Lab 4: Natural Number Recursion  
*12:00 PM, Oct 1, 2017*

## Contents

1. Recursion on Natural Numbers  
2. `succ?` and `zero?` and teachpacks, oh my!  
3. Factorial Fun!  
4. Primitive Recursion  
5. Take and Drop  
6. Crossover

## Objectives

By the end of this lab, you will know:

- what natural numbers (really) are

By the end of this lab, you’ll be able to:

- follow the design recipe to write procedures that recur on natural numbers

## 1. Recursion on Natural Numbers

So far, we have written Racket procedures that recur on lists. Today, we are going to write a few more recursive procedures—again following the design recipe—but these procedures are going to recur on natural numbers.

**Natural numbers** are non-negative integers. But let’s describe them in a way that should look very familiar: a natural number is either:

- 0, or
- the **successor** of (i.e., 1 more than) another natural number
- Nothing else is a natural number.
If you’ve been paying attention during your data definitions as you wrote recursive programs, you should recognize this as being parallel to “an int list is either the empty list, or (cons n lst), where n is an int and lst is an int list. Nothing else is an int list.”

That data definition — a description of the structure of the data — led to a particular form in which we wrote all our procedures, following the rule that “the structure of the code matches the structure of the data.” We’re going to apply that same idea to recursive programs involving integers.

Note that the definition of natural numbers is recursive. It has a base case: the smallest natural number is 0. And it has another case, referred to as the inductive case; to create a natural number, we take the successor (i.e., add 1) to an existing natural number.

We can define a way to construct natural numbers that’s analogous to the way we make lists:

```scheme
;; Data Definition:
;; a natural number is either 0 or
;; the successor of a natural number
;; succ: int -> int
;; Input:
;; num, a nonnegative integer (i.e., a natural number)
;; Output:
;; another natural number, the successor of num.
;; nothing else is a natural number

(define (succ num)
  (+ 1 num))

(check-expect (succ 0) 1)
(check-expect (succ 1) 2)
(check-expect (succ 5) 6)
```

Let us also define `pred`, which computes the predecessor of a nonzero natural number.

```scheme
;; pred: int -> int
;; Input: num, a nonzero natural number
;; Output: another integer, the predecessor of num.

(define (pred num)
  (- 1 num))

(check-expect (pred 10) 9)
(check-expect (pred 1) 0)
```

Both of these procedures, as well as a few more, are defined for you in the CS17 Teachpack. Furthermore, `succ?`, `pred`, and `zero?` gives us a suite of tools that are analogous to those for lists:

```scheme
0 ← empty
zero? ← empty?
succ ← cons
succ? ← cons?
pred ← rest
n ← first
```
where the \( n \) in the last example is the natural number itself.

That means that recursive programs that operate on natural numbers have a structure that’s strongly comparable to the template for list-recursion:

```
(define (my-proc n)
  (cond
    [(zero? n) ...]
    [(succ? n) ... (my-proc (pred n)) ... n ... ]))
```

where the \( n \) in the last line is analogous to the \((\text{first lst})\) that appears in the list-recursion template.

We will now use the design recipe to construct a sum procedure.

1. Provide data definitions for all non-atomic data types. (If multiple procedures in a single file use the same non-atomic types, provide these data definitions once at the top of the file, not once-per-procedure).

```
;; Data Definition
;; a natural number is either
;; - zero
;; - the successor of a natural number
```

2. Provide examples of the data the procedure will process and produce.

```
;; Data Definition
;; a natural number is either
;; - zero
;; - the successor of a natural number

;; Examples of Data
0
(succ 0) ;; 1
(succ (succ 0)) ;; 2
```

3. Specify the procedure’s type signature, which describes the type of data the procedure consumes, and the type it produces.

```
;; Data Definition
;; a natural number is either
;; - zero
;; - the successor of a natural number

;; Examples of Data
0
(succ 0) ;; 1
(succ (succ 0)) ;; 2

;; summorial: int → int
```
4. Following the type signature, describe the procedure's call structure, i.e., give names to the procedure and its arguments. (This involves writing the start of a Racket program rather than writing a comment.)

```racket
;; Data Definition
;; a natural number is either
;;  - zero
;;  - the successor of a natural number

;; Examples of Data
0
(succ 0);; 1
(succ (succ 0));; 2

;; summorial: int → int
(define (summorial num)
  ...)
```

5. Write a specification for the procedure. That is, in words, not code, state the relationship between the procedure’s input and output (make sure to use the argument names you created in the call structure). This goes in a comment above the call structure.

```racket
;; Data Definition
;; a natural number is either
;;  - zero
;;  - the successor of a natural number

;; Examples of Data
0
(succ 0);; 1
(succ (succ 0));; 2

;; summorial: int → int

;; Input: a natural number, num
;; Output: the sum of all natural numbers from 0 to num
(define (summorial num)
  ...)
```

6. Provide test cases that exemplify the procedure’s operation. These tests must follow its call structure and satisfy its specification.

```racket
;; Data Definition
;; a natural number is either
;;  - zero
;;  - the successor of a natural number

;; Examples of Data
0
(succ 0);; 1
```
(succ (succ 0));; 2
;; summorial: int → int
;; Input: a natural number, num
;; Output: the sum of all natural numbers from 0 to num
(define (summorial num)
  ...)
;; Test cases
(check-attempt (summorial 2) 3)
(check-attempt (summorial 1) 1)
(check-attempt (summorial 0) 0)

Now, we are almost ready to write our procedure. But first, we must create a template:

(define (summorial num)
  (cond
    [(zero? num) 0]
    [(succ? num) ( ... num ... (summorial (pred num)) ... )]))

This template describes how most recursive procedures operate on natural numbers. Exactly as for lists, this template mimics the data definition by using a cond expression. The base case is: is the natural number zero?; and the inductive step is: is the natural number a succ?: i.e., the successor of another natural number.

The fact that the summorial procedure operates on a recursive data type suggests that summorial should itself be recursive. This means that the summorial of a natural number is going to be computed in terms of the summorial of a smaller natural number. Hence, we include in the template a recursive call—in this case, the application of the summorial procedure to (pred num).

To fill in our template, we must determine two things: what to do in the base case, and what to do in the inductive step. Like the length procedure, what to do in the base case is straightforward.

(define (summorial num)
  (cond
    [(zero? num) 0]
    [(succ? num) ( ... num ... (summorial (pred num)) ... )]))

To figure out what to do in the inductive step, let’s draw a couple of recursive diagrams. Based on the specification, we know what the original output and the recursive output should be, given the original input and the recursive input, respectively. What’s left is to figure out:

1. how to derive the recursive input from the original input
2. how to derive the original output from the recursive output

To help us infer the appropriate logic to accomplish the second step, let’s write down a couple of sample recursive diagrams:
• original input: 4
  – recursive input: 3
  – recursive output: 6

• original output: 10

How can we derive the original output 10 from the recursive output 6?

• original input: 3
  – recursive input: 2
  – recursive output: 3

• original output: 6

What about deriving the original output 6 from the recursive output 3?

We just add the original input 3 to the recursive output 3, and similarly, in the previous case we add the original input 4 to the recursive output 6.

More generally, to derive the procedure’s output from the recursive output, add the original input \texttt{num} to \texttt{(summorial (pred num))} as follows:

\begin{verbatim}
(define (summorial num)
  (cond
   [(zero? num) 0]
   [(succ? num) (+ num (summorial (pred num)))])
)
\end{verbatim}

And there you have it. A procedure to compute the summorial function. Here is our final product:

\begin{verbatim}
;; Data Definition
;; a natural number is either
;; - zero
;; - the successor of a natural number

;; Examples of the Data
0
(succ 0) ;; 1
(succ (succ 0)) ;; 2

;; summorial: int → int

;; Input: a natural number, num
;; Output: the sum of all natural numbers from 0 to num
(define (summorial num)
  (cond
   [(zero? num) 0]
   [(succ? num) (+ num (summorial (pred num)))])
)
\end{verbatim}
Test cases for summorial
(check-expect (summarial 2) 3)
(check-expect (summarial 1) 1)
(check-expect (summarial 0) 0)

If you type the code above into DrRacket’s definitions window and click Run, the output in the interactions window should be “All tests passed.”

2 succ? and zero? and teachpacks, oh my!

Here in CS0170, we’ve been using a no magic policy: every procedure you use, you’ll learn how to implement. So how do succ, pred, zero?, and succ? work?

succ, pred, zero? are Racket built-in procedures, but it’s easy to guess at how they would work:

```racket
;; succ : int -> int
;; Inputs: a natural number to increment
;; Output: the number incremented by 1
(define (succ x) (+ x 1))

;; pred : int -> int
;; Inputs: a number to decrement
;; Output: the number decremented by 1
(define (pred x) (- x 1))

;; zero? : int -> bool
;; Inputs: a natural number to check if zero
;; Output: true if the number is zero
;; false if the number isn't zero
(define (zero? x) (= x 0))
```

succ? is a function we provide in the teachpack, but it works similarly to zero?, with a few tweaks:

```racket
;; succ? : int -> boolean
;; Inputs: a natural number to check if an integer and positive
;; Output: true if the number is a positive integer
;; false if the number isn't an integer, or positive
(define (succ? x)
  (if (number? x)
      (and (integer? x) (> x 0))
      (error 'succ? "expects argument of type <real number>")))
```

While you don’t have to write these procedures yourself (since they’re given by Racket and the teachpack), now you know how they work.

\textsuperscript{1}Hint: it’s not magic
3 Factorial Fun!

Task: Ariel has five gadgets lined up and wants to see in which order they look best! To determine the best arrangement, she wants to figure out how many different ways she can order them in a line. Modify the summorial procedure to compute factorial instead. (A factorial is the product of the integers from 1 to the number in question, where 0! = 1.)

Hint: Pay close attention to what happens in the base case.

4 Primitive Recursion

Lilo is trying to teach Stitch how to add by counting on its fingers.

Task: Pretend that you can’t use + for addition. Write a procedure, plus, that takes as input two natural numbers and outputs their sum. Do not use +, -, *, or /.

Hint: Use the succ and pred procedures:

\[
\begin{align*}
(succ \ 17) & \Rightarrow 18 \\
(pred \ 18) & \Rightarrow 17
\end{align*}
\]

Now that Stitch has mastered addition, Lilo wants to show him how to multiply.

Task: Pretend that you can’t use * for multiplication. Write a procedure, times, that takes as input two natural numbers and outputs their product. As above, do not use +, -, *, or /.

Hint: Use the addition procedure, plus, that you just wrote.

| You’ve reached a checkpoint! Please call over a lab TA to review your work.

5 Take and Drop

Now let’s practice two argument recursion!

Task: Write the procedure \( \text{my-take} \ n \ \text{alod} \) that takes as input a natural number \( n \) and a list of datum \( \text{alod} \) such that \( n \) is at most the length of \( \text{alod} \), and outputs the list consisting of the first \( n \) elements of \( \text{alod} \), in the order in which they appear in \( \text{alod} \).

Examples:

\[
\begin{align*}
(\text{my-take} \ 3 \ (\text{list} \ "\text{Mickey}" \ "\text{Minnie}" \ "\text{Daisy}" \ "\text{Donald}" \ "\text{Pluto}" \ "\text{Goofy}")) & \Rightarrow (\text{list} \ "\text{Mickey}" \ "\text{Minnie}" \ "\text{Daisy"}) \\
(\text{my-take} \ 0 \ (\text{list} \ 0 \ 1 \ 2 \ 3 \ 4 \ 5)) & \Rightarrow \text{empty} \\
(\text{my-take} \ 5 \ (\text{list} \ "i" \ "\text{love}" \ "cs" \ "17" \ "!")) & \Rightarrow (\text{list} \ "i" \ "\text{love}" \ "cs" \ "17" \ "!")
\end{align*}
\]
Note: The procedure take is built-in to Racket’s list library. To use it, you must include the expression (require racket/list) in your code. Needless to say, you should not use take for your implementation of my-take, but you may use it on future assignments.

Task: Write a procedure (my-drop n alo1d) that takes as input a natural number n and a list of datum alo1d such that n is at most the length of alo1d, and outputs the list consisting of all but the first n elements of alo1d, in the order in which they appear in alo1d.

Example:

(my-drop 3 (list "Mickey" "Minnie" "Daisy" "Donald" "Pluto"))
=> (list "Donald" "Pluto")

(my-drop 6 (list 0 1 2 3 4 5))
=> empty

(my-drop 0 (list "i" "love" "cs17")
=> (list "i" "love" "cs17")

Note: The procedure drop is built-in to Racket’s list library. To use it, you must include the expression (require racket/list) in your code. As above, you should not use it on this task, but you may use it on future assignments.

6 Crossover

Task: Without using any helpers (e.g., take and drop), write the procedure (crossover alod1 alod2 num) which, given a natural number num and two lists of datum alo1d1 and alo1d2 of length at least num, outputs a list of datum containing the first num elements in alod1 followed by all but the first num elements in alod2.

Example:

(crossover (list 1 2 3 4 5 6) (list 60 40 20 0) 3)
=> (1 2 3 0)

Task: Write another procedure, crossover2, with the same specification as crossover, but which uses take and drop as helpers.

Hint: Your crossover2 procedure should be one line.

And by that we of course mean:

(define (crossover2 alod1 alod2 num)
  ***one line of code***)

and definitely not:

(define (crossover2 alod1 alod2 num) ***one line of code***)

9
Once a lab TA signs off on your work, you’ve finished the lab! Congratulations! Before you leave, make sure both partners have access to the code you’ve just written.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS17 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/cs017/feedback]