Lecture 11: Two-Argument Recursion
10:00 AM, Sep 27, 2019

Contents

1 Warmup ................................. 1
   1.1 Natural Number Recursion .......................................... 2
2 Bignum ................................ 3
3 Two-Argument Recursion ............... 3
   3.1 Zip ........................................ 5
   3.2 List-longer? ..................................... 6
4 List Reversal ............................ 7
5 Summary .................................. 10
   5.1 Ideas ....................................... 10
   5.2 New tools ..................................... 10

Announcements

- You are required to attend lab each week, and attend your lab section. If you do not go to the lab you are assigned to, you will not get credit, even if you later go to the correct lab. This is because we have limited resources in terms of computers and TA staff.

Objectives

By the end of this lecture, you will be able to:

- write several kinds of two-argument recursive procedures

1 Warmup

Class began with the idea that almost all today’s procedures would operate on things of type (int list), and so presented the Data Description for int lists:

;; An int list is either empty, or (cons item lst),
;; where item is an int and lst is an int list.
Spike had students quickly write two non-recursive programs. Second which returns the second item of a list. The code is as follows:

```scheme
;; return the second item in an a list
;; of length at least two.
;; second: 'a list -> 'a
;;
;; input: alod, a list containing at least two items
;; output: the second item in alod
(define (second alod) (first (rest alod)))
```

When we try to run this code in racket it fails because second is actually a built in that returns the second element of a list, there are similar functions like third, fourth...

Now we are going to quickly write swap which swaps the items of a list with two elements. Here is the code:

```scheme
(swap (list 1 2)) => (list 2 1)
;; Swap the items in a list of length two
;; swap: 'a list -> 'a list
;; input:
;;   alod, a list containing exactly two items
;; output: a list with the same two elements in the opposite order
(define (swap alod) (list (second alod) (first (rest alod))))
```

Note that from now on you can use list instead of cons, this will make writing check-expects easier.

### 1.1 Natural Number Recursion

Natural number recursion is similar to recursion on one list in that there are two cases that lead to a template. A natural number is either zero or greater than zero. We are going to write an example program called zlist, that takes a natural number and returns a list with that many number of zeros. First we have the template:

```scheme
;; zlist : num -> num list
;;
;; NB: a natural number is either
;;   zero
;; the successor of (i.e., one greater than) a natural number
;; nothing else is a natural number
;; input: n, a natural number
;; output: a num list containing n copies of the number 0
(define (zlist n)
```
Then we can fill it in.

;; zlist : num -> num list
;;; NB: a natural number is either
;;; zero
;;; the successor of (i.e., one greater than) a natural number
;;; nothing else is a natural number
;;; input: n, a natural number
;;; output: a num list containing n copies of the number 0
(define (zlist n)
  (cond
   [(zero? n) empty]
   [(> n 0) (cons 0 (zlist (- n 1)))]))

There are two procedures that we provide pred and succ? that will be useful with natural number recursion. pred gives you the natural number just before n (i.e., n-1), the predecessor. Succ? tests whether a natural number has a predecessor, i.e., whether it’s a successor (i.e., nonzero) Template ends up looking more like the one for lists:

(define (proc n) (cond [(zero? n) empty] [(succ? n) (... (proc...(pred n)...))]))

2 Bignum

Our first project, Bignum has been released! Projects are basically glorified homeworks in that they have bigger tasks that take longer and you work in pairs. Keep in mind that you can only work with the same partner ONCE. To help keep you on track, we have design checks where you will talk through the design for your program with a TA and ensure you aren’t going down a rabbit hole. Get started early, read the handout now! Your main task in bignum is to write a recursive procedure that adds two numbers.

3 Two-Argument Recursion

We previously wrote improve, which replaced every integer in an int list with the number 17:

(define (improve aloi)
  (cond
   [(empty? aloi) empty]
   [(cons? aloi) (cons 17 (improve (rest aloi)))]))

Now let’s look at a different version, better, which replaces items in the list not with 17, but with any number you choose. So better takes two arguments, a list and a replacement integer.
This procedure will be recursive, and we can write a recursive diagram for it.

Orig. in.: (list 3 7 10), 14  
Rec. In: (list 7 10), 14  
Rec. out: (list 14 14)  
Orig out: (list 14 14 14)

Just as in all the procedures we’ve seen so far, the recursive input uses the rest of the original input. But what about the number? Should we change that, too? Nah. We want to replace everything in the tail of the list with 14 as well. So we write:

Orig. in.: (list 3 7 10), 14  
Rec. In: (list 7 10), 14  
Rec. out: (list 14 14)  
cons on "num" to the recursive output, perhaps?  
Orig out: (list 14 14 14)

And with that (and perhaps a second or third example to build confidence) we can write the program:

```scheme
(define (better num aloi)
  (cond
    [(empty? aloi) empty]
    [(cons? aloi) (cons num (better (rest aloi)))]))
```

More generally, if we have a recursive procedure that consumes a natural number (rather than an arbitrary integer) and a list, we might do one of two things to simplify the input: we might reduce the natural number (eventually reaching zero) or we might take the rest of the list (eventually reaching an empty list). Or we might do both. Or a couple of other possibilities, which we’ll encounter as we go along. Some of the other possibilities include:

- reducing natnum by 1, keep the list the same; base case is when natnum is zero
- reduce natnum by 1, take "rest" of the list (i.e., reduce both)
- reduce natnum by 1, somehow enlarge the list

This weekend, you’ll be doing natural number recursion during lab, so you’ll be more familiar with it as you think about combining natural number and list recursion.
Here’s one example of a “natural number and int list” recursion example: finding the $k$th item in an int list (that contains at least $k$ items). The secret, which you could discover by writing out a few recursive diagrams, is that the way to find the 4th item in a list is to find the third item in the rest of the list, which is the 2nd item in the rest of that list, which is the first item in the rest of that list. In this case, the recursive call involves simplifying both the natural number and the list. The code’s still a little messy, because we only operate on nonempty lists, and only operate on natural numbers larger than zero, so I won’t write out the details. We’ll return to this later.

But there are some generalities to observe:

- When we have more arguments, the choice of recursive input gets more complex.
- When we have more arguments, the main cond in the procedure may involve more cases — the structure of each argument ends up being involved.

We’ll see our first two-list-arg recursion shortly.

Now we are going to broaden our horizons: we are going to write procedures that consume instances of two recursive data types. While there are many forms of this, we’re going to focus on two specifically:

For two list arguments, we have some options:

- One list shortens, other remains unchanged
- Both lists shorten

### 3.1 Zip

We’re going to write a procedure `zip` that takes in two lists of the same length, and produces another list which consists of pairs (two item lists) of corresponding elements in the input lists. Here are a few examples:

```scheme
(check-expect (zip (list 1 2 5)    (list 1 3 5))
             (list (list 1 1)    (list 2 3)    (list 5 5)))
(check-expect (zip empty empty) empty)
(check-expect (zip (list 4) (list 4)) (list (list 4 4)))
```

Now that we know what’s supposed to happen, where should we start? How about our trusty recursive diagram:

Orig. in.: (list 1 2 5) (list 1 3 5)
Rec. in.: ???
Rec. out.: ...

...  
Orig out.: (list (list 1 1) (list 2 3) (list 5 5))
Hm... On which input(s) should we recur? As mentioned, there are a few different options, but for our purposes we are going to recur on both lists; otherwise, the recursive input doesn’t meet our input specification (two lists of equal length). With this in mind, let’s finish our diagram.

Original inputs: (list 1 2 5) (list 1 3 5)
Recursion inputs: (list 2 5) (list 3 5)
Recursive result: (list (list 2 3)
                  (list 5 5)

Cons a pair onto the recursive result
Pair contains "first" of each of the input lists

Original result: (list (list 1 1) (list 2 3) (list 5 5))

Now with our completed recursive diagram in tow, we can write the program, keeping in mind that our input is two lists that can each be either empty or cons.

; zip: (num list) * (num list) -> (num list) list
; Input: lists alon and alon2, of the same length
; Output: a list consisting of "pairs" (two item lists), the first output pair containing the first item in each list; the second output pair containing the second item for each list, and so on.
(define (zip alon alon2)
  (cond
   [(and (empty? alon) (empty? alon2)) empty]
   [(and (cons? alon) (cons? alon2))
     (cons (list (first alon) (first alon2))
           (zip (rest alon) (rest alon2)))]))

(check-expect (zip (list 1 2 5)
                   (list 1 3 5))
              (list (list 1 1)
                    (list 2 3)
                    (list 5 5)))
(check-expect (zip empty empty) empty)
(check-expect (zip (list 4) (list 4)) (list (list 4 4)))

Note: Within our cond, we could also check the cases where one list is empty and the other is cons (when aloi is empty and aloi2 is cons, and vice versa). However, since we are explicitly working with equal-length lists, these cases never occur and are unnecessary for this procedure.

Great! Let’s try our hand at another example.

### 3.2 List-longer?

We’re going to write a procedure that takes in two lists and produces true if the first list is longer than the second, and false otherwise. While a “cheap” answer may be using the length built-in, and writing (> (length aloi1)(length aloi2)), we’re going to write a recursive procedure. To start, we’ll use a recursive diagram!
original input: (list 1 2 3) (list 4 5)
  recursive input: (list 2 3) (list 5)
  recursive output: true
original output: true

Let’s do another example to get an idea of the false case:

original input: (list 8) (list 8 3)
  recursive input: empty (list 3)
  recursive output: false
original output: false

It seems pretty clear. If both lists are empty, we should return false as the first list is not longer than the second. If the first list is empty, and the second is not, we should return false as the first list is not longer than the second. If the first list is not empty, and the second list is empty, we should return true. Finally, in the event that both lists have contents, we should recur, removing the first item of each list. With this logic in mind, let’s write the procedure!

```
;; Data definition:
;; (int list):
;;  - empty
;;  - (cons int (int list))
;;
;; list-longer?: (int list) * (int list) -> bool
;; input: aloi1, a list of integers
;;    aloi2, another list of integers
;; output: a boolean that is true if the first list is longer than the second list; false otherwise.
(define (list-longer? aloi1 aloi2)
  (cond
    [(and (empty? aloi1) (empty? aloi2)) false]
    [(and (empty? aloi1) (cons? aloi2)) false]
    [(and (cons? aloi1) (empty? aloi2)) true]
    [(and (cons? aloi1) (cons? aloi2))
      (list-longer? (rest aloi1) (rest aloi2))])))
```

(check-expect (list-longer? (list 1 2 3) (list 2)) (length (list 1 2 3)) (length (list 2)))
(check-expect (list-longer? (list 1) empty) true)
(check-expect (list-longer? empty (list 1)) false)
(check-expect (list-longer? empty empty) false)
4 List Reversal

To illustrate the value of helpers (and, again, the trick of simplifying a problem by solving a more complicated problem!), let’s reverse a list. You’ll also see that the types of two-argument recursion we defined above are not complete.

We’ll start with a recursive diagram:

- original input: (list 1 3 4)
  - recursive input: (list 3 4)
  - recursive output: (list 4 3)
- original output: (list 4 3 1)

It sure seems as if reversing a nonempty list means reversing the rest of the list and then adding something on to the end, which we’re generally avoiding (for reasons that’ll become evident very soon). As we get down to a zero- or one-element list, reversing gets easy, but the more general case is a mess. For the empty list, the answer’s just empty. For a list (list v), the result is (list v). But beyond that ... it’s a mess. Here’s where we use the “bigger helper” trick. Note: we have to call our procedure my-reverse because Racket has reverse built into it.

We’re going to define reverse-with-tail, which takes two arguments, alod and tail (another list of data), and produces the reverse of alod with tail appended to it.

> (reverse-with-tail (list 1 2 3) (list 4 5))
(list 3 2 1 4 5)
> (reverse-with-tail (list 1 2 3) empty)
(list 3 2 1)

It’s pretty clear, if you’ve got reverse-with-tail, how to write my-reverse:

;; Data definition:
;; (`a list):
;; - empty
;; - (cons `a (`a list))
;; my-reverse: (`a list) -> (`a list)
;; input: alod, a list of data
;; output: a new list, consisting of all element of alod, in reverse order

(define (my-reverse alod)
  (reverse-with-tail alod empty))
(check-expect (my-reverse empty) empty)
(check-expect (my-reverse (list 1 2)) (list 2 1))
(check-expect (my-reverse (list 1 2 3)) (list 3 2 1))

So how do we write reverse-with-tail?
Let’s look at some recursive cases. To do so, we have to decide how we’re going to recur. Let’s try this example.

- original input: (list 1 2) (list 17 18)
  - recursive input: (list 2) (list 17 18)
  - recursive output: (list 2 1 17 18)
- original output: (list 2 1 17 18)

That looks horrible. We need to perform surgery to stick that “1” in there. How about this:

- original input: (list 1 2) (list 17 18)
  - recursive input: (list 2) (list 1 17 18)
  - recursive output: (list 2 1 17 18)
- original output: (list 2 1 17 18)

Wow! By changing both lists— one getting simpler, the other getting more complicated — we’ve got ourselves a solution!

```scheme
;; Data definition:
;; (’a list):
;; - empty
;; - (cons ’a (’a list))
;; reverse-with-tail: (’a list) * (’a list)-> (’a list)
;; input: alod, a list of data
;; start, another list of data of the same type as alod
;; output: a new list, consisting of all element of alod, in reverse order,
;; followed by those of start, in their original order.

(define (reverse-with-tail alod start)
  (cond
   [(empty? alod) start]
   [(cons? alod) (reverse-with-tail (rest alod)
     (cons (first alod) start))])

(check-expect (reverse-with-tail empty empty) empty)
(check-expect (reverse-with-tail (list 1 2) empty) (list 2 1))
(check-expect (reverse-with-tail (list 1 2 3) (list 4 5)) (list 3 2 1 4 5))
```

Done! Woo hoo! The overall result is
;; - (cons `a (`a list))
;; reverse-with-tail: (`a list) (`a list) -> (`a list)
;; input: alod, a list of data
;; start, another list of data of the same type as alod
;; output: a new list, consisting of all element of alod, in reverse order,
;; followed by those of start, in their original order.

(define (reverse-with-tail alod start)
  (cond
   [(empty? alod) start]
   [(cons? alod) (reverse-with-tail (rest alod) (cons (first alod) start))]))

;; my-reverse: (`a list) -> (`a list)
;; input: alod, a list of data
;; output: a new list, consisting of all element of alod, in reverse order

(define (my-reverse alod)
  (reverse-with-tail alod empty))

(check-expect (my-reverse empty) empty)
(check-expect (my-reverse (list 1 2)) (list 2 1))
(check-expect (my-reverse (list 1 2 3)) (list 3 2 1))

Now you might ask yourself a couple of questions here:

- I see that it solves the problem, but what on earth motivated you to write reverse-with-tail, Spike?

- How many arguments can a helper-procedure like this have? Could it have 3, or 4, or twelve?

The answer to the second question is pretty much “two or three”. There’s a tendency to use these things in a way that’s sloppy and that makes it hard to express what your procedure actually does. If you can’t write a crisp specification for the helper, you’re probably doing something wrong.

As for the first question, the answer is that in looking at the recursive diagram, I saw that right from the start (unlike in the length procedure), I already had part of the answer. In reversing (list 1 2 3), the first of the input list was in fact, something that needed to be right at the start of building up an answer. That meant that I wanted to have a gradually accumulating answer, and the “tail” was a way to keep track of that as I went deeper and deeper into the recursion.

Perhaps a better name for our helper would be reverse-with-partial-result.

Notice that in this two-argument recursion, while one argument shrinks, the other is growing at every step of the recursion. That’s in contrast to zip, where both grew smaller.

**Question:** Are there any other possibilities, like one argument staying the same and the other getting more complicated, or both getting more complicated?

**Answer:** In those cases, repeated recursive invocations of the procedure process more and more complicated data, and never terminate: the (empty? lst) clause never gets evaluated, and the other clause always ends up making a recursive call. (There’s one more case: both arguments stay
the same size; that doesn’t work either.) It’s essential, in writing recursive procedures in CS17, that at least one argument gets simpler in each call, or proving that the procedure terminates becomes infeasible.

5 Summary

5.1 Ideas

- We’ve seen many procedures that make use of two-argument recursion— both with one list and one atomic argument and with two list arguments. While there are many variations, one of the arguments must always shrink.

5.2 New tools

- pred which takes a natural number and returns that number -1.
- succ? which takes a natural number, n, and returns true if n is nonzero, false otherwise.
- second, third which takes a list and return the second or third item in the list respectively.
- reverse which takes a list and outputs the a list with the elements from the input list in reverse order.
- You can now write lists like this list(1 2 3) instead of (cons 1(cons 2(cons 3 empty)))

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/csci0170/feedback