More lists

Readings: HtDP, sections 11, 12, 13 (Intermezzo 2).

Topics:

- Sorting a list
- List abbreviations
- Lists containing lists
- Dictionaries and association lists
- Lists of lists as 2D data
- Processing two lists simultaneously
- Consuming a list and a number

Sorting a list

When writing a function to consume a list, we may find that we need to create an 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 nondecreasing order
;; sort: (listof Num) → (listof Num)
(check-expect (sort (cons 2 (cons 0 (cons 1 empty)))) ...)

(define (sort lon)
  (cond [(empty? lon) ...]
        [else (... (first lon) ... (sort (rest lon)) ...)]))
```

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 nondecreasing order
;; sort: (listof Num) → (listof Num)
(check-expect (sort (cons 2 (cons 0 (cons 1 empty)))) ...)

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

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

We again use the list template for `insert`.

```
;; (insert n slon) inserts the number n into the sorted list slon
;;     so that the resulting list is also sorted.
;; insert: Num (listof Num) → (listof Num)
;;     requires: slon is sorted in nondecreasing order
(define (insert n slon)
  (cond [(empty? slon) ...]
        [else (... (first slon) ...
                   (insert n (rest slon)) ...)]))
```

If `slon` is empty, the result is the list containing just `n`.

If `slon` is not empty, another conditional expression is needed.

`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)`.

> 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**.

List abbreviations

Now that we understand lists, we can abbreviate them.

In DrRacket, "Beginning Student With List Abbreviations" provides new syntax for list abbreviations, and a number of additional convenience functions.

Remember to follow the instructions in Module 01 when changing language levels.

The expression

```
(cons exp1 (cons exp2 (... (cons expn empty)...)))
```

can be abbreviated as

```
(list exp1 exp2 ... expn)
```

The result of the trace we did on the last slide can be expressed as `(list 1 2 4 5)`.

`(second my-list)` is an abbreviation for `(first (rest my-list))`.

`third`, `fourth`, and so on up to `eighth` are also defined.

Use these **sparingly** to improve readability.

The templates we have developed remain very useful.

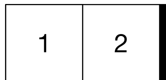
Note that `cons` and `list` have different results and different purposes.

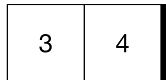
We use `list` to construct a list of fixed size (whose length is known when we write the program).

We use `cons` to construct a list from one new element (the first) and a list of arbitrary size (whose length is known only when the second argument to `cons` is evaluated during the running of the program).

Lists containing lists

Here are two different two-element lists.





We now know two different ways to construct these lists:

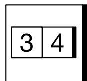
```
(cons 1 (cons 2 empty))
(cons 3 (cons 4 empty))
```

OR

```
(list 1 2)
(list 3 4)
```

Lists can contain anything, including other lists, at which point these abbreviations can improve readability.

Here is a one-element list whose single element is one of the two-element lists we saw above.



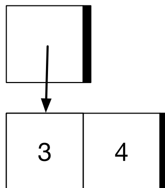
As before, we now know two different ways we could construct this.

```
(cons (cons 3 (cons 4 empty))
      empty)
```

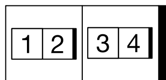
OR

```
(list (list 3 4))
```

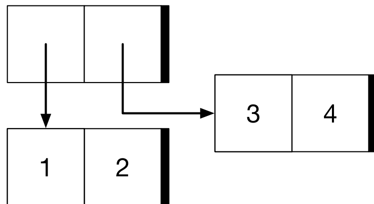
When the thing a list contains is complicated, we may draw an arrow to it, as shown on the right. This visualization represents the same list as the one above.



We can create a two-element list, each of whose elements is itself a two-element list. These are two different visualizations of **the same list.**



OR



Such a list can be created in code two different ways:

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

OR

```
(list (list 1 2)
      (list 3
            4))
```

Clearly, the abbreviations are more expressive.

> Example: processing a payroll

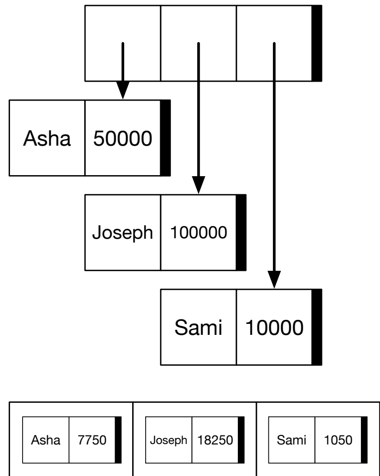
A company needs to process their payroll – a list of employee names and their salaries. It produces a list of each employee name and the tax owed. The tax owed is computed with `tax-payable` from Module 04.

Payroll:

```
(list (list "Asha" 50000)
      (list "Joseph" 100000)
      (list "Sami" 7000))
```

TaxOwed:

```
(list (list "Asha" 7750)
      (list "Joseph" 18250)
      (list "Sami" 1050))
```



» Data definitions

```
;; A Payroll is one of:
;; * empty
;; * (cons (list Str Num) Payroll)
```

```
;; A TaxOwed is one of:
;; * empty
;; * (cons (list Str Num) TaxOwed)
```

Note the use of `(list Str Num)` rather than `(listof X)`.

```
;; (payroll-template pr)
;; payroll-template: Payroll → Any
(define (payroll-template pr)
  (cond [(empty? pr) ...]
        [(cons? pr) ... (first pr) ...
                    ... (payroll-template (rest pr)) ...]))
```

A payroll is just a list, so this looks exactly like the `(listof X)` template – **so far...**

Some information from our data definition is not yet captured in the template: The list's first item is known to be of the form `(list Str Num)`.

It's useful to reflect that fact in the template:

- It reminds us of all the data available to us when solving the problem.
- Our solutions (derived from the template) will often access the parts of the sublist.

```
;; (payroll-template pr)
;; payroll-template: Payroll → Any
(define (payroll-template pr)
  (cond [(empty? pr) ...]
        [(cons? pr) (... (first (first pr)) ...
                     ... (first (rest (first pr))) ...
                     ... (payroll-template (rest pr)) ...)]))
```

Some short helper functions will make our code more readable.

```
;; (name lst) produces the first item from lst -- the name.
(define (name lst) (first lst))
;; (amount lst) produces the second item from lst -- the amount.
(define (amount lst) (first (rest lst)))
```

```
;; (payroll-template pr)
;; payroll-template: Payroll → Any
(define (payroll-template pr)
  (cond [(empty? pr) ...]
        [(cons? pr) (... (name (first pr)) ...
                     ... (amount (first pr)) ...
                     ... (payroll-template (rest pr)) ...)]))
```

Non-recursive helper functions only need a purpose.

» Start design recipe; fill in template

```
;; (compute-taxes payroll) calculates the tax owed for each
;; employee/salary pair in the payroll.
;; compute-taxes: Payroll → TaxOwed
(check-expect (compute-taxes test-payroll) test-taxes)

(define (compute-taxes payroll)
  (cond [(empty? payroll) ...]
        [(cons? payroll) (... (name (first payroll)) ...
                          ... (amount (first payroll)) ...
                          ... (compute-taxes (rest payroll)) ...)]))
```

» Finish compute-taxes

```
;; (compute-taxes payroll) calculates the tax owed for each
;; employee/salary pair in the payroll.
;; compute-taxes: Payroll → TaxOwed
(check-expect (compute-taxes test-payroll) test-taxes)

(define (compute-taxes payroll)
  (cond [(empty? payroll) empty]
        [(cons? payroll)
         (cons (list (name (first payroll))
                     (tax-payable (amount (first payroll))))
               (compute-taxes (rest payroll)))]))
```

» Alternate solution

```
(define (compute-taxes-alt payroll)
  (cond [(empty? payroll) empty]
        [(cons? payroll) (cons (sr->tr (first payroll))
                               (compute-taxes-alt (rest payroll)))]))

;; (sr->tr salary-rec) consumes a salary record and produces the
;; corresponding tax record
;; sr->tr: (list Str Num) -> (list Str Num)
(define (sr->tr salary-rec)
  (list (name salary-rec) (tax-payable (amount salary-rec))))
```

```
;; (payroll-template pr)
;; payroll-template: Payroll → Any
(define (payroll-template pr)
  (cond [(empty? pr) ...]
        [(cons? pr) (... (salary-rec-template (first pr)) ...
                    ... (payroll-template (rest pr)) ...)]))

(define (salary-rec-template sr) (... (name sr) ... (amount sr) ...))
```

Different kinds of lists

When we introduced lists in module 06, the items they contained were not lists. These were **flat lists**.

We have just seen **lists of lists**. A `Payroll` is a list containing a two-element flat list.

In later lecture modules, we will use lists containing unbounded flat lists.

We will also see **nested lists**, in which lists may contain lists that contain lists, and so on to an arbitrary depth.

Dictionaries

Once upon a time, a dictionary was a book in which you look up a word to find a definition. Nowdays, a dictionary is an app:

More generally, a **dictionary** contains a number of unique **keys**, each with an associated **value**.

Examples:

- A dictionary: keys are words; values are definitions.
- Your contacts list: keys are names; values are telephone numbers.
- Your seat assignment for midterms: keys are userids; values are seat locations.
- Stocks: keys are symbols; values are prices.

Many two-column tables can be viewed as dictionaries. The previous examples can all be viewed as two-column tables.

> Dictionary operations

What *operations* might we wish to perform on dictionaries?

- **lookup**: given a key, produce the corresponding value
- **add**: add a (key,value) pair to the dictionary
- **remove**: given a key, remove it and its associated value

One simple solution uses an **association list**, which is just a list of (key, value) pairs.
We store the pair as a two-element list. For simplicity, we will use numbers as keys and strings as values.

```
;; An association list (AL) is one of:
;; ⋆ empty
;; ⋆ (cons (list Num Str) AL)
;;       Requires: each key (Num) is unique
```

Example:

```
(list (list 8 "Asha")
      (list 2 "Joseph")
      (list 5 "Sami"))
```

8 Asha
2 Joseph
5 Sami

We can create association lists based on other types for keys and values. We use `Num` and `Str` here just to provide a concrete example.

Since we have a data definition, we could use `AL` for the type of an association list, as given in a contract.

Another alternative is to use `(listof (list Num Str))`.

We can use the data definition to produce a template.

```
;; al-template: AL → Any
(define (al-template alst)
  (cond [(empty? alst) ...]
        [else (... (first (first alst)) ... ; first key
                   (second (first alst)) ... ; first value
                   (al-template (rest alst)))]))
```

> Lookup operation

Recall that `lookup` consumes a key and a dictionary (association list) and produces the corresponding value.

In coding `lookup`, we have to make a decision. What should it produce if the lookup fails?

Since every string (including `""`) is a valid value, `lookup` can produce `false` to indicate that the key was not present in the association list.

```
(check-expect (lookup 2 (list (list 8 "Asha")
                              (list 2 "Joseph")
                              (list 5 "Sami")) "Joseph")
(check-expect (lookup 1 (list (list 8 "Asha")
                              (list 2 "Joseph")
                              (list 5 "Sami")) false)
```

```
;; (lookup-al k alst) produces the value corresponding to key k,
;;     or false if k not present
;; lookup-al:

(define (lookup-al k alst)
  (cond [(empty? alst) false]
        [(= k (first (first alst))) (second (first alst))]
        [else (lookup-al k (rest alst))]))
```

What is the contract for `lookup-al`?

We need a way to indicate that it can produce either a string or `false`.

> (anyof ...) notation in contracts

Use (anyof X Y ...) to mean any of the listed types or values.

Examples:

- (anyof Num Str)
- (anyof Str Num Bool)
- (anyof 1 2 3)
- (listof (anyof Str false))

```
;; foo: Num → (anyof Str Bool Num)
(define (foo x)
    (cond [(< x 0) "negative"]
          [(= x 0) false]
          [(= x 1) true]
          [else x]))
```

Dictionaries: summary

We will leave the `add` and `remove` functions as exercises.

The association list solution is simple enough that it is often used for small dictionaries.

For a large dictionary, association lists are inefficient. For example, consider the case where the key is not present and the whole list must be searched.

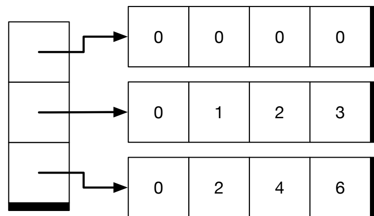
In a future module, we will impose structure to improve this situation.

Two-dimensional data

Another use of lists of lists is to represent a two-dimensional table.

For example, here is a multiplication table:

```
(mult-table 3 4) ⇒
(list (list 0 0 0 0)
      (list 0 1 2 3)
      (list 0 2 4 6))
```



The cth entry of the rth row (numbering from 0) is $r \times c$.

We can write `mult-table` using two applications of the “count up” idea.

Make one row of the table but counting the columns from 0 up to `nc`, doing the required multiplication for each one.

This will be a helper function in the final solution.

```
;; cols-to: Nat Nat Nat → (listof Nat)
;; Example:
(check-expect (cols-to 0 3 5) (list 0 3 6 9 12))
(check-expect (cols-to 0 4 5) (list 0 4 8 12 16))

(define (cols-to c r nc)
  (cond [(>= c nc) empty]
        [else (cons (* r c) (cols-to (add1 c) r nc))]))
```

> Put multiple rows together

```
;; (mult-table nr nc) produces multiplication table
;;     with nr rows and nc columns
;; mult-table: Nat Nat → (listof (listof Nat))

(define (mult-table nr nc)
  (rows-to 0 nr nc))

;; (rows-to r nr nc) produces mult. table, rows r...(nr-1)
;; rows-to: Nat Nat Nat → (listof (listof Nat))
(define (rows-to r nr nc)
  (cond [(>= r nr) empty]
        [else (cons (cols-to 0 r nc) (rows-to (add1 r) nr nc))]))
```

Processing two lists simultaneously

We now look at a more complicated recursion, namely writing functions which consume two lists (or two data types, each of which has a recursive definition).

Following the textbook, we will distinguish three different cases, and look at them in order of complexity.

The simplest case is when one of the lists does not require recursive processing.

As an example, consider the function `my-append`.

```
;; (my-append lst1 lst2) appends lst2 to the end of lst1
;; my-append: (listof Any) (listof Any) → (listof Any)
;; Examples:
(check-expect (my-append empty (list 'a 'b 'c)) (list 'a 'b 'c))
(check-expect (my-append (list 3 4) (list 1 2 5)) (list 3 4 1 2 5))

(define (my-append lst1 lst2)
  ...)
```

```
(define (my-append lst1 lst2)
  (cond [(empty? lst1) lst2]
        [else (cons (first lst1)
                    (my-append (rest lst1) lst2))]))
```

The code only does simple recursion on `lst1`.

The parameter `lst2` is “along for the ride”.

`append` is a built-in function in Racket.

» A condensed trace

```
(my-append (list 1 2 3) (list 4 5 6))
⇒ (cons 1 (my-append (list 2 3) (list 4 5 6)))
⇒ (cons 1 (cons 2 (my-append (list 3) (list 4 5 6))))
⇒ (cons 1 (cons 2 (cons 3 (my-append (list ) (list 4 5 6)))))
⇒ (cons 1 (cons 2 (cons 3 (list 4 5 6))))
```

The last line is the same as

`(cons 1 (cons 2 (cons 3 (cons 4 (cons 5 (cons 6 empty))))))`.

That's the same as `(list 1 2 3 4 5 6)`.

> Case 2: processing in lockstep

To process two lists `lst1` and `lst2` in lockstep, they must be the same length and be consumed at the same rate.

`lst1` is either `empty` or a `cons`, and the same is true of `lst2` (four possibilities in total).

However, because the two lists must be the same length, `(empty? lst1)` is `true` if and only if `(empty? lst2)` is `true`.

This means that out of the four possibilities, two are invalid for proper data.

The template is thus simpler than in the general case.

```
(define (lockstep-template lst1 lst2)
  (cond [(empty? lst1) ... ]
        [else
         (... (first lst1) ... (first lst2) ...
              (lockstep-template (rest lst1) (rest lst2)) ... )]))
```

» Example: dot product

To take the dot product of two vectors, we multiply entries in corresponding positions (first with first, second with second, and so on) and sum the results.

Example: the dot product of (1 2 3) and (4 5 6) is
$1 \cdot 4 + 2 \cdot 5 + 3 \cdot 6 = 4 + 10 + 18 = 32$.

We can store the elements of a vector in a list, so (1 2 3) becomes `(list 1 2 3)`.

For convenience, we define the empty vector with no entries, represented by `empty`.

» dot-product

```
;; (dot-product lon1 lon2) computes the dot product
;;       of vectors lon1 and lon2
;; dot-product: (listof Num) (listof Num) → Num
;;    requires: lon1 and lon2 are the same length
(check-expect (dot-product empty empty) 0)
(check-expect (dot-product '(2) '(3)) 6)
(check-expect (dot-product '(2 3 4 5) '(6 7 8 9)) (+ 12 21 32 45))


(define (dot-product lon1 lon2)
  ...)
```

» dot-product

```
;; (dot-product lon1 lon2) computes the dot product
;;       of vectors lon1 and lon2
;; dot-product: (listof Num) (listof Num) → Num
;;    requires: lon1 and lon2 are the same length
(check-expect (dot-product empty empty) 0)
(check-expect (dot-product '(2) '(3)) 6)
(check-expect (dot-product '(2 3 4 5) '(6 7 8 9)) (+ 12 21 32 45))


(define (dot-product lon1 lon2)
  (cond
    [(empty? lon1) 0]
    [else (+ (* (first lon1) (first lon2))
             (dot-product (rest lon1) (rest lon2)))]))
```

» A condensed trace

```
 (dot-product (list 2 3 4)
              (list 5 6 7))
⇒ (+ 10 (dot-product (list 3 4)
                     (list 6 7)))
⇒ (+ 10 (+ 18 (dot-product (list 4)
                           (list 7))))
⇒ (+ 10 (+ 18 (+ 28 (dot-product (list )
                                 (list )))))
⇒ (+ 10 (+ 18 (+ 28 0)))
⇒ (+ 10 (+ 18 28))
⇒ (+ 10 46)
⇒ 56
```

Exercise 1

Write a recursive function `vector-add` that adds two vectors.

`(vector-add '(3 5) '(7 11))` ⇒ `'(10 16)`

`(vector-add '(3 5 1 3) '(2 2 9 3))` ⇒ `'(5 7 10 6)`

Exercise 2

Complete `join-names`.

```
(define gnames '("Joseph" "Burt" "Douglas" "James" "David"))
(define snames '("Hagey" "Matthews" "Wright" "Downey" "Johnston"))
;; (join-names G S) Make a list of full names from G and S.
;; join-names: (listof Str) (listof Str) → (listof Str)
;; Example:
(check-expect (join-names gnames snames)
              '("Joseph Hagey" "Burt Matthews" "Douglas Wright"
                                "James Downey" "David Johnston"))
```

> Case 3: processing at different rates

If the two lists `lst1`, `lst2` being consumed are of different lengths, all four possibilities for their being empty/nonempty are possible:

```
(and (empty? lst1) (empty? lst2))
(and (empty? lst1) (cons? lst2))
(and (cons? lst1) (empty? lst2))
(and (cons? lst1) (cons? lst2))
```

Exactly one of these is true, but all must be tested in the template.

The template so far

```
(define (twolist-template lst1 lst2)
  (cond [(and (empty? lst1) (empty? lst2)) ...]
        [(and (empty? lst1) (cons? lst2)) ...]
        [(and (cons? lst1) (empty? lst2)) ...]
        [(and (cons? lst1) (cons? lst2)) ...]))
```

The first case is a base case; the second and third may or may not be.

» Refining the template

```
(define (twolist-template lst1 lst2)
  (cond
    [(and (empty? lst1) (empty? lst2)) ...]
    [(and (empty? lst1) (cons? lst2))
     (... (first lst2) ... (rest lst2) ...)]
    [(and (cons? lst1) (empty? lst2))
     (... (first lst1) ... (rest lst1) ...)]
    [(and (cons? lst1) (cons? lst2))
     ??? ]))
```

The second and third cases may or may not require recursion.

The fourth case definitely does, but its form is unclear.

There are many different possible natural recursions for the last `cond` answer `???`:

```
... (first lst2) ... (twolist-template lst1 (rest lst2)) ...

... (first lst1) ... (twolist-template (rest lst1) lst2) ...

... (first lst1) ... (first lst2)
... (twolist-template (rest lst1) (rest lst2)) ...
```

We need to reason further in specific cases to determine which is appropriate.

» Example: merging two sorted lists

We wish to design a function `merge` that consumes two lists.

Each list is sorted in ascending order (no duplicate values).

`merge` will produce one such list containing all elements.

As an example:

```
(merge (list 1 8 10) (list 2 4 6 12)) ⇒ (list 1 2 4 6 8 10 12)
```

We need more examples to be confident of how to proceed.

» Example: merging two sorted lists

```
(merge empty empty) ⇒ empty

(merge empty (list 2 6 9)) ⇒ (list 2 6 9)

(merge (list 1 3) empty) ⇒ (list 1 3)

(merge (list 1 4) (list 2)) ⇒ (list 1 2 4)

(merge (list 3 4) (list 2)) ⇒ (list 2 3 4)
```

» Reasoning about merge

If `lon1` and `lon2` are both nonempty, what is the first element of the merged list?

It is the smaller of `(first lon1)` and `(first lon2)`.

If `(first lon1)` is smaller, then the rest of the answer is the result of merging `(rest lon1)` and `lon2`.

If `(first lon2)` is smaller, then the rest of the answer is the result of merging `lon1` and `(rest lon2)`.

```
(define (merge lon1 lon2)
  (cond [(and (empty? lon1) (empty? lon2)) empty]
        [(and (empty? lon1) (cons? lon2)) lon2]
        [(and (cons? lon1) (empty? lon2)) lon1]
        [(and (cons? lon1) (cons? lon2))
         (cond [(< (first lon1) (first lon2))
                (cons (first lon1) (merge (rest lon1) lon2))]
               [else (cons (first lon2) (merge lon1 (rest lon2)))])]))
```

» A condensed trace

```
(merge (list 3 4)
       (list 2 5 6))
⇒ (cons 2 (merge (list 3 4)
                 (list 5 6))))
⇒ (cons 2 (cons 3 (merge (list 4)
                         (list 5 6))))
⇒ (cons 2 (cons 3 (cons 4 (merge empty
                                 (list 5 6)))))
⇒ (cons 2 (cons 3 (cons 4 (cons 5 (cons 6 empty)))))
```

Testing list equality

```
;; (list=? lst1 lst2) determines if lst1 and lst2 are equal
;; list=?: (listof Num) (listof Num) → Bool
(define (list=? lst1 lst2)
  (cond
    [(and (empty? lst1) (empty? lst2)) ...]
    [(and (empty? lst1) (cons? lst2))
     (... (first lst2) ... (rest lst2) ...)]
    [(and (cons? lst1) (empty? lst2))
     (... (first lst1) ... (rest lst1) ...)]
    [(and (cons? lst1) (cons? lst2))
     (???)]))
```

Two empty lists are equal; if one is empty and the other is not, they are not equal.

If both are nonempty, then their first elements must be equal, and their rests must be equal.

The natural recursion in this case is

```
(list=? (rest lst1) (rest lst2))
```

List equality code

```
(define (list=? lst1 lst2)
  (cond
    [(and (empty? lst1) (empty? lst2)) true]
    [(and (empty? lst1) (cons? lst2)) false]
    [(and (cons? lst1) (empty? lst2)) false]
    [(and (cons? lst1) (cons? lst2))
     (and (= (first lst1) (first lst2))
          (list=? (rest lst1) (rest lst2)))]))
```

Some further simplifications are possible.

> Built-in list equality

As you know, Racket provides the predicate `equal?` which tests structural equivalence. It can compare two simple values (numbers, strings, symbols, etc) or two lists containing any mixture of simple values and other lists.

We will soon add structures to our repertoir of values. Structures may also contain other structures or lists or simple values.

`equal?` can compare any of these for equality.

At this point, you can see how you might write `equal?` if it were not already built in. It would involve testing the type of data supplied, and doing the appropriate comparison, recursively if necessary.

Consuming a list and a number

We defined recursion on natural numbers by showing how to view a natural number in a list-like fashion.

We can extend our idea for computing on two lists to computing on a list and a number, or on two numbers.

A predicate “Does *elem* appear at least *n* times in this list?”

Example: “Does 2 appear at least 3 times in the list `(list 4 2 2 3 2 4)`?” produces `true`.

> The function at-least?

```
;; (at-least? n elem lst) determines if elem appears
;;        at least n times in lst.
;; at-least?: Nat Any (listof Any) → Bool
(check-expect (at-least? 0 'red (list 1 2 3)) true)
(check-expect (at-least? 3 "hi" empty) false)
(check-expect (at-least? 2 'red (list 'red 'blue 'red 'green)) true)
(check-expect (at-least? 3 'red (list 'red 'blue 'red 'green)) false)
(check-expect (at-least? 1 7 (list 5 4 0 5 3)) false)


(define (at-least-template? n elem lst)
```

The recursion involves the parameters `n` and `lst`, once again giving four possibilities:

```
(define (at-least-template? n elem lst)
  (cond [(and (zero? n) (empty? lst)) ...]
        [(and (zero? n) (cons? lst)) ...]
        [(and (> n 0) (empty? lst)) ...]
        [(and (> n 0) (cons? lst)) ...]))
```

Once again, exactly one of these four possibilities is true.

In which cases can we produce the answer without further processing?
In which cases do we need further recursive processing to discover the answer?
Which of the natural recursions should be used?

In working out the details for each case, it becomes apparent that some of them can be combined.

If `n` is zero, it doesn't matter whether `lst` is `empty` or not. Logically, every element always appears at least 0 times.

This leads to some rearrangement of the template, and eventually to the code that appears on the next slide.

```
(define (at-least? n elem lst)
  (cond [(zero? n) true]
        [(empty? lst) false]
        ; list is nonempty, n ≥ 1
        [(equal? (first lst) elem) (at-least? (sub1 n) elem (rest lst))]
        [else (at-least? n elem (rest lst))]))
```

> Two condensed traces

```
(at-least? 3 'green (list 'red 'green 'blue)) ⇒
(at-least? 3 'green (list 'green 'blue)) ⇒
(at-least? 2 'green (list 'blue)) ⇒
(at-least? 2 'green empty) ⇒ false

(at-least? 1 8 (list 4 8 15 16 23 42)) ⇒
(at-least? 1 8 (list 8 15 16 23 42)) ⇒
(at-least? 0 8 (list 15 16 23 42)) ⇒ true
```

Goals of this module

- You should understand the principle of insertion sort, and how the functions involved can be created using the design recipe.
- You should be able to use list abbreviations for lists where appropriate.
- You should be able to construct and work with lists that contain lists.
- You should understand the three approaches to designing functions that consume two lists (or a list and a number, or two numbers) and know which one is suitable in a given situation.

Exercise 3

Write a recursive function `(sum L)` that consumes a `(listof Num)` and returns the sum of the values in the list.

Write a recursive function `divide-each` that allows `portions-r` to achieve its purpose.

```
;; (portions-r L) divide each value in L by sum of L.
;; portions-r: (listof Num) → (listof Num)
;; Examples:
(check-expect (portions-r '(1 1 2)) '(0.25 0.25 0.5))
(check-expect (portions-r '(6 1 3)) '(0.6 0.1 0.3))

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

Exercise 4

Write a function `(add-first-each L)` that consumes a `(listof Int)` and adds to each value in the list the first in the list.

Your function should be a wrapper around a recursive helper function.

`(add-first-each '(3 2 7 6 5))` ⇒ `'(6 5 10 9 8)`