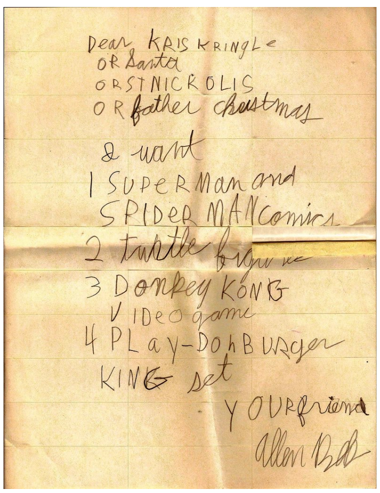
06: Lists

What is a list?



https://www.flickr.com/photos/genealogyphotos/3095254725

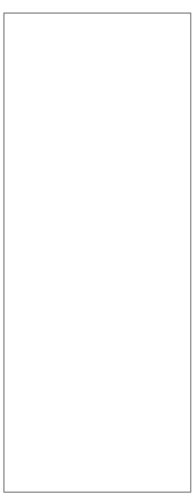
Numbers, strings, symbols and Boolean values can represent a single data item. However, for a Christmas wish list a single item is clearly insufficient.

There are many circumstances in which we need more than a single item of data: groceries to buy at a store, names of all the students in a class, today's transactions on your credit card, etc.

The amount of data is often unbounded, meaning it may grow or shrink – and you don't know how much. The order of values may also be important.

Many programming languages meet this need with **lists**.

Empty Lists



This is the simplest possible list – a list with nothing on it.

In Racket, it's represented as

```
empty
```

`empty` is a value, just like `49`, `"comic book"`, `'earth` and `(make-inventory "dry lentils" 0.79 42)`.

Like any other value, it can be named as a constant:

```
(define wish-list empty)
```

An empty list drawn like this: 

Non-empty Lists

comic book

A list with one wish.
In Racket, we can add one item to an empty list:

```
(cons "comic book" empty)
```

`(cons "comic book" empty)` is a value,
like `empty`, `42`, and `(make-inventory "dry lentils" 0.79 42)`.
Notice that it uses constructor syntax, like structures.

We can name it: `(define wish-list (cons "comic book" empty))`

A one-element list is drawn like this:

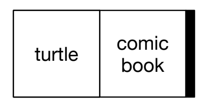
Non-empty Lists

turtle
comic book

A list with two wishes.
In Racket, we can add "turtle" to our previous, one-element, list:

```
(cons "turtle" (cons "comic book" empty))
```

A two-element list is drawn like this:



In Racket, lists grow at the “front”. This explains the unusual ordering of the list items on the left – we’re trying to match how Racket does it.

Non-empty Lists

bicycle
video game
play-doh
turtle
comic book

`cons` consumes a value and a list. It produces a new list that is one element longer than the one it was given.

```
(cons "bicycle"
      (cons "video game"
            (cons "play-doh"
                  (cons "turtle"
                        (cons "comic book"
                              empty)))))
```

Use `cons` to build a list as long as you want, one item at a time.

`cons`, the formalities

- `empty` is a list.
- `(cons v lst)`, where `v` is a value and `lst` is a list, produces a new list, `lst'`, such that:
  - The first value of `lst'` is `v`.
  - The rest of the items in `lst'` are the same as the values in `lst` in the same order.

Deconstructing a list

A list can be taken apart:

- `(first lst)` produces the first element of `lst`; `lst` must not be empty.
- `(rest lst)` produces the rest of the elements of `lst`; `lst` must not be empty.

Examples:

```
(define wishes (cons "bicycle" (cons "video game" (cons "comic book" empty))))

(first wishes) ⇒ "bicycle"
(rest wishes) ⇒ (cons "video game" (cons "comic book" empty))

(first (rest wishes)) ⇒ "video game"
(first (rest (rest wishes))) ⇒ "comic book"
(first (rest (rest (rest wishes))))   ⇒
       first: expects a non-empty list, given: empty
```

Ex. 1

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`).

Predicates

There are several predicates that work with lists:

- `(empty? v)` Consumes a value; produces `true` if `v` is `empty` and `false` otherwise.
- `(cons? v)`: Consumes a value; produces `true` if `v` is a `cons` value and `false` otherwise.
- `(list? v)`: Equivalent to `(or (cons? v) (empty? v))`.

```
(empty? empty) ⇒ true
(empty? (cons 1 (cons 2 (cons 3 empty)))) ⇒ false
(empty? 'earth) ⇒ false

(cons? empty) ⇒ false
(cons? (cons 1 (cons 2 empty))) ⇒ true
(cons? 'earth) ⇒ false

(list? empty) ⇒ true
(list? (cons 1 (cons 2 empty))) ⇒ true
(list? 'earth) ⇒ false
```

A **cons value** is a value produced by `(cons v lst)`. In other words, a list that contains at least one value.

Summary: 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))`.

> Simple functions on lists

Using these built-in functions, we can write our own simple functions on lists.

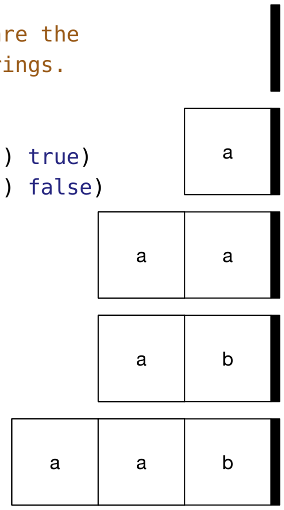
```
;; (add-replace lst-nums) replaces the first two numbers in the list
;;    with their sum.
;; Examples:
(check-expect (add-replace (cons 2 (cons 3 (cons 4 empty))))
              (cons 5 (cons 4 empty)))
(check-expect (add-replace (cons 10 (cons 20 empty)))
              (cons 30 empty))

;; add-replace: (listof Num) → (listof Num)
;; Requires: (length lst-nums) >= 2
(define (add-replace lst-nums)
  (cons (+ (first lst-nums)
           (first (rest lst-nums)))
        (rest (rest lst-nums))))
```

> Simple functions on lists (cont.)


```
;; (first-two-equal? los) determines if first two strings are the
;;     same; produces false if there are fewer than two strings.
(check-expect (first-two-equal? empty) false)
(check-expect (first-two-equal? (cons "a" empty)) false)
(check-expect (first-two-equal? (cons "a" (cons "a" empty))) true)
(check-expect (first-two-equal? (cons "a" (cons "b" empty))) false)

;; first-two-equal?: (listof Str) → Bool
(define (first-two-equal? los)
  (and (not (empty? los))
       (not (empty? (rest los)))
       (string=? (first los) (first (rest los)))))
```



Ex. 2

Write `first-two-equal-cond?` which is the same as `first-two-equal?` except that it uses a **`cond`** expression instead of **`and`**.
Does the order of the question-answer pairs matter? Why or why not?

Ex. 3

Write a function `remove-second` that consumes a list of length at least 2, and produces a list containing the same items, with the second item removed.

```
(remove-second (cons 'Mercury (cons 'Venus empty)))
⇒ (cons 'Mercury empty)
(remove-second (cons 2 (cons 4 (cons 6 (cons 0 (cons 1 empty))))))
⇒ (cons 2 (cons 6 (cons 0 (cons 1 empty))))
```

You'll need `cons` in addition to `first` and `rest`.

Contracts involving lists

What is the contract for `(add-replace lst-nums)`?

We could use "`List`" for `lst-nums`.

However, we almost always need to answer the question "list of what?". A list of numbers? A list of strings? A list of any type at all?

> (listof X) notation in contracts

We'll use (listof X) in contracts, where X may be replaced with any type.

Examples:

- (listof Str): a wish list or grocery list.
- ;; add-replace: (listof Num) → (listof Num)
- ;; first-two-equal?: (listof Str) → Bool

Other examples: (listof Bool), (listof Sym), and (listof Any).

Replace X with the most appropriate type available.

(listof X) always includes the empty list, empty.

! It's (listof Str), **not** (Listof Str) or (list-of Str).

Syntax and semantics: Values

List values are

1. empty
2. (cons v lst), where v is any Racket value (including list values) and lst is a list value (which includes empty).

Note that values and expressions look very similar!

Value: (cons 1 (cons 2 (cons 3 empty)))

Expression: (cons 1 (cons (+ 1 1) (cons 3 empty))) ;; (+ 1 1) is not a value

Racket list values are traditionally given using **constructor notation** – the same notation we would use to construct the value. We will see other representations in Module 8.

Syntax and semantics: Substitution rules

The substitution rules are:

(first (cons a lst)) ⇒ a, where a and lst are values.

(rest (cons a lst)) ⇒ lst, where a and lst are values.

(empty? empty) ⇒ true.

(empty? a) ⇒ false, where a is any Racket value other than empty.

(cons? (cons a lst)) ⇒ true, where a and lst are values.

(cons? a) ⇒ false, where a is any Racket value not created using cons.

Most interesting functions will process the entire consumed list.

- How many wishes are on the list?
- How many times does "bicycle" appear?
- What is the largest value in a list of numbers?
- What's the sum of 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.

As we did with structures, we'll develop a data definition and template for lists.

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).

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

This is a recursive data definition; the definition refers to itself. Here, the definition of `(listof Str)` refers to a `(listof Str)` in the second clause.

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))
```

Replace `X` with a specific type such as `Str`, `Int`, or `Char`.

Templates and data-directed design

One of CS135's main ideas 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.

> Template for processing a `(listof X)`

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.

In the last case we **know** from the data definition that there is a first `X` and the rest of the list of `X`'s, so...

> 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)

```
;; 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 wishes in a list.

Processing lists: how many wishes?

Problem: Write a function to count the number of wishes in a list of wishes.

1. Write the purpose, examples, and contract.
2. Copy the listof-X-template.
3. Rename the function and any parameters.
4. Fill in the details.

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

;; count-wishes: (listof Str) → Nat
(define (count-wishes wishes)
  (cond [(empty? wishes) 0]
        [(cons? wishes) (+ 1
                           (count-wishes (rest wishes)))]))
```

Processing lists: thinking about list functions

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

1. What should the function produce in the base case?
2. What should applying the function to the rest of the list produce?
3. What should the function do to the first element in a non-empty list?
4. How should the function combine #2 and #3 to produce the answer for the entire list?

Processing lists: count-wishes


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

;; count-wishes (listof Str) → Nat
(define (count-wishes los)
  (cond [(empty? los) 0]
        [else (+ 1 (count-wishes (rest los)))]))
```

count-wishes 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-wishes


```
(count-wishes (cons "a" (cons "b" empty)))
⇒ (cond [(empty? (cons "a" (cons "b" empty)) 0]
        [else (+ 1 (count-wishes (rest (cons "a" (cons "b" empty)))))])

⇒ (cond [false 0]
        [else (+ 1 (count-wishes (rest (cons "a" (cons "b" empty)))))])

⇒ (cond [else (+ 1 (count-wishes (rest (cons "a" (cons "b" empty)))))])

⇒ (+ 1 (count-wishes (rest (cons "a" (cons "b" empty)))))

⇒ (+ 1 (count-wishes (cons "b" empty)))

⇒ (+ 1 (cond [(empty? (cons "b" empty)) 0]
             [else (+ 1 (count-wishes (rest (cons "b" empty))))]))
```

```
⇒ (+ 1 (cond [false 0]
             [else (+ 1 (count-wishes (rest (cons "b" empty))))]))

⇒ (+ 1 (cond [else (+ 1 (count-wishes (rest (cons "b" empty))))]))

⇒ (+ 1 (+ 1 (count-wishes (rest (cons "b" empty)))))

⇒ (+ 1 (+ 1 (count-wishes empty)))

⇒ (+ 1 (+ 1 (cond [(empty? empty) 0]
                  [else (+ 1 (count-wishes (rest empty)))])))

⇒ (+ 1 (+ 1 (cond [true 0][else (+ 1 (count-wishes (rest empty)))])))

⇒ (+ 1 (+ 1 0))
⇒ (+ 1 1)
⇒ 2
```

> 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.

```
(count-wishes (cons "a" (cons "b" empty)))
⇒ (+ 1 (count-wishes (cons "b" empty)))
⇒ (+ 1 (+ 1 (count-wishes 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.

> Condensed traces

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.

Ex. 4

Write a recursive function `sum` that consumes a `(listof Int)` and produces the sum of all the values in the list.

`(sum (cons 6 (cons 7 (cons 42 empty))))` ⇒ 55

Consider the four questions:

1. What should the function produce in the base case?
2. What should the function do to the first element in a non-empty list?
3. What should applying the function to the rest of the list produce?
4. How should the function combine #2 and #3 to produce the answer for the entire list?

Termination

It's important that our functions always **terminate** (stop running and produce an answer). Why does `count-wishes` 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 `count-wishes` **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!

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 $n+1$.

> Example: count-bicycles


```
;; (count-bicycles wishes) produces the number of occurrences
;;      of "bicycle" in wishes
;; Examples:
(check-expect (count-bicycles empty) 0)
(check-expect (count-bicycles (cons "bicycle" empty)) 1)
(check-expect (count-bicycles (cons "comic"
                                    (cons "play-doh" empty))) 0)

;; count-bicycles (listof Str) → Nat
(define (count-bicycles wishes) ...)
```

The template is a good place to start writing code. Write the template. Then, alter it according to the specific function you want to write.

> Example: count-string

We can generalize `count-bicycles` to a function which also consumes the string to be counted.

```
;; (count-string s los) counts the number of occurrences of s in los.
;; Example:
(check-expect
   (count-string "ab" (cons "bc" (cons "ab" (cons "d" empty)))) 1)

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

> Refining the `(listof X)` template


```
;; listof-X-template: (listof X) → Any
```

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

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

We assume this generic data definition and template from now on.

> Templates as generalizations

A template provides the basic shape of the code as suggested by the data definition.

Later in the course, we will learn about an abstraction mechanism (higher-order functions) that can reduce the need for templates.

We will also discuss alternatives for tasks where the basic shape provided by the template is not right for a particular computation.

Patterns of recursion

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.

> Simple recursion

In simple recursion, every argument in a recursive function application is either:

- unchanged, or
- *one step* closer to a base case according to a data definition

```
(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
```

length

A closer look at `count-wishes` reveals that it will work just fine on any list.

In fact, it is a built-in function in Racket, under the name `length`.

Misusing `length`

Just like `count-wishes`, `length` always “handles” every element in the list. This can be very inefficient compared to alternatives. Consider the following seemingly equivalent functions:

```
(define (count-bicycles-v1 wishes)
  (cond [(empty? wishes) 0]
        [(cons? wishes) (+ (cond [(string=? (first wishes) "bicycle") 1]
                                 [else 0])
                           (count-bicycles-v1 (rest wishes)))]))


(define (count-bicycles-v2 wishes)
  (cond [(= (length wishes) 0) 0]
        [(cons? wishes) (+ (cond [(string=? (first wishes) "bicycle") 1]
                                 [else 0])
                           (count-bicycles-v2 (rest wishes)))]))
```


Producing lists from lists

Consider `negate-list`, which consumes a list of numbers and produces the same list with each number negated (3 becomes −3).

```
;; (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) ... )
```

Since `negate-list` consumes a `(listof Num)`, we use the general list template to write it.

> negate-list with template


```
;; (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) ...]
        [else (... (first lon)
                   (negate-list (rest lon)))]))
```

> negate-list completed


```
;; (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))
```

Ex. 5

Write a recursive function `keep-evens` that consumes a `(listof Int)` and returns the list of even values.

```
(keep-evens (cons 4 (cons 5 (cons 8 (cons 10 (cons 11 empty)))))) ⇒ (cons 4
(cons 8 (cons 10 empty)))
(keep-evens (cons 5 empty)) ⇒ empty
(keep-evens (cons 4 empty)) ⇒ (cons 4 empty)
```

> Non-empty lists

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. Develop a template for a function that consumes a `(ne-listof X)`. Finally, write a function to find the maximum of a non-empty list of numbers.

Design recipe refinements

In an ideal world, each type used in a program would be defined with a data definition and a template derived from that data definition. All data definitions and templates are placed between the top of the program, before the first one is used. This information is only needed **once** per type.

In practise,

- types for which we have developed data definitions and templates in class, the slides, or the style guide (eg `(listof X)` and `(ne-listof X)`) do not need data definitions or templates included in assignments.
- we will explicitly identify types for which you **must** provide data definitions and templates in assignments.
- for all others, we strongly encourage you to write data definitions and templates because we believe they will help you write better code.

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.

Ex. 6

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)
```

Racket has a built-in function `(string-length s)` that produces the length of the string `s`.

Strings and lists of characters

Processing text is an extremely common task for computer programs. Text is usually represented in a computer by strings.

In Racket (and in many other languages), a string is really a sequence of characters in disguise.

Racket provides the function `string->list` to convert a string to an explicit list of characters.

The function `list->string` does the reverse: it converts a list of characters into a string.

> Racket notation for characters

Racket's notation for the character 'a' is `#\a`.

The result of evaluating `(string->list "test")` is
the list `(cons #\t (cons #\e (cons #\s (cons #\t empty))))`.

This is unfortunately ugly, but the `#` notation is part of a more general way of specifying values in Racket.

Racket has a number of built-in functions for characters:

- `char<=?`, `char<?`, `char=?`, `char>? char>=?`: character comparisons
- `char-numeric?`, `char-whitespace?`, etc: character classification predicates
- `char-upcase`, `char-downcase`: character conversions

> Removing characters from a string

Write a function that removes every occurrence of a specified character from a string.

```
;; (remove-char ch s) removes all ch characters from the string s.
;; Examples:
(check-expect (remove-char #\e "beekeeper") "bkpr")
(check-expect (remove-char #\e "string without E") "string without E")
(check-expect (remove-char #\e "") "")

;; remove-char: Char Str -> Str
(define (remove-char ch s) ...)
```

We haven't seen any functions to remove a character from a string.

Perhaps use a helper function together with `string->list` and `list->string`?

```
(define (remove-char ch s)
  (list->string (remove-char/lst ch (string->list s))))
```

> Removing characters from a list of characters


```
;; (remove-char/lst ch loc) removes all ch characters from the list of
;;      characters, loc.
;; Examples:
(check-expect (remove-char/lst #\e empty) empty)
(check-expect (remove-char/lst #\e (cons #\e (cons #\a (cons #\e empty)))) (cons
   #\a empty))
(check-expect (remove-char/lst #\e (string->list "beekeeper"))
              (cons #\b (cons #\k (cons #\p (cons #\r empty)))))

;; remove-char/lst: Char (listof Char) -> (listof Char)
(define (remove-char/lst ch loc)
  (cond [(empty? loc) empty]
        [(char=? ch (first loc)) (remove-char/lst ch (rest loc))]
        [else (cons (first loc)
                    (remove-char/lst ch (rest loc)))]))
```

> Wrapper functions

`remove-char` uses the helper function `remove-char/lst`. It's the helper function that does almost all of the "work". `remove-char` just sets up the problem for `remove-char/lst`.

We call `remove-char` a **wrapper function** – a simple function that "wraps" the main function and takes care of housekeeping details like converting the string to a list.

Wrapper functions:

- are short and simple
- always apply another function that does much more
- sets up the appropriate conditions for using the other function, usually by transforming one or more of its parameters or providing a starting value for one of its arguments

Ex. 7

Write a function, `e->*`, which consumes a `Str` and replaces each `e` with `*`.

```
;; (e->* s) Replace each "e" in s with "*".
;; Examples:
(check-expect (e->* "beekeeper") "b**k**p*r")
```

Your solution will consist of a wrapper function, `e->*`, and another function, `(e->*/lst loc)`, where `loc` is a `(listof Char)`. It replaces each `\#e` in `loc` with `\#*`.

```
(check-expect (e->*/lst (cons #\h (cons #\e (cons #\y (cons #\! empty)))))
              (cons #\h (cons #\* (cons #\y (cons #\! empty)))))

;; e->*/lst: (listof Char) → (listof Char)
```

Ex. 8

Write a function, `add-first`, which consumes a non-empty list of numbers and adds the first number to all the numbers in the list (including itself).

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

Your solution should consist of a wrapper function (`add-first`) and another function, `(add-item item lon)`, where `item` is a `Num` and `lst` is a `(listof Num)`. It adds `item` to each number in `lon`.

sorting a list

When writing a function to consume a list, we may find that we need to create a helper function to do some of the work. The helper function may or may not be recursive itself.

sorting a list of numbers provides a good example; in this case the solution follows easily from the templates and design process.

In this course and CS 136, we will see several different sorting algorithms.

> Filling in the list template


```
;; (sort lon) sorts the elements of lon in non-decreasing order
;; sort: (listof Num) → (listof Num)
(define (sort lon)
  (cond [(empty? lon) ...]
        [else (... (first lon)
                   (sort (rest lon)))]))

(check-expect (sort (cons 2 (cons 0 (cons 1 empty))))
              (cons 0 (cons 1 (cons 2 empty))))
```

If the list `lon` is empty, so is the result.

Otherwise, the template suggests doing something with the first element of the list, and the sorted version of the rest.

```
;; (sort lon) sorts the elements of lon in non-decreasing order
;; sort: (listof Num) → (listof Num)
(define (sort lon)
  (cond [(empty? lon) empty]
        [else (insert (first lon)
                      (sort (rest lon)))]))

(check-expect (sort (cons 2 (cons 0 (cons 1 empty)))) ...)
```

`insert` is a recursive helper function that consumes a number and a sorted list, and inserts the number into the sorted list.

> A condensed trace of `sort` and `insert`


```
(sort (cons 2 (cons 4 (cons 3 empty))))
⇒ (insert 2 (sort (cons 4 (cons 3 empty))))
⇒ (insert 2 (insert 4 (sort (cons 3 empty))))
⇒ (insert 2 (insert 4 (insert 3 (sort empty))))
⇒ (insert 2 (insert 4 (insert 3 empty)))
⇒ (insert 2 (insert 4 (cons 3 empty)))
⇒ (insert 2 (cons 3 (cons 4 empty)))
⇒ (cons 2 (cons 3 (cons 4 empty)))
```

> The helper function `insert`

We again use the list template for `insert`.

```
;; (insert n slon) inserts the number n into the sorted list slon...
;; Examples:
(define test-result (cons 1 (cons 2 (cons 3 empty))))
(check-expect (insert 1 empty) (cons 1 empty))
(check-expect (insert 1 (cons 2 (cons 3 empty))) test-result)
(check-expect (insert 2 (cons 1 (cons 3 empty))) test-result)

;; insert: Num (listof Num) → (listof Num)
;; Requires: slon is sorted in non-decreasing order
(define (insert n slon)
  (cond [(empty? slon) ...]
        [else (... (first slon)
                   (insert n (rest slon)))]))
```

The helper function `insert`

Use our four questions:

1. What should be produced for the base case?
2. What does the recursive application produce?
3. What should happen to the first element?
4. How should 2 and 3 be combined to solve the entire problem?

1. If `slon` is empty, the result is the list containing just `n`.
2. The rest of the list with `n` inserted in the correct place.
3. `n` is the first number in the result if it is less than or equal to the first number in `slon`. Otherwise, the first number in the result is the first number in `slon`, and the rest of the result is what we get when we insert `n` into `(rest slon)`.
4. See #3.

> Insert


```
(define (insert n slon)
  (cond [(empty? slon) (cons n empty)]
        [(<= n (first slon)) (cons n slon)]
        [else (cons (first slon) (insert n (rest slon)))]))
```

```
(insert 4 (cons 1 (cons 2 (cons 5 empty))))
⇒ (cons 1 (insert 4 (cons 2 (cons 5 empty))))
⇒ (cons 1 (cons 2 (insert 4 (cons 5 empty))))
⇒ (cons 1 (cons 2 (cons 4 (cons 5 empty))))|
```

Our `sort` with helper function `insert` are together known as **insertion sort**.

Ex. 9
Modify insertion sort to order the list from largest to smallest instead of smallest to largest.
Hint: It's a pretty small change.

Tracing in DrRacket


```
(require htdp-trace)

(define/trace (sort lon)
  (cond [(empty? lon) empty]
        [else
         (insert (first lon)
                 (sort (rest lon)))]))

(define/trace (insert n slon)
  (cond [(empty? slon) (cons n empty)]
        [(<= n (first slon)) (cons n slon)]
        [else (cons (first slon)
                    (insert n (rest slon)))]))

(check-expect
 (sort (cons 3 (cons 2 (cons 5 (cons 4 empty)))))
 (cons 2 (cons 3 (cons 4 (cons 5 empty)))))
```

```
>(sort (3 2 5 4))
> (sort (2 5 4))
> >(sort (5 4))
> > (sort (4))
> > >(sort ())
< < <()
> > (insert 4 ())
< < (4)
> >(insert 5 (4))
> > (insert 5 ())
< < (5)
< <(4 5)
> (insert 2 (4 5))
< (2 4 5)
>(insert 3 (2 4 5))
> (insert 3 (4 5))
< (3 4 5)
<(2 3 4 5)
The test passed!
```

Goals of this module

- 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 `(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.
- You should be able to use recursiver helper functions such as `insert` in the `sort` function.

Ex. 10

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

```
;; drop-first: (listof Sym) → (listof Sym)
;; Requires: lst is non-empty.
(define (drop-first lst)
  (remove-each (first lst) lst))
```

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?

The following functions and special forms have been introduced in this module:

```
char-alphabetic? char-downcase char-lower-case? char-numeric? char-upcase
    char-upper-case? char-whitespace? char<=? char<? char=? char>=? char>? char?
    cons cons? define/trace empty? first length list->string list? rest
    string->list
```

You should complete all exercises and assignments using only these and the functions and special forms introduced in earlier modules. The complete list is:

```
* + - ... / < <= = > >= abs and boolean? ceiling char-alphabetic? char-downcase
    char-lower-case? char-numeric? char-upcase char-upper-case? char-whitespace? char<=?
    char<? char=? char>=? char>? char? check-error check-expect check-within cond cons
    cons? cos define define-struct define/trace e else empty? error even? exp expt first
    floor integer? length list->string list? log max min modulo negative? not
    number->string number? odd? or pi positive? quotient remainder rest round sgn sin sqr
    sqrt string->list string-append string-downcase string-length string-lower-case?
    string-numeric? string-upcase string-upper-case? string<=? string<? string=? string>=?
    string>? string? substring symbol=? symbol? tan zero?
```