Goals of this tutorial
You should be able to...

- understand and write the data definitions for lists
- understand and use the template for processing lists to write recursive functions consuming this type of data.
- understand the difference between structure and list

Review: List data definition
;; A (listof Any) is one of:
;; * empty
;; * (cons Any (listof Any))

From the data definition, a list of values of any type is either empty or it consists of a first value followed by a list of values (the rest of the list).

This is a recursive definition. It contains a base case, and a recursive (self-referential) case.

Recursive types should be operated with recursive functions.
Review: Basic list constructs

- empty: A value representing a list with 0 items.
- cons: Consumes an item and a list and produces a new, longer list.
- first: Consumes a nonempty list and produces the first item.
- rest: Consumes a nonempty list and produces the same list without the first item.
- empty?: Consumes a value and produces true if it is empty and false otherwise.
- cons?: Consumes a value and produces true if it is a cons value and false otherwise.

Clicker Question - box-and-pointer

Which of the following nested box representations match this box-and-pointer representation?

Template for processing a list of symbols

Here we use list of symbols as an example to show a general method for processing a list.

```scheme
;; my-los-fn: (listof Sym) -> Any
(define (my-los-fn los)
 (cond [(empty? los) ...]
      [(cons? los) ...]))
```

Since cons is a recursive structure type, we can use its selectors in the structure template to get the contents.

The second conditional question can now be replaced by else.
(define (my-los-fn los)
  (cond [(empty? los) . . . ]
        [else . . . (first los) . . . (rest los) . . . ])))

Now we have the first item and the rest of the list.
Since (rest los) is a list of symbols, we should apply the same computation to it – that is, we apply my-los-fn onto the rest of the symbols until we have nothing left.

The resulting template is a recursive function that consumes a list, and applies the necessary steps to work towards the base case of empty:

(define (my-los-fn los)
  (cond [(empty? los) . . . ]
        [else . . . (first los) . . . (my-los-fn (rest los)) . . . ])))

We can now fill in the dots for a specific example.

**Group Problem - list of Strings - Data Definition**

Write a data definition for a list of Strings. You can find a similar example on module 5, slide 15 (slide 3 of this tutorial as well). Keep this solution for the next problem.
Group Problem - list of Strings - Template

Based on the previous data definition, write an appropriate template. Module 5, slides 16-18 also show an example of a template (slide 6-8 of this tutorial as well). Keep this for the next problem.

Group Problem: string-length-adder

Based on the template on slide 10, write the contract and function definition for string-length-adder which consumes a list of Strings and produces the length of all the strings added together. If you are given empty, produce 0.

Recall that the function string-length consumes a Str and produces a Nat. An example would be (string-length "apple") \(\rightarrow\) 5.

;; (string-length-adder los) adds up the length of all strings in los, a given (listof Str).
;; Examples:
;; (check-expect (string-length-adder
;; (cons "MA" (cons "MAT" (cons "MATH" (cons "MATH rocks" empty)))))) 19
;; (check-expect (string-length-adder (cons "Turkey" empty)) 6)

Group Problem: list-replace

Write a function list-replace which consumes a list of numbers, a target number, and a replacement number. The function produces a new list, where each element is equal to the corresponding element of the consumed list, except for all occurrences of the target number (if any), which are replaced with the replacement number.

;; Examples:
;; (check-expect (list-replace (cons 1 (cons 2 (cons 2 empty))) 2 3)
;; (cons 1 (cons 3 (cons 3 empty))))
;; (check-expect (list-replace (cons 1 (cons 3 (cons 3 empty))) empty)
;; (check-expect (list-replace empty 3 4) empty)

;;always include base case in Examples
;; (check-expect (list-replace empty 3 4) empty)
Review: Defining Structures

The special form
\( \text{(define-struct sname (fname1 \ldots \ fnamen))} \)
defines the structure type \text{sname} and automatically defines the following
primitive functions:

- **Constructor:** \text{make-sname}
- **Selectors:** \text{sname-fname1 \ldots \ sname-fnamen}
- **Predicate:** \text{sname?}

The \text{sname?} predicate tests if its argument is a \text{sname}.
\text{Sname} may be used in contracts if the respective data definition has been
stated.

Review: Difference between Structures and Lists

When you have a fixed amount of data and you want to group data
together, you may use a structure to represent it.

For example, suppose we want to represent information associated
with downloaded MP3 files which contains: performer, title, length,
genre(rap, country, etc.), we can define a structure as follows.

\( \text{(define-struct mp3info (performer title length genre))} \)

;; An Mp3Info is a (make-mp3info Str Str Num Sym)

Review: Difference between Structures and Lists

When the amount of data is unbounded, meaning it may grow or
shrink – and you don’t know how much, so you cannot use a
structure for that kind of data.

For example, suppose you enjoy attending concerts of local
musicians and want a list of the upcoming concerts you plan to
attend. The number will change as time passes. We will also be
cconcerned about order. So we may use a \text{(listof Str)} to represent the
concerts data.