08: More Lists

Fixed-length lists

In Module 06 we introduced unbounded lists. A wish list, for example, could have any number of items – including none. Unbounded lists usually grow and/or shrink as the program executes.

A **fixed-length list** has a size that is determined by the problem. A fixed-length list does not grow or shrink.

A fixed-length list is similar to a structure: it is often used to group things that belong together.

Tools for fixed-length lists

A number of functions work particularly well with fixed-length lists:

- `(list x1 ... xn)` constructs a list of n items.
- `(first lst)` produces the first item of the non-empty list `lst`.
- `(second lst)` produces the second item (if it exists; error otherwise).
- `(third lst)` produces the third item (if it exists; error otherwise).
- `(fourth lst)` produces the fourth item (if it exists; error otherwise).
- ...
- `(eighth lst)` produces the eighth item (if it exists; error otherwise).

To make use of these functions, change DrRacket's language level to "Beginning Student with List Abbreviations".

List abbreviations

The expression

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

produces the same result as (is equivalent to)

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

Note that `empty` is included explicitly when using `cons` but not when using `list`. It's still present in the list that `list` produces but `empty` is not one of the arguments.

Ex. 1

- You want to add one more element to the list `lst`. Do you use `(cons elem lst)` or `(list elem lst)`? What’s the difference between them?
- Why is `(list 1 2)` legal but `(cons 1 2)` is not?
- What’s the difference between `(cons 1 empty)` and `(list 1 empty)`?

Fixed-length lists in contracts

An unbounded list's type is `(listof X)`.

A fixed-length list's type is `(list T1 ... Tn)` where `T1` to `Tn` are the types of each of the `n` elements in the list.

Examples:

- `(list Str Num)` – could be used for an employee's name and their salary
- `(listof (list Str Num))` – an arbitrarily long list of two-element lists; it could be used to store the names and salaries of all the employees of a company.

As with other types, we can give meaningful names:

- `A SalaryRec is a (list Str Num)`.
- `A Payroll is a (listof SalaryRec)`.
- Alternatively, `A Payroll is a (listof (list Str Num))`.

Example: a payroll

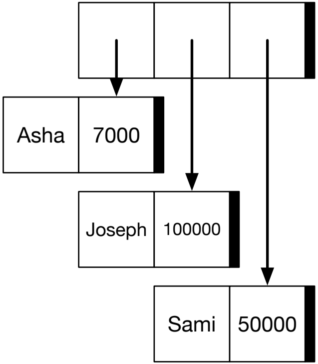
A company's payroll is a list of employee names and their salaries.

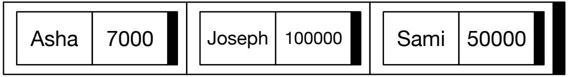
The payroll is an unbounded list (it grows and shrinks with the workforce) of fixed-length lists (the data kept for each employee is always the same – name and salary).

Example payroll:

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

We have two different ways to visualize this **list of lists**.

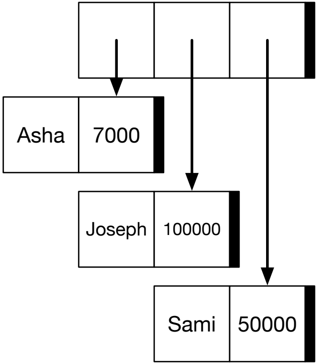




Example: processing a payroll

Write `compute-taxes`, a function which consumes a payroll (a list of employee names and their salaries) and produces a list of each employee name and the tax owed. The tax owed is computed with `tax-payable` from Module 04.

```
(check-expect (compute-taxes
               (cons (list "Asha" 7000)
                     (cons (list "Joseph" 100000)
                           (cons (list "Sami" 50000)
                                 empty))))

              (cons (list "Asha" 700)
                    (cons (list "Joseph" 16500)
                          (cons (list "Sami" 5500)
                                empty))))
```



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

Note the use of `(list Str Num)` for the fixed-length list but `cons` for the payroll itself (which is unbounded).

This data definition is equivalent to `(listof X)` where `X` is `(list Str Num)`. We use the expanded form because it makes template development easier.

> Template

```
;; (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...**

> Template

```
;; (payroll-template pr)
;; payroll-template: Payroll → Any
(define (payroll-template pr)
  (cond [(empty? pr) ...]
        [(cons? pr) (... (first (first pr))
                         (second (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.
;; name: (list Str Num) → Str
(define (name lst) (first lst))
;; (salary lst) produces the second item from lst -- the salary.
;; salary: (list Str Num) → Num
(define (salary lst) (second lst))


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

> Start design recipe; fill in template

```
;; (compute-taxes payroll) calculates the tax owed for each
;; employee/salary pair in the payroll.
;; Examples:
(check-expect (compute-taxes empty) empty)
(check-expect (compute-taxes (cons (list "Asha" 7000) empty))
              (cons (list "Asha" 700) empty))
(check-expect (compute-taxes test-payroll) test-taxes)

;; compute-taxes: Payroll → (listof (list Str Num))
(define (compute-taxes payroll)
  (cond [(empty? payroll) ...]
        [(cons? payroll) (... (name (first payroll))
                              (salary (first payroll))
                              (compute-taxes (rest payroll)))]))
```

> Finish `compute-taxes`

```
;; (compute-taxes payroll) calculates the tax owed for each
;; employee/salary pair in the payroll.
;; Examples:
(check-expect (compute-taxes empty) empty)
(check-expect (compute-taxes (cons (list "Asha" 7000) empty))
              (cons (list "Asha" 700) empty))
(check-expect (compute-taxes test-payroll) test-taxes)

;; compute-taxes: Payroll → TaxRoll
(define (compute-taxes payroll)
  (cond [(empty? payroll) empty]
        [(cons? payroll)
         (cons (list (name (first payroll))
                     (tax-payable (salary (first payroll))))
               (compute-taxes (rest payroll)))]))
```

» Alternate solution

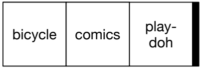
```
(define (compute-taxes-alt payroll)
  (cond [(empty? payroll) empty]
        [(cons? payroll) (cons (sr->tr (first payroll))
                               (compute-taxes-alt (rest payroll)))]))

;; (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 (salary salary-rec))))
```

> Alternate templates leading to the second solution

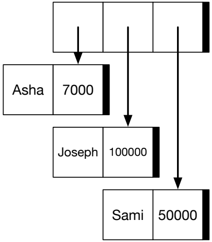
```
;; A SalaryRec is a (list Str Num).
;; A Payroll is one of:
;; * empty
;; * (cons SalaryRec Payroll)

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

;; (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)))]))
```

Different kinds of lists

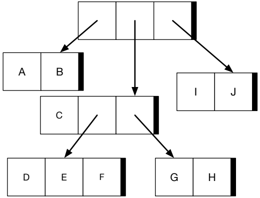
When we introduced lists in module 06, the items they contained were not lists. These were **flat lists**.



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



`cons` vs. `list`

`cons`

- Consumes **exactly two** arguments: an `Any` and a list, which may be `empty`.
- Used to add one more item to the front of a list of arbitrary size; the length is often known only when the program is running.
- Lists constructed with `cons` will explicitly show `empty` at the end of the list.

`list`

- Consumes **any number** of arguments and creates a list exactly that length.
- Used to construct a list that has fixed size; the length is known when we write the program.
- Lists constructed with `list` will not explicitly show `empty` at the end (although it may contain `empty` as an element).

`list` is very useful for creating test data, even for functions that consume an unbounded list. Example: `(check-expect (sort (list 3 1 4 2)) (list 1 2 3 4))`

Except for creating tests, data, and other lists of known length, you should almost always use `cons` instead of `list`.

Ex. 2

What is the length of `gear`?

```
(define gear (list (list "hat" "boots") "coat"
       (list 32.3 (list "mitts")) empty "scarf"))
```

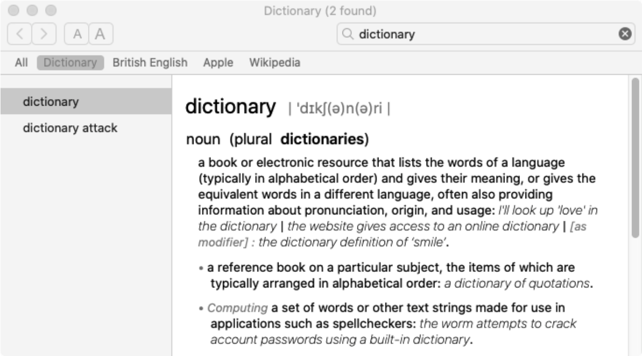
Determine the answer by hand, then use the `length` function to check your answer.

Ex. 3

Write an expression, `e`, using `cons` but not `list` so that `(check-expect? gear e)` passes.

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 or a website:



dictionary | ˈdɪkʃ(ə)n(ə)ri |

noun (plural **dictionaries**)

a book or electronic resource that lists the words of a language (typically in alphabetical order) and gives their meaning, or gives the equivalent words in a different language, often also providing information about pronunciation, origin, and usage: *I'll look up 'love' in the dictionary* | *the website gives access to an online dictionary* | *[as modifier] : the dictionary definition of 'smile'*.

- a reference book on a particular subject, the items of which are typically arranged in alphabetical order: *a dictionary of quotations*.
- *Computing* a set of words or other text strings made for use in applications such as spellcheckers: *the worm attempts to crack account passwords using a built-in dictionary*.

But in all cases there is a correspondence between a word and its definition.

> Example dictionaries

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

Examples:

- A book of word definitions: keys are words; values are definitions.
- Your contacts list: keys are names; values are telephone numbers, twitter handle, email address, etc.
- Course marks: keys are student numbers; values are marks.
- 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.

`Payroll` was a dictionary.

> 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

Association lists

One simple dictionary implementation uses an **association list**, which is just a list of (key, value) pairs.

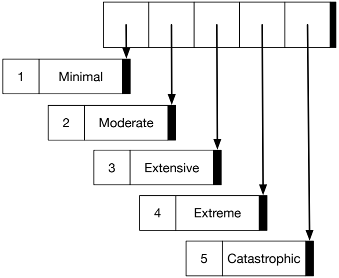
We store each pair as a two-element list.

For simplicity, we will use natural numbers as keys and strings as values.

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

Example:

```
(define hurricane-damage
  (list (list 1 "Minimal")
        (list 2 "Moderate")
        (list 3 "Extensive")
        (list 4 "Extreme")
        (list 5 "Catastrophic")))
```



We can create association lists based on other types for keys and values. We use `Nat` 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 name for the same type is `(listof (list Nat Str))`, still with `;; Requires: each key (Nat) is unique`.

Once we have considered the template, we will write functions implementing the dictionary operations on association lists: `lookup-al`, `add-al`, and `remove-al`.

Constructing the `al-template`

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)))]))
```

A better implementation (except for the lack of documentation):

```
(define (key kv) (first kv))
(define (val kv) (second kv))

(define (al-template alst)
  (cond [(empty? alst) ...]
        [else (... (key (first alst))
                   (val (first alst))
                   (al-template (rest alst)))]))
```

> Lookup operation

Recall that **lookup** consumes a key and a dictionary (association list) and produces the corresponding value when it's found. But what should `lookup-al` produce if it fails?

For now, we'll just produce an empty string – `""`. We'll come up with a better solution soon.

```
(define hurricane-damage
  (list (list 1 "Minimal")
        (list 2 "Moderate")
        (list 3 "Extensive")
        (list 4 "Extreme")
        (list 5 "Catastrophic")))


(check-expect (lookup-al-v1 2 hurricane-damage) "Moderate")
(check-expect (lookup-al-v1 8 hurricane-damage) "")
```

> `lookup` implementation

```
(define (key kv) (first kv))
(define (val kv) (second kv))

(define (al-template alst)
  (cond [(empty? alst) ...]
        [else (... (key (first alst))
                   (val (first alst))
                   (al-template (rest alst)))]))


;; (lookup-al k alst) produces the value corresponding
;;    to key k, or "" if k not present.
;; lookup-al: AL -> Str
(define (lookup-al-v1 k alst)
  (cond [(empty? alst) ""]
        [(= k (key (first alst))) (val (first alst))]
        [else (lookup-al-v1 k (rest alst))]))
```

> `lookup` improvements

But what if we wanted to add a "category 0" hurricane with no damage?

```
(define hurricane-damage
  (list (list 0 "")
        (list 1 "Minimal")
        (list 2 "Moderate")
        (list 3 "Extensive")
        (list 4 "Extreme")
        (list 5 "Catastrophic")))
```

`lookup-al` returns `""` for **both** category 0 and a key that’s not found!

> `lookup` improvements

```
;; (lookup-al-v2 k alst) produces the value corresponding
;;    to key k, or false if k not present.
;; Examples:
(check-expect (lookup-al-v2 2 hurricane-damage) "Moderate")
(check-expect (lookup-al-v2 0 hurricane-damage) "")
(check-expect (lookup-al-v2 8 hurricane-damage) false)

;; lookup-al-v2: AL -> ???
(define (lookup-al-v2 k alst)
  (cond [(empty? alst) false]
        [(= k (key (first alst))) (val (first alst))]
        [else (lookup-al-v2 k (rest alst))]))
```

But what’s the type for the contract?

> `(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]))
```

We can now explain the type `Any` more precisely: it is an abbreviation for `(anyof Nat Int Num Sym Bool Str ...)` where ... is every other type in your program.

Dictionaries: summary

We will leave the `add-al` and `remove-al` 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.

Ex. 4

Write `add-al` to implement the **add** operation. For example:

```
;; (add-al assoc alst) adds assoc to alst.  If alst already contains assoc's




;; Closed-box tests:
(check-expect (add-al (list 8 "Asha") empty) (list (list 8 "Asha")))

(check-expect         ; alst does not contain this key.
 (add-al (list 7 "Bo")
         (list (list 8 "Asha") (list 2 "Joseph") (list 5 "Sami")))
```

Ex. 5

Write `remove-al` to implement the **remove** operation. For example:

```
;; Closed-box tests:
```

Ex. 6

Think about maintaining the association list in sorted order.

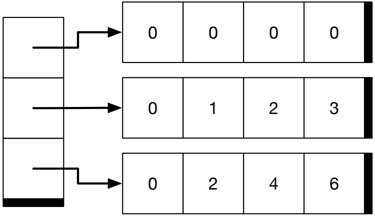
- How would you modify `lookup-al` to avoid searching the *whole* list most of the time?
- How would you modify `insert-al`? Does it do more “work” or less “work” than inserting into an unordered list?

Two-dimensional data

Another use of lists of (fixed-length) 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 c^{th} entry of the r^{th} row (numbering from 0) is $r \times c$.

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

> Make one row

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.

```
;; (make-a-row c r nc) produces entries c...(nc-1) of rth row of mult. table
;; Examples:
(check-expect (make-a-row 0 3 5) (list 0 3 6 9 12))
(check-expect (make-a-row 0 4 5) (list 0 4 8 12 16))

;; make-a-row: Nat Nat Nat → (listof Nat)
(define (make-a-row c r nc)
  (cond [(>= c nc) empty]
        [else (cons (* r c) (make-a-row (add1 c) r nc))]))
```

> Put multiple rows together

```
;; (mult-table nr nc) produces multiplication table
;;     with nr rows and nc columns
;; Example:
(check-expect (mult-table 3 4)
              (list (list 0 0 0 0)
                    (list 0 1 2 3)
                    (list 0 2 4 6)))

;; mult-table: Nat Nat → (listof (listof Nat))
(define (mult-table nr nc)
  (generate-rows 0 nr nc))

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

We will distinguish four different cases, and look at them in order of complexity.

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

> Case 1: processing just one list

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

```
;; (my-append lst1 lst2) adds each element of lst1 to the
;;    beginning of lst2, preserving order
;; Examples:
(check-expect (my-append empty (list 1 2))
              (list 1 2))
(check-expect (my-append (list 3 4) (list 1 2 5))
              (list 3 4 1 2 5))

;; my-append: (listof Any) (listof Any) → (listof Any)
(define (my-append lst1 lst2)
```

» `my-append`

```
(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)`.

`cons` vs. `list` vs. `append`

`cons`, `list`, and `append` are all different and have different uses, but are often confused.

- `(cons v lst)` is used to make `lst` one element longer by adding the value `v` at the beginning of the list.
  The length of the result will be `(add1 (length lst))`.
- `(list v_1 v_2 ... v_n)` is used to construct a list with the *n* values given.
  The length of the result will be `n`.
- `(append lst1 lst2)` appends `lst2` to the end of `lst1`.
  The length of the result will be `(+ (length lst1) (length lst2))`.

In each case, the values and lists involved might be the result of evaluating an expression.

Ex. 7

Write a function `expand-each` that consumes two lists. For each item in the first list, make a list that contains that item, followed by all the items in the second list.

```
(check-expect (expand-each (list 12 13 'x)
                           (list 42 "zorkmids" 'Q))
              (list (list 12 42 "zorkmids" 'Q)
                    (list 13 42 "zorkmids" 'Q)
                    (list 'x 42 "zorkmids" 'Q)))
```

Remember: the second list is “along for the ride”; it does not change.

> Case 2: processing in lockstep; equal length lists

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.

» Lockstep template

```
;; lockstep-template: (listof X) (listof Y) → Any
;; Requires:  (length lst1) = (length lst2)
(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
;; Examples:
(check-expect (dot-product empty empty) 0)
(check-expect (dot-product (list 2) (list 3)) 6)
(check-expect (dot-product (list 2 3 4 5) (list 6 7 8 9))
              (+ 12 21 32 45))

;; dot-product: (listof Num) (listof Num) → Num
;; Requires: lon1 and lon2 are the same length
(define (dot-product lon1 lon2)
```

» dot-product

```
;; (dot-product lon1 lon2) computes the dot product
;;       of vectors lon1 and lon2
;; Examples:
(check-expect (dot-product empty empty) 0)
(check-expect (dot-product (list 2) (list 3)) 6)
(check-expect (dot-product (list 2 3 4 5) (list 6 7 8 9))
              (+ 12 21 32 45))

;; dot-product: (listof Num) (listof Num) → Num
;; Requires: lon1 and lon2 are the same length
(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
```

Ex. 8

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

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

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

Ex. 9

Complete `join-names`.

```
(define gnames (list "Joseph" "Burt" "Douglas" "James" "David"))
(define snames (list "Hagey" "Matthews" "Wright" "Downey" "Johnston"))

;; (join-names g s) Make a list of full names from g (given names) and
;;                  s (surnames).


;; Closed-box tests:
(check-expect (join-names gnames snames)
              (list "Joseph Hagey" "Burt Matthews" "Douglas Wright"
                    "James Downey" "David Johnston"))
```

Case 3: lockstep but (possibly) different lengths

`(nlist=? lon1 lon2)` produces true if the number in each position of `lon1` is equal to the number in the corresponding position in `lon2`.

```
;; (nlist=? lon1 lon2) determines if lon1 and lon2 are equal
;; Examples:
(check-expect (nlist=? empty empty) true)
(check-expect (nlist=? (list 1 2 3) (list 1 2 3)) true)
(check-expect (nlist=? (list 1 2 3) (list 1 2 4)) false)
(check-expect (nlist=? (list 1 2 3) (list 1 2 3 4)) false)

;; nlist=?: (listof Num) (listof Num) → Bool
(define (nlist=? lon1 lon2)
  (cond
    [(and (empty? lon1) (empty? lon2)) ...]
    [(and (empty? lon1) (cons? lon2)) ...]
    [(and (cons? lon1) (empty? lon2)) ...]
    [(and (cons? lon1) (cons? lon2)) ...]))
```

> Reasoning about list equality

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

```
;; (nlist=? lon1 lon2) determines if lon1 and lon2 are equal
;; nlist=?: (listof Num) (listof Num) → Bool
(define (nlist=? lon1 lon2)
  (cond
    [(and (empty? lon1) (empty? lon2)) true]
    [(and (empty? lon1) (cons? lon2)) false]
    [(and (cons? lon1) (empty? lon2)) false]
    [(and (cons? lon1) (cons? lon2)) ...]))
```

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

The natural recursion in this case is

```
(... (first lon1) (first lon2) (nlist=? (rest lon1) (rest lon2)))
```

List equality code

```
;; (nnlist=? lon1 lon2) determines if lon1 and lon2 are equal
;; Examples:
(check-expect (nlist=? (list 1 3 5) (list 1 3)) false)
(check-expect (nlist=? (list 1 3 5) (list 1 4 5)) false)
(check-expect (nlist=? (list 1 3) (list 1 3 5)) false)
(check-expect (nlist=? (list 1 3 5) (list 1 3 5)) true)

;; nlist=?: (listof Num) (listof Num) → Boolean
(define (nlist=? lon1 lon2)
  (cond
    [(and (empty? lon1) (empty? lon2)) true]
    [(and (empty? lon1) (cons? lon2)) false]
    [(and (cons? lon1) (empty? lon2)) false]
    [(and (cons? lon1) (cons? lon2))
     (and (= (first lon1) (first lon2))
          (nlist=? (rest lon1) (rest lon2)))]))
```

Ex. 10

The code for `nlist=?` can be transformed in various ways. Each problem stands alone, starting with the code on the previous slide. Whether the result is "better" or not depends on the metrics used.

Modify the implementation of `nlist=?` in the following ways:

1. Combine the second and third question/answer pairs.
2. Combine the first and second question/answer pairs; simplify the third.
3. Use **`else`**.
4. Combine 1 and 3.
5. Combine 2 and 3.
6. Get rid of the **`cond`** completely.

Ex. 11

Our "basic types" so far are `Num`, `Str`, `Bool`, and `Sym`. Let's give these a name:

```
;; an Atom is (anyof Num Str Bool Sym)
```

- Write a (non-recursive) function `atom=?` that determines if two `Atom` are equal.
- Expand your `nlist=?` function so it works on two `(listof Atom)`.

You may use `boolean=?` for this question, but in general, avoid it. We're **not** adding it to our toolbox.

Ex. 12

If you want a significantly greater challenge:

```
;; a PrettyMuchAny is a (anyof Atom (listof PrettyMuchAny))
```

Expand your `nlist=?` function so it works on `(listof PrettyMuchAny)`.

> Built-in list equality

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

```
(equal? (list 1 (list 2 3)) (list 1 (list 2 3))) ⇒ true
(equal? 'a 'b) ⇒ false    ;; Bad style!  Use symbol=?
(equal? (list "one" 'two 3) (list 1 2 3)) ⇒ false
(equal? (make-point 3 4) (make-point 3 4)) ⇒ true
(equal? (make-point 3 4) (make-circle 3 4)) ⇒ false
```

How would you write `equal?` if it were not already built in?

!

WARNING: Do not over-use `equal?`.
If there is a type-specific predicate that works, use it.

> Case 4: processing at different rates

If the two lists being consumed are of different lengths, all four possibilities for their being empty/nonempty are possible and need to be checked in the template:

```
(define (twolist-template lon1 lon2)
  (cond [(and (empty? lon1) (empty? lon2)) ...]
        [(and (empty? lon1) (cons? lon2)) ...]
        [(and (cons? lon1) (empty? lon2)) ...]
        [(and (cons? lon1) (cons? lon2)) ...]))
```

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

» Refining the template

```
(define (twolist-template lon1 lon2)
  (cond
    [(and (empty? lon1) (empty? lon2)) ...]
    [(and (empty? lon1) (cons? lon2)) (... (first lon2) (rest lon2))]
    [(and (cons? lon1) (empty? lon2)) (... (first lon1) (rest lon1))]
    [(and (cons? lon1) (cons? lon2))  ??? ]))
```

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

The fourth possibility definitely requires recursion, but its form is unclear.

Further refinements

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

```
(... (first lon1)
     (twolist-template (rest lon1) lon2))

(... (first lon2)
     (twolist-template lon1 (rest lon2)))

(... (first lon1) (first lon2)
     (twolist-template (rest lon1) (rest lon2)))
```

Which of these is appropriate depends on the specific problem we're trying to solve and will require further reasoning.

» Example: merging two sorted lists

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

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

`merge` will produce one list containing all elements, also in ascending order.

As an example:

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

The effect of `(merge lst1 lst2)` is the same as `(sort (append lst1 lst2))` but will take advantage of `lst1` and `lst2` already being sorted.

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

» Example: merging two sorted lists

```
;; Base cases:
(check-expect (merge empty empty) empty)
(check-expect (merge empty (list 2 6 9)) (list 2 6 9))
(check-expect (merge (list 1 3) empty) (list 1 3))

;; Recursive cases:
(check-expect (merge (list 1 4) (list 2)) (list 1 2 4))
(check-expect (merge (list 3 4) (list 2)) (list 2 3 4))
```

Ex. 13

Before you proceed, try to write your own `merge` function.

» 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)`.

» Merge code

```
;; merge: (listof Num) (listof Num) → (listof Num)
;; Requires: lon1 and lon2 are already in ascending order.
(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)))))
```

Mergesort

The `merge` algorithm is the core of `mergesort`, a sorting algorithm invented by John von Neumann in 1945. `mergesort` is more complicated than insertion sort but on longer lists it is **much** faster.

```
;; (mergesort lon) puts lon in increasing order
;; mergesort: (listof Num) → (listof Num)
(define (mergesort lon)
  (cond [(empty? lon) empty]
        [(empty? (rest lon)) lon]
        [else (merge (mergesort (one-half lon))
                     (mergesort (other-half lon)))]))
```

`one-half` and `other-half` each produce half of the provided list. Perhaps one produces the first half and the other the last half, or one produces the items at even-numbered positions and the other produces items at odd-numbered positions, or ....

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.

Write 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`.

> Examples for `at-least?`

```
;; (at-least? n elem lst) determines if elem appears
;;       at least n times in lst.
;; Examples:
(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)

;; at-least?: Nat Any (listof Any) → Bool
(define (at-least? n elem lst)
```

> Developing the code

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

```
(define (at-least? 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?

> Improving at-least?

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 code, and eventually to the code that appears on the next slide.

> Improved at-least?

```
(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 be able to work with fixed-length lists, including lists of fixed-length lists such as dictionaries.
- You should know the differences between `cons`, `list`, and `append`, and know the circumstances where each is appropriate.
- You should be able to construct and work with lists that contain lists.
- You should understand the four 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.

Summary: built-in functions

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

`append` `eighth` `equal?` `fifth` `fourth` `list` `second` `seventh` `sixth` `third`

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` `add1` **`and`** `append` `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` `eighth` **`else`** `empty?` `equal?` `error` `even?` `exp` `expt` `fifth` `first` `floor` `fourth` `integer?` `length` `list` `list->string` `list?` `log` `max` `min` `modulo` `negative?` `not` `number->string` `number?` `odd?` **`or`** `pi` `positive?` `quotient` `remainder` `rest` `round` `second` `seventh` `sgn` `sin` `sixth` `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?` `sub1` `substring` `symbol=?` `symbol?` `tan` `third` `zero?`