Module 6: 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.

Natural numbers

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

;; * 0

;; * 1 + Nat

The analogy to the self-referential definition of lists can be made clearer by defining a "1 plus" function:

```
(define (add1 n) (+ 1 n))
(add1 0) ⇒ 1
(add1 (add1 0)) ⇒ 2
(add1 (add1 (add1 0))) ⇒ 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 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 using

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

Comparing the templates

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

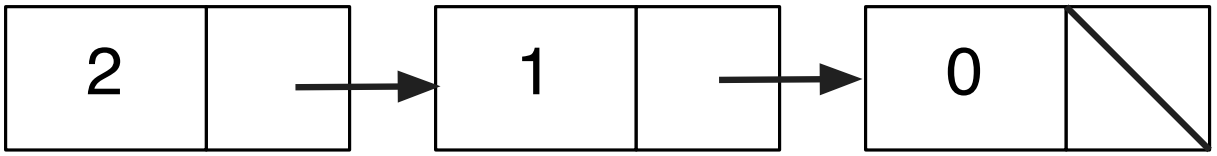
Example: producing a decreasing list

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)))]))
```

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 data definition:

```
;; A natural number greater than or equal to 7 (Nat7) is one of:
;; * 7
;; * (add1 Nat7)
```

Template for a natural number greater than or equal to 7

```
;; my-downto-7-fun: Nat7 → Any
(define (my-downto-7-fun n)
  (cond
    [(<= n 7) ...]
    [else ... n ... (my-downto-7-fun (sub1 n))]))
```

```
;; (countdown-to-7 n) produces a decreasing list of Nats starting with n
;;    and ending with 7.
;; countdown-to-7: Nat7 → (listof Nat7)
;; Examples:
(check-expect (countdown-to-7 7) (cons 7 empty))
(check-expect (countdown-to-7 9) (cons 9 (cons 8 (cons 7 empty))))
(define (countdown-to-7 n) ...)
```

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

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

This corresponds to the definition:

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

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

The template my-nat-fun 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) . . . )
```

```
(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)?

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.

$\mathbb{Z}_{\geq b}$ defines the integers greater than or equal to b.

Going the other way

What if we want an increasing count?

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

;; A **NonPosInt** is one of:

;; * 0

;; * (sub1 NonPosInt)

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

```
;; my-nonpos-fun: Int → Any
;; requires: n <= 0
(define (my-nonpos-fun n)
  (cond
    [(zero? n) ...]
    [else ... n ... (my-nonpos-fun (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 an increasing list of ints from n up to zero.
;; countup: Int → (listof Int)
;; requires: n <= 0
;; Examples:
(check-expect (countup 0) (cons 0 empty))
(check-expect (countup -2) (cons -2 (cons -1 (cons 0 empty))))

(define (countup n) ...)
```

```
(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 for integers in

integer in $\mathbb{Z}_{\leq b}$

For $b = 14$, 14 is an integer in $\mathbb{Z}_{\leq 14}$.

13 is an integer in $\mathbb{Z}_{\leq 14}$ because it is $14 - 1$.

12 is because it is $13 - 1$, and so on.

In other words, 12 is (sub1 (sub1 14)).

We will use (add1 n) in our recursive step.

```
;; my-upto-fun: Int Int → Any
;; requires: n <= b
(define (my-upto-fun n b)
  (cond
    [(>= n b) ... b ...]
    [else ... n ... (my-upto-fun (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) (cons 5 empty))
(check-expect (countup-to 6 8) (cons 6 (cons 7 (cons 8 empty))))

(define (countup-to n b) ...)
```

```
(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)))

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.

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

Instead, learn to fix your mistake by starting with the right template.

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 (my-downto-fun n b)
  (cond
    [(= n b) ... b ...]
    [else ... n ... (my-downto-fun (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") $\Rightarrow$

(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 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.

Even numbers

```
;; An EvenNat is one of:
;; * 0
;; (+ 2 EvenNat)

(define (my-even-fun n)
  (cond
    [(zero? n) ...]
    [else ... n ... (my-even-fun (- n 2)) ...]))
```

Integer powers of ten

```
;; A Power10 is one of:
;; * 1
;; * (* 10 Power10)

(define (my-power-ten-fun n)
  (cond
    [(= 1 n) ...]
    [else ... n ... (my-power-ten-fun (/ n 10)) ...]))
```

More complicated situations

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

Sorting a list of numbers provides a good example.

In CS 115 and 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.

Filling in the list template

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

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

⇒ (insert 7 (sort (cons 4 (cons 3 empty))))

⇒ (insert 7 (insert 4 (sort (cons 3 empty))))

⇒ (insert 7 (insert 4 (insert 3 (sort empty))))

⇒ (insert 7 (Insert 4 (insert 3 empty)))

⇒ (insert 7 (insert 4 (cons 3 empty)))

⇒ (insert 7 (cons 3 (cons 4 empty)))

⇒ (cons 3 (cons 4 (cons 7 empty)))

The auxiliary 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) → (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 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.

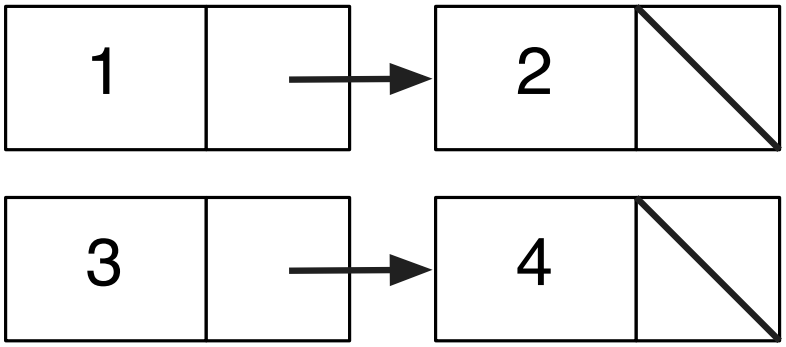
We use list to construct a list of fixed size (whose length is known when we write the program).

We 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).

Lists containing lists

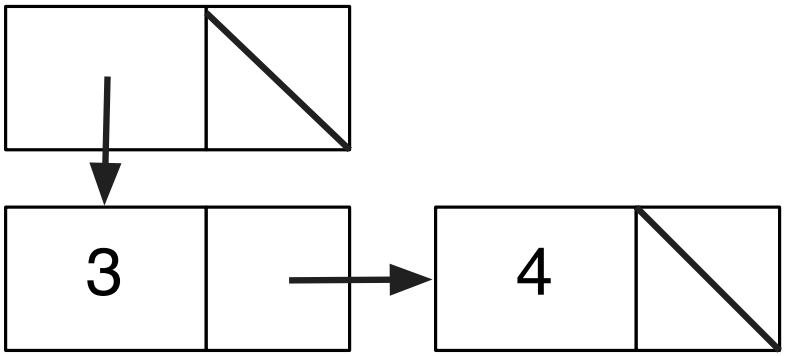
Lists can contain anything, including other lists, at which point these abbreviations can improve readability.

Here are two different two-element lists.




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

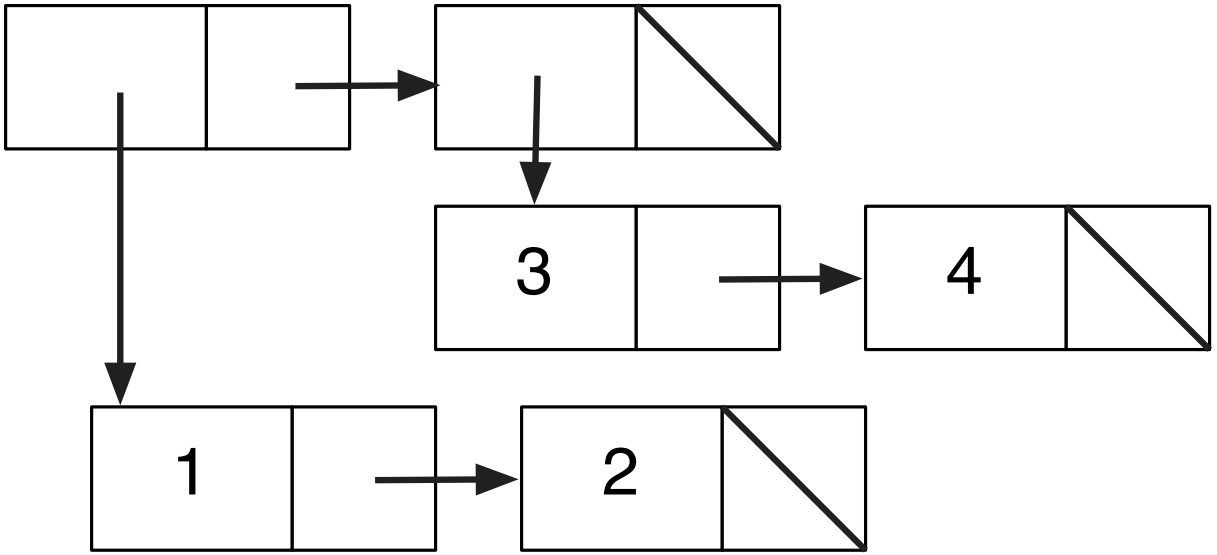
Here is a one-element list whose single element is one of the two-element lists we saw above.




```
(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.

Lists versus structures

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

For example, we defined a list of students as a list of four-field structures.

We could have instead defined an **slist** as a list of four-element lists.

```
(list (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 lists containing four-element lists.

```
;; A Student is (list Str Num Num Num)
;; requires:
;; * 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.
;;
;; An SList is (listof Student)
```

```
(define (my-slist-fun 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
          ... (my-slist-fun (rest sl))...]))
```

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

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

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 (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)))]))
```

Why use lists containing lists?

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.

Our original structure-based definitions of lists of students and grades 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 grades 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.

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 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 the 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 Module 8 we will see how to avoid this.

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) $\Rightarrow$

(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 $r * c$.

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 multiplication table, rows r through nr
;; 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 multiplication 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 through nc of rth row of
;;    mulitplication 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))]))
```

Different kinds of lists

When we introduced lists in Module 5, 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.

In Module 10, 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 similar uses of structures and fixed-size lists, and be able to write functions that consume either data.

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