Working with recursion

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

We can extend the idea of a self-referential definition to defining the natural numbers, which leads to the use of recursion in order to write functions that consume numbers.

We also look at function which do recursion on more than one item (lists and/or numbers) simultaneously.

From definition to template

We'll review how we derived the list template.

;; A List is one of:

;; ⋆ empty

;; ⋆ (cons Any List)

Suppose we have a list lst.

The test (empty? lst) tells us which case applies.

If (empty? lst) is false, then lst is of the form (cons f r).

How do we compute the values f and r?

f is (first lst).

r is (rest lst).

Because r is a list, we recursively apply the function we are constructing to it.

```
;; my-list-fn: (listof Any) → Any
(define (my-list-fn lst)
  (cond [(empty? lst) ...]
        [else (... (first lst) ...
                   (my-list-fn (rest lst)) ...)]))
```

We can repeat this reasoning on a recursive definition of natural numbers to obtain a template.

Natural numbers

```
;; A Nat is one of:
;; ⋆ 0
;; ⋆ (add1 Nat )
```

Here add1 is the built-in function that adds 1 to its argument.

The natural numbers start at 0 in computer science and some branches of mathematics (e.g. logic).

We'll now work out a template for functions that consume a natural number.

Suppose we have a natural number n.

The test (zero? n) tells us which case applies.

If (zero? n) is false, then n has the value (add1 k) for some k.

To compute k, we subtract 1 from n, using the built-in sub1 function.

Because the result (sub1 n) is a natural number, we recursively apply the function we are constructing to it.

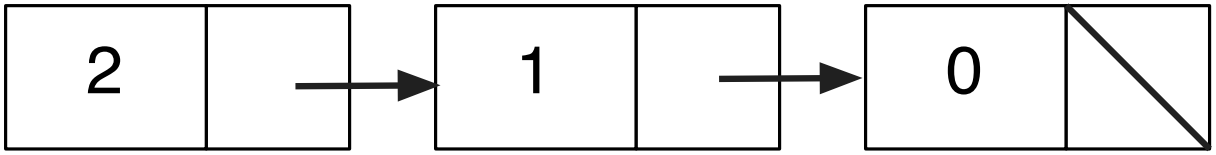
```
(define (my-nat-fn n)
  (cond [(zero? n) ...]
        [else (... (my-nat-fn (sub1 n)) ...)]))
```

Example: a decreasing list

Goal: countdown, which consumes a natural number n and produces a decreasing list of all natural numbers less than or equal to n.

(countdown 0) $\Rightarrow$ (cons 0 empty)

(countdown 2) $\Rightarrow$ (cons 2 (cons 1 (cons 0 empty)))



We start filling in the template:

```
(define (countdown n)
  (cond [(zero? n) ...]
        [else (... (countdown (sub1 n)) ...)]))
```

If n is 0, we produce the list containing 0, and if n is nonzero, we cons n onto the countdown list for $n - 1$.

```
;; (countdown n) produces a list of Nats from n...0
;; countdown: Nat → (listof Nat)
;; Example:
(check-expect (countdown 2) (cons 2 (cons 1 (cons 0 empty))))

(define (countdown n)
  (cond [(zero? n) (cons 0 empty)]
        [else (cons n (countdown (sub1 n)))]))
```

A condensed trace

(countdown 2)

⇒ (cons 2 (countdown 1))

⇒ (cons 2 (cons 1 (countdown 0)))

⇒ (cons 2 (cons 1 (cons 0 empty)))

If the function countdown is applied to a negative integer, it will not terminate. Even though we make no guarantees when the contract is violated, we should program “defensively” where possible.

We could check for this and signal an error, or we could change the test to catch this (and some other situations).

```
(define (countdown n)
  (cond [(<= n 0) (cons 0 empty)]
        [else (cons n (countdown (sub1 n)))]))
```

Some useful notation

The symbol $\mathbb{Z}$ is often used to denote the integers.

We can add subscripts to define subsets of the integers.

For example, $\mathbb{Z}_{\geq 0}$ defines the non-negative integers, also known as the natural numbers.

Other examples: $\mathbb{Z}_{>4}$, $\mathbb{Z}_{<-8}$, $\mathbb{Z}_{\leq 1}$.

Subintervals of the natural numbers

If we change the base case test from (zero? n) to (= n 7), we can stop the countdown at 7.

This corresponds to the following definition:

;; An integer in $\mathbb{Z}_{\geq 7}$ is one of:

;; $\star$ 7

;; $\star$ (add1 $\mathbb{Z}_{\geq 7}$)

(In practice, we add a require section to our contract.)

```
;; (countdown-to-7 n) produces a decreasing list from n...7
;; countdown-to-7: Nat → (listof Nat)
;; requires: n >= 7
;; Example:
(check-expect (countdown-to-7 9) (cons 9 (cons 8 (cons 7 empty))))

(define (countdown-to-7 n)
  (cond [(= n 7) (cons 7 empty)]
        [else (cons n (countdown-to-7 (sub1 n)))]))
```

Again, making the base case be (<= n 7) is more robust.

We can generalize both countdown and countdown-to-7 by providing the base value (e.g. 0 or 7) as a second parameter b (the “base”).

Here, the stopping condition will depend on b.

The parameter b has to go “along for the ride” in the recursion.

```
;; (countdown-to n b) produces a list from n...b
;; countdown-to: Int Int → (listof Int)
;; requires: n >= b
;; Example:
(check-expect (countdown-to 4 2) (cons 4 (cons 3 (cons 2 empty))))

(define (countdown-to n b)
  (cond [(= n b) (cons b empty)]
        [else (cons n (countdown-to (sub1 n) b))]))
```

Another condensed trace

(countdown-to 4 2)

⇒ (cons 4 (countdown-to 3 2))

⇒ (cons 4 (cons 3 (countdown-to 2 2)))

⇒ (cons 4 (cons 3 (cons 2 empty)))

countdown-to works just fine if we put in negative numbers.

```
(countdown-to 1 -2)
⇒ (cons 1 (cons 0 (cons -1 (cons -2 empty))))
```

Here is the template for counting down to b.

```
(define (my-downto-b-fn n b)
  (cond [(= n b) (... b ...)]
        [else (... (my-downto-b-fn (sub1 n) b) ... )]))
```

Going the other way

What if we want an increasing count?

Consider the non-positive integers $\mathbb{Z}_{\leq 0}$.

;; A integer in $\mathbb{Z}_{\leq 0}$ is one of:

;; $\star$ 0

;; $\star$ (sub1 $\mathbb{Z}_{\leq 0}$)

Examples: -1 is (sub1 0), -2 is (sub1 (sub1 0)).

If an integer i is of the form (sub1 k), then k is equal to (add1 i). This suggests the following template.

Notice the additional requires section.

```
;; my-nonpos-fn: Int → Any
;; requires: n <= 0
(define (my-nonpos-fn n)
  (cond [(zero? n) ...]
        [else (... (my-nonpos-fn (add1 n)) ...)]))
```

We can use this to develop a function to produce lists such as (cons −2 (cons −1 (cons 0 empty))).

```
;; (countup n) produces a list from n...0
;; countup: Int → (listof Int)
;; requires: n <= 0
;; Example:
(check-expect (countup -2) (cons -2 (cons -1 (cons 0 empty))))

(define (countup n)
  (cond [(zero? n) (cons 0 empty)]
        [else (cons n (countup (add1 n)))]))
```

As before, we can generalize this to counting up to b, by introducing b as a second parameter in a template.

```
(define (my-upto-b-fn n b)
  (cond [(= n b) (... b ...)]
        [else (... (my-upto-b-fn (add1 n) b) ... )]))
```

```
;; (countup-to n b) produces a list from n...b
;; countup-to: Int Int → (listof Int)
;; requires: n <= b
;; Example:
(check-expect (countup-to 6 8) (cons 6 (cons 7 (cons 8 empty))))

(define (countup-to n b)
  (cond [(= n b) (cons b empty)]
        [else (cons n (countup-to (add1 n) b))]))
```

Yet another condensed trace

(countup-to 6 8)

⇒ (cons 6 (countup-to 7 8))

⇒ (cons 6 (cons 7 (countup-to 8 8)))

⇒ (cons 6 (cons 7 (cons 8 empty)))

Many imperative programming languages offer several language constructs to do repetition:

for i $=$ 1 to 10 do $\{ \ldots \}$

Racket offers one construct – recursion – that is flexible enough to handle these situations and more.

We will soon see how to use Racket's abstraction capabilities to abbreviate many common uses of recursion.

When you are learning to use recursion, sometimes you will “get it backwards” and use the countdown pattern when you should be using the countup pattern, or vice-versa.

Avoid using the built-in list function reverse to fix your error. It cannot always save a computation done in the wrong order.

Instead, learn to fix your mistake by using the right pattern.

- ⋆ You may **not** use reverse on assignments unless we say otherwise.

More complicated situations

As before, we may need to introduce auxiliary functions during the composition of a function. These may or may not be recursive themselves.

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

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

Filling in the list template

```
;; (sort lon) sorts the elements of lon in nondecreasing order
;; sort: (listof Num) → (listof Num)
(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.

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

insert is a recursive auxiliary function which consumes a number and a sorted list, and inserts the number to the sorted list.

A condensed trace of sort and insert

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

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

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

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

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

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

⇒ (insert 2 (cons 3 (cons 4 empty)))

⇒ (cons 2 (cons 3 (cons 4 empty)))

The auxiliary function insert

We again use the list template for insert.

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

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

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

n is the first number in the result if it is less than or equal to the first number in slon.

Otherwise, the first number in the result is the first number in slon, and the rest of the result is what we get when we insert n into (rest slon).

```
(define (insert n slon)
  (cond [(empty? slon) (cons n empty)]
        [(<= n (first slon)) (cons n slon)]
        [else (cons (first slon) (insert n (rest slon)))]))
```

A condensed trace of insert

(insert 4 (cons 1 (cons 2 (cons 5 empty))))

⇒ (cons 1 (insert 4 (cons 2 (cons 5 empty))))

⇒ (cons 1 (cons 2 (insert 4 (cons 5 empty))))

⇒ (cons 1 (cons 2 (cons 4 (cons 5 empty))))

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

List abbreviations

Now that we understand lists, we can abbreviate them.

The expression

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

can be abbreviated as

(list exp1 exp2 . . . expn)

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

Beginning Student With List Abbreviations also provides some shortcuts for accessing specific elements of lists.

Recall slide 01-23 to change language in DrRacket.

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

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

Use these **sparingly** to improve readability.

The templates we have developed remain very useful.

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

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

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

Quoting lists

If lists built using list consist of just symbols, strings, and numbers, the list abbreviation can be further abbreviated using the quote notation we used for symbols.

(cons 'red (cons 'blue (cons 'green empty))) can be written '(red blue green).

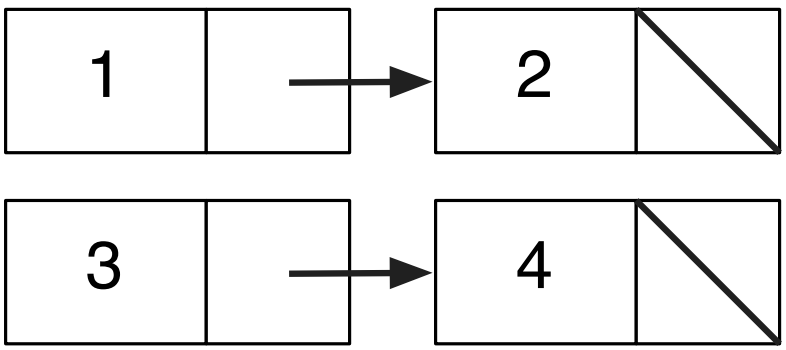
(list 5 4 3 2) can be written '(5 4 3 2), because quoted numbers evaluate to numbers; that is, '1 is the same as 1.

What is '() ?

Lists containing lists

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

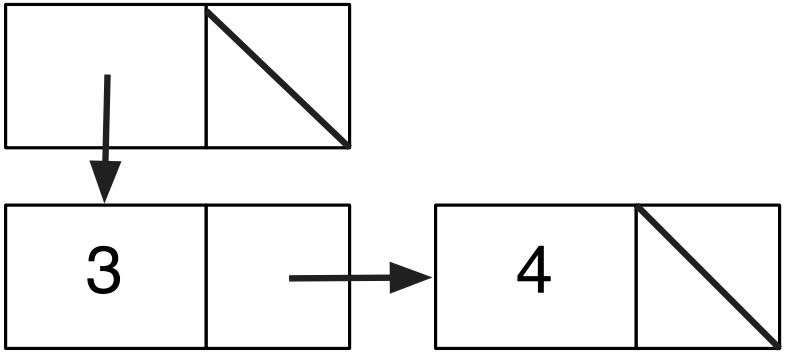
Here are two different two-element lists.



(cons 1 (cons 2 empty))

(cons 3 (cons 4 empty))

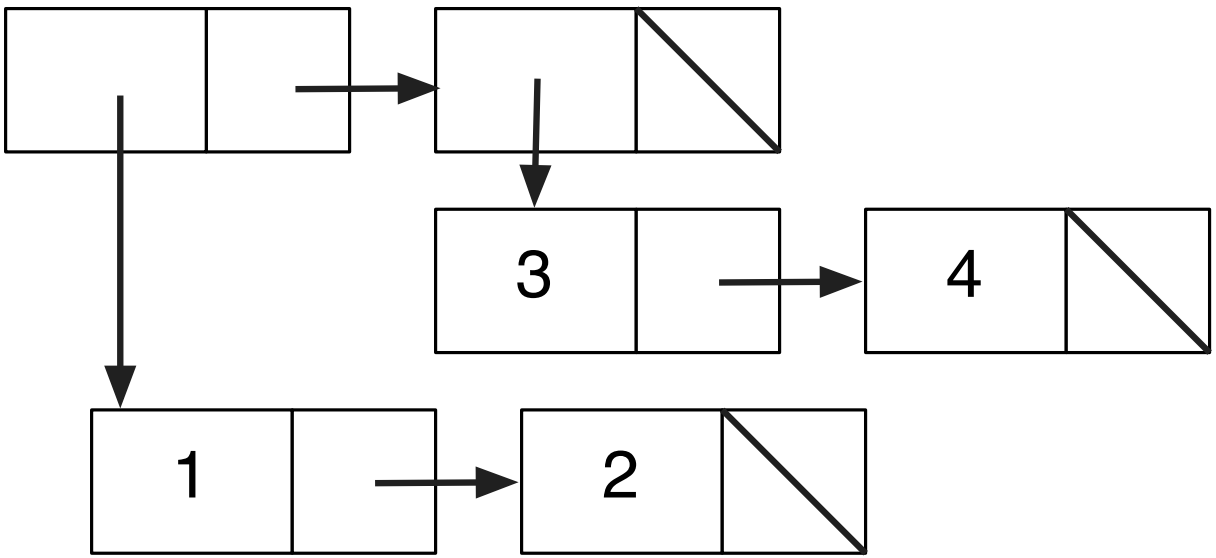
Here is a one-element list whose single element is one of the two-element lists we saw above.




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

We can create a two-element list by consing the other list onto this one-element list.

We can create a two-element list, each of whose elements is itself a two-element list.




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

We have several ways of expressing this list in Racket:

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

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

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

Clearly, the abbreviations are more expressive.

Lists versus structures

Since lists can contain lists, they offer an alternative to using structures.

Previously, we defined a salary record as a two-field structure but we could have defined a salary record as a two element list.

```
;; An Alternative Salary Record (AltSR) is a (list Str Num)

(list (list "Jane Doe" 50000)
      (list "Da Kou" 15500)
      (list "MuzaAlKhwarismi" 100000))
```

We can use our AltSR data definition to generate a template function, and for a list of AltSR, we can combine the templates.

```
;; An Alternative Salary Record (AltSR) is a (list Str Num)
;; my-altsr-fn: (list Str Num) → Any
(define (my-altsr-fn asr)
  (... (first asr) ... (second asr) ...))
;; my-listof-altsr-fn: (listof (list Str Num)) → Any
(define (my-listof-altsr-fn lst)
  (cond [(empty? lst) ...]
        [else (... (first (first lst)) ... ; name of first
                   (second (first lst)) ... ; salary of first
                   (my-listof-altsr-fn (rest lst))...)]))
```

Example: a function name-list which consumes a list of AltSR and produces the corresponding list of names.

```
;; name-list: (listof (list Str Num)) → (listof Str)
(define (name-list lst)
  (cond [(empty? lst) empty]
        [else (cons (first (first lst)) ; name of first
                    (name-list (rest lst)))]))
```

This code is less readable, because it uses only lists, instead of structures, and so is more generic-looking.

We can fix this with a few definitions.

```
(define (name x) (first x))
(define (salary x) (second x))

;; name-list: (listof (list Str Num)) → (listof Str)
(define (name-list lst)
  (cond [(empty? lst) empty]
        [else (cons (name (first lst))
                    (name-list (rest lst)))]))
```

This is one of the ways that structures can be simulated. There are others, as we will see.

Why use lists containing lists?

We could reuse the name-list function to produce a list of names from an list of tax records.

Our original structure-based salary record and tax record definitions would require two different (but very similar) functions.

We will exploit this ability to reuse code written to use “generic” lists when we discuss abstract list functions later in the course.

Why use structures?

Structure is often present in a computational task, or can be defined to help handle a complex situation.

Using structures helps avoid some programming errors (e.g., accidentally extracting a list of salaries instead of names).

Our design recipes can be adapted to give guidance in writing functions using complicated structures.

Most mainstream programming languages provide structures for use in larger programming tasks.

Some statically typed programming languages provide mechanisms that avoid having to duplicate code for types that are structurally identical or similar.

Inheritance in object-oriented languages is one such mechanism.

Such features could be easily built into the teaching languages, but are not, because the goal is to make a transition to other languages.

The full version of Racket (which goes beyond the standard, as we will see in CS 136) does have such features.

Different kinds of lists

When we introduced lists in module 05, the items they contained were not lists. These were **flat lists**.

We have just seen **lists of lists** in our example of 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

You know dictionaries as books in which you look up a word and get a definition or a translation.

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

Example: the telephone directory. Keys are names, and values are telephone numbers.

Example: your seat assignment for midterms. Keys are userids, and values are seat locations.

More generally, any two-column table can be viewed as a dictionary.

Example: your report card, where the first column contains courses, and the second column contains corresponding marks.

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

- **lookup**: given key, produce corresponding value
- **add**: add a (key,value) pair to the dictionary
- **remove**: given key, remove it and 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 make the keys numbers, and the values strings.

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

We can, in an association list, use other types for keys and values. We have used Num and Str here just to provide a concrete example.

We impose the additional restriction that an association list contains at most one occurrence of any key.

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

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

We can use the data definition to produce a template.

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

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

Since all valid values are strings, we can produce false to indicate that the key was not present in the association list.

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

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

We will leave the add and remove functions as exercises.

This solution is simple enough that it is often used for small dictionaries.

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

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

Two-dimensional data

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

For example, here is a multiplication table:

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

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

We can write mult-table using two applications of the "count up" idea.

```
;; (mult-table nr nc) produces multiplication table
;;     with nr rows and nc columns
;; mult-table: Nat Nat → (listof (listof Nat))
(define (mult-table nr nc)
  (rows-from 0 nr nc))

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

```
;; (row r nc) produces rth row of mult. table of length nc
;; row: Nat Nat → (listof Nat)
(define (row r nc)
  (cols-from 0 r nc))

;; (cols-from c r nc) produces entries c...(nc-1) of rth row of mult. table
;; cols-from: Nat Nat Nat → (listof Nat)
(define (cols-from c r nc)
  (cond [(>= c nc) empty]
        [else (cons (* r c) (cols-from (add1 c) r nc))]))
```

Processing two lists simultaneously

We now skip ahead to section 17 material.

This involves more complicated recursion, namely in 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
;; my-append: (listof Any) (listof Any) → (listof Any)
;; Examples:
(check-expect (my-append empty '(1 2)) '(1 2))
(check-expect (my-append '(3 4) '(1 2 5)) '(3 4 1 2 5))

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

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

The code only does structural recursion on lst1.

The parameter lst2 is “along for the ride”.

A condensed trace

(my-append (cons 1 (cons 2 empty)) (cons 3 (cons 4 empty)))

⇒ (cons 1 (my-append (cons 2 empty) (cons 3 (cons 4 empty))))

⇒ (cons 1 (cons 2 (my-append empty (cons 3 (cons 4 empty)))))

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

Case 2: processing in lockstep

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

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

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

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

The template is thus simpler than in the general case.

```
(define (my-lockstep-fn lst1 lst2)
  (cond [(empty? lst1) ... ]
        [else
         (... (first lst1) ... (first lst2) ...
              (my-lockstep-fn (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 '(1 2 3).

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

The function dot-product

```
;; (dot-product lon1 lon2) computes the dot product
;;     of vectors lon1 and lon2
;; dot-product: (listof Num) (listof Num) → Num
;; requires: lon1 and lon2 are the same length
(check-expect (dot-product empty empty) 0)
(check-expect (dot-product '(2) '(3)) 6)
(check-expect (dot-product '(2 3) '(4 5)) 23)

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

```
(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 (cons 2 (cons 3 empty))
             (cons 4 (cons 5 empty)))
⇒ (+ 8 (dot-product (cons 3 empty)
                    (cons 5 empty))
⇒ (+ 8 (+ 15 (dot-product empty
                          empty)))
⇒ (+ 8 (+ 15 0)) ⇒ 23
```

Case 3: processing at different rates

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

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

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

The template so far

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

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

Refining the template

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

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

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

Further refinements

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

... (first lst2) ... (my-twolist-fn lst1 (rest lst2)) ...

... (first lst1) ... (my-twolist-fn (rest lst1) lst2) ...

... (first lst1) ... (first lst2) ... (my-twolist-fn (rest lst1) (rest lst2)) ...

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

Example: merging two sorted lists

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

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

merge will produce one such list containing all elements.

As an example:

(merge (list 1 8 10) (list 2 4 6 12)) $\Rightarrow$ (list 1 2 4 6 8 10 12)

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

(merge empty empty) $\Rightarrow$ empty

(merge empty

(cons 2 empty)) $\Rightarrow$ (cons 2 empty)

(merge (cons 1 (cons 3 empty))

empty) $\Rightarrow$ (cons 1 (cons 3 empty))

(merge (cons 1 (cons 4 empty))

(cons 2 empty)) $\Rightarrow$ (cons 1 (cons 2 (cons 4 empty)))

(merge (cons 3 (cons 4 empty))

(cons 2 empty)) $\Rightarrow$ (cons 2 (cons 3 (cons 4 empty)))

Reasoning about merge

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

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

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

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

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

A condensed trace

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

A condensed trace (with lists)

```
(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)))))
```

Consuming a list and a number

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

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

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

Example: "Does 2 appear at least 3 times in the list (list 4 2 2 3 2 4)?" returns true.

The function at-least?

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

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

Developing the code

The recursion will involve 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.

Refining at-least?

```
(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)) ???]))
```

In which cases can we return 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 [(= n 0) 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

Testing list equality

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

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))
```

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

Some further simplifications are possible.

Built-in list equality

As you know, Racket provides the predicate equal? which tests structural equivalence. It can compare two atomic values, two structures, or two lists. Each of the nonatomic objects being compared can contain other lists or structures.

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.

Goals of this module

You should understand the recursive definition of a natural number, and how it leads to a template for recursive functions that consume natural numbers.

You should understand how subsets of the integers greater than or equal to some bound m, or less than or equal to such a bound, can be defined recursively, and how this leads to a template for recursive functions that “count down” or “count up”. You should be able to write such functions.

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 and quote notation for lists where appropriate.

You should be able to construct and work with lists that contain lists.

You should understand the similar uses of structures and fixed-size lists, and be able to write functions that consume either type of data.

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.