Lab 4: Natural Number Recursion
12:00 PM, Sep 30, 2019

Contents

1 The Principle of Maximal Laziness 1
2 Recursion on Natural Numbers 2
3 succ? and zero? and teachpacks, oh my! 8
4 Factorial Fun! 9
5 Primitive Recursion 10
6 Take and Drop 10
7 Crossover 11

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 The Principle of Maximal Laziness

One of the ways to describe recursion is as an implementation of the principle of maximal laziness. That is, recursive procedures delegate as much work as possible to the recursive calls. Here’s an example of how Old MacDonald’s farm might recursively compute (summorial 4).

Old MacDonald. (to his cow:) Compute the summorial of 4. E-I-E-I-O.
Cow (with a moo moo here and a moo moo there): Well ... let’s see. Is my number zero? No. So I can delegate the computation of the summorial of 3. Once I know the summorial for that, I can add back 4 to get the summorial of 4. I think I’ll pass off some of the work. (to pig:) Compute the summorial of 3. E-I-E-I-O.
Pig (with an oink oink here and an oink oink there): Is my number 0? No...I'll pass off some of the work. (to duck:) Compute the summorial of 2. E-I-E-I-O.

Duck (with a quack quack here and a quack quack there): Why do I have to do this? I don’t even do math! Is my number 0? No...I’ll pass off some of the work. (to horse:) Compute the summorial of 1. E-I-E-I-O.

Horse (with a neigh neigh here and a neigh neigh there): Duck, stop whining. Is my number 0? No...I’ll pass off some of the work. (to lamb:) Compute the summorial of 0. E-I-E-I-O.

Lamb (with a baa baa here and a baa baa there): Is my number 0? Hey, it is! (to horse:) My answer is 0. E-I-E-I-O.

Horse: I’m supposed to add 1 to that...0 plus 1 is 1. (to duck:) My answer is 1.

Duck: Me again?! I’m supposed to add 2 to that...2 plus 1 is 3. (to pig:) My answer is 3.

Pig: I’m supposed to add 3 to that...3 plus 3 is 6. (to cow:) My answer is 6.

Cow: I’m supposed to add 4 to that...4 plus 6 is 10. (to Old MacDonald:) My answer is 10.

Old MacDonald: Ah. The answer is 10.

Note that it is necessary that each farm member pass off only some of the work (smaller and smaller jobs) to the others. If everyone were as lazy as Old MacDonald, nothing would ever get done!

Old MacDonald (to cow:) Compute the summorial of 4.

Cow (to self:) Ugh! Too much work. I’ll delegate the job. (to pig:) Compute the summorial of 4.

Pig (to self:) Ugh! Too much work. I’ll delegate the job. (to Duck:) Compute the summorial of 4. . . .

It is also necessary that there exist a base case in which the recursion bottoms out. Otherwise, we are at risk of infinite regress.

Old MacDonald (to cow:) Compute the summorial of 4.

Cow (to self:) I can delegate the computation of the summorial of a smaller number...how about the number that is 1 smaller? Once I know the summorial of 3, I can add 4 to get the summorial of 4. (to pig:) Compute the summorial of 3.

Pig (to duck:) Compute the summorial of 2.

Duck (to horse:) Compute the summorial of 1.

Horse (to lamb:) Compute the summorial of 0.

Lamb (to chickens:) Compute the summorial of −1.

Chickens: (“Wait a minute here, I don’t see an end in sight.”)

Lamb: Says here in the script that you should ask someone for the summorial of a natural number that’s 1 smaller than your number.

Chickens: (“But −1 is not a natural number!”)

Lamb: Did I forget the base case? How silly of me. Do forgive me!

Who ever said there are no skits in CS 17?
2 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 “a num list is either the empty list, or (cons n lst), where n is an num 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:

```racket
;; Data Definition:
;; a natural number is either 0 or
;; the successor of a natural number
;; succ: num -> num
;; Input:
;; n, a nonnegative integer (i.e., a natural number)
;; Output:
;; another natural number, the successor of n.
;; nothing else is a natural number
(define (succ n)
  (+ 1 n))
```

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

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

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

\[
\begin{align*}
0 & \leftrightarrow \text{empty} \\
\text{zero?} & \leftrightarrow \text{empty?} \\
\text{succ} & \leftrightarrow \text{cons} \\
\text{succ?} & \leftrightarrow \text{cons?} \\
\text{pred} & \leftrightarrow \text{rest} \\
\text{n} & \leftrightarrow \text{first}
\end{align*}
\]

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:

\[
(\text{define } (\text{my-proc } n))
(\text{cond})
[[(\text{zero? } n) \:\ldots]]
[[(\text{succ? } n) \:\ldots (\text{my-proc } (\text{pred } n)) \:\ldots n \:\ldots]]
\]

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

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

Note: Although you no longer have to provide data definitions and example data for atomic data types, for this lab you will be mandated to provide data definitions and example data for natural numbers as demonstrated below in order to reinforce the concept that natural numbers are structurally very similar to lists.

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

```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
```

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

```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
```

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.

```
;; 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, n
;; Output: the sum of all natural numbers from 0 to n
(define (summorial n)
  ...
)
```

6. Provide **test cases** and that exemplify the procedure’s operation. These tests must follow its call structure and satisfy its specification. Also make sure to write **recursion diagrams** to determine the structure of your procedure's recursive call(s).

```
;; 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

;; Recursion diagrams
;; OI: 3
;;   RI: 2
;;   RO: 3
;; OO: 6

;; OI: 4
;;   RI: 3
;;   RO: 6
;; OO: 10

;; summorial: int → int

;; Input: a natural number, n
```
Now, we are almost ready to write our procedure. But first, we must create a template:

```
(define (summarial n)
  (cond
    [(zero? n) ... ]
    [(succ? n) ( ... n ... (summarial (pred n)) ... )]))
```

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

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 (summarial n)
  (cond
    [(zero? n) 0]
    [(succ? n) ( ... n ... (summarial (pred n)) ... )]))
```

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 to \((\text{sumsorial} (\text{pred } n))\) as follows:

\[
(\text{define} \ (\text{sumsorial} \ n))
\begin{cases}
  (\text{cond})
  & [(\text{zero?} \ n) \ 0] \\
  & [(\text{succ?} \ n) \ (+ \ n \ (\text{sumsorial} \ (\text{pred } n)))]
\end{cases}
\]

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

;; 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

;; sumsorial: num → num

;; Input: a natural number, n
;; Output: the sum of all natural numbers from 0 to n
(define sumsorial n)
(cond
  [(zero? n) 0] \\
  [(succ? n) (+ n (sumsorial (pred n)))]
)
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.”

3  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:

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

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

;;; zero? : num -> bool
;;; Inputs: a natural number, x, to check if zero
;;; Output: the boolean true if the number is zero, and
;;; the boolean 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:

;;; succ? : num -> boolean
;;; Inputs: a natural number, x, to check if an integer and positive
;;; Output: the boolean true if the number is a positive integer,
;;; and the boolean 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.

4 Factorial Fun!

Task: Old MacDonald has five farm tools 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. The notation for the factorial of a number \( x \) is \( x! \), and \( 0! = 1 \).)

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

5 Primitive Recursion

Peppa is trying to teach her friend 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:

(succ 17) => 18
(pred 18) => 17

Now that Peppa Pig has mastered addition, her friend wants to show her 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.

6 Take and Drop

Now let’s practice two argument recursion!

\(^1\)Hint: it’s not magic
Task: Write the procedure \texttt{(my-take n alod)} that takes as input a natural number \texttt{n} and a list of datum \texttt{alod} such that \texttt{n} is at most the length of \texttt{alod}, and outputs the list consisting of the first \texttt{n} elements of \texttt{alod}, in the order in which they appear in \texttt{alod}.

Examples:

\begin{verbatim}
(my-take 3 (list "Peter Rabbit" "Flopsy" "Mopsy" "Cottontail" "Mr. McGregor" "Benjamin Bunny"))
=> (list "Peter Rabbit" "Flopsy" "Mopsy")
\end{verbatim}

\begin{verbatim}
(my-take 0 (list 0 1 2 3 4 5))
=> empty
\end{verbatim}

\begin{verbatim}
(my-take 5 (list "i" "love" "cs" "17" "!"))
=> (list "i" "love" "cs" "17" "!")
\end{verbatim}

Note: The procedure \texttt{take} is built-in to Racket’s list library. To use it, you must include the expression \texttt{(require racket/list)} in your code. Needless to say, you should not use \texttt{take} for your implementation of \texttt{my-take}, but you may use it on future assignments.

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

Examples:

\begin{verbatim}
(my-drop 3 (list "corn" "wheat" "hay" "barley" "sorghum"))
=> (list "barley" "sorghum")
\end{verbatim}

\begin{verbatim}
(my-drop 6 (list 0 1 2 3 4 5))
=> empty
\end{verbatim}

\begin{verbatim}
(my-drop 0 (list "i" "love" "cs17")
=> (list "i" "love" "cs17")
\end{verbatim}

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

7 Crossover

Task: Without using any helpers (e.g., \texttt{take} and \texttt{drop}), write the procedure \texttt{(crossover alod1 alod2 n)} which, given a natural number \texttt{n} and two lists of datum \texttt{alon1} and \texttt{alon2} of length at least \texttt{n}, outputs a list of datum containing the first \texttt{n} elements in \texttt{alon1} followed by all but the first \texttt{n} elements in \texttt{alon2}.

Example:
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:

```scheme
(define (crossover2 alod1 alod2 n) ***one line of code***)
```

and definitely not:

```scheme
(define (crossover2 alod1 alod2 n) ***one line of code***)
```

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 CS 17 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/csci10170/feedback](http://cs.brown.edu/courses/csci10170/feedback).