\text{var}_i$  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  $\text{var}_i$  (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 `defines` 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!***

# *Avoid repeated recursion*

Consider this code and the recursive calls it makes

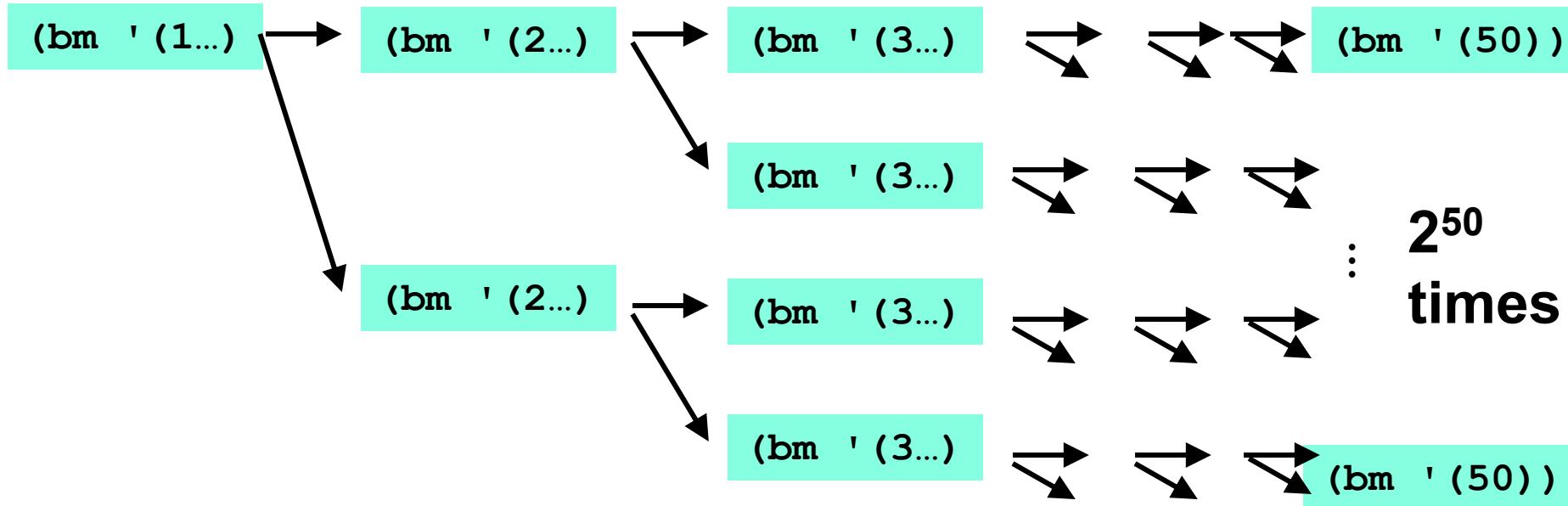
- Don't worry about calls to `null?`, `car`, and `cdr` because they do a small constant amount of work

```
(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  $50 \times 10^{-7}$  seconds
- And `(bad-max ' (1 2 ... 50))` takes  $2.25 \times 10^8$  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))
```

`(gm ' (50...)) → (gm ' (49...)) → (gm ' (48...)) → → → (gm ' (1))`

`(gm ' (1...)) → (gm ' (2...)) → (gm ' (3...)) → → → (gm ' (50))`

# 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))
```

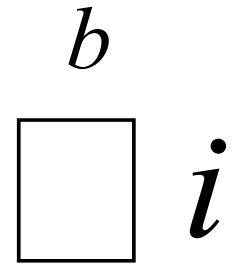
- What is **z**?
  - Would depend on how we implemented **sort-pair**
    - Would have to decide carefully and document **sort-pair**
  - But without mutation, we can implement “either way”
    - No code can ever distinguish aliasing vs. identical copies
    - No need to think about aliasing; focus on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

# *First-Class Functions*

# An Example

- What if we wanted to add up all the numbers from a to b?

```
(define (sum a b)
  (if (> a b)
      0
      (+ a
          (sum (+ a 1) b))))
```



b  
i

$i=a$

# An Example

- What if we wanted to add up all the **squares** numbers from a to b?

```
(define (sum a b)
  (if (> a b)
      0
      (+ (expt a 2)
          (sum (+ a 1) b))))
```

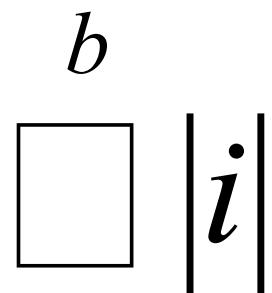
$$\boxed{ } \ i^2$$

$$i=a$$

# An Example

- What if we wanted to add up all the **absolute values of the numbers** from a to b?

```
(define (sum a b)
  (if (> a b)
      0
      (+ (abs a)
          (sum (+ a 1) b)))))
```



$b$   
 $i=a$   
 $|i|$

# *These functions are all very similar*

- All three of these functions differ only in how the sequence of integers from a to b are transformed before they are all added together.
- The adding process itself is identical in all of the functions:

```
(define (sum-something a b)
  (if (> a b)
      0
      (+ (do something to a)
          (sum-something (+ a 1) b))))
```

- What if there were a general sum function that could sum up any sequence of this form?

# *A function that takes a function*

- Imagine a function that could take another function as an argument:

```
(define (sum-any func a b)
  (if (> a b)
      0
      (+ (func a)
          (sum-any func (+ a 1) b)))))
```

# *Sum-any in action!*

```
(sum-any sqrt 1 10)
=> sqrt(1) + sqrt(2) + sqrt(3) + ...
=> about 22.5
```

```
(define (square x) (* x x))
(sum-any square 1 4)
=> 1^2 + 2^2 + 3^2 + 4^2 => 1 + 4 + 9 + 16 => 30
```

```
(define (identity x) x)
(sum-any identity 1 4)
=> 10
```

# *How to use sum-any*

- You can put the name of any function in place of `sqrt`, `square`, or `identity`, and `sum-any` will compute

$$f(a) + f(a + 1) + f(a + 2) + \dots + f(b)$$

- Provided `f` is a function of a single numeric argument.
- What if you want to compute  $f(a^2/2) + f((a+1)^2/2) + \dots$ 
  - Fine to do:  
`(define (silly-function x) (/ (* x x) 2))`  
`(sum-any silly-function 1 10)`
- Wouldn't it be nicer if we didn't have to name that silly function?

# *Anonymous Functions*

- Functional programming languages allow us to create functions without names.
- In Racket, we use the keyword **lambda** for this:  
`(lambda (arg1 arg2...) body)`
- This expression represents an *anonymous function*.
  - Kind of like a "function literals."

# Aside: lambda calculus

- Formal system for computation based on function abstraction and application.
- *Church-Turing thesis* (1936-37) proved lambda calculus is equivalent in power to Turing machines.



**Alonzo  
Church**

# *Anonymous Functions*

- Use an anonymous function when you need a "temporary" function:

```
(sum-any (lambda (x) (/ (* x x) 2)) 1 10)
```

is better style than

```
(define (silly-function x) (/ (* x x) 2))
(sum-any silly-function 1 10)
```

- Compare:

```
(sum-any (lambda (x) (* x x)) 1 10)
```

and

```
(define (square (x) (* x x))
(sum-any square 1 10))
```

# *Using anonymous functions*

- Most common use: Argument to a higher-order function
  - Don't need a name just to pass a function
- But: Cannot use an anonymous function for a recursive function
  - `(define (triple x) (* 3 x)) ; named version`
  - `(lambda (x) (* 3 x)) ; anonymous version`

# *Named functions vs anonymous functions*

- Named functions are mostly indistinguishable from anonymous functions.
- In fact, naming a function with **define** uses the anonymous form behind the scenes:

```
(define (func arg1 arg2 ...) expression)
```

is converted to:

```
(define func (lambda (arg1 arg2 ...) expression))
```

- It is poor style to define unnecessary functions in the global (top-level) environment
  - Use either nested defines, or anonymous functions.