More lists

We're now returning to working with lists – but upping our game to include more complex situations.

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 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 non-decreasing order
(check-expect (sort (cons 2 (cons 8 (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 \( (\text{rest} \ slon) \).

\[
> \text{Insert 8/58}
\]

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

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

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

Modify insertion sort to order the list from largest to smallest instead of smallest to largest.
Now that we’ve had some experience writing recursive functions on lists, we will return to expressing lists in list notation, instead of constructor notation. But keep the constructor notation in mind: \((\text{list } 2 3 5)\) is “really” a shorter way of writing \((\text{cons } 2 \ (\text{cons } 3 \ (\text{cons } 5 \ \text{empty})))\).

In DrRacket, switch to the language level “Beginning Student With List Abbreviations”. This provides the syntax for list abbreviations, and a number of additional convenience functions.

The expression
\((\text{cons } \text{exp1} \ (\text{cons } \text{exp2} \ (\ldots \ (\text{cons } \text{expn} \ \text{empty}) \ldots)))\)
can be abbreviated as
\((\text{list } \text{exp1} \ \text{exp2} \ \ldots \ \text{expn})\)

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 \((\text{sort } \ (\text{list } 4 2 1 5)).\)

\((\text{second my-list})\) is an abbreviation for \((\text{first } \ (\text{rest my-list})).\)
\(\text{third, fourth,}\) and so on up to \(\text{eighth}\) are also defined.
Use these \textbf{sparingly} to improve readability.

The templates we have developed remain very useful.
You want to add one more element to the list \texttt{lst}. Do you use \texttt{(cons elem lst)} or \texttt{(list elem lst)}? What's the difference between them?

Why is \texttt{(list 1 2)} legal but \texttt{(cons 1 2)} is not?

What's the difference between \texttt{(cons 1 empty)} and \texttt{(list 1 empty)}?

What is the length of:
\texttt{(list (list "hat" "boots") "coat"
(list 32.3 (list "mitts")) empty "scarf")}

Determine the answer by hand, then use the \texttt{length} function to check your answer.

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:

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

```scheme
;; 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"))
```

Using the `listof` data definition, we could equally say:

```scheme
;; An AL is a (listof (list Nat Str))
;;     Requires: each key (Nat) is unique
```

---
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)))])))

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?

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

(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 is not present.
;; lookup-al: Num AL -> (anyof Str false)
(define (lookup-al k alst)
  (cond [(empty? alst) false]
        [(= k (key (first alst))) (val (first alst))]
        [else (lookup-al k (rest alst))])))

Notice the anyof in the contract. This means it can produce either a Str (when k is present in alst), or false (when it’s not).
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.

**Exercise**

Write `add-al` to implement the `add` operation.

Write `remove-al` to implement the `remove` operation.

Consider: can you improve the performance of an association list if the dictionary is large (using techniques from this lecture module)? Can you think of a way to avoid search the whole list most of the time?

---

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 $c^{th}$ 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 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.

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

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

Put multiple rows together

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

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

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`.

```racket
;; (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))
```

```racket
;; 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.

```racket
(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)`. 
Exercise

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.

```scheme
(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

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

```scheme
;; 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 \text{empty}.

```
;; (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)))]))
```
A condensed trace

\[
\begin{align*}
\text{(dot-product (list 2 3 4) (list 5 6 7))} & \Rightarrow (+ 10 \text{(dot-product (list 3 4) (list 6 7))}) \\
& \Rightarrow (+ 10 (+ 18 \text{(dot-product (list 4) (list 7))}) \\
& \Rightarrow (+ 10 (+ 18 (+ 28 \text{(dot-product (list) (list))})}) \\
& \Rightarrow (+ 10 (+ 18 (+ 28 0})) \\
& \Rightarrow (+ 10 (+ 18 28)) \\
& \Rightarrow (+ 10 46) \\
& \Rightarrow 56
\end{align*}
\]

Exercise

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

\[
\begin{align*}
\text{(vector-add (list 3 5) (list 7 11))} & \Rightarrow (\text{(list 10 16)}) \\
\text{(vector-add (list 3 5 1 3) (list 2 2 9 3))} & \Rightarrow (\text{(list 5 7 10 6)})
\end{align*}
\]

Exercise

Complete join-names.

\[
\begin{align*}
\text{(define gnames (list "Joseph" "Burt" "Douglas" "James" "David"))} \\
\text{(define snames (list "Hagey" "Matthews" "Wright" "Downey" "Johnston"))}
\end{align*}
\]

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

;; Example:

\[
\begin{align*}
\text{(check-expect (join-names gnames snames)} \\
& (\text{list "Joseph Hagey" "Burt Matthews" "Douglas Wright" "James Downey" "David Johnston"})
\end{align*}
\]

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

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

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

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

```scheme
(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.

```scheme
(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))

(check-expect (merge (list 1 4) (list 2)) (list 1 2 4))
(check-expect (merge (list 3 4) (list 2)) (list 2 3 4))
```

Before you proceed, try to write your own `merge` function.
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: (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))))])])

(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)))))
Write a function `list=?`, that consumes two `(listof Num)`. If the two lists are the same, it produces `true`; otherwise, it produces `false`.

(Don't use `equal?`; the point is to see how to do it using recursion!)

### Exercise

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

- an Atom is `(anyof Num Str Sym)`

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

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

If you want a significantly greater challenge:

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

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

### 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?`

```scheme
;; (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? 2 'red (list 'red 'blue 'red 'green)) false)
;; (check-expect (at-least? 3 'red (list 'red 'blue 'red 'green)) true)
;; (check-expect (at-least? 1 7 (list 5 4 0 5 3)) false)
```

```scheme
;; at-least?: Nat Any (listof Any) -> Bool
(define (at-least? n elem lst)
)```
Developing the code

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

\[
\text{Cond} \begin{cases} 
& (\text{and (zero? } n) (\text{empty? lst}) \ldots) \\
& ((\text{and (zero? } n) (\text{cons? lst}) \ldots) \\
& ((\text{and (> } n \ 0) (\text{empty? lst}) \ldots) \\
& ((\text{and (> } n \ 0) (\text{cons? lst}) \ldots))
\end{cases}
\]

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 \( \text{lst} \) is \text{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?

\[
\text{Define (at-least? } n \ \text{elem lst)}
\text{(cond } [(\text{zero? } n) \text{ true}]
\text{[(empty? lst) false]}
\text{; list is nonempty, } n \geq 1
\text{[(equal? (first lst) elem) (at-least? (sub1 } n \ \text{elem (rest lst))]} \\
\text{[else (at-least? } n \ \text{elem (rest lst))])}
\]

Evaluating a Polynomial

We will represent a polynomial as a list of coefficients. For example, \((\text{list } 5 4 2)\) represents the polynomial \(5 + 4x + 2x^2\).

Notice that we can write this as \(5x^0 + 4x^1 + 2x^2\).

We will write:

```
;; a Polynomial is a (listof Num)
;; Requires: the last value is not zero.
```

Write a function \(\text{evaluate-polynomial-from poly x i}\). It consumes a Polynomial, a Num, and a Nat representing the exponent. The function produces the value of the polynomial at the given value of \(x\).

For example,

\(\text{evaluate-polynomial-from (list 5 4 2) 3 0} \Rightarrow 35\)

Since \(5 + 4 \cdot 3 + 2 \cdot 3^2 = 35\).

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.
In this module we added the following to our toolbox:

`length list`

These are the functions and special forms currently in our toolbox: