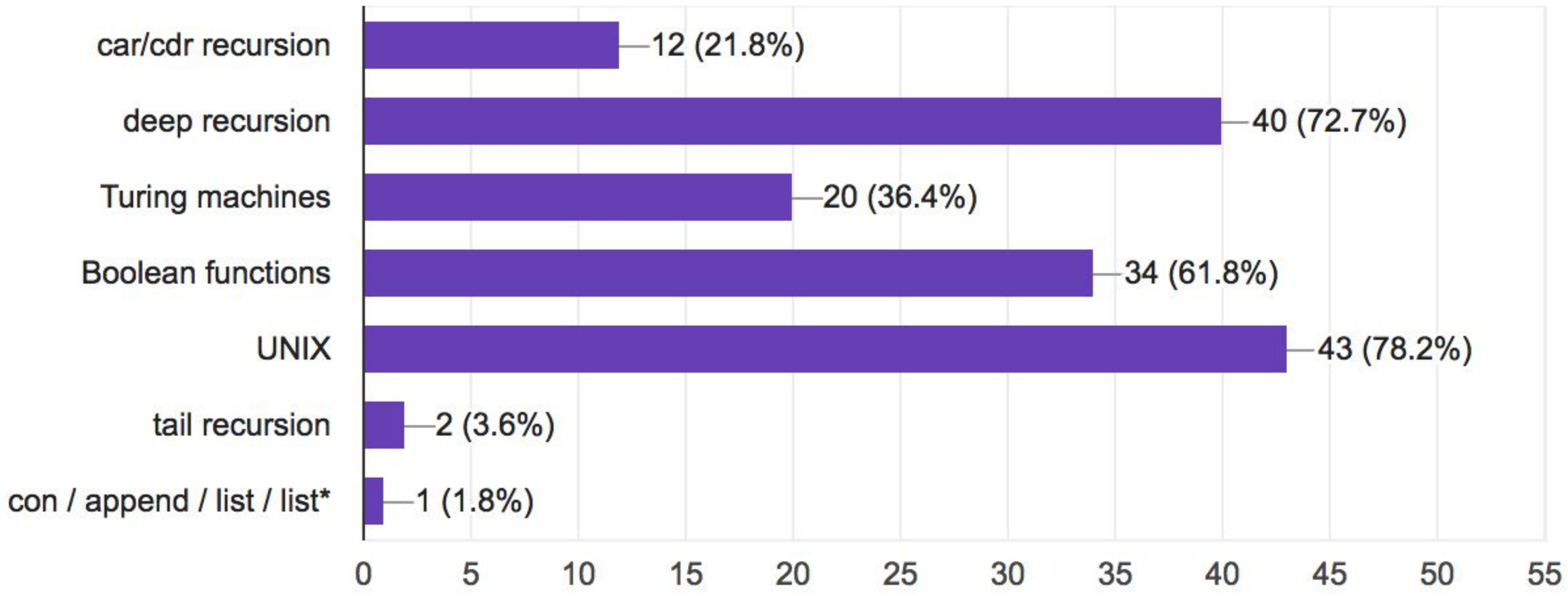
CS 201 midterm review

10/7/18

Juliana Viola

What topics should we go over?

55 responses



Agenda

1. How to get better at UNIX
2. `cons` vs. `append`
3. Deep recursion (review `lambda` functions)
4. Boolean functions
5. Turing machines (review `let`)
6. Time permitting: more questions!

Please interrupt at any time with questions or suggestions :)

UNIX

How can I get better at UNIX?

1) UNIX tutorial on the Zoo! ssh into the Zoo; then in your home folder, type the following command: `python3 /c/cs201/www/unixtutorial.py`

2) Practice typing commands on the Zoo

General tips:

- Be familiar with the *output* of each command (important in context of the transcript!)

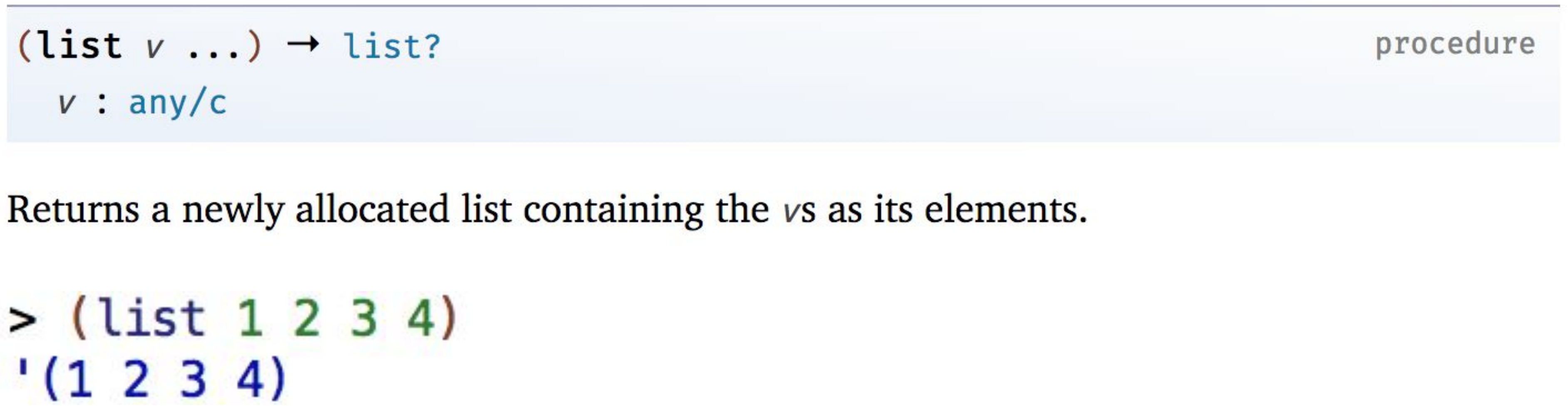
Any specific UNIX questions?

cons vs. append

First, a note on pairs vs. lists...

From the Racket guide:

- "A *pair* combines exactly two values."
- "A *list* is recursively defined: it is either the constant `null`, or it is a pair whose second value is a list."

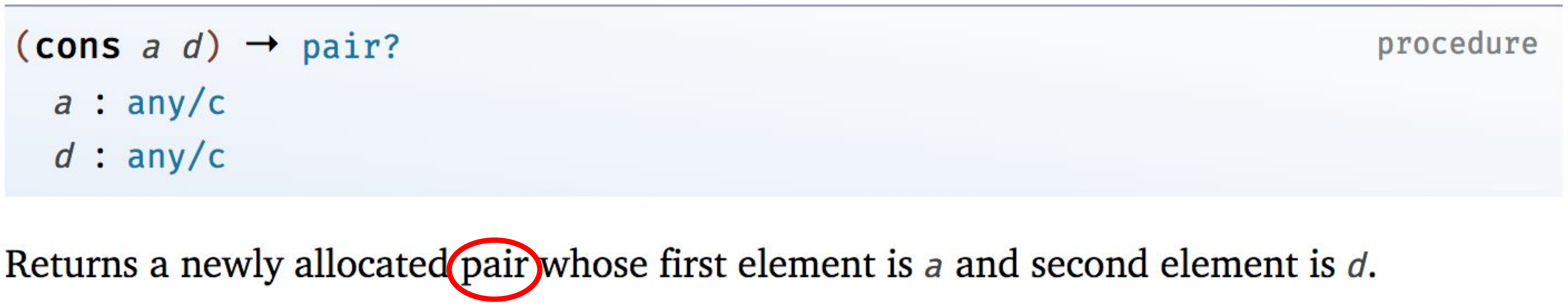


```
(list v ...) → list?                                    procedure
  v : any/c
```

Returns a newly allocated list containing the *v*s as its elements.

```
> (list 1 2 3 4)
'(1 2 3 4)
```

`cons`



```
(cons a d) → pair?                                        procedure
  a : any/c
  d : any/c
```

Returns a newly allocated pair whose first element is *a* and second element is *d*.

⚠️ `(cons 1 2)` returns the *pair* `'(1 . 2)`
This pair is not a list because the cdr is not `null`*!*

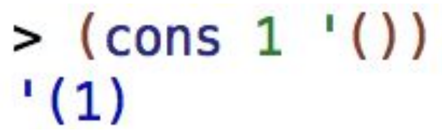
If you want to construct the list `'(1 2)`, use
`(cons 1 (cons 2 '()))` or
`(cons 1 '(2))` or
`(cons 1 (list 2))`

`cons` continued

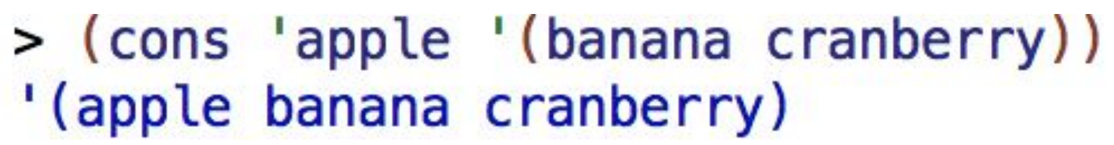
In general, here is how I use/conceptualize `cons` to return a list (not merely a pair):

- Let `a` be any data type; let `b` be a list
- `(cons a b)` inserts `a` as the first element of the list `b`

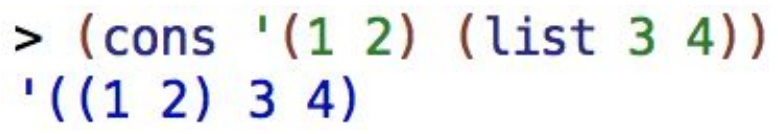
Examples:



```
> (cons 1 '())
'(1)
```



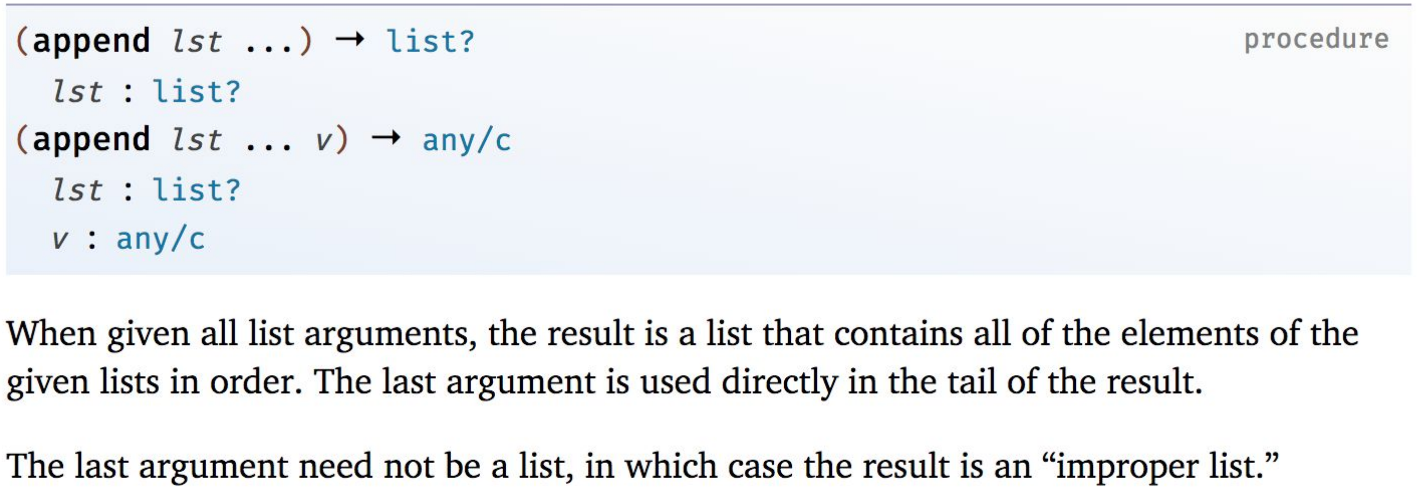
```
> (cons 'apple '(banana cranberry))
'(apple banana cranberry)
```



```
> (cons '(1 2) (list 3 4))
'((1 2) 3 4)
```

Key takeaway:
The second argument for `cons` should almost always be a list (unless you want to return a pair); the first argument can be whatever you want and will be inserted as the first element of the list supplied

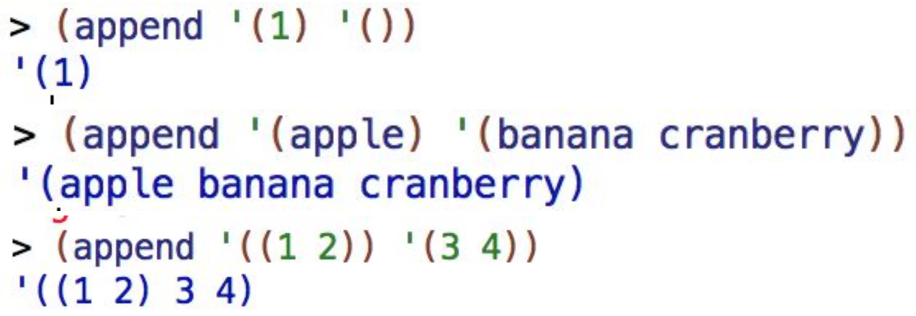
append



```
(append lst ...) → list?                                  procedure
  lst : list?
(append lst ... v) → any/c
  lst : list?
  v : any/c
```

When given all list arguments, the result is a list that contains all of the elements of the given lists in order. The last argument is used directly in the tail of the result.

The last argument need not be a list, in which case the result is an "improper list."



```
> (append '(1) '())
'(1)
> (append '(apple) '(banana cranberry))
'(apple banana cranberry)
> (append '((1 2)) '(3 4))
'((1 2) 3 4)
```

In general, here is how I use/conceptualize `append` to return a list:

- Let `a` and `b` be lists
- `(append a b)` essentially merges the lists `a` and `b`

Key takeaway:

Generally speaking, all arguments for `append` should be lists (unless you want to return an improper list)

```
> (cons 1 '())
'(1)

> (cons 'apple '(banana cranberry))
'(apple banana cranberry)

> (cons '(1 2) (list 3 4))
'((1 2) 3 4)
```

```
> (append '(1) '())
'(1)

> (append '(apple) '(banana cranberry))
'(apple banana cranberry)

> (append '((1 2)) '(3 4))
'((1 2) 3 4)
```

Questions?

Practice with `cons` vs. `append`

1. `(cons 1 2)`
2. `(cons 1 ‘())`
3. `(cons 1 ‘(2))`
4. `(cons ‘(1) 2)`
5. `(cons ‘(1) ‘(2))`
6. `(append ‘(1) ‘(2))`
7. `(append ‘((1)) ‘((2)))`
8. Define `my-lst` to be `‘(hello “hi” #t 7).`
   a. How would you use `cons` to get the list `‘(1 hello “hi” #t 7)`?
   b. How would you use `append` to get the list `‘(1 hello “hi” #t 7)`?

Solutions

1. `(cons 1 2) => '(1 . 2)`
2. `(cons 1 '()) => '(1)`
3. `(cons 1 '(2)) => '(1 2)`
4. `(cons '(1) 2) => '((1) . 2)`
5. `(cons '(1) '(2)) => '((1) 2)`
6. `(append '(1) '(2)) => '(1 2)`
7. `(append '((1)) '((2))) => '((1) (2))`
8. Define `my-lst` to be `'(hello "hi" #t 7).`
   a. How would you use `cons` to get the list `'(1 hello "hi" #t 7)`? `=>` `(cons 1 my-lst)`
   b. How would you use `append` to get the list `'(1 hello "hi" #t 7)`? `=>` `(append '(1) my-lst)`

Deep recursion

Practice with deep recursion

Write a procedure

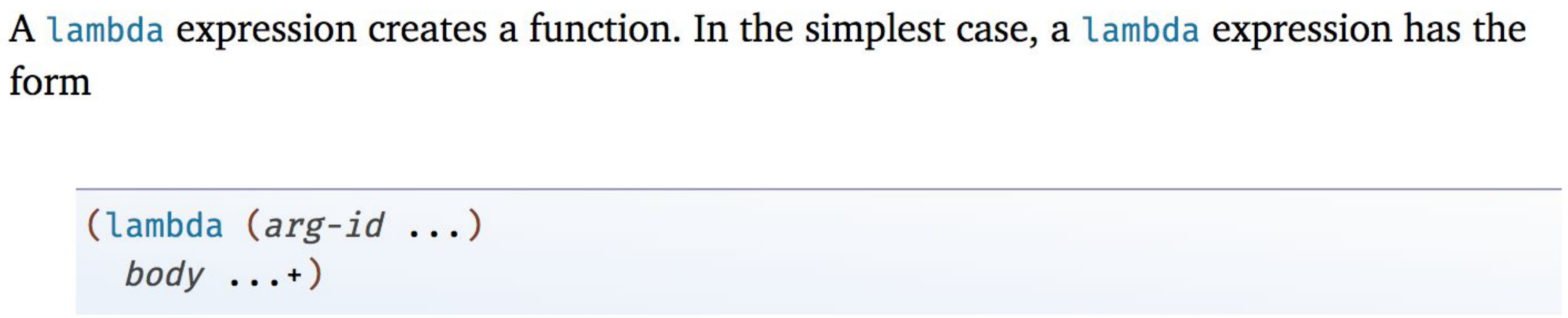
```
(count-if pred tree)
```

which returns the number of leaves of the tree that satisfy the given predicate `pred`

Examples

```
(count-if odd? '(1 2 3)) => 2
(count-if even? '(1 2 3)) => 1
(count-if integer? '(1 (2 (3)))) => 3
(count-if string? '()) => 0
(count-if even? '((((((8 8 8))))))) => 3
(count-if (lambda (x) (> x 5)) '((((((9 9 9))))))) => 3
```

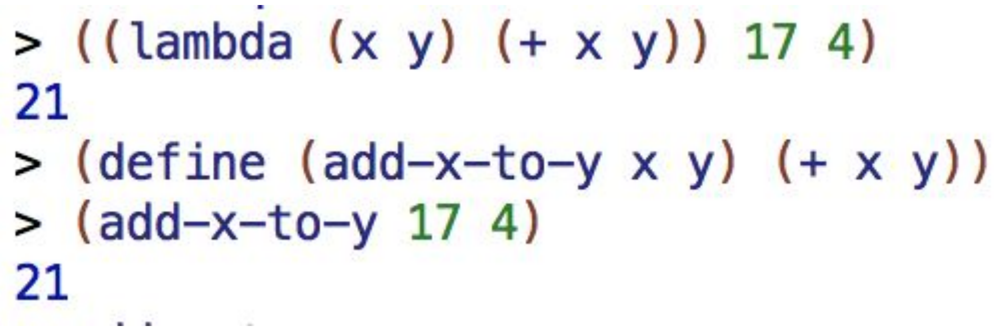
Quick review: what is a `lambda` function?



A `lambda` expression creates a function. In the simplest case, a `lambda` expression has the form

```
(lambda (arg-id ...)
  body ...+)
```

Example:



```
> ((lambda (x y) (+ x y)) 17 4)
21
> (define (add-x-to-y x y) (+ x y))
> (add-x-to-y 17 4)
21
```

Back to deep recursion...

Write a procedure

```
(count-if pred tree)
```

which returns the number of leaves of the tree that satisfy the given predicate `pred`

Examples

```
(count-if odd? '(1 2 3)) => 2
(count-if even? '(1 2 3)) => 1
(count-if integer? '(1 (2 (3)))) => 3
(count-if string? '()) => 0
(count-if even? '((((((8 8 8))))))) => 3
(count-if (lambda (x) (> x 5)) '((((((9 9 9))))))) => 3
```

Back to deep recursion...

Write a procedure

```
(count-if pred tree)
```

which returns the number of leaves of the tree that satisfy the given predicate `pred`

Examples

```
(count-if odd? '(1 2 3)) => 2
(count-if even? '(1 2 3)) => 1
(count-if integer? '(1 (2 (3)))) => 3
(count-if string? '()) => 0
(count-if even? '((((((8 8 8))))))) => 3
(count-if (lambda (x) (> x 5)) '((((((9 9 9))))))) => 3
```

Think/pair/share

Sample solution

```
(define (count-if pred tree)
  (cond [(empty? tree) 0]
        [(list? (first tree)) (+ (count-if pred (first tree)) (count-if pred (rest tree)))]
        [(pred (first tree)) (+ 1 (count-if pred (rest tree)))]
        [else (count-if pred (rest tree))]))
```

Sample solution

```
(define (count-if pred tree)
  (cond [(empty? tree) 0]
        [(list? (first tree)) (+ (count-if pred (first tree)) (count-if pred (rest tree)))]
        [(pred (first tree)) (+ 1 (count-if pred (rest tree)))]
        [else (count-if pred (rest tree))]))
```

Questions? Comments? Do you agree or disagree?

Sample solution

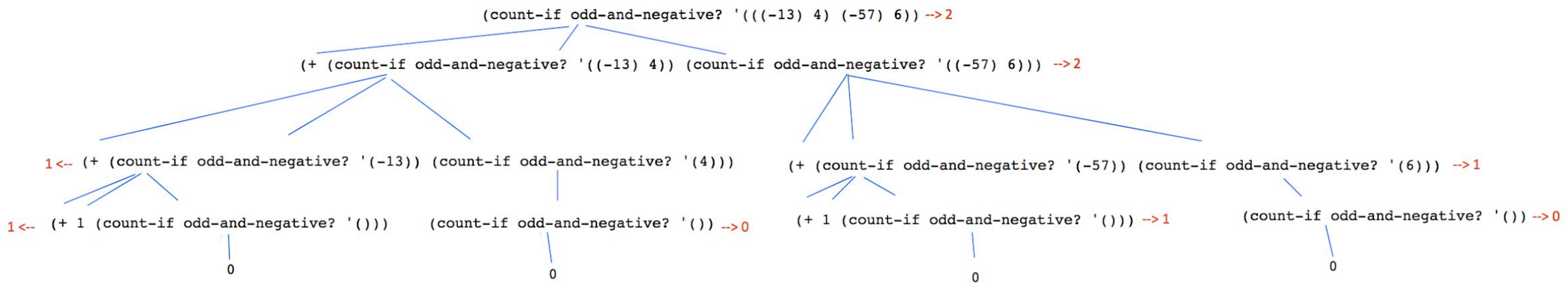
```
(define (count-if pred tree)
  (cond [(empty? tree) 0]
        [(list? (first tree)) (+ (count-if pred (first tree)) (count-if pred (rest tree)))]
        [(pred (first tree)) (+ 1 (count-if pred (rest tree)))]
        [else (count-if pred (rest tree))]))
```

Questions? Comments? Do you agree or disagree?

Time to draw a tree of recursive calls

```
(count-if (lambda (x) (and (negative? x) (odd? x))) '(((-13) 4) (-57) 6))
```

Sample tree of recursive calls



```
(count-if odd-and-negative? '(((-13) 4) (-57) 6)) --> 2

(+ (count-if odd-and-negative? '((-13) 4)) (count-if odd-and-negative? '((-57) 6))) --> 2

1 <-- (+ (count-if odd-and-negative? '(-13)) (count-if odd-and-negative? '(4)))
(+ (count-if odd-and-negative? '(-57)) (count-if odd-and-negative? '(6))) --> 1

1 <-- (+ 1 (count-if odd-and-negative? '()))
(count-if odd-and-negative? '()) --> 0
(+ 1 (count-if odd-and-negative? '())) --> 1
(count-if odd-and-negative? '()) --> 0

0
0
0
0
```

Boolean functions

Key ideas

- Truth tables
- Operations:
  - and *
  - or +
  - not ‵
- Sum of products algorithm

Truth table example – write an expression for f(x,y,z)

How I approach truth tables:

x	y	z	f(x,y,z)
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Truth table example – write an expression for f(x,y,z)

x	y	z	f(x,y,z)
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

How I approach truth tables:

1. Find all the true values in the output column

Truth table example – write an expression for f(x,y,z)

x	y	z	f(x,y,z)	
0	0	0	0	
0	0	1	0	
0	1	0	0	
0	1	1	1	x’*y*z
1	0	0	0	
1	0	1	1	x*y’*z
1	1	0	1	x*y*z’
1	1	1	1	x*y*z

How I approach truth tables:

1. Find all the true values in the output column
2. Write Boolean expressions for the corresponding rows

Truth table example – write an expression for f(x,y,z)

x	y	z	f(x,y,z)	
0	0	0	0	
0	0	1	0	
0	1	0	0	
0	1	1	1	x’*y*z
1	0	0	0	
1	0	1	1	x*y’*z
1	1	0	1	x*y*z’
1	1	1	1	x*y*z

How I approach truth tables:

1. Find all the true values in the output column
2. Write Boolean expressions for the corresponding rows
3. Add these expressions together to get your final sum of products:
   (x’*y*z)+(x*y’*z)+(x*y*z’)+(x*y*z)

Hooray! You’re done! You’ve written a Boolean expression for f(x,y,z) using the sum-of-products algorithm

Equivalently:
(x*y)+(y*z)+(x*z)
Can you see why this is true?

Practice with truth tables – write an expression for f(x,y,z)

x	y	z	f(x,y,z)
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

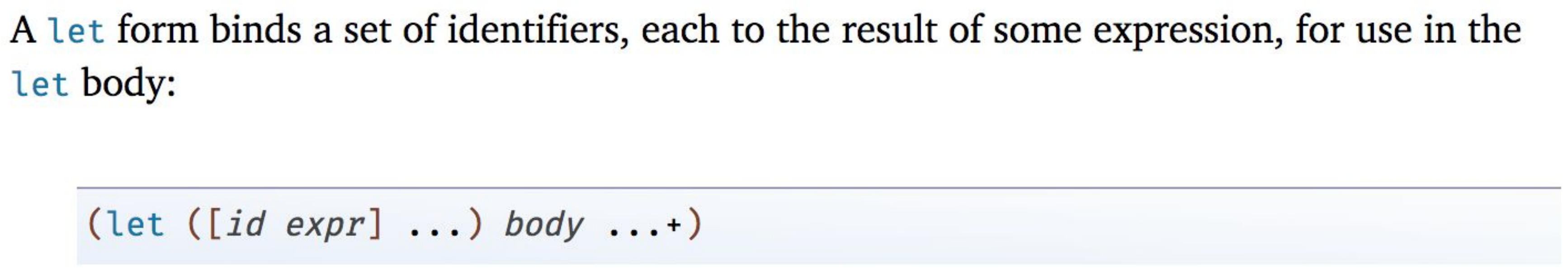
Practice with truth tables – write an expression for f(x,y,z)

x	y	z	f(x,y,z)
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

Solution:
(x’*y’*z’)+(x’*y*z’)+(x*y’*z’)+(x*y’*z)

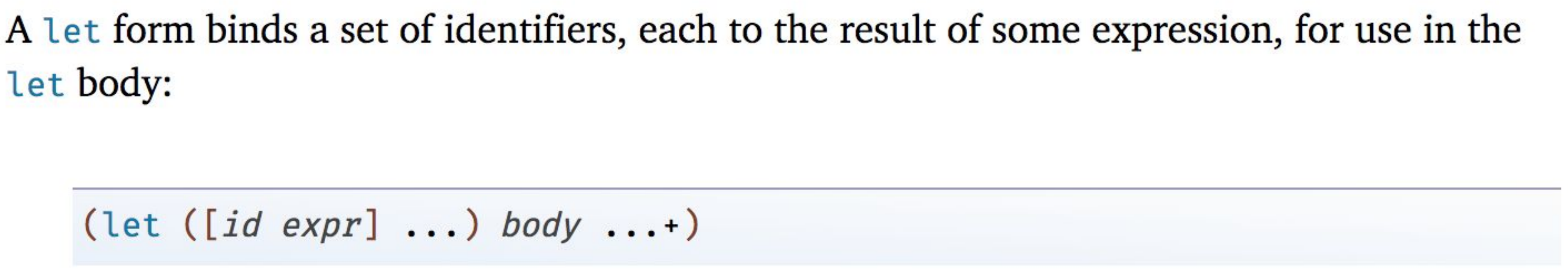
Turing machines

First, a quick review of `let` – what is it good for?


A `let` form binds a set of identifiers, each to the result of some expression, for use in the `let` body:

```
(let ([id expr] ...) body ...+)
```

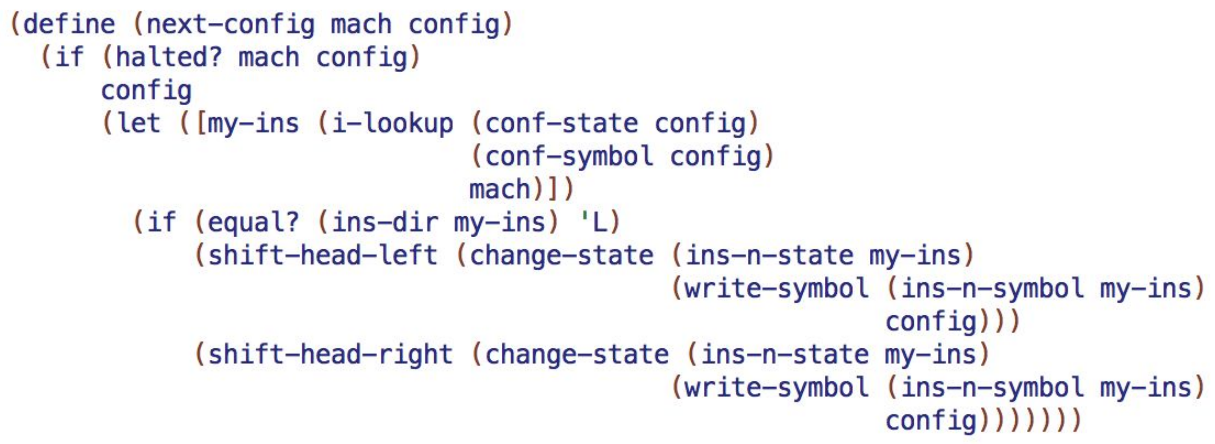
First, a quick review of `let` – what is it good for?



A `let` form binds a set of identifiers, each to the result of some expression, for use in the `let` body:

```
(let ([id expr] ...) body ...+)
```

Example from my hw3.rkt:



```
(define (next-config mach config)
  (if (halted? mach config)
      config
      (let ([my-ins (i-lookup (conf-state config)
                              (conf-symbol config)
                              mach)])
        (if (equal? (ins-dir my-ins) 'L)
            (shift-head-left (change-state (ins-n-state my-ins)
                                           (write-symbol (ins-n-symbol my-ins)
                                                         config)))
            (shift-head-right (change-state (ins-n-state my-ins)
                                            (write-symbol (ins-n-symbol my-ins)
                                                          config)))))))
```

What's the advantage here?

Think/pair/share – what is this Turing machine doing?

```
(define tm-mystery
  (list (ins 'q1 0 'q1 0 'R)
        (ins 'q1 1 'q1 1 'R)
        (ins 'q1 'b 'q2 'b 'L)
        (ins 'q2 0 'q3 1 'L)
        (ins 'q2 1 'q4 0 'L)
        (ins 'q3 0 'q3 1 'L)
        (ins 'q3 1 'q4 0 'L)
        (ins 'q3 'b 'q5 'b 'R)
        (ins 'q4 0 'q4 0'L)
        (ins 'q4 1 'q4 1 'L)
        (ins 'q4 'b 'q5 'b 'R)))
```

(Assume inputs will be >= 1 in binary)

Solution

Subtract 1 from the n-bit input; the output is n-bit difference

Examples:

1 => 0

10 => 01

11 => 10

100 => 011

Questions?