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.

Natural numbers

;; **natural number** (Nat) is either:

;; * 0

;; * 1 + 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

;; list-template: (listof Any) → Any

(define (list-template alist)

  (cond

    [(empty? alist) . . .]

    [else (. . . (first alist) . . . (list-template (rest alist)) . . .)]))

;; nat-template: Nat → Any

(define (nat-template n)

  (cond

    [(zero? n) . . .]

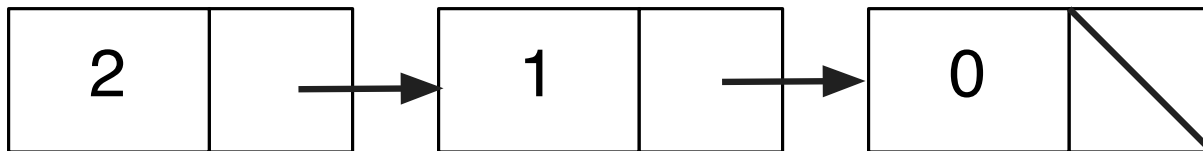   [else (. . . n . . . (nat-template (sub1 n)) . . .)]))

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) $\Rightarrow$ (cons 0 empty)

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

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) (cons 0 empty))
(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)))]))
```

Condensed trace of countdown

(countdown 2)

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

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

$\Rightarrow$ (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)))]))
```

Counting down to a different base

We can generalize countdown by providing a base value as a parameter b.

This function deals with the following range of integers:

An **integer greater than or equal to** b is either

- b or

- 1 plus *an integer greater than or equal to b*.

The parameter b (for "base") has to "go along for the ride" in the recursion.

Template for an integer greater than or equal to a base

;; downto-template: Int Int $\rightarrow$ 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 $\rightarrow$ (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)

$\Rightarrow$ (cons 4 (countdown-to 3 2))

$\Rightarrow$ (cons 4 (cons 3 (countdown-to 2 2)))

$\Rightarrow$ (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 an increasing count up to an integer b?

Consider the following definition:

An **integer less than or equal to b** is either:

- b

- An *(integer less than or equal to b) minus 1*

Based on this definition, the following are integers less than or equal to $b = 14$: 14, (sub1 14), (sub1 (sub1 14)), ...

Our recursive step n=k-1 will use (add1 n).

5: Working with recursion

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

5: Working with recursion

```
;; (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) (cons 5 empty))
(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))]))
```

Condensed trace of countup-to

(countup-to 6 8)

$\Rightarrow$ (cons 6 (countup-to 7 8))

$\Rightarrow$ (cons 6 (cons 7 (countup-to 8 8)))

$\Rightarrow$ (cons 6 (cons 7 (cons 8 empty)))

Consider another variation

;; countup-by: Int Int Nat $\rightarrow$ (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)

5: Working with recursion

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 $\rightarrow$ (listof Int)

(range 4 7 1) $\Rightarrow$ (cons 4 (cons 5 (cons 6 empty)))

(range 5 0 −1) $\Rightarrow$ (cons 5 (cons 4 (cons 3 (cons 2 (cons 1 empty)))))

(range −4 4 3) $\Rightarrow$ (cons −4 (cons −1 (cons 2 empty)))

(range 2 2 5) $\Rightarrow$ empty

5: Working with recursion

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") ⇒

(cons "" (cons "a" (cons "ab" (cons "abc" empty))))

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 $\rightarrow$ 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.

(cons 3 (cons 5 (cons 9 empty)))

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

5: Working with recursion

Filling in the list template

;; (my-sort alon) produces a list containing same values as alon sorted in

;;   nondecreasing order.

;; my-sort: (listof Num) $\rightarrow$ (listof Num)

(define (my-sort alon)

  (cond

    [(empty? alon) . . .]

    [else (. . . (first alon) . . . (my-sort (rest alon)) . . .)]))

If the list alon 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 alon)
  (cond
    [(empty? alon) empty]
    [else (insert (first alon) (my-sort (rest alon)))]))
```

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)))

5: Working with recursion

The helper function **insert**

We again use the list template for insert.

;; (insert n alon) produces the sorted (in nondecreasing order) list

;;   formed by adding the number n to the sorted list alon.

;; insert: Num (listof Num) $\rightarrow$ (listof Num)

;; requires: alon is sorted in nondecreasing order

(define (insert n alon)

  (cond

    [(empty? alon) . . .]

    [else (. . . (first alon) . . . (insert n (rest alon)) . . .)]))

Reasoning about insert

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

If alon 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 alon).

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

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

Tracing insert

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

$\Rightarrow$ (cons 2 (insert 5 (cons 4 (cons 6 empty))))

$\Rightarrow$ (cons 2 (cons 4 (insert 5 (cons 6 empty))))

$\Rightarrow$ (cons 2 (cons 4 (cons 5 (cons 6 empty))))

This is known as **insertion sort**.

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 result of the trace we did on the last slide can be expressed as

(list 2 4 5 6).

Beginning Student Racket with List Abbrevations 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.

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

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.

Lists containing lists

Lists can contain anything, including other lists, at which point these abbreviations can improve readability.
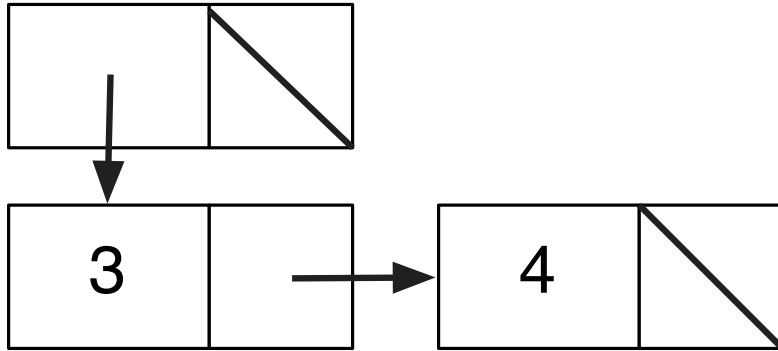
Here are two different two-element lists.
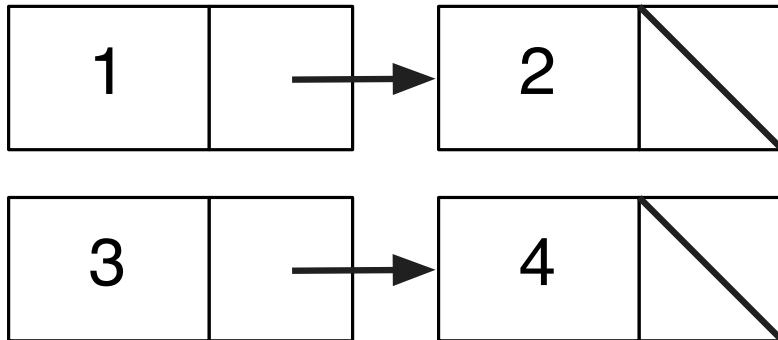
(cons 1 (cons 2 empty))

(cons 3 (cons 4 empty))
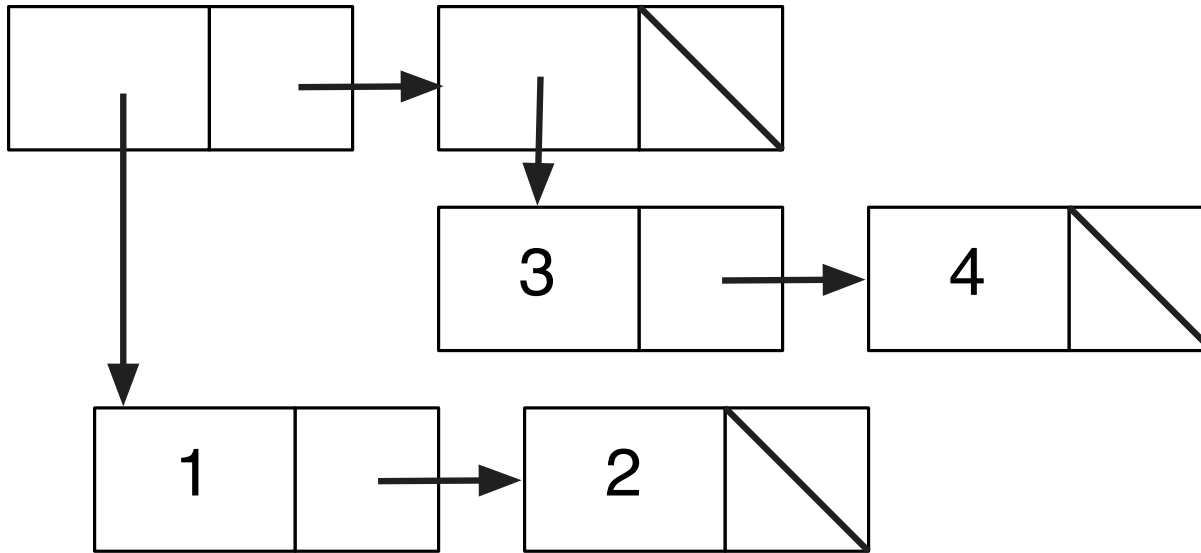
Here is a one-element list whose single element is one of the two-element lists we saw above.

(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))

Expressing the list

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))

Many find the abbreviations more expressive and easier to use.

Using lists for related data

Aside from lists, the types we've seen so far have been very simple: a number, a string, a Boolean value.

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 (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

;; * and all grades are between 0 and 100, inclusive.

Using (listof Student)

We can develop a template for this new type of list.

```
(define (slist-template sl)
  (cond
    [(empty? sl) ...]
    [else (...  (first (first sl)) ; name of first
           ...  (second (first sl)) ; assts of first
           ...  (third (first sl)) ; mid of first
           ...  (fourth (first sl)) ; final of first
           ...  (slist-template (rest sl))...)]))
```

5: Working with recursion

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 (name x) (first x))

(define (assts x) (second x))

(define (mid x) (third x))

(define (final x) (fourth x))

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

5: Working with recursion

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")))

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 (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.

Without these data definitions, we can write the type as (listof (list Num Str))

;; (lookup-al k alst) produces the value associated with 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.

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.