CS115 - Module 6 - Recursion

Cameron Morland

Fall 2017

Reminder: if you have not already, ensure you:

- Read *How to Design Programs*, sections 11, 12, and 13 (Intermezzo 2).

Playing Peano

We have been working with natural numbers without defining them. Let's change that.
The **Peano axioms** use a successor function, $S(n)$, to define natural numbers as follows:

1. 0 is a natural number.
2. For every natural number n, $S(n)$ is a natural number.
3. For all natural numbers m and n, $m = n$ if and only if $S(m) = S(n)$.
4. There is no natural number whose successor is 0.

I can represent 1 as $S(0)$, 2 as $S(S(0))$, 3 as $S(S(S(0)))$, and so on.
In Racket we have **`add1`** instead of S. So we can say:

A `Nat` is either:

- 0 or
- `(`**`add1`**` r)` where `r` is a `Nat`

Does this look familiar?

A list is either:

- `'()` or
- `(`**`cons`**` f r)` where `f` is a value and `r` is a list.

We have a recursive definition for a list, and can do the same for `Nat`.

Recursive Analogies

So where previously we could build a list using **cons**:

```
(list 1 2 3) ⇔ (cons 1 (cons 2 (cons 3 empty)))
```

now we can do something similar for a `Nat` using **add1**:

```
3 ⇔ (+ 1 1 1) ⇔ (add1 (add1 (add1 0)))
```

Also,
On lists we used **rest** to remove one **cons**:

```
(rest (cons 1 (cons 2 (cons 3 empty)))) ⇔ (cons 2 (cons 3 empty))
```

With a `Nat` we can use **sub1** to remove one **add1**:

```
(sub1 (add1 (add1 (add1 0)))) ⇔ (add1 (add1 0)) ⇔ 2
```

Template for countdown functions on `Nat`

Let's rewrite the factorial function we wrote earlier, now using **`sub1`**:

```
;; (fact n) compute n!
;; fact: Nat -> Nat
;; Example:
(check-expect (fact 4) 24)

(define (fact n)
  (cond [(= n 0) 1]
        [else (* n (fact (sub1 n)))]))
```

This suggests a template to count down on natural numbers:

```
;; a template for countdown on Nat:
(define (countdown-template n)
  (cond [(zero? n) ... ]
        [else (... n ... (countdown-template (sub1 n)) ...)]))
```

Countdown function

The template for `Nat` closely matches that for lists.

```
;; a template for countdown on Nat:
(define (countdown-template n)
  (cond [(zero? n) ... ]
        [else (... n ... (countdown-template (sub1 n)) ...)]))
```

This function ties the two together:

```
;; (countdown n) return a list of the natural numbers from n down to zero.
;; countdown: Nat -> (listof Nat)
;; Examples:
(check-expect (countdown 3) (cons 3 (cons 2 (cons 1 (cons 0 empty)))))
(check-expect (countdown 5) (list 5 4 3 2 1 0))
```

Exercise: Complete `countdown`.

Stopping away from zero

```
;; (countdown n) return a list of the natural numbers from n down to zero.
;; countdown: Nat -> (listof Nat)
(define (countdown n)
  (cond [(zero? n) (cons 0 empty)]
        [else (cons n (countdown (sub1 n)))]
        ))
```

There's no reason we have to stop at zero!

```
;; (countdown-to n bottom) return a list of the natural numbers
;;   from n down to bottom.
;; countdown-to: Nat Nat -> (listof Nat)
;; Examples:
(check-expect (countdown-to 3 0) (cons 3 (cons 2 (cons 1 (cons 0 empty)))))
(check-expect (countdown-to 5 2) (list 5 4 3 2))
```

Exercise: Make a copy of your `countdown` function, then modify it to make `countdown-to`.

Counting up

Suppose I wanted to count up, to get `(list 2 3 4 5)`.
I *could* use `(countdown-to 5 2)`, then use the built-in **`reverse`** function:

`(reverse (countdown-to 5 2))` ⇒ `(list 2 3 4 5)`

But there is a better solution. *Do not* use **`reverse`** on any of your assignments or exams.
In `countdown-to` we used **`sub1`** to get the next number for the recursive call. To write `countup-to`, use **`add1`** instead:

```
;; (countup-to n top) return a list of the natural numbers from n up to top.
;; countup-to: Nat Nat -> (listof Nat)
;; Example:
(check-expect (countup-to 2 5) (list 2 3 4 5))

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

Templates for down and up:

```
;; a template for countdown-to on Nat:
(define (countdown-to-template n bottom)
  (cond [(= n bottom) ... ]
        [else (... n ...
                   (countdown-to-template (sub1 n) bottom) ...)]))

;; a template for countup-to on Nat:
(define (countup-to-template n top)
  (cond [(= n top) ... ]
        [else (... n ...
                   (countup-to-template (add1 n) top) ...)]))
```

Prefixes of a `Str`

I want to find **prefixes** of a `Str`. For example, `"abc"` has prefixes of `""`, `"a"`, `"ab"`, and `"abc"`.

I could write using a countdown template on the length of the string:

```
;; (prefixes-down s) return list of all prefixes of s, starting with longest.
;; prefixes: Str -> (listof Str)
;; Example:
(check-expect (prefixes-down "abc") (list "abc" "ab" "a" ""))

(define (prefixes-down s)
  (cond [(= 0 (string-length s)) (cons "" empty)]
        [else (cons s (prefixes-down
                       (substring s 0 (sub1 (string-length s)))))]))
```

But I want them in the other order.

Prefixes of a `Str`

```
;; a template for countdown-to on Nat:
(define (countdown-to-template n bottom)
  (cond [(= n bottom) ... ]
        [else (... n ...
                   (countdown-to-template (sub1 n) bottom) ...)]))

;; a template for countup-to on Nat:
(define (countup-to-template n top)
  (cond [(= n top) ... ]
        [else (... n ...
                   (countup-to-template (add1 n) top) ...)]))
```

Exercise: Use the `countup-to` template to write a function to list prefixes of a `Str`.

```
(check-expect (prefixes-up "abc" 0) (list "" "a" "ab" "abc"))
```

Summing Multiples

Exercise: Write a function `(sum-multiples n d1 d2)` that adds up all the numbers up to `n` that are multiples of `d1` or `d2`.
For example, `(sum-multiples 10 3 5) => 33`, the sum of 3, 5, 6, 9, and 10.

Exercise: Rewrite this function to create `(sum-multiples-list n D)`, that consumes a `(listof Nat)` representing the divisors to check.

```
(sum-multiples-list 10 (list 3 5)) => 33
(sum-multiples-list 10 (list 3 5 2)) => 47
```

Course Philosophy: Structural Recursion

In this course we have been attempting to consider only *structural recursion*. In structural recursion, the structure of the recursion matches the structure of the data.

- A list is defined as `'()` or **`cons`** of a value and a list; we recurse on them using **`rest`** until we reach `'()`.
- A `Nat` is defined as 0 or **`add1`** of a `Nat`; we recurse on them using **`sub1`** until we reach 0.

If we insist recursion must match the structure, can we count by numbers other than one? E.g. counting by twos: $\{0, 2, 4, 6, 8, \dots\}$?
Yes! We can make a recursive data definition for each such situation! For example:

```
;; An EvenNat is either:
;;   0 or
;;   (+ 2 n) where n is an EvenNat
```

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

Integer powers of 10

```
;; A Power10 is either:
;;   1 or
;;   (* 10 n) where n is a Power10
```

```
;; my-power-ten-fun: Power10 -> Any
(define (my-power-ten-fun n)
  (cond
    [(= 1 n) ... ]
    [else (... n ...
               (my-power-ten-fun (/ n 10))
               ...)]))
```

Exercise: *;; my-log10: Power10 -> Nat*

Use the `Power10` template to write a function that returns the $\log_{10}$ of a `Power10`.

Examples:

```
(my-log10 100) => 2
(my-log10 10000) => 4
```

Recursion in Recursion

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

A beautiful example is *insertion sort*.

Insertion Sort

A few notes:

- If I have a sorted list, I can make a larger sorted list by inserting another item after all the items smaller than it.
- An empty list is sorted.

If I want to sort `(list 5 7 9 2 1)`:

1. Somehow sort the **`rest`** of the list, giving `(list 1 2 7 9)`
2. Insert the **`first`** of the list in the right place:

   `(insert 5 (list 1 2 7 9))` $\Rightarrow$ `(list 1 2 5 7 9)`

Exercise: Write your own `insertion-sort` and `insert` functions.

```
;; (insertion-sort L) sort L, using insertion sort.
;; insertion-sort: (listof Num) -> (listof Num)
;; Examples:
(check-expect (insertion-sort (list 3 8 5 2 6)) (list 2 3 5 6 8))
```

Insertion Sort

Consider how to use the list template:

```
;; my-listof-X-fun: (listof X) -> Any
(define (my-listof-X-fun L)
  (cond [(empty? L) ... ]
        [else (... (first L) ... (my-listof-X-fun (rest L)) ...)]))
```

Strategy: sort the rest of the list, then insert the first item into the sorted rest.

```
;; (insertion-sort L) sort L, using insertion sort.
;; insertion-sort: (listof Num) -> (listof Num)
;; Examples:
(check-expect (insertion-sort (list 3 8 5 2 6)) (list 2 3 5 6 8))

(define (insertion-sort L)
  (cond [(empty? L) empty]
        [else (insert (first L)
                      (insertion-sort (rest L)))]))
```

Insertion Sort

All that remains to complete is `insert`.

```
;; (insert item L) insert item into L so the result remains sorted.
;; insert: Num (listof Num) -> (listof Num)
;; Requires: L is sorted.
;; Examples:
(check-expect (insert 5 (list  1 2 7 9)) (list 1 2 5 7 9))

(define (insert item L)
  (cond [(empty? L) (cons item empty)]
        [(> item (first L)) (cons (first L) (insert item (rest L)))]
        [else (cons item L)]))
```

Key thing with recursion: functions do what they are supposed to! If I'm writing a function that returns a sorted list, and it needs a smaller sorted list, it can use itself to make the smaller list.

Selection Sort

There are many ways to sort lists! For extra practice let's think about another one.

Strategy: find the smallest item in the list, and remove it from the whole list. Then construct a new list from the smallest item and the modified whole.[1]

```
(selection-sort (list 5 1 2 7 9))
```

Need: `(cons 1 (selection-sort (list (5 2 7 9))))`

One of our functions is built-in to Racket:

```
;; (remove item L) removes the first occurence of item from L
;; remove: Any (listof Any) -> (listof Any)
;; Example:
(check-expect (remove 4 (list 2 4 6 8 4)) (list 2 6 8 4))
```

[1]Note we are not recursing on **rest**. This is not structural recursion, so it is technically beyond the scope of CS 115, but still good practice.

Selection Sort

We previously wrote `list-max`. Writing `list-min` should be similar. Suggestion: use `(empty (rest L))` as a base case.

Exercise: Complete `(smallest L)`

```
;; (smallest L) return the smallest item in L
;; smallest: (listof Num) -> Num
;; Example:
(check-expect (smallest (list -3 5 -7 2 4)) -7)
```

Exercise: Complete `(selection-sort L)`.

```
;; (selection-sort L) return L, sorted in increasing order.
;; selection-sort: (listof Num) -> (listof Num)
;; Example:
(check-expect (selection-sort (list 2 4 6 8 4)) (list 2 4 4 6 8))
```

List Abbreviation

We are now permitted to use list abbreviation on assignments.
Please change your language level to *Beginning Student with List Abbreviation*.

Instead of writing a list `(cons 1 (cons 2 (cons 3 empty)))`, we will prefer to write `(list 1 2 3)`.

During recursion we will still use **`cons`** most of the time, otherwise we will get lists of lists.

Using lists to speed up computations

Suppose I have a series of numbers that I use frequently, but which take work to compute, such as the Catalan numbers (used in combinatorics; `https://oeis.org/A000108`):

$$C_n = \frac{\binom{2n}{n}}{n+1} \qquad C = [1, 1, 2, 5, 14, 42, \ldots]$$

You may assume I have defined the function which creates one:

```
;; (catalan n) return the n-th Catalan number.
;; catalan: Nat -> Nat
```

If I every time my program need one of these, it computes it, it may compute the same number many times, which takes time. Instead, I can calculate each just once, and save them in a list.

```
;; (catalans-interval bottom top) return all the catalan numbers
;;   starting at index bottom, and ending before index top.
;; catalans-interval: Nat Nat -> (listof Nat)
(define (catalans-interval bottom top)
  (cond [(= bottom top) empty]
        [else (cons (catalan bottom) (catalans-interval (add1 bottom) top))]))
```

Getting stuff out

We can make a list of numbers, but can we get them back out?

Exercise: Complete `n-th-item`.

```
;; (n-th-item n L) return the n-th item in L, where (first L) is the 0th.
;; n-th-item: Nat (listof Any) -> Any
;; Example:
(check-expect (n-th-item 0 (list 3 7 31 2047 8191)) 3)
(check-expect (n-th-item 3 (list 3 7 31 2047 8191)) 2047)
```

By creating a list to store a sequence of numbers, then extracting the *n*th item of the list, we can speed computations, sometimes significantly.

A few reminders about **`first`** and **`rest`**

Consider a few `(listof Nat)`:

- `(`**`first`**` (`**`list`**` 1 2 3))` ⇒ 1, which is a `Nat`.
- `(`**`rest`**` (`**`list`**` 1 2 3))` ⇒ `(`**`list`**` 2 3)`, which is a `(listof Nat)`.

- `(`**`first`**` (`**`list`**` 2 3))` ⇒ 2, which is a `Nat`.
- `(`**`rest`**` (`**`list`**` 2 3))` ⇒ `(`**`list`**` 3)`, which is a `(listof Nat)`.

- `(`**`first`**` (`**`list`**` 3))` ⇒ 3, which is a `Nat`.
- `(`**`rest`**` (`**`list`**` 3))` ⇒ `'()`, (the same as **`empty`**), which is a `(listof Nat)`.

Lists of lists and **cons**

Writing lists only using **cons** is messy enough as it is, but it's quite atrocious if the items in the lists are lists.

`(list (list 5 4 3) (list 2 3 1))` would be

```
(cons (cons 5 (cons 4 (cons 3 '()))) (cons (cons 2 (cons 3 (cons 1 '()))) '()))
```

I can't read that! "*Many* find the abbreviations more expressive and easier to use."

cons ⇔ list

Exercise: Read carefully, and re-write using **list**:

```
(cons (cons 'a (cons '() '()))
      (cons 'b (cons (cons 'c '())
                     (cons 'd '()))))
```

Check your answer in Racket with language level:
Beginning Student.

Exercise: Re-write using **cons**:

```
(list (list 'a '() 'c) (list))
```

Check your answer in Racket with language level:
Beginning Student with List Abbreviation.

Lists versus structures

Recall working with structures:

```
(define-struct polarcoord (r theta))
;; a Polarcoord is a (make-polarcoord Num Num)
;; Requires:
;;   r is the distance to the point. r > 0.
;;   theta is angle from the x-axis.
```

This implicitly made four functions: `make-polarcoord`, `polarcoord?`, `polarcoord-r`, and `polarcoord-theta`.

Lists versus structures

I could get the same functionality using lists instead of structures:

```
;; A PolarCoord is a (list Num Num)
;; Requires:
;;   the first item is the distance to the point. r > 0.
;;   the second item is the angle from the x-axis.

(define (make-polarcoord r theta) (list r theta))          ; fake constructor
(define (polarcoord? L) (and (list? L) (= 2 (length L)))) ; not great, but OK.
(define (polarcoord-r P) (first P))                        ; fake selector
(define (polarcoord-theta P) (second P))                   ; fake selector
```

...but it's usually better to use lists where they are good (variable size data!) and structures where they are good (fixed size data!).

Two-dimensional data

You may know how to compute binomial coefficients, used in combinatorics:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

If I have n items, this tells me how many ways there are to choose k of them. Requires: $k \leq n$. Suppose we want to save these, instead of recomputing as needed. How can we store the data? A list of lists!

```
(binomials 4) => (list
                   (list 1)            ; 0 choose 0
                   (list 1 1)          ; 1 choose 0, 1 choose 1
                   (list 1 2 1)        ; 2 choose 0, 2 choose 1, 2 choose 2
                   (list 1 3 3 1)      ; ...
                   (list 1 4 6 4 1))   ; ...
```

I can get one row out of this: `(n-th-item 4 binomials)` ⇒ `(list 1 4 6 4 1)`

...and an item out of that row: `(n-th-item 2 (n-th-item 4 binomials))` ⇒ 6

Creating two-dimensional data

How can I build a table like this?

```
(binomials 4) => (list
                  (list 1)              ; 0 choose 0
                  (list 1 1)            ; 1 choose 0, 1 choose 1
                  (list 1 2 1)          ; 2 choose 0, 2 choose 1, 2 choose 2
                  (list 1 3 3 1)        ; ...
                  (list 1 4 6 4 1))     ; ...
```

I will count up, twice, once inside the other. (See another example in the notes.)

```
;; (make-binomial-row r i) make the rest of the r-th row of the
;;   binomial table, starting from i.
;; make-binomial-row: Nat Nat -> (listof Nat)
;; Example:
(check-expect (make-binomial-row-from 4 0) (list 1 4 6 4 1))

(define (make-binomial-row-from r i)
  (cond [(> i r) empty]
        [else (cons (binomial r i) (make-binomial-row-from r (+ 1 i)))]))
```

Creating two-dimensional data

Since `make-binomial-row-from` makes one row of the table, now I just need to call it repeatedly, once for each row. I can do this with another count up recursion.

```
;; (binomial-rows low high) make all the rows of binomials from low to high.
;; binomial-rows: Nat Nat -> (listof (listof Nat))

(define (binomial-rows low high)
  (cond [(= low high) empty]
        [else (cons (make-binomial-row-from low 0)
                    (binomial-rows (+ 1 low) high))]))
```

Exercise: Create a function (and necessary helper functions) to create the times tables up to a given value. For example,

```
(times-tables 4) => (list (list 0 0 0 0)
                          (list 0 1 2 3)
                          (list 0 2 4 6)
                          (list 0 3 6 9))
```

Dictionaries

Suppose I want to keep track of students by ID number, to store, say, name and program. I could make a list, and put item *n* in the *n*th position in my list, using our `n-th-item` function to get the nth student.

```
(define-struct student (name programme))

(define students
  (list (make-student "Al Gore" 'government)
        (make-student "Barack Obama" 'law)
        (make-student "Ben Bernanke" 'economics)
        (make-student "Bill Gates" 'appliedmath)
        (make-student "Conan O'Brien" 'history))
  )
```

But there is no student 0, no student 1, no student 2, ... I could fill these in with **`empty`**, but then I have millions of empty elements just to store thousands of student records!

Dictionaries

A better way: make a list, where each item in the list is itself a list, containing two items: a key (ID number) and a value (information about students)

```
(define-struct student (name programme))

(define students
  (list (list 6938 (make-student "Al Gore" 'government))
        (list 7334 (make-student "Bill Gates" 'appliedmath))
        (list 7524 (make-student "Ben Bernanke" 'economics))
        (list 8535 (make-student "Conan O'Brien" 'history))
        (list 8838 (make-student "Barack Obama" 'law))))
```

(I could do this by adding another field to my student structure, but this way I could store any data type, or even a mix of types.)

Dictionaries

```
;; An association (As) is a (list Num Any) where
;;   the first item is the key
;;   the second item is the associated value.

;; An association list (AL) is a (listof As).
;;
;; That is, an AL is either
;;   empty or
;;   (cons asoc r) where asoc is an As and r is an AL.
;;
;; Requires: keys must be distinct.
```

Dictionaries

Exercise: Complete `lookup-al`.

```
;; (lookup-al key L) return false if key is not in L, or the value in L
;;   associated with key otherwise.
;; lookup-al: Num AL -> Any
;; Example:
(check-expect
 (lookup-al 1000 (list (list 42 "forty-two")
                       (list 1000 "square root of a million")))
 "square root of a million")
```

Exercise: `(define-struct association (key value))`
This would perhaps be clearer if each association was a `(make-structure Num Any)` instead of a list of length 2. Rewrite `lookup-al` to use this format.

Dictionaries

Exercise: Compete `add-al`.

```
;; (add L key value) return L, with (key value) added to it.
;;   if key already exists, overwrite value.
;; add: AL Num Any -> AL
;; Example:
(check-expect (add-al (list (list 42 "forty-two")
                            (list 1000 "square root of a million"))
                      17 "seventeen")
              (list (list 17 "seventeen")
                    (list 42 "forty-two")
                    (list 1000 "square root of a million")))
```

Try to write your own, but here is a solution `add-al` for comparison:

```
;; (add L key value) return L, with (key value) added to it.
;;   if key already exists, overwrite value.
;; add: AL Num Any -> AL

(define (add-al L key value)
  (cond [(empty? L) (cons (list key value) empty)]
        [(= key (first (first L))) ; overwrite value
         (cons (list key value) (rest L))]
        [(> key (first (first L))) ; seek through list to insert
         (cons (first L) (add-al (rest L) key value))]
        [else ; this is the place to insert. Don't use rest, keep whole AL.
         (cons (list key value) L)]))
```

Module summary

List abbreviation is permitted! Yay! Make sure you change your language level to *Beginning Student with List Abbreviation*.

Understand that it is possible to define `Nat` recursively. This allows us to make recursive functions on `Nat` while still using structurally recursive code.

Be comfortable writing recursive functions that count up and count down.

Understand the principle of insertion sort, and be able to write such functions.

Work with lists that contain lists, either fixed-size (such as association lists) or variable-sized (including tables of values such as binomial coefficients).

Before we begin the next module, please

- Read *How to Design Programs*, Section 17.