Formally Defining lists

Up to this point, we have worked with lists intuitively. We have assumed that it's clear what we mean by a “list”, and we have tools like `map` and `filter` that work with these values.

But we want to understand lists at a deeper level: how can we store an arbitrary number of values? How do our higher order functions actually work?

In this module we introduce a fundamental concept of computation:

```
recursion
```

To study recursion, we put aside what we know about lists.

We will define them, from the ground up, using a very small collection of very simple tools.
What is Recursion?

Simply put, recursion is any thing that refers to itself.

Some examples:

- This sentence is recursive, since it is talks about itself.
- Many fractals, including the Sierpinski Triangle (drawn here by a Racket program!):

![Sierpinski Triangle]

Recursion is an important concept; we need a deep understanding of what it can do.
For now, switch your language level back to *Beginning Student.*

This makes working with lists more awkward, but exposes their underlying recursive nature, allowing us to study it more easily.

Refrain, for the time being, from using any of the following functions:

```
filter foldl foldr lambda length list map range
```

We will get them all back. I promise.
In Racket a **list** is a recursive structure – it is defined in terms of a smaller list.

Consider a list of concerts:

- A list of 4 concerts is a concert followed by a list of 3 concerts.
- A list of 3 concerts is a concert followed by a list of 2 concerts.
- A list of 2 concerts is a concert followed by a list of 1 concert.
- A list of 1 concert is a concert followed by a list of 0 concerts.

A list of zero concerts is special. We’ll call it the **empty list**. It is represented in Racket by `empty`.

`(cons v lst)` creates a list by adding the value `v` to the beginning of list `lst`. 

A sad state of affairs – no upcoming concerts to attend:

```
(define concerts0 empty)
```

A list with one concert to attend:

```
(define concerts1 (cons "Waterboys" empty))
```

A new list just like `concerts1` but with a new concert at the beginning:

```
(define concerts2 (cons "DaCapo" concerts1))
```
Another way to write \textit{concerts2}:

\begin{verbatim}
(define concerts2alt (cons "DaCapo"
                     (cons "Waterboys"
                     empty)))
\end{verbatim}

A list with one U2 and two DaCapo concerts:

\begin{verbatim}
(define concerts3 (cons "U2"
                     (cons "DaCapo"
                     (cons "DaCapo"
                     empty))))
\end{verbatim}
**Basic list constructs**

- **empty**: A value representing an empty list.
- **(cons v lst)**: Consumes a value and a list; produces a new, longer list.
- **(first lst)**: Consumes a non-empty list; produces the first value.
- **(rest lst)**: Consumes a non-empty list; produces the same list without the first value.
- **(empty? v)**: Consumes a value; produces `true` if it is `empty` and `false` otherwise.
- **(cons? v)**: Consumes a value; produces `true` if it is a `cons` value and `false` otherwise.
- **(list? v)**: Equivalent to `(or (cons? v) (empty? v))`. 
(define clst (cons "U2"
               (cons "DaCapo" (cons "Waterboys" empty))))

First concert:
(first clst) ⇒ "U2"

Concerts after the first:
(rest clst) ⇒ (cons "DaCapo" (cons "Waterboys" empty))

Second concert:
(first (rest clst)) ⇒ "DaCapo"
Given a shopping list

```
(define slst (cons "milk"
  (cons "eggs"
    (cons "bread"
      (cons "PB" empty)))))
```

use a combination of only `first`, `rest` and `slst` to

- produce the string "bread".
- produce the list `(cons "bread" (cons "PB" empty))`.
- produce the empty list (starting with `slst`).
Using these built-in functions, we can write our own simple functions on lists.

;; (next-concert loc) produces the next concert to attend or
;; the empty string if loc is empty

;; Examples:
(check-expect (next-concert (cons "a" (cons "b" empty))) "a")
(check-expect (next-concert empty) "")

;; next-concert: (listof Str) -> Str
(define (next-concert loc)
  (cond [(empty? loc) ""
        [else (first loc)])
```
(same-consec? loc) determines if next two concerts are the same

Examples:
(check-expect (same-consec? empty) false)
(check-expect (same-consec? (cons "a" empty)) false)
(check-expect (same-consec? (cons "a" (cons "a" empty))) true)
(check-expect (same-consec? (cons "a" (cons "b" empty))) false)

;; same-consec?: (listof Str) -> Bool
(define (same-consec? loc)
  (and (not (empty? loc))
       (not (empty? (rest loc)))
       (string=? (first loc) (first (rest loc)))))
Write \texttt{same-consec-cond?} which is the same as \texttt{same-consec?} except that it uses a \texttt{cond} expression instead of \texttt{and}.

Does the order of the question-answer pairs matter? Why or why not?
Write a function \texttt{remove-second} that consumes a list of length at least 2, and produces a list containing the same items, with the second item removed.

\begin{verbatim}
(\texttt{remove-second (cons 'Mercury (cons 'Venus empty)))}
⇒ (\texttt{cons 'Mercury empty})
(\texttt{remove-second (cons 2 (cons 4 (cons 6 (cons 0 (cons 1 empty)))))))
⇒ (\texttt{cons 2 (cons 6 (cons 0 (cons 1 empty)))))
\end{verbatim}

You’ll need \texttt{cons} in addition to \texttt{first} and \texttt{rest}.
List values are

1. **empty**
2. \((\text{cons } v l)\)

where \(v\) is any Racket value (including list values) and \(l\) is a list value (which includes \(\text{empty}\)).

Note that values and expressions look very similar!

**Value:** \((\text{cons } 1 (\text{cons } 2 (\text{cons } 3 \text{ empty})))\)

**Expression:** \((\text{cons } 1 (\text{cons } (+ 1 1) (\text{cons } 3 \text{ empty})))\)

Racket list values are traditionally given using **constructor notation** – the same notation we would use to construct the value.

Earlier we saw **list notation**, like \((\text{list } 1 2 3)\). Constructor notation helps emphasize the recursive structure, so we’ll stick with it until module 08.
The following are valid expressions:

- `(cons e1 e2)`, where `e1` and `e2` are expressions
- `(first e1)`
- `(rest e1)`
- `(empty? e1)`
- `(cons? e1)`
- `(list? e1)`
The substitution rules are:

- \((\text{first } (\text{cons } a \ b)) \Rightarrow a\), where \(a\) and \(b\) are values.
- \((\text{rest } (\text{cons } a \ b)) \Rightarrow b\), where \(a\) and \(b\) are values.
- \((\text{empty? } \text{empty}) \Rightarrow \text{true}\).
- \((\text{empty? } a) \Rightarrow \text{false}\), where \(a\) is any Racket value other than \text{empty}.
- \((\text{cons? } (\text{cons } a \ b)) \Rightarrow \text{true}\), where \(a\) and \(b\) are values.
- \((\text{cons? } a) \Rightarrow \text{false}\), where \(a\) is any Racket value not created using \text{cons}. 
Most interesting functions will process the entire consumed list. How many concerts are on the list? How many times does "Waterboys" appear? Which artists are duplicated in the list?

The structure of a function often mirrors the structure of the data it consumes. As we encounter more complex data types, we will find it useful to be precise about their structures.

We will do this by developing **data definitions**.

We can even go so far as developing function **templates** based on the data definitions of the values it consumes.
Informally: a list of strings is either empty, or consists of a **first** string followed by a list of strings (the **rest** of the list).

```scheme
;; A (listof Str) is one of:
;; * empty
;; * (cons Str (listof Str))
```

This is a **recursive** data definition; the definition refers to itself. At least one case refers to itself (is self-referential).

A **base** case does not refer to itself.

We can use this data definition to show rigorously that

```
(cons "a" (cons "b" empty)) is a (listof Str).
```
We can generalize lists of strings to other types by using an $X$.

;; A (listof $X$) is one of:
;; * empty
;; * (cons $X$ (listof $X$))

The $X$ represents a specific type, such as Str or Int.
One of the main ideas of the HtDP textbook is that the form of a program often mirrors the form of the data.

A **template** is a general framework within which we fill in specifics.

We create a template once for each new form of data, and then apply it many times in writing functions that consume that type of data.

A template is derived from a data definition.
We start with the data definition for a (listof X):

`; A (listof X) is one of:
`;  * empty
`;  * (cons X (listof X))

A function consuming a (listof X) will need to distinguish between these two cases.
> Template for processing a (listof X)

;; A (listof X) is one of:
;; * empty
;; * (cons X (listof X))

;; listof-X-template: (listof X) -> Any
(define (listof-X-template lox)
  (cond [(empty? lox) ...]
        [(cons? lox) ...]))

The ... represents a place to fill in code specific to the problem.

From the data definition, when the (listof X) is not empty, we know it is a (cons X (listof X)). We want to “undo” the cons. The function that does this is rest:

(rest (cons a b)) ⇒ b for all valid values a and b.
> Template for processing a (listof X)

;; listof-X-template: (listof X) -> Any
(define (listof-X-template lox)
  (cond [[(empty? lox) ...]
        [(cons? lox) (... (first lox) ... (rest lox) ...)])

Now we go a step further.

Because (rest lox) is of type (listof X), we apply the same computation to it – that is, we apply listof-X-template.
> Completed template for processing a (listof X)

```scheme
;; listof-X-template: (listof X) -> Any
(define (listof-X-template lox)
  (cond [(empty? lox) ...]
        [(cons? lox) (... (first lox) ...
                         (listof-X-template (rest lox)))]))
```

This is the template for a function consuming a (listof X). Its form parallels the data definition.

We can now fill in the dots for a specific example – counting the number of concerts in a list.
**Problem:** Write a function to count the number of concerts in a list of concerts.

We begin with writing the purpose, examples, contract, and then copying the template and renaming the function and parameters.

```scheme
;; (count-concerts loc) counts the number of concerts in loc
;; Examples:
(check-expect (count-concerts empty) 0)
(check-expect (count-concerts (cons "a" (cons "b" empty))) 2)

;; count-concerts: (listof Str) -> Nat
(define (count-concerts loc)
  (cond [(empty? loc) ...]
          [else (... (first loc) ... [else (... (first loc) ...
                                      ... (count-concerts (rest loc)) ...)])])
```

Here are three crucial questions to help think about functions consuming a list:

- What does the function produce in the base case?
- What does the function do to the first element in a non-empty list?
- How does the function combine the value produced from the first element with the value obtained by applying the function to the rest of the list?
(check-expect (count-concerts empty) 0)
(check-expect (count-concerts (cons "a" (cons "b" empty))) 2)

;; count-concerts: (listof Str) -> Nat
(define (count-concerts loc)
  (cond [(empty? loc) 0]
        [else (+ 1 (count-concerts (rest loc)))]))

This is a recursive function (it uses recursion).
A function is recursive when the body of the function involves an application of the same function.

This is an important technique which we will use quite frequently throughout the course.

Fortunately, our substitution rules allow us to trace such a function without much difficulty.
> Tracing count-concerts

(count-concerts (cons "a" (cons "b" empty)))
⇒ (cond [(empty? (cons "a" (cons "b" empty))) 0]
  [else (+ 1 (count-concerts
       (rest (cons "a" (cons "b" empty))))))])
⇒ (cond [false 0]
  [else (+ 1 (count-concerts
       (rest (cons "a" (cons "b" empty))))))])
⇒ (cond [else (+ 1 (count-concerts
       (rest (cons "a" (cons "b" empty))))))]
⇒ (+ 1 (count-concerts (rest (cons "a" (cons "b" empty))))))
⇒ (+ 1 (count-concerts (cons "b" empty)))]
⇒ (+ 1 (cond [(empty? (cons "b" empty)) 0]
        [else (+ 1 (count-concerts (rest (cons "b" empty))))))])
\[
\begin{align*}
&\Rightarrow (+ 1 (\text{cond} \ [\text{false} \ 0]) \\
&\quad [\text{else} (+ 1 (\text{count-concerts} (\text{rest} (\text{cons} \ "b" \ \text{empty}))))])]) \\
&\Rightarrow (+ 1 (\text{cond} \ [\text{else} (+ 1 (\text{count-concerts} (\text{rest} (\text{cons} \ "b" \ \text{empty}))))])]) \\
&\Rightarrow (+ 1 (+ 1 (\text{count-concerts} (\text{rest} (\text{cons} \ "b" \ \text{empty})))))) \\
&\Rightarrow (+ 1 (+ 1 (\text{count-concerts} \ \text{empty}))) \\
&\Rightarrow (+ 1 (+ 1 (\text{cond} \ [(\text{empty?} \ \text{empty}) \ 0]) \\
&\quad [\text{else} (+ 1 (\text{count-concerts} (\text{rest} \ \text{empty}))))])]) \\
&\Rightarrow (+ 1 (+ 1 (\text{cond} \ [\text{true} \ 0] [\text{else} (+ 1 (\text{count-concerts} (\text{rest} \ \text{empty}))))])))) \\
&\Rightarrow (+ 1 (+ 1 0)) \\
&\Rightarrow (+ 1 1) \\
&\Rightarrow 2
\end{align*}
\]
Remember that if \texttt{los} is a \texttt{(listof Str)}, we can get that first value (\texttt{first los}), and the rest of the values with (\texttt{rest los}).

**Exercise**

Write a function that gives the total length of all the \texttt{Str} in a \texttt{(listof Str)}.

\[
\text{(total-length (cons "U2" (cons "DaCapo" (cons "DaCapo" empty))))}
\]

\Rightarrow (+ 2 (+ 6 (+ 6 0))) \Rightarrow 14

Key things to keep in mind:

- Our base case is when the list is \texttt{empty}. What is the answer in this case?
- If the list is not empty, image we already have the answer for a list 1 smaller. How can we combine that answer with the value from \texttt{first} to get the answer for the bigger list?
Condensed traces

The full trace contains too much detail, so we instead use a condensed trace of the recursive function. This shows the important steps and skips over the trivial details.

This is a space saving tool we use in these slides, not a rule that you have to understand.
The condensed trace of our example

(count-concerts (cons "a" (cons "b" empty)))
⇒ (+ 1 (count-concerts (cons "b" empty)))
⇒ (+ 1 (+ 1 (count-concerts empty)))
⇒ (+ 1 (+ 1 0))
⇒ 2

This condensed trace shows more clearly how the application of a recursive function leads to an application of the same function to a smaller list, until the base case is reached.
From now on, for the sake of readability, we will tend to use condensed traces. At times we will condense even more (for example, not fully expanding constants).

If you wish to see a full trace, you can use the Stepper.

But as we start working on larger and more complex forms of data, it becomes harder to use the Stepper, because intermediate expressions are so large.
Write a recursive function \( \text{sum} \) that consumes a \( \text{(listof} \ \text{Int}) \) and produces the sum of all the values in the list.

\[
(\text{sum} \ (\text{cons} \ 6 \ (\text{cons} \ 7 \ (\text{cons} \ 42 \ \text{empty}))))) \Rightarrow 55
\]

Consider:

- If I add up no items, what must the total be?
- If I have the first item, and a list containing all the other items, what is the sum of all the items?
It's important that our functions always terminate (stop running and produce an answer). Why does \texttt{count-concerts} always terminate?

There are two conditions. Either

- it's the base case, which produces 0 and immediately terminates
- or, it's the recursive case which applies \texttt{count-concerts} to a shorter list. Each recursive application is to a shorter list, which must eventually become empty and terminate.

We will eventually generalize “a shorter list” to “a smaller version of the same problem” where “a smaller version” depends on the nature of the problem. Perhaps a smaller number terminating at 0 or fewer elements that meet a certain criteria.

Does this remind you of induction? It should!
Exercise: Take the code to `count-concerts` and remove the application of `rest` so that the recursive application of `count-concerts` is on a list of the same size. What does DrRacket do when you run it? Be specific!
Thinking recursively

The similarity of recursion to induction suggests a way to think about developing recursive functions.

- Get the base case right.
- **Assume** that your function correctly solves a problem of size $n$ (e.g. a list with $n$ items).
- Figure out how to use that solution to solve a problem of size $n + 1$. 
The template is a good place to start writing code. Write the listof-X-template. Then alter it to complete count-waterboys.

;; (count-waterboys los) produces the number of occurrences of "Waterboys" in los

;; Examples:
(check-expect (count-waterboys empty) 0)
(check-expect (count-waterboys (cons "Waterboys" empty)) 1)
(check-expect (count-waterboys (cons "DaCapo"
                                       (cons "U2" empty))) 0)

;; count-waterboys: (listof Str) -> Nat
(define (count-waterboys los) ...)
Generalize count-waterboys to a function which also consumes the string to be counted.

;; (count-string s los) counts the number of occurrences of s in los.
;; Examples:

(check-expect
  (count-string "ab" (cons "bc" (cons "ab" (cons "d" empty))))) 1)

;; count-string: Str (listof Str) -> Nat
(define (count-string s los) ...)

The recursive function application will be (count-string s (rest los)).
> Refining the \((\text{listof } X)\) template

Sometimes, each \(X\) in a \((\text{listof } X)\) may require further processing. Indicate this with a template for \(X\) as a helper function.

\[
;; \text{listof-}X\text{-template}: (\text{listof } X) \rightarrow \text{Any}
\]

\[
(\text{define} \ (\text{listof-}X\text{-template} \ \text{lox})
(\text{cond} \ [(\text{empty?} \ \text{lox}) \ ...]
[\text{else} \ (\ ... \ (\text{X-template} \ (\text{first} \ \text{lox})) \ ... \ (\text{listof-}X\text{-template} \ (\text{rest} \ \text{lox})) \ ...])))
\]

We assume this generic data definition and template from now on.
A template provides the basic shape of the code as suggested by the data definition.

The `listof-X-template` corresponds to what can be done with `foldr` (or `foldl`).

But there exist computations that cannot be handled by either!

We need to become skillful with recursion. Later in the course we will use recursion to solve tasks that do not fit into the template.
The list template has the property that the form of the code matches the form of the data definition.

We will call this **simple recursion**.

There are other patterns of recursion which we will see later on in the course.

Until we do, the functions we write (and ask you to write) will use simple recursion (and hence will fit the form described by such templates).

**Use the templates.**
In simple recursion, every argument in a recursive function application is either:

- unchanged, or
- one step closer to a base case, using the inverse of the function in the data definition.

(For lists, which are defined using `cons`, the inverse function is `rest`.)

(define (func lst) (... (func (rest lst)) ...)) ;; Simple
(define (func lst x) (... (func (rest lst) x) ...)) ;; Simple
(define (func lst x) (... (func (process lst) x) ...)) ;; NOT Simple
(define (func lst x)
  (... (func (rest lst) (math-function x)) ...)) ;; NOT Simple
A closer look at `count-concerts` reveals that it will work just fine on any list. In fact, it is a built-in function in Racket, under the name `length`.

Another useful built-in function is `member?`, which consumes an element of any type and a list, and returns `true` if the element is in the list, or `false` if it is not present.
Write your own implementation of `member?`. You’ll need to name it `my-member?` because Racket complains if you try to redefine a built-in function.

```racket
;; (my-member? item lst) produce true if item is in lst;
;; produce false otherwise.
;; Examples:
(check-expect (my-member? 1 empty) false)
(check-expect (my-member? 1 (cons 1 empty)) true)
(check-expect (my-member? "b" (cons 0 (cons "a" empty))) false)
```

To allow your function to work on any data type, use the built-in function `equal?`. It produces `false` if its arguments are not of the same type, and `true` if they have the same type and value.
Consider \texttt{negate-list}, which consumes a list of numbers and produces the same list with each number negated (3 becomes $-3$).

\begin{verbatim}
;; (negate-list \textit{lon}) produces a list with every number in \textit{lon} negated
(check-expect (negate-list empty) empty)
(check-expect (negate-list (cons 2 (cons -12 empty)))
             (cons -2 (cons 12 empty)))
\end{verbatim}

Since \texttt{negate-list} consumes a \texttt{(listof Num)}, we use the general list template to write it.
(check-expect (negate-list empty) empty)
(check-expect (negate-list (cons 2 (cons -12 empty)))
  (cons -2 (cons 12 empty)))

;; negate-list: (listof Num) -> (listof Num)
(define (negate-list lon)
  (cond [(empty? lon) ...]
        [else (... (first lon) ... (negate-list (rest lon)) ... )])))
;; (negate-list lon) produces a list with every number in lon negated
;; Examples:
(check-expect (negate-list empty) empty)
(check-expect (negate-list (cons 2 (cons -12 empty)))
  (cons -2 (cons 12 empty)))

;; negate-list: (listof Num) -> (listof Num)
(define (negate-list lon)
  (cond [(empty? lon) empty]
        [else (cons (- (first lon)) (negate-list (rest lon)))]))
> A condensed trace

(negate-list (cons 2 (cons -12 empty)))
⇒ (cons (- 2) (negate-list (cons -12 empty)))
⇒ (cons -2 (negate-list (cons -12 empty)))
⇒ (cons -2 (cons (- -12) (negate-list empty)))
⇒ (cons -2 (cons 12 (negate-list empty)))
⇒ (cons -2 (cons 12 empty))
Write a recursive function \texttt{keep-evens} that consumes a \texttt{(listof Int)} and returns the list of even values.

\begin{enumerate}
\item \texttt{(keep-evens (\texttt{cons} 4 (\texttt{cons} 5 (\texttt{cons} 8 (\texttt{cons} 10 (\texttt{cons} 11 empty))))))} \Rightarrow \texttt{(cons} 4 \texttt{(cons} 8 (\texttt{cons} 10 empty)))
\item \texttt{(keep-evens (\texttt{cons} 5 empty))} \Rightarrow \texttt{empty}
\item \texttt{(keep-evens (\texttt{cons} 4 empty))} \Rightarrow \texttt{(cons} 4 \texttt{empty)}
\end{enumerate}

That is, use recursion to duplicate the following function:

\begin{verbatim}
(define (keep-evens L) (filter even? L))
\end{verbatim}
Sometimes a given computation makes sense only on a non-empty list — for instance, finding the maximum of a list of numbers.

Exercise

Create a self-referential data definition for (ne-listof X), a non-empty list of X.

Exercise

Develop a template for a function that consumes an (ne-listof X).

Exercise

Finally, write a function to find the maximum of a non-empty list of numbers.
When we introduce new types, like \texttt{(ne-listof X)}, we need to include it in the design recipe. For each new type, place the following someplace between the top of the program and the first place the new type is used. This information is only needed \textit{once}.

- The data definition
- The template derived from that data definition

Assignments do \textit{not} need to include the data definition or template for \texttt{(listof X)}. \texttt{(ne-listof X)} (which does not appear in the slides) and other types you may define \textit{should} be included in your assignments, unless the assignment states otherwise.

The design recipe requirements for each function remain unchanged.
Example:
;; A (listof X) is one of:
;;  * empty
;;  * (cons X (listof X))

Every data definition will have a name (e.g. (listof X) that can be used in contracts.

In a self-referential data definition (like (listof X)):

- at least one clause (and possibly more) will use the definition’s name to show how to build a “larger” version of the data.
- at least one clause (and possibly more) must *not* use the definition’s name; these are base cases.
The template follows directly from the data definition.

The overall shape of a self-referential template will be a cond expression with one clause for each clause in the data definition.

Self-referential data definition clauses lead to recursive expressions in the template.

Base case clauses will not lead to recursion.
Write a recursive function `longest-word` that consumes `(ne-listof Str)` and produces the length of the longest word in the list.

;; (longest-word words) produces the length of the longest word in the list of words.

;; Examples:

(check-expect (longest-word (cons "and" empty)) 3)
(check-expect (longest-word (cons "and" (cons "then" empty))) 4)

Recall the built-in function `string-length`.
we have already worked with strings by converting them to \(\text{(listof Char)}\), then working with higher order functions. Recall the functions \(\text{string->list}\) and \(\text{list->string}\).

Now let's take another look at that same idea, using recursion.
Write a function to count the number of occurrences of a specified character in a 
(listof Char). Later we will see how to make this work on a Str.

;; (count-char/list ch loc) counts the number of occurrences 
;; of ch in loc.

;; Examples:
(check-expect (count-char/list #\e (string->list "")) 0)
(check-expect (count-char/list #\e (string->list "beekeeper")) 5)
(check-expect (count-char/list
    #\o (cons #\f (cons #\o (cons #\o (cons #\d empty))))) 2)

;; count-char/list: Char (listof Char) -> Nat
(define (count-char/list ch loc) ... )
;; (count-char/list ch loc) counts the number of occurrences
;; of ch in loc.
;; Examples:
(check-expect (count-char/list #\e (string->list "")) 0)
(check-expect (count-char/list #\e (string->list "beekeeper")) 5)

;; count-char/list: Char (listof Char) -> Nat
(define (count-char/list ch loc)
  (cond [(empty? loc) 0]
    [else (+ (cond [(char=? ch (first loc)) 1]
               [else 0])
           (count-char/list ch (rest loc)))]))
Write a function \texttt{e->*list} that consumes a \texttt{(listof Char)}, and replaces each \#\texttt{e} with a \#\texttt{*}. Use recursion.

\begin{verbatim}
;; (e->*list lst) Replace each \#\texttt{e} in lst with \#\texttt{*}
;; Examples:
(check-expect (e->*list (cons \#h (cons \#e (cons \#y (cons \#! empty)))))
  (cons \#h (cons \#* (cons \#y (cons \#! empty))))
\end{verbatim}
Our functions should be easy to use. The problem statement was to count characters in a string, not in a list of characters.

We shouldn’t expect the user of our function to know that to use `count-char/list` they need to convert their string to a list of characters.

In such cases it’s good practice to include a **wrapper function** — a simple function that “wraps” the main function and takes care of housekeeping details like converting the string to a list.
(define (count-char ch s) (count-char/list ch (string->list s)))
Wrapper functions:

- are short and simple
- call another function that does much more
- sets up the appropriate conditions for calling the other function, usually by transforming one or more of its parameters or providing a starting value for one of its arguments
Write a wrapper function `e->*` that consumes a `Str`, and replaces each `e` with `*`. Use `e->*/list` to do the recursive part.

;; (e->* s) Replace each "e" in s with "*".

;; Examples:
(check-expect (e->* "hello world, how are you?"
               )
              "h*llo world, how ar* you?")
First write a function `(add-item item lon)` that consumes a `Num` and a `(listof Num)`. It produces a list where each value in `lon` has had `item` added to it.

;; Examples:
(check-expect (add-item 7 (cons 7 (cons 3 (cons 5 empty))))
  (cons 14 (cons 10 (cons 12 empty))))

Then use `add-item` as a helper function to write a function `(add-first numbers)` that consumes a `(listof Num)` and adds the first value in `numbers` to each item in `numbers`.

;; Examples:
(check-expect (add-first (cons 7 (cons 3 (cons 5 empty))))
  (cons 14 (cons 10 (cons 12 empty))))

`add-first` is an example of a wrapper function.
You should understand the data definitions for lists, how the template mirrors the definition, and be able to use the template to write recursive functions consuming this type of data.

You should understand the additions made to the semantic model of Beginning Student to handle lists, and be able to do step-by-step traces on list functions.

You should understand and use \((\text{listof } X)\) notation in contracts.

You should understand strings, their relationship to characters and how to convert a string into a list of characters (and vice-versa).

You should understand when a wrapper function is appropriate and be able to write one.
In this module we added the following to our toolbox:

... cons? equal?

We also are putting the following aside, for now:

filter foldl foldr lambda length list map range

These are the functions and special forms currently in our toolbox:

Write a function `drop-first` that consumes a non-empty \((\text{listof Any})\), and produces a \((\text{listof Any})\) with all copies of the first item removed.

\[
\text{(check-expect}
\begin{align*}
\text{(drop-first (cons 1 (cons 'V (cons 4 (cons 1 (cons "!" empty)))))))} \\
\text{(cons 'V (cons 4 (cons "!" empty))))}
\end{align*}
\text{)}
\]

Again, use `equal?` to compare equality of values of any type.

You likely will write `drop-first` as a wrapper around a recursive function with an extra parameter. What does the recursive function need to do?
Now write another function, `remove-first-char`, that consumes a `Str` and removes all copies of the first character.

```
(check-expect (remove-first-char "aardvark") "rdvrk")
(check-expect (remove-first-char "would a woodchuck chuck wood?"
  "ould a oodchuck chuck ood?"
)
```

*This function will also be a wrapper, possibly around another wrapper. You can potentially have many “layers” of wrapper!*