CS115 - Module 5 - Lists

Cameron Morland

Fall 2017

Reminder: if you have not already, ensure you:

- Read *How to Design Programs*, sections 9 and 10.

What is a list?

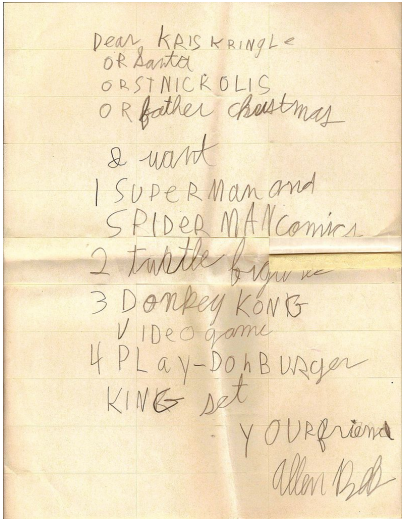


Photo Credit: Genealogyphotos

The word *list* comes from Old English "líste", meaning a strip (such a strip of cloth or paper).

> *"His targe wiþ gold list He carf atvo."*
> *(Guy of Warwick, ca. 1330)*

→ A strip of paper with items written on it.
→ An ordered collection of items.
We can make a list really easily:

```
(define wishes
  (list "comics" "turtle figures"
        "Donkey Kong" "Play-Doh Burger King"))

(define primes (list 2 3 5 7 11 13 17 19))
```

List Introduction

```
(define wishes
  (list "comics" "turtle figures"
        "Donkey Kong" "Play-Doh Burger King"))

(define primes (list 2 3 5 7 11 13 17 19))
```

We have two functions, **first** and **rest**, to extract items from lists:

(**first** wishes) ⇒ "comics"

(**rest** wishes) ⇒ (**list** "turtle figures" "Donkey Kong" "Play-Doh Burger King") [1].

(**first** (**rest** wishes)) ⇒ "turtle figures"

(**first** (**rest** (**rest** primes))) ⇒ 5

We also have some extra functions, **second**, **third**, . . . , **eighth**, which might be useful:

(**fourth** wishes) ⇒ "Play-Doh Burger King"

(**eighth** primes) ⇒ 19

[1] Select Language: Beginning Student with List Abbreviation

How does a list work?

Lists constructed in this way match our familiar experiences with lists. But how can a computer store items of arbitrary length?
Structures have a fixed number of fields. We want to be able to store any number of items.
Different programming languages implement lists in different ways.
The solution in Racket: a recursive definition.

A list is either:

- `'()`, which is sometimes written `empty`
- or a pair of exactly two items, `(cons f r)`, where
  - `f` is a value
  - `r` is a list

`cons` is a built-in function.

Examples[a]:

```
(list) ⇒ '()
(list 42) ⇒ (cons 42 '())
(list 1 2 3)
⇒ (cons 1 (cons 2 (cons 3 '())))
(rest (list 1 2 3))
⇒ (cons 2 (cons 3 '()))
```

[a] Select Language: Beginning Student.

Constructing lists

Choose Language: Beginning Student reveals the recursive nature of Racket lists.
Choose Language: Beginning Student with List Abbreviations hides the recursive nature.

For now we will avoid using the **`list`** function. We will prefer to construct our lists one item at a time, using **`cons`**.
Every list except the empty list `'()` is constructed of a value and another list.

```
(cons 'blue '())

(cons 42 (cons 7 '()))

(list 2 (make-posn 5 6) 3) ⇒ (cons 1 (cons (make-posn 5 6) (cons 3 '())))

(cons (* 2 3) (cons (* 3 4) (cons (* 4 5) '()))) ⇒
(cons 6 (cons 12 (cons 20 '())))
```

Substitution rules for lists

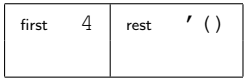
Formally, we implement **`first`** and **`rest`** as follows:

- `(`**`first`** `(`**`cons`** `element alist)) ⇒ element`
- `(`**`rest`** `(`**`cons`** `element alist)) ⇒ alist`

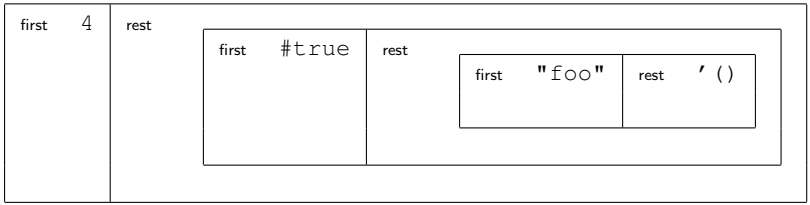
Note: the functions consume non-empty lists only.

Nested Box Visualization

```
(cons 4 '())
```

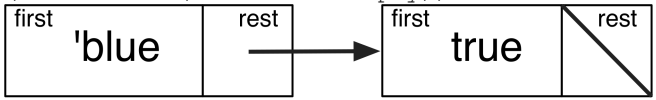



```
(cons 4 (cons #true (cons "foo" '())))
```

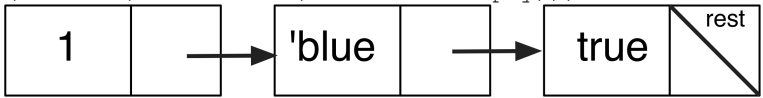


Box-and-pointer visualization

```
(cons 'blue (cons #true empty))
```




```
(cons 1 (cons 'blue (cons #true empty)))
```



Dynamic Typing and Lists

Since Racket is dynamically typed, the system allows us to have lists containing any value, even lists!

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

This is equivalent to `(list (list 1 2) (list 3 4))`

Lists and the Design Recipe

Lists of whatever are really handy, but we want to be able to specify what our lists are in the design recipe.
We use a **data definition**, but like lists themselves, it is recursive!

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

There is a **base case**, which is `'()`, and a **recursive case**.
Some examples of `(listof Sym)`:

- `'()`
- `(cons 'blue '())`
- `(cons 'red (cons 'blue '()))`

We do not need to write a data definition for each data type. `(listof Int)` is understood to be a list that is either empty, or contains values of type `Int`. Similarly for every data type.
To indicate a list which may contain different types, we will use `(listof Any)`.

Writing recursive functions on lists

Consider the following recursive function on the natural numbers:

```
;; (fact n) compute n!
;; fact: Nat -> Nat

(define (fact n)
  (cond [(= n 0) 1]
        [else (* n (fact (- n 1)))]))
```

This has one **base case**: it stops recursion when $n = 0$.

With lists often a base case is when the list is empty:

- `(empty? mylist)` returns `#true` when `mylist` is `empty`
- `(cons? mylist)` returns `#true` when `mylist` is a non-empty list. (We often prefer `else`.)

```
;; (list-len mylist) return the number of items in mylist.
;; list-len: (listof Any) -> Nat

(define (list-len mylist)
  (cond [(empty? mylist) 0]
        [else (+ 1 (list-len (rest mylist)))]))
```

A Template for Recursion on Lists

Many, many recursive functions on lists follow this basic form:

```
;; my-los-fun: (listof Sym) -> Any
(define (my-los-fun alos)
  (cond
    [(empty? alos) ... ]
    [else (... (first alos) ...
               (my-los-fun (rest alos)) ...)]))
```

We do something with the first, and we call ourselves on `rest`. This is OK since `(rest alos)` is a `(listof Sym)`. It satisfies the contract.

This is recursion: something defined in terms of itself.

Note the parallels between the code of the template, and the data definition:

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

The template is a `cond` with one clause for each clause in the data definition.

Tracing a Recursive Function

Consider tracing a call to a recursive function:

```
(define (list-len mylist)
  (cond [(empty? mylist) 0]
        [else (+ 1 (list-len (rest mylist)))]))

(list-len (cons 'a (cons 'b empty)))
⇒  (cond [(empty? (cons 'a (cons 'b empty))) 0]
         [else (+ 1 (list-len (rest (cons 'a (cons 'b empty)))))]))

⇒  (cond [#false 0]
         [else (+ 1 (list-len (rest (cons 'a (cons 'b empty)))))]))

⇒  (cond [else (+ 1 (list-len (rest (cons 'a (cons 'b empty)))))]))

⇒  (+ 1 (list-len (rest (cons 'a (cons 'b empty)))))

⇒  (+ 1 (list-len (cons 'b empty)))
```

```
(define (list-len mylist)
  (cond [(empty? mylist) 0]
        [else (+ 1 (list-len (rest mylist)))]))
```

...continuing from `(+ 1 (list-len (cons 'b empty)))`

⇒ `(+ 1 (cond [(empty? (cons 'b empty)) 0]`
`[else (+ 1 (list-len (rest (cons 'b empty))))]))`

⇒ `(+ 1 (cond [#false 0]`
`[else (+ 1 (list-len (rest (cons 'b empty))))]))`

⇒ `(+ 1 (cond [else (+ 1 (list-len (rest (cons 'b empty))))]))`

⇒ `(+ 1 (+ 1 (list-len (rest (cons 'b empty)))))`

⇒ `(+ 1 (+ 1 (list-len empty)))`

```
(define (list-len mylist)
  (cond [(empty? mylist) 0]
        [else (+ 1 (list-len (rest mylist)))]))
```

...continuing from `(+ 1 (+ 1 (list-len empty)))`

⇒ `(+ 1 (+ 1 (cond [(empty? empty) 0]`
`[else (+ 1 (list-len (rest empty)))])))`

⇒ `(+ 1 (+ 1 (cond [#true 0]`
`[else (+ 1 (list-len (rest empty)))])))`

⇒ `(+ 1 (+ 1 0))`

⇒ `(+ 1 1)`

⇒ `2`

Phew!

Condensed Trace

```
(define (list-len mylist)
  (cond [(empty? mylist) 0]
        [else (+ 1 (list-len (rest mylist)))]))

(list-len (cons 'a (cons 'b empty)))
```

$\Rightarrow$ `(+ 1 (list-len (cons 'b empty)))`

$\Rightarrow$ `(+ 1 (+ 1 (list-len empty)))`

$\Rightarrow$ `(+ 1 (+ 1 0))`

$\Rightarrow$ 2

The full trace contains too much detail, so we define the condensed trace with respect to a recursive function `my-fn` to be the following lines from the full trace:

- Each application of `my-fn`, showing its arguments;
- The result once the base case has been reached;
- The final value (if above expression was not simplified).

Structural versus non-Structural Recursion

So far, the form of our recursive code has matched the form of the data. Lists are defined recursively, and the form of the recursive code matches this definition.
This is called **structural recursion**.
It is possible to build code which is recursive without being structurally recursive.
Consider a function to sort `mylist` in the following manner:

1. Split `mylist` into `first-half` and `second-half`
2. Recursively sort `first-half` and `second-half`
3. Merge these two sorted lists into one list.

This would be recursive. But since the structure of a list is not "first half" and "second half", parts of it would not count as structural recursion.

You are not expected to do any non-structural recursion in this course.
Templates can help you write structurally recursive code.

Now we many have self-referential data. Every self-referential data type must have at least one base case which is *not* self-referential.

Consider:

```
;; A Symbol-Or-Num-List is one of:
;; * (cons Sym Symbol-Or-Num-List)
;; * (cons Num Symbol-Or-Num-List)
```

A problem! There is no way to stop.

Instead:

```
;; A Symbol-Or-Num-List is one of:
;; * empty
;; * (cons Sym Symbol-Or-Num-List)
;; * (cons Num Symbol-Or-Num-List)
```

Contracts

The types we can use now include:

- Atomic types (`Num`, `Int`, `Nat`, `Sym`, `Bool`, `Str`)
- Multiple possibilities using a `anyof` comment, e.g.

```
;; myfun: (anyof Int Str) -> (anyof Bool Nat)
```

- Any type defined in a data definition, e.g.

```
(make-struct foo (bar baz))
;; A Foo is a (Int (anyof Foo Num))
;; Requires:
;;   bar is a doohickey
;;   baz is a whatzit
```

- Lists, e.g. `(listof Int)`, or `(listof (anyof Foo Str))`.
  You may use `(listof X)` where `X` is any valid type, even a list.

A Generic Template

A template for a list of any type X is as follows:

```
;; my-listof-X-fun: (listof X) -> Any
(define (my-listof-X-fun lst)
  (cond
    [(empty? lst) ... ]
    [else (... (first lst) ... (my-listof-X-fun (rest lst)) ...)]))
```

Hint for template use: start with the base case(s), then do the recursive case(s).

Use the template to

Exercise: write a recursive function `list-sum` which consumes a `(listof Num)` and returns the sum of the numbers in the list.

Exercise: write a recursive function `list-max` which consumes a `(listof Nat)` and returns the largest number in the list. It's trickier if I ask for `(listof Int)`; why? Try it!

This `list-max` counts an empty list as having a max of zero, so it doesn't work with all negative numbers:

```
;; (list-max mylist) return the largest value in mylist
;; list-max: (listof Nat) -> Nat
;; Examples:
(check-expect (list-max (list 3 5 4)) 5)
(check-expect (list-max (list -3 -5 -4)) 0)

(define (list-max mylist)
  (cond
    [(empty? mylist) 0]
    [else (max (first mylist) (list-max (rest mylist)))]))
```

Instead, detect when the list is *almost* empty, and use that one item as our base case, like so:

```
    [(empty? (rest mylist)) (first mylist)]
```

Also add: `;; Requires: mylist is not empty.`

A little more complicated

Suppose I wanted to make a function which finds the maximum item in a list, but only of the items less than some upper value.

```
;; (list-max-under lst upper) return the largest value in lst
;;   that is less than upper.
;; list-max-under: (listof Nat) Nat -> Nat
;; Requires: lst is not empty.
;; Examples:
(check-expect (list-max-under (list 3 5 4) 10) 5)
(check-expect (list-max-under (list 3 5 4) 4) 3)
```

Exercise: Complete `list-max-under`.

A little more complicated Template

This example suggests a modification to the template:

```
;; my-extra-info-list-fun: (listof Any) Any -> Any
(define (my-extra-info-list-fun alist info)
  (cond
  [(empty? alist) ... ]
  [else (... (first alist) ... info ...
        (my-extra-info-list-fun (rest alist) info) ...)]))
```

Now we can store values which are unchanged (or even changed) in `info`.

Finding items in a list

Exercise: Complete `in?`.

```
;; (in? item mylist) return true if some value in mylist is item,
;;   else false.
;; in?: Any (listof Any) -> Bool
;; Examples:
(check-expect (in? 4 (list 1 2 3 5)) #false)
(check-expect (in? 3 (list 1 2 3 5)) #true)
(check-expect (in? 'blue (list 1 2 3 5)) #false)
(check-expect (in? 'blue (list 'red 'blue 'green)) #true)
```

In real code, you shouldn't use this function! There is a built-in function **`member?`** which does the same thing. Use it instead.

Built-in list functions

There are several built-in functions that consume lists. Usually, it is better to use an existing function than to write your own.
You are encouraged to use:

`cons`	to construct lists
`first`	extract the first item of a list
`rest`	to extract the list with the first item removed
`empty?`	to determine if a list is empty (often as a base case)
`cons?`	to determine if a list is non-empty (you may prefer to use `else`)
`length`	to find the length of the list
`member?`	to tell if an item is in a list.

For now, please do not use other built-in functions, including **`list`**, on your assignments.

Producing lists from lists

I want to do a little work on each item in a list. For example, I want to take each element x, and replace it with $10\sqrt{x}$. I can do this to a *single* value as follows:

```
(define (10rootx x) (* 10 (sqrt x)))
```

Suppose I want to do this to each value in the list:

`(fixit (list 81 49 4))` ⇒ `(list 90 70 20)`

`(fixit (cons 81 (cons 49 (cons 4 empty))))` ⇒ `(cons 90 (cons 70 (cons 20 empty)))`

Basecase: `empty?`, as usual.

Recursive call: will need to be `(cons <newvalue> <recursive call>)`

```
;; (fixit mylist) compute the function for each item in mylist
;; fixit: (listof Num) -> (listof Num)
;; Example:
(check-expect (fixit (list 81 49 4)) (list 90 70 20))
```

Exercise: Complete `fixit`, using `10rootx` as a helper function.

(Later we will have a wonderful function called **`map`** to do this for us.)

Removing items from a list

Consider the following function:

```
;; (removal mylist target) return mylist with all occurrences of target removed.
;; (listof Any) Any -> (listof Any)
;; Example:
(check-expect (removal (list 2 5 2 3 6 2) 2) (list 5 3 6))
```

Removing an item from the list is sort of like applying a function to each case.

Exercise: Complete `removal`.

```
(define (removal mylist target)
  (cond [(empty? mylist) empty]
        [(equal? target (first mylist)) (removal (rest mylist) target)]
        [else (cons (first mylist) (removal (rest mylist) target))]))
```

Removing duplicates

```
(define my-ints (list 2 5 3 2 5 7 2))
```

What does `(removal my-ints (first my-ints))` give us?

```
(list 5 3 5 7)
;; (de-duplicate mylist) return mylist with duplicates removed.
;; (listof Any) -> (listof Any)
;; Example:
(check-expect (de-duplicate (list 2 5 2 3 6 3)) (list 2 5 3 6))
```

Exercise: Complete `de-duplicate` using `removal` as a helper function.

Wrapper functions

Sometimes your recursion requires extra parameters, or needs to have data in a different format than provided. A wrapper function can help here.

An example using a wrapper: `gcd`

The greatest common divisor (gcd) of two natural numbers is the largest natural number that divides evenly into both.

```
(my-gcd 10 25) => 5
(my-gcd 20 22) => 2
(my-gcd 47 21) => 1
```

One way to compute `gcd` is to count down from one of the numbers until you find a number which divides both.

```
;; (my-gcd-under a b t) return the largest
;;   number t or less, that divides a and b.
;; Examples:
(check-expect (my-gcd-under 60 40 100) 20)
(check-expect (my-gcd-under 60 40 18) 10)
```

Exercise: Complete `my-gcd-under`.
You may use the following helper function:

```
(define (divisible? n d) (= 0 (remainder n d)))
```

But I want `my-gcd` to have the contract `Nat Nat -> Nat`.
I can fix this with a wrapper:

```
(define (my-gcd a b) (my-gcd-under a b a))
```

Strings and characters

In addition to using **`string`**`=?`, `substring`, and the other string manipulating functions, we can convert between strings, `Str`), and lists of characters, `(listof Char)`. We can also manipulate `Char` values.

- Convert to `(listof Char)`:
  `(string->`**`list`** `"foobar")` ⇒ `(`**`list`** `#\f #\o #\o #\b #\a #\r)`
- Convert to `Str`:
  `(list->`**`string`** `(`**`list`** `#\f #\o #\o #\b #\a #\r))` ⇒ `(`**`list`** `"foobar")`
- Compare two `Char`:
  `(`**`char`**`<? #\a #\c)` ⇒ `#true`
- Check if a `Char` is an uppercase letter:
  `(char-upper-case? #\q)` ⇒ `#false`
  `(char-upper-case? #\Q)` ⇒ `#true`
  `(char-upper-case? #\space)` ⇒ `#false`

Wrapper around a `(listof Char)` function

```
;; (removal mylist target) return mylist with all occurrences of target removed.
;; (listof Char) Char -> (listof Char)
;; Example:
(check-expect (removal (list 2 5 2 3 6 2) 2) (list 5 3 6))
(check-expect (removal (cons #\n (cons #\o (cons #\n (cons #\e empty)))) #\n)
              (cons #\o (cons #\e empty)))
```

To process a `Str`, you may:

- Convert the `Str` to `(listof Char)` using `string->`**`list`**
- Process this list using recursion
- Convert back to a `Str` using `list->`**`string`**

Writing lists with `Char` in them directly is annoying. You may write tests for a wrapper which consumes and returns `Str` instead of `(listof Char)`. An example is on the next slide.

Wrapper around a `(listof Char)` function

```
;; (removal mylist target) return mylist with all occurrences of target removed.
;; (listof Char) Char -> (listof Char)
(define (removal mylist target)
  (cond [(empty? mylist) empty]
        [(equal? target (first mylist)) (removal (rest mylist) target)]
        [else (cons (first mylist) (removal (rest mylist) target))]))


;; (remove-char s c) remove every instance of c in s.
;; remove-char: Str Char -> Str
;; Example:
(check-expect (remove-char "his mass" #\s) "hi ma")

(define (remove-char s c)
  (list->string (removal (string->list s) c)))
```

If a function consumes `(listof Char)` and has a wrapper function that consumes `(Str)`, it is sufficient to test just the wrapper function.

Wrapper around a `(listof Char)` function

To process a `Str`, you may:

- Convert the `Str` to `(listof Char)` using `string->`**`list`**
- Process this list using recursion
- Convert back to a `Str` using `list->`**`string`**

Exercise: Using the approach described above, create wrapper function `flatten-uppercase`, and the necessary helper function.

```
;; (flatten-uppercase s) replace all uppercase letters with "-".
;; flatten-uppercase: Str -> Str
;; Example:
(check-expect (flatten-uppercase "Hello World!") "-ello -orld!")
```

Some bits you will need:

- `(`**`first`** `(string->`**`list`** `"-"))` ⇒ `#\-`
- `(char-upper-case? #\q)` ⇒ `#false` and `(char-upper-case? #\Q)` ⇒ `#true`

Determining portions of a total

We wish to write a function `portions`:

```
;; (portions L) return the list of fractions of the total of each number in L.
;; (listof Num) -> (listof Num)
;; Example:
(check-expect (portions (list 6 1 3)) (list 0.6 0.1 0.3))
```

We can make this work if `portions` itself isn't recursive!

```
(define (portions L)
  (divide-each L (list-sum L))
```

All that remains is to write a helper function:

```
;; (divide-each L v) divide each value in L by v.
;; divide-each: (listof Num) Num -> (listof Num)
;; Examples:
(check-expect (divide-each (list 2 4 6) 2) (list 1 2 3))
```

Exercise: Complete `divide-each`.

Structures containing lists

Suppose we store the name of a server along with a list of tips collected. How might we store the information?

```
(define-struct server (name tips))
;; A Server is a (make-server Str (listof Num))
;; requires:
;;   numbers in tips are non-negative
```

Consider the template for a function that consumes a `Server`:

```
(define (my-server-fun s)
  (...(server-name s)...
   (my-lon-fun (server-tips s))...))
```

...and the template for a function that consumes a `(listof Num)`:

```
(define (my-lon-fun alon)
  (cond
    [(empty? alon) ... ]
    [else (...(first alon)...
           (my-lon-fun (rest alon))...
          )]))
```

Structures containing lists

```
;; (total-tips tips) return the sum of the tips.
;; total-tips: (listof Num) -> Num
;; Example:
(check-expect (total-tips (list 2 7 4 5)) 18)

(define (total-tips tips)
  (cond [(empty? tips) 0]
        [else (+ (first tips) (total-tips (rest tips)))]))


;; (server-total-tips person) add up person's tips.
;; server-total-tips: Server -> Num
;; Example:
(check-expect (server-total-tips (make-server "Bob" (list 2 4 6 7 3))) 22)

(define (server-total-tips person)
  (total-tips (server-tips person)))
```

Working with a list which is inside a structure is just like working with a list not in a structure.

Lists of structures

```
;; (biggest-take workers) return the amount of money received by the
;;   best-tipped person in workers.
;; biggest-take: (listof Server) -> Num
;; Requires: workers is not empty.
;; Example:
(check-expect (biggest-take
               (list (make-server "Ali" (list 3 5 7
                     (make-server "Bob" (list 1 1 1
                     (make-server "Kim" (list 4 8 4)))) 16)

(define (biggest-take workers)
  (cond [(empty? (rest workers)) (server-total-tips (first workers))]
        [else (max (server-total-tips (first workers))
                   (biggest-take (rest workers)))]))
```

Recursion with a list of structures is no different than with a list of numbers. Use selectors to access the fields, just as with non-recursive code.

Exercise: Write a function `(find-multiples L d)` that returns a list containing all the elements of `L` that are divisible by `d`.

```
;; find-multiples: (listof Nat) Nat -> listof Nat
```

Hint: remember the `divisible?` function we used earlier:

```
(define (divisible? n d) (= 0 (remainder n d)))
```

Exercise: Write a function `(find-two-multiples L d1 d2)` that returns a list containing all the elements of `L` that are divisible by `d1` or `d2`.

```
;; find-multiples: (listof Nat) Nat Nat -> listof Nat
```

Exercise: Write a function `(find-multi-multiples L divisors)` that returns a list containing all the elements of `L` that are divisible by at least one element in `divisors`.

```
;; find-multi-multiples: (listof Nat) (listof Nat) -> listof Nat
```

Hint: write a predicate helper function that consumes a `Num` and a `(listof Num)`, and returns `#true` if the `Num` is divisible by some item in the list.

Module summary

Be familiar with the terms recursion, recursive, self-referential, and base case.
Become comfortable using lists. See how to convert between lists in `(list 1 2 3)` format and `(cons 1 (cons 2 (cons 3 empty)))` format. Be comfortable constructing lists using **`cons`**, and taking them apart using **`first`** and **`rest`**.
Become comfortable writing recursive functions on lists. Use wrapper functions when they are necessary.
Use `Requires` and `(listof <type>)` appropriately in design recipes.

Before we begin the next module, please

- Read *How to Design Programs*, sections 11, 12, and 13 (Intermezzo 2).