Module 5: 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.

Note that natural numbers can be defined non-recursively. But, our focus is on a recursive view.

First, however, we will introduce some shortcuts for working with lists.

List abbreviations

Now that we understand Racket lists, we can abbreviate them.

The expression

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

can be abbreviated as

(list exp1 exp2 ... expn)

The list (cons 2 (cons 4 (cons 5 (cons 5 empty)))) can be expressed as (list 2 4 5 6).

Racket also provides some shortcuts for accessing specific elements of lists.

(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, and use list to construct long lists for testing.

There will still remain situations when using cons is the best choice.

We should now switch to Beginning Racket with list abbreviations.

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 will mostly use list in our examples and tests.

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).

Our function implementations are more likely to use cons than list.

Natural numbers

```
;; natural number (Nat) is either:
;; * 0
;; * (add1 Nat)
```

The analogy to the self-referential definition of lists can be made clearer by using the built-in add1 function:

(add1 0) $\Rightarrow$ 1

(add1 (add1 0)) $\Rightarrow$ 2

(add1 (add1 (add1 0))) $\Rightarrow$ 3

```
;; A List is one of:
;; * empty
;; * (cons Any List)
```

To derive a template, we used a cond for the two cases.

We broke up the nonempty list (cons f r) using

- the selector first to extract f,
- the selector rest to extract r, and
- an application of the function on r.

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

To derive a template for a natural number n, we will use a cond for the two cases.

We will break up the non-zero case $n=k+1$ using

- the function sub1 to extract k and
- an application of the function on k (that is, on (sub1 n)).

Comparing the templates

```
;; listof-any-template: (listof Any) → Any
(define (listof-any-template alist)
  (cond
    [(empty? alist) . . . ]
    [else (. . .  (first alist) . . .  (listof-any-template (rest alist)) . . . )]))
;; nat-template: Nat → Any
(define (nat-template n)
  (cond
    [(zero? n) . . . ]
    [else (. . .  n . . .  (nat-template (sub1 n)) . . . )]))
```

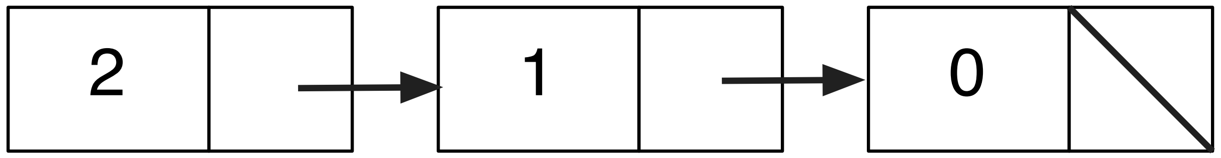
Example: producing a decreasing list

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

Use the data definition to derive examples.

(countdown 0) ⇒ (list 0)

(countdown 2) ⇒ (list 2 1 0)



Developing countdown

Using the natural number template:

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

If n is 0, we produce the list containing 0.

If n is nonzero, we cons n onto the countdown list for n-1.

```
;; (countdown n) produces a decreasing list of nats starting at n
;; and ending with 0
;; countdown: Nat → (listof Nat)
;; Examples:
(check-expect (countdown 0) (list 0))
(check-expect (countdown 2) (list 2 1 0))

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

Condensed trace of countdown

```
(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 argument, it will not terminate.

The following variation is a little more robust.

It can handle negative arguments more gracefully.

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

What if we want to count down to a different value?

Suppose we want to countdown to 11, not 0. We would then be dealing with integers that are greater or equal to 11:

```
;; A Nat11, an integer greater than or equal to 11, is either
;; 11 or
;; (add1 Nat11)
```

The template differs only in the base case:

```
(define (countdown-to11 n)
  (cond [(= n 11) ...]
        [else (... n ... (countdown-to11 (sub1 n)) ...)]))
```

Further generalization

We can modify the templates to stop at any base b by changing the base case test. The parameter b (for “base”) has to “go along for the ride” in the recursion.

```
;; downto-template: Int Int → Any
;; requires: n >= b
(define (downto-template n b)
  (cond
    [(= n b) (... b ...)]
    [else (... n ... (downto-template (sub1 n) b)...)]))
```

The template nat-template is a special case where b is zero. Since we know the value zero, it doesn't need to be a parameter.

The function countdown-to

```
;; (countdown-to n b) produces a decreasing list of ints starting with n
;;    and ending with b.
;; countdown-to: Int Int → (listof Int)
;; requires: n >= b
(define (countdown-to n b)
  (cond
    [(<= n b) (cons b empty)]
    [else (cons n (countdown-to (sub1 n) b))]))
```

Condensed trace of countdown-to

(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)))

- What is the result of (countdown-to 1 -2)?
- Of (countdown-to -4 -2)?

Going the other way

What if we want to count up to 0? Then we are dealing with non-positive integers:

```
;; A NonPosInt is one of:
;; * 0
;; * (sub1 NonPosInt)
```

Examples: (sub1 0), (sub1 (sub1 0)), ...

Our base case will again be checking for 0.

Our recursive case will be on (add1 n).

```
;; nonposint-template: NonPosInt → Any
(define (nonposint-template n)
  (cond
        [(zero? n) ...]
        [else ... (nonposint-template (add1 n)) ...]))
```

```
;; countup: NonPosInt → Any
(define (countup n)
  (cond
        [(zero? n) (cons 0 empty)]
        [else (cons n (countup (add1 n)))]))
```

Generalizing counting up

What if we want an increasing count up to an integer b? We can generalize the nonposint-template to consume an additional parameter b, which will go along for the ride.

```
;; upto-template: Int Int → Any
;; requires: n <= b
(define (upto-template n b)
  (cond
    [(>= n b) (... b ...)]
    [else (... n ... (upto-template (add1 n) b) ...)]))
```

```
;; (countup-to n b) produces an increasing list of ints starting with n
;;    and ending with b
;; countup-to: Int Int → (listof Int)
;; requires: n <= b
;; Examples:
(check-expect (countup-to 5 5) (list 5))
(check-expect (countup-to 6 8) (list 6 7 8))

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

Condensed trace of countup-to

```
(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)))
```

What does this do?

```
;; countup-by: Int Int Nat → (listof Int)
(define (countup-by n b inc)
  (cond [(>= n b) empty]
        [else (cons n (countup-by (+ n inc) b inc))]))

(countup-by 2 5 1)

(countup-by −10 20 7)

(countup-by 10 5 1)
```

range function

Racket has a built-in function range that generalizes countup-by, and provides a short-cut for basic counting up or down.

```
;; (range a b c) produces a list of integers from a to b, but not
;;    including b, stepping by c.
;; range: Int Int Int → (listof Int)

(range 4 7 1) ⇒ (list 4 5 6)
(range 5 0 −1) ⇒ (list 5 4 3 2 1)
(range −4 4 3) ⇒ (list −4 −1 2)
(range 2 2 5) ⇒ empty
```

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

```
for i = 1 to 10 do { . . . }
```

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

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

When you are learning to use recursion with integers, 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 starting with the right template.

Do not use reverse in your labs, assignments, or exams. You will lose many marks if you do.

Example: factorial

Suppose we wish to compute $n! = n \cdot (n-1) \cdot (n-2) \cdots 1$, or (factorial n), for $n \geq 0$ ($0!$ is defined to be 1).

To choose between the templates, we consider what information each gives us.

Counting down: (factorial (sub1 n))

Counting up: (factorial (add1 n))

Filling in the template

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

(define (factorial n)
  (cond
    [(zero? n) 1]
    [else (* n (factorial (sub1 n)))]))
```

Example: String prefixes

Suppose we want to find all the prefixes of a string, starting from the shortest prefix (the empty string) to the string itself.

For example,

```
(prefixes "abc") ⇒
       (list "" "a" "ab" "abc")
```

How is this a problem on natural numbers?

We need to include: (substring s 0 0), (substring s 0 1), (substring s 0 2), etc.

Example: Greatest common divisor

The greatest common divisor (gcd) of a group of positive natural numbers is the largest integer that divides evenly into each. For example, the gcd of 21, 35, 14 is 7.

Suppose we want to find the gcd of three natural numbers: n1, n2, n3.

What range of numbers should we check?

In what order should we check them?

How do we check for a common divisor?

Example: Checking for a common divisor

How do we check if a particular number is a common divisor?

```
;; (common? k n1 n2 n3) produces true if k divides evenly into
;;    n1, n2, and n3, and false otherwise.
;; common?: Nat Nat Nat Nat → Boolean
;; requires: k > 0
(define (common? k n1 n2 n3)
        (and (zero? (remainder n1 k))
             (zero? (remainder n2 k))
             (zero? (remainder n3 k))))
```

```
;; gcd-three: Nat Nat Nat → Nat
;; requires: n1, n2, n2 > 0
(define (gcd-three n1 n2 n3)
  . . . )
```

How do we countdown?

gcd-three needs a recursive helper function.

A more complicated situation - Sorting

Sometimes a recursive function will use a helper function that itself is recursive.

Sorting a list of numbers provides a good example.

In CS 116, we will see several different sorting algorithms.

We will sort from lowest number to highest. `(list 3 5 9)`

A list is sorted if no number is followed by a smaller number.

Filling in the list template

```
;; (my-sort alon) produces a list containing same values as values sorted in
;;    nondecreasing order.
;; my-sort: (listof Num) → (listof Num)
(define (my-sort values)
  (cond
    [(empty? values) ...]
    [else (... (first values) ... (my-sort (rest values)) ...)]))
```

If the list values is empty, so is the result. Otherwise, the template suggests somehow combining the first element and the sorted version of the rest.

```
(define (my-sort values)
  (cond
    [(empty? values) empty]
    [else (insert (first values) (my-sort (rest values)))]))
```

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

Tracing my-sort

(my-sort (cons 7 (cons 4 (cons 3 empty))))

$\Rightarrow$ (insert 7 (my-sort (cons 4 (cons 3 empty))))

$\Rightarrow$ (insert 7 (insert 4 (my-sort (cons 3 empty))))

$\Rightarrow$ (insert 7 (insert 4 (insert 3 (my-sort empty))))

$\Rightarrow$ (insert 7 (Insert 4 (insert 3 empty)))

$\Rightarrow$ (insert 7 (insert 4 (cons 3 empty)))

$\Rightarrow$ (insert 7 (cons 3 (cons 4 empty)))

$\Rightarrow$ (cons 3 (cons 4 (cons 7 empty)))

The helper function insert

We again use the list template for insert.

```
;; (insert n values) produces the sorted (in nondecreasing order) list
;;    formed by adding the number n to the sorted list values.
;; insert: Num (listof Num) → (listof Num)
;; requires: values is sorted in nondecreasing order
(define (insert n values)
  (cond
    [(empty? values) . . . ]
    [ else (. . . (first values) . . . (insert n (rest values)) . . . )]))
```

Reasoning about insert

If values is empty, then the result is the list containing just n.

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

n is the first number in the resulting list if it is less than or equal to (first values).

Otherwise, (first values) is the first number in the resulting list, and we get the rest of the resulting list by inserting n into (rest values).

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

Tracing insert

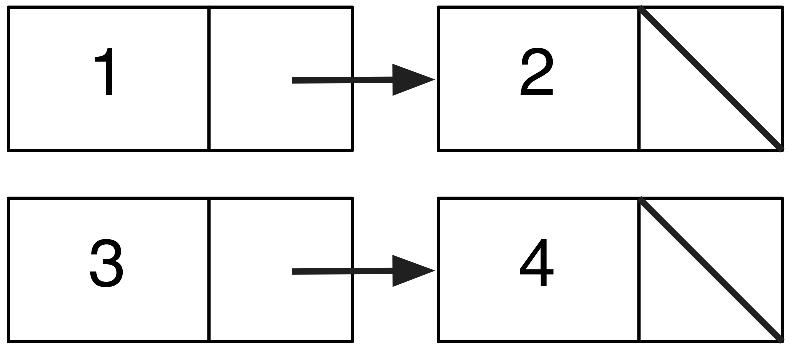
```
(insert 5 (cons 2 (cons 4 (cons 6 empty))))

⇒ (cons 2 (insert 5 (cons 4 (cons 6 empty))))

⇒ (cons 2 (cons 4 (insert 5 (cons 6 empty))))

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

This is known as **insertion sort**.

Lists containing lists

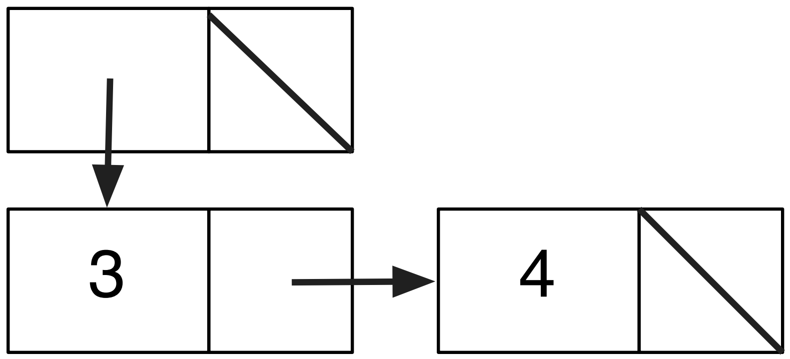
Lists can contain anything, including other lists, at which point these abbreviations can improve readability.

Here are two different two-element lists.



(cons 1 (cons 2 empty)) or (list 1 2) (cons 3 (cons 4 empty)) or (list 3 4)

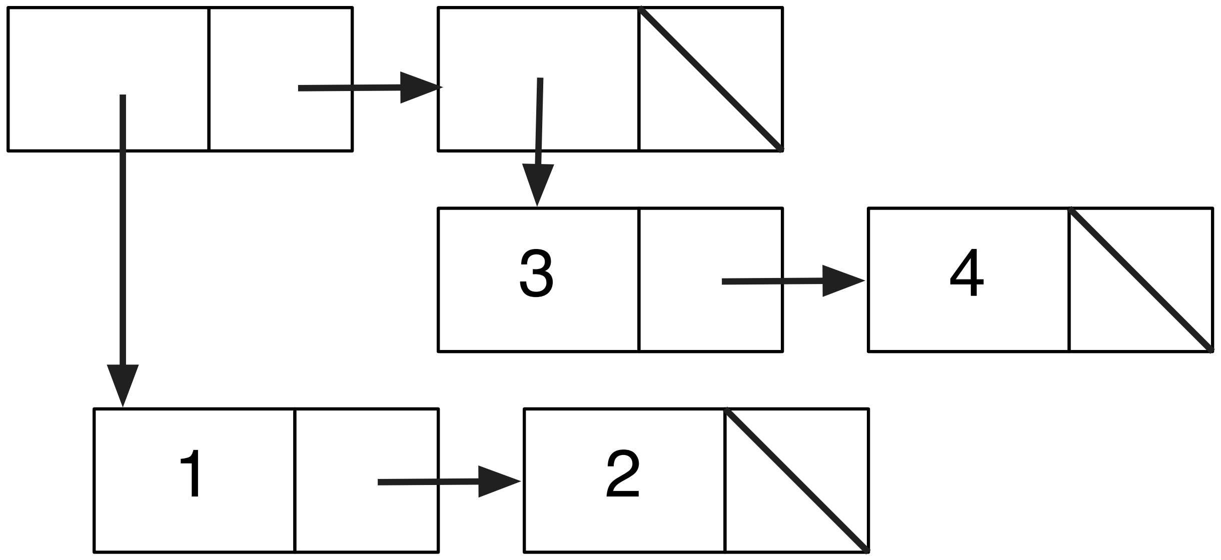
Here is a one-element list whose single element is one of the two-element lists we saw above.



(cons (cons 3 (cons 4 empty)) empty) or (list (list 3 4)).

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

Using lists for related data

Aside from lists, the types we've seen so far have been very simple: Num, Str, Bool, Char.

Sometimes, it would be nice to group together related pieces of information into a single value. For example, suppose we need the name of a student, along with their grades on assignments, the midterm exam, and the final exam. We could use the four different values separately, or we could create a list with the four values inside it, and call it a Student.

For example, the following are three Student values:

```
(list "Virginia Woolf" 100 100 100)
(list "Alan Turing" 90 80 40)
(list "Anonymous" 30 55 10)
```

We can use a data definition to be precise about our new Student type.

```
;; A **Student** is a (list Str Num Num Num)
;; where:
;; * the first item is the name of a student
;; * the second item is the assignment grade
;; * the third item is the midterm exam grade
;; * the fourth item is the final exam grade
;; requires:
;; * all grades are between 0 and 100, inclusive.
```

We can construct a Student value with a new function that we define:

```
;; make-student: Str Num Num Num → Student
;; requires 0 ≤ asst, mid, final ≤ 100
(define (make-student name assts mid final)
          (list name assts mid final))
```

A template for Student

```
;; student-template: Student → Any
(define (student-template s)
      (... (first s)   ; name of first
       ... (second  s)  ; assts of first
       ... (third  s)  ; mid of first
       ... (fourth sl) ... ))  ; final of first
```

A template for (listof Student)

```
;; listof-student-template: (listof Student) → Any
(define (slist-template sl)
  (cond [(empty? sl) ...]
        [else (... (student-template (first sl))
                   ... (slist-template (rest sl))...)]))
```

Example: a function name-list which consumes a (listof Student) and produces the corresponding list of student names.

```
(define (name-list sl)
  (cond
    [(empty? sl) empty]
    [else (cons (first (first sl)) ; name of first
                (name-list (rest sl)))]))
```

This code is not very readable because the meaning of (first (first sl)) is not not clear without the data definition.

We can fix this with a few definitions.

```
(define (student-name x) (first x))
(define (student-assts x) (second x))
(define (student-mid x) (third x))
(define (student-final x) (fourth x))

(define (student-name-list sl)
  (cond
    [(empty? sl) empty]
    [else (cons (student-name (first sl))
                (name-list (rest sl)))]))
```

name-list is re-usable

If we define a **glist** as a list of two-element lists, each sublist holding name and grade, we could reuse the name-list function to produce a list of names from a glist.

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

What happens with the following call?

```
(name-list (list (list 1 "a") (list 3 "c") (list 2 "b")))
```

An exercise with a new type: Point

We often need to store information about a point in space or a grid location, that is, about an (x, y) pairing.

```
;; A Point is a (list Num Num)
;; where: the first item is the x-co-ordinate, and
;;          the second is the y-co-ordinate.
```

Write a template for any function that consumes a Point value.

Write a function to calculate the distance between two points.

Dictionaries

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

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

- Seat assignment look-up (keys=usernames, values=seats)
- Reverse telephone lookup (keys=phone numbers, values=names).

Our task is to store the set of (key,value) pairs to support the operations *lookup*, *add*, and *remove*.

Association lists

;; An **Association** (As) is a (list Num Str),

;; where

;; * the first item is the key,

;; * the second item is the associated value.

;; An **Association List** (AL) is one of:

;; * empty

;; * (cons As AL)

;; Note: All keys must be distinct.

Racket has a built-in lookup operation for Association Lists called assoc. This is our version of it.

```
;; (my-assoc k alst) produces the association with key k, or
;;    false if k not present.
;; my-assoc: Num AL → (anyof As false)
(define (my-assoc k alst)
  (cond
    [(empty? alst) false]
    [(equal? k (first (first alst))) (first alst)]
    [else (my-assoc 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.

Keeping the list in sorted order might improve some searches, but there is still a case where the whole list is searched.

In a later module, we will see how to avoid this.

Different kinds of lists

When we introduced lists in Module 4, the items they contained were not lists. These were **flat lists**.

We have just seen **lists of lists** in our example of lists containing two-element flat lists.

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

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 b, 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 for lists where appropriate.

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

You should understand the uses of fixed-size lists, and be able to write functions that consume or produce such data.

You should be able to use association lists to implement dictionaries.