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 a helper function to do some of the work. The helper function may or may not be recursive itself.

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

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

---

> **Filling in the list template**

```scheme
;; (sort lon) sorts the elements of lon in non-decreasing order
(check-expect (sort (cons 2 (cons 0 (cons 1 empty)))) ...) 

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

;; sort: (listof Num) → (listof Num)
(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 into the sorted list.

> A condensed trace of sort and insert

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

> The helper function insert

We again use the list template for insert.

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

;; insert: Num (listof Num) → (listof Num)
;; requires: slon is sorted in non-decreasing order
(define (insert n slon)
  (cond [(empty? slon) ...]
        [else (... (first slon) ...
                 (insert n (rest slon)) ...)]))
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 \( (\text{rest } slon) \).

\[
\text{insert} \quad (\text{define} \quad (\text{insert} \quad n \quad \text{slon})
\text{cond}
[\text{(empty? } \text{slon}) \quad (\text{cons} \quad n \quad \text{empty})]
[\text{(<=} \quad n \quad (\text{first} \quad \text{slon})) \quad (\text{cons} \quad n \quad \text{slon})]
[\text{else} \quad (\text{cons} \quad (\text{first} \quad \text{slon}) \quad (\text{insert} \quad n \quad (\text{rest} \quad \text{slon})))])
\]

\[
\text{(insert} \quad 4 \quad (\text{cons} \quad 1 \quad (\text{cons} \quad 2 \quad (\text{cons} \quad 5 \quad \text{empty}))))
\Rightarrow \quad (\text{cons} \quad 1 \quad (\text{insert} \quad 4 \quad (\text{cons} \quad 2 \quad (\text{cons} \quad 5 \quad \text{empty}))))
\Rightarrow \quad (\text{cons} \quad 1 \quad (\text{cons} \quad 2 \quad (\text{insert} \quad 4 \quad (\text{cons} \quad 5 \quad \text{empty}))))
\Rightarrow \quad (\text{cons} \quad 1 \quad (\text{cons} \quad 2 \quad (\text{cons} \quad 4 \quad (\text{cons} \quad 5 \quad \text{empty}))))
\]

Our \text{sort} with helper function \text{insert} are together known as \text{insertion sort}.

Exercise 1
Modify insertion sort to order the list from largest to smallest instead of smallest to largest.

1 \( \text{(define} \quad (sort} \quad \text{lon})
2 \quad \text{(cond}
3 \quad [\text{(empty? } \text{lon}) \quad \text{empty}]
4 \quad \text{[else} \quad (\text{insert} \quad (\text{first} \quad \text{lon})]
5 \quad (\text{sort} \quad (\text{rest} \quad \text{lon}))))])
6 \quad \text{)}
7 \( \text{(define} \quad (\text{insert} \quad n \quad \text{slon})
8 \quad \text{(cond}
9 \quad [\text{(empty? } \text{slon}) \quad (\text{cons} \quad n \quad \text{empty})]
10 \quad [\text{(<=} \quad n \quad (\text{first} \quad \text{slon})) \quad (\text{cons} \quad n \quad \text{slon})]
11 \quad \text{[else} \quad (\text{cons} \quad (\text{first} \quad \text{slon})]
12 \quad (\text{insert} \quad n \quad (\text{rest} \quad \text{slon}))))])}
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 02 when changing language levels.

> List abbreviations

The expression

\[
(\text{cons } \text{exp1} \ (\text{cons } \text{exp2} \ (... \ (\text{cons } \text{expn} \ \text{empty})...)))
\]

can be abbreviated as

\[
(\text{list } \text{exp1} \ \text{exp2} \ ... \ \text{expn})
\]

The result of \(\text{sort} \ (\text{cons} \ 4 \ (\text{cons} \ 2 \ (\text{cons} \ 1 \ (\text{cons} \ 5 \ \text{empty}))))\) can be expressed as \(\text{list} \ 1 \ 2 \ 4 \ 5\). The application itself can be expressed as \(\text{sort} \ (\text{list} \ 4 \ 2 \ 1 \ 5)\).

> cons VS. list

Note that \text{cons} and \text{list} have different results and different purposes.

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

We use \text{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 \text{cons} is evaluated during the running of the program).
Other list abbreviations

(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.

Exercise 2

- 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)`?

Lists containing lists

Here are two different two-element lists.

```
1 2
3 4
```

We now know two different ways to construct these lists:

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

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

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

\[
\begin{array}{c}
3 \\
4
\end{array}
\]

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

\[
(\text{cons} \ (\text{cons} \ 3 \ \text{empty}) \ \text{empty}) \quad \text{OR} \quad (\text{list} \ (\text{list} \ 3 \ 4))
\]

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.

> Lists containing lists

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.

\[
\begin{array}{c}
1 \\
2 \\
3 \\
4
\end{array}
\]

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

\[
(\text{cons} \ (\text{cons} \ 1 \ \text{empty}) \ \text{empty}) \quad \text{OR} \quad (\text{list} \ \text{list} \ 1 \ 2)
\]

Clearly, the abbreviations are more expressive.

> Example: processing a payroll

A company needs to process its 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:
\[
(\text{list} \ (\text{list} \ "\text{Asha}" \ 50000) \\
 (\text{list} \ "\text{Joseph}" \ 100000) \\
 (\text{list} \ "\text{Sami}" \ 7000))
\]

TaxRoll:
\[
(\text{list} \ (\text{list} \ "\text{Asha}" \ 7750) \\
 (\text{list} \ "\text{Joseph}" \ 18250) \\
 (\text{list} \ "\text{Sami}" \ 1050))
\]

Asha 50000
Joseph 100000
Sami 10000

Asha 50000
Joseph 100000
Sami 10000
» Data definitions

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

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

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

» Template

;; (payroll-template pr)
;; payroll-template: Payroll → Any
(define (payroll-template pr)
  (cond [(empty? ... List abbrev Lists of lists Dictionaries 2D data Processing two lists List & number
18/69 08: More Lists

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

» Template

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.
(define (payroll-template pr)
  (cond [(empty? pr) ...]
        [(cons? pr) (... (first (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))

;; (amount lst) produces the second item from lst -- the amount.
;; amount: (list Str Num) → Num
(define (amount lst) (first (rest lst)))

(define (payroll-template pr)
  (cond [(empty? pr) ...]
        [(cons? pr) ... (name (first pr)) ...
            ...
            ...
            ... (payroll-template (rest pr)) ...]
        )])

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

(define (compute-taxes payroll)
  (cond [(empty? payroll) ...]
        [(cons? payroll) ... (name (first payroll)) ...
            ...
            ...
            ... (compute-taxes (rest payroll)) ...]
        )])
;; (compute-taxes payroll) calculates the tax owed for each employee/salary pair in the payroll.
(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 (amount (first payroll))))
                                 (compute-taxes (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. Nowadays, a dictionary is an app:

- Examples dictionaries

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.
- Course marks: keys are userids; 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.
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 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 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:

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

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 alternative is to use `(listof (list Nat Str))`. 
> 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 produce if it fails?

When the key is not found in the association list, we can not produce a string. Every string, even "", is a valid value and might be the result of a successful lookup. The “not found” condition needs to be distinguishable, from successful searches. We’ll use false to indicate failure.

(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 implementation

(define (key kv) (first kv))
(define (val kv) (second kv))
;; (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 (key (first alst))) (val (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.

Exercise 3

- Implement add.
- Implement remove.
- How could you improve the performance of an association list if the dictionary is large (using techniques from this lecture module)? Is there a way to avoid search the whole list most of the time?
Two-dimensional data

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

For example, here is a multiplication table:

\[
(\text{mult-table } 3 4) \Rightarrow \\
(\text{list } (\text{list } 0 0 0 0) \\
(\text{list } 0 1 2 3) \\
(\text{list } 0 2 4 6))
\]

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.

\[
;; (\text{cols-to } c \ r \ nc) \text{ produces entries } c...(nc-1) \text{ of } r\text{th row of mult. table} \\
;; \text{ Example:} \\
(\text{check-expect } (\text{cols-to } 0 3 5) (\text{list } 0 3 6 9 12)) \\
(\text{check-expect } (\text{cols-to } 0 4 5) (\text{list } 0 4 8 12 16))
\]

\[
;; \text{cols-to: } \text{Nat Nat Nat} \to (\text{listof Nat}) \\
(\text{define } (\text{cols-to } c \ r \ nc) \\
 (\text{cond} [ (\geq c \ nc) \text{ empty}] \\
 [\text{else} (\text{cons } (* r c) (\text{cols-to } (\text{add1 } c) \ r \ nc))])
\]

> Put multiple rows together

\[
;; (\text{mult-table } nr \ nc) \text{ produces multiplication table} \\
;; \quad \text{with } nr \text{ rows and } nc \text{ columns} \\
;; \text{ Example:} \\
(\text{define } (\text{mult-table } nr \ nc) \\
 (\text{rows-to } 0 \ nr \ nc))
\]

\[
;; (\text{rows-to } r \ nr \ nc) \text{ produces mult. table, rows } r...(nr-1) \\
;; \text{rows-to: } \text{Nat Nat Nat} \to (\text{listof } \text{listof Nat}) \\
(\text{define } (\text{rows-to } r \ nr \ nc) \\
 (\text{cond} [ (\geq r \ nr) \text{ empty}] \\
 [\text{else} (\text{cons } (\text{cols-to } 0 \ r \ nc) (\text{rows-to } (\text{add1 } r) \ nr \ nc))])
\]
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.

Case 1: processing just one list

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

```
;; (my-append lst1 lst2) appends lst2 to the end of lst1
;; 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))

;; my-append: (listof Any) (listof Any) → (listof Any)
(define (my-append 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.

» 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)) ... )]))
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 \((\text{list } 1 \ 2 \ 3)\).

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

```scheme
;; (dot-product lon1 lon2) computes the dot product
;; of vectors lon1 and lon2
(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 lon1 lon2) computes the dot product
;; of vectors lon1 and lon2
(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)))]))
Exercise 4

Write a recursive function \texttt{vector-add} that adds two vectors.

\begin{verbatim}
(vector-add '(3 5) '(7 11)) ⇒ '(10 16)
(vector-add '(3 5 1 3) '(2 2 9 3)) ⇒ '(5 7 10 6)
\end{verbatim}

Exercise 5

Complete \texttt{join-names}.

\begin{verbatim}
(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 (given names) and s (surnames).
;; Example:
(check-expect (join-names gnames snames)
'("Joseph Hagey" "Burt Matthews" "Douglas Wright" "James Downey" "David Johnston"))
\end{verbatim}
Case 3: processing at different rates

If the two lists *lon1, lon2* being consumed are of different lengths, all four possibilities for their being empty/nonempty are possible:

\[
\begin{align*}
&\text{(and (empty? lon1) (empty? lon2))} \\
&\text{(and (empty? lon1) (cons? lon2))} \\
&\text{(and (cons? lon1) (empty? lon2))} \\
&\text{(and (cons? lon1) (cons? lon2))}
\end{align*}
\]

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

The template so far:

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

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

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

```
;; 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))
```
If \( \text{lon1} \) and \( \text{lon2} \) are both nonempty, what is the first element of the merged list?

It is the smaller of \((\text{first lon1})\) and \((\text{first lon2})\).

If \((\text{first lon1})\) is smaller, then the rest of the answer is the result of merging \((\text{rest lon1})\) and \(\text{lon2}\).

If \((\text{first lon2})\) is smaller, then the rest of the answer is the result of merging \(\text{lon1}\) and \((\text{rest lon2})\).

### Merge code

```scheme
(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

```scheme
(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)) 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)))]))

Reasoning about list equality

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.
Exercise 6

Like `merge`, the code for `list=?` can be transformed in various ways. Each problem stands alone, starting with the code on the previous slide. Like `merge`, whether the result is “better” or not depends on the metrics used.

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.

> 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 repertoire 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.

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.
(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-template? n elem lst)

> Developing the code

The recursion involves the parameters \( n \) and \( lst \), once again giving four
possibilities:

(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 template, 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, \( n \geq 1 \)
       [(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? 2 'green (list 'green '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.