The **signatures** for the functions are very important!

Your functions need to always receive and produce the right types of data.

If a function **consumes** a list of words (i.e., a list of lists of 1Strings), then it will fail if you call it with a word as input.

Be sure all cases of your functions **produce** the right type of data. If your recursive case produces a word (i.e., a list of 1strings), your base case can’t produce a string.
(permutations (list "h" "e" "y"))
(permutations (list "h" "e" "y"))
(permutations (list "h" "e" "y"))

→

(insert-everywhere/in-all-words
  (first w)
  (permutations (rest w)))

_Word_, i.e., `[List-of 1String]`
\textbf{Word, i.e., [List-of 1String]}

\((\text{permutations (list "h" "e" "y"))})

\rightarrow

\((\text{insert-everywhere/in-all-words (first w) 1String, i.e., a letter (permutations (rest w)))})\)
(permutations (list "h" "e" "y"))

→

(insert-everywhere/in-all-words
 (first w)
 (permutations (rest w)))

Word, i.e., [List-of 1String]

1String, i.e., a letter

[List-of Word] i.e., [List-of [List-of 1String]]
(permutations (list "h" "e" "y"))

→

(insert-everywhere/in-all-words
  (first w)
  (permutations (rest w)))

→

(insert-everywhere/in-all-words
  "h"
  (permutations (rest w)))
(permutations (list "h" "e" "y"))

→

(insert-everywhere/in-all-words
  (first w)
  (permutations (rest w)))

→

(insert-everywhere/in-all-words
  "h"
  (list (list "e" "y")
        (list "y" "e")))
\[(\text{permutations} \ (\text{list} \ "h" \ "e" \ "y"))\]

\[
\rightarrow
\]

\[(\text{insert-everywhere/in-all-words} \ (\text{first} \ \text{w}) \ (\text{permutations} \ (\text{rest} \ \text{w})))\]

\[
\rightarrow
\]

\[(\text{insert-everywhere/in-all-words} \ "h" \ (\text{list} \ (\text{list} \ "e" \ "y") \ (\text{list} \ "y" \ "e"))))\]
(permutations (list "h" "e" "y"))

→

(insert-everywhere/in-all-words
   (first w)
   (permutations (rest w)))

→

(insert-everywhere/in-all-words
   "h"
   (list (list "e" "y")
         (list "y" "e")))

→

(list
  (list "h" "e" "y")
  (list "e" "h" "y")
  (list "e" "y" "h")
  (list "h" "y" "e")
  (list "y" "h" "e")
  (list "y" "e" "h"))
(permutations (list "h" "e" "y"))

→

(insert-everywhere/in-all-words (first w) (permutations (rest w)))

→

(insert-everywhere/in-all-words "h" (list (list "e" "y") (list "y" "e")))

→

(list Insert h first Generate possibilities where h is later
  (list "h" "e" "y") (list "e" "h" "y") (list "e" "y" "h")
  (list "h" "y" "e") (list "y" "h" "e") (list "y" "e" "h"))
(permutations (list "h" "e" "y"))

→

(insert-everywhere/in-all-words
  (first w)
  (permutations (rest w)))

→

(insert-everywhere/in-all-words
  "h"
  (list (list "e" "y")
        (list "y" "e")))

→

(list Insert h first Generate possibilities where h is later
  (list "h" "e" "y") (list "e" "h" "y") (list "e" "y" "h")
  (list "h" "y" "e") (list "y" "h" "e") (list "y" "e" "h"))
(permutations (list "h" "e" "y"))

→

(insert-everywhere/in-all-words
   (first w)
   (permutations (rest w)))

→

(insert-everywhere/in-all-words
   "h"
   (list (list "e" "y")
         (list "y" "e")))

→

(list "h" "e" "y") (list "e" "h" "y") (list "e" "y" "h")
(list "h" "y" "e") (list "y" "h" "e") (list "y" "e" "h")

These possibilities all have the same first letter, so we can use a helper to add that letter to the possibilities!

Insert h first Generate possibilities where h is later
You should write the functions in order, “top down”, i.e., from the `permutations` function to the last helper it depends on, `insert-front/of-all-words`.

But, if your code doesn’t work, you should check your work from the bottom up.

Since `permutations` depends on three other functions, if it doesn’t work, that could be something going wrong in any of those functions. So, start with the function that doesn’t depend on any helpers, and be very certain that it works correctly, then move up to the function that only depends on it, and so forth.
When you’re debugging your code, you can use the stepper to walk through it, but you might not care about every step of the evaluation.

Instead, you might just want to see what arguments each function is being called on and what each one returns. This is a job for trace!
(require racket/trace)

(define (foo bar)
  (cond [(empty? bar) '()] [(cons? bar)
    (cons (+ 1 (first bar))
      (foo (rest bar)))]))

(trace foo)

(foo '(1 2 3 4))
None of the recursive calls return a value until after they get to the end of the list and there are no more recursive calls. We build up the list we’re returning by consing one element at a time onto the list formed by our base case, namely '().
Updated due date: 10:30 a.m., Wednesday, 27 November
Assignment 3 graded soon
Assignment 4 due a week from today
Exam 2 on Friday
Functions and names
Consider some values we can use in Racket:

**Numbers**: 1, 17.8, 4/5

**Booleans**: #true, #false

**Lists**: '(), (cons 7 '())

...

**Function names**: less-than-5?, first-is-apple?

given

(define (less-than-5? n) ...)

(define (first-is-apple? a b) ...)
Consider some values we can use in Racket:

**Numbers:** 1, 17.8, 4/5

**Booleans:** #true, #false

**Lists:** '(), (cons 7 '())

... 

**Function names:** less-than-5?, first-is-apple?

given

(define (less-than-5? n) ...)

(define (first-is-apple? a b) ...) 

Why do only function values require names?
Needing to name every kind of value would be painful:

```
(local [(define (first-is-apple? a b)
          (symbol=? a 'apple))]
        (choose '(apple banana)
             '(cherry cherry)
             first-is-apple?))
```

would need to be

```
(local [(define (first-is-apple? a b)
          (symbol=? a 'apple))
          (define a1 '(apple banana))
          (define b1 '(cherry cherry))
        (choose a1 b1 first-is-apple?))
```

But, fortunately, we don’t need to name every list.
So, can we avoid naming functions?

In other words, instead of writing

```scheme
(local [(define (first-is-apple? a b) (symbol=? a 'apple))]
  ... first-is-apple? ...
)
```

we’d like to write

```scheme
... function that takes a and b and produces (symbol=? a 'apple)
... 
```
We can do this in the *Intermediate Student with lambda* language.
Lambda expressions are special forms that evaluate to procedures.
The meaning of a *lambda* expression

\((\text{lambda } (x) (* x x))\)

- A procedure
- Of one argument “x”
- That returns the square of x
The meaning of a lambda expression

\((\text{lambda} \ (x \ y) \ (/ \ (+ \ x \ y) \ 2))\)

- A procedure
- Of two arguments “x” and “y”
- That returns the average of x and y
Calling this keyword **lambda** is a convention derived from the lambda calculus that Alonzo Church developed in the 1930s, where λ is used to mean “function”.

Church’s choice of this notation was arbitrary, but arbitrary choices can have long-lasting consequences. For another example, consider why programmers use “foo” and “bar” as placeholder names.
An example of an anonymous function value is

\[
\text{(lambda (a b) (symbol=? a 'apple))}
\]

So, using lambda the original example becomes

\[
\text{(choose ' (apple banana) ' (cherry cherry) (lambda (a b) (symbol=? a 'apple)))}
\]
In DrRacket, we get this evaluation:

```scheme
> (lambda (x) (+ x 10))
(lambda (a1) ...)
```

Unlike most kinds of values, there’s no one shortest name:

- The argument name is arbitrary
- The body can be implemented in many different ways

So DrRacket gives up – it invents argument names and hides the body.
(lambda (x) (+ x 10)) 17

27

The function position of an application (i.e., a function call) no longer needs to be an identifier.

As such, some former syntax errors are now runtime errors:

> (2 3)

procedure application: expected procedure, given 2
Positional association

\(((\text{lambda} \ (x \ y) \ (\div \ (+ \ x \ y) \ 2)) \ 1066 \ 2019)\)

Which actual argument is substituted for \(x\)?

The one in the first position, since \(x\) is first.

Which actual argument is substituted for \(y\)?

The one in the second position, since \(y\) is second.
So, what’s the difference between

```
(define (f a b)
  (+ a b))
```

and

```
(define f
  (lambda (a b)
    (+ a b)))
```
So, what’s the difference between

\[
(\text{define (f a b)}
  (+ a b))
\]

and

\[
(\text{define f}
  (\lambda (a b)
    (+ a b)))
\]

?

Nothing! The former is now a shorthand (“syntactic sugar”) for the latter.
Anonymous functions are great for when we want to pass a function as an argument to another function.

Just as we can call a function on 5 rather than write `(define N 5)` and call it on N, we can now call a function on another function value without first defining it.
Design a function `each-matched-pair?` that takes three inputs: lists `lst1` and `lst2` and a predicate `P` that takes two inputs.

The `each-matched-pair?` function applies `P` to each corresponding pair of members of `lst1` and `lst2`, e.g., if `lst1` is `(1 2 3)` and `lst2` is `(a b c)`, it evaluates `(P 1 'a)`, `(P 2 'b)`, and `(P 3 'c)`.

The `each-matched-pair?` procedure returns `#true` if `P` returns `#true` each and every time; otherwise it returns `#false`. If `lst1` and `lst2` have different lengths, `each-matched-pair?` ignores the extra members at the end of the longer list.
;; [List-of X] [List-of Y] [X Y -> Boolean] -> Boolean
;; Check if corresponding elements of lst1 and lst2
;; satisfy P.
(define (each-matched-pair? lst1 lst2 P)
  (or (null? lst1)
      (null? lst2)
      (and (P (first lst1) (first lst2))
           (each-matched-pair? (rest lst1) (rest lst2) P))))

(check-expect (each-matched-pair? '(1 2 3) '(4 5 6) =) #false)
(check-expect (each-matched-pair? '(1 2 3) '(4 5 6) <) #true)
(check-expect
  (each-matched-pair? '(1 2 3) '(0 3 2)
   (lambda (x y) (<= (abs (- x 1)) 1)))
  #true)
Acknowledgments

This lecture incorporates material from:

Matthias Felleisen
Robert Bruce Findler
Matthew Flatt
Gregor Kiczales
Shriram Krishnamurthi
Marc Smith