

CS51 Assignment 8: Lazy Sunday

Due: Tuesday, April 21st, 2009, 11:59PM

Total Points: 55 (including 5 style points)

1 Lazy Trees

(23 points total)

Exercise 1.

23 points

In lecture, you learned about creating lazy lists (also called streams) using `define-syntax-rule` to delay evaluation. Here, we will ask you to write a lazy tree using a similar technique.

To help you, we've defined a struct `lbranch`, which contains a value (`val`), a left child (`left`) and a right child (`right`). For testing, we've also provided the function (`print-n-levels n sometree`), which prints the first `n` levels of `sometree` on a single line.

(a) [10 points]

Write the functions `make-lt`, `lt-left`, `lt-right`. `lt-left` and `lt-right` should return the left and right children, respectively, of a lazy tree. `(make-lt val left right)` should return an `lbranch` struct with `lbranch-val` set to `val`, and with `lbranch-left` and `lbranch-right` set to *unevaluated* functions wrapping `left` and `right`, respectively. For example, consider the following definition:

```
> (define tree1 (make-lt 1
                        tree1
                        tree1))

> tree1
#<lbranch>
> (lt-val tree1)
1
> (lt-left tree1)
#<lbranch>
> (lbranch-left tree1)
#<procedure:f>
> (lt-val (lt-left tree1))
1
```

```
> (print-n-levels 1 tree1)
1(1.1)
```

This represents a tree with every `val` set to 1. We have included the call to `lbranch-left` only for clarification. You should instead call `lt-left` whenever possible.

(b) [5 points] Write `ltmap`, a map function for lazy trees. This takes a function and an `lbranch` representing the root of a lazy tree, and returns a new lazy tree with the function applied to every element of the original. For example:

```
> (define tree2 (ltmap incr tree1))
> (print-n-levels 1 tree2)
2(2.2)
```

(c) [5 points] As you may have noticed, we made our `lbranch` struct mutable. We can call `(set-lbranch-left! parenttree childtree)` to set the left child of `parenttree` to `childtree`. `set-lbranch-right!` and `set-lbranch-val!` work similarly.

Mutable branches allow us to implement *memoization* to avoid the repeated computation of values, in a similar manner to the lazy list memoization described in lecture (See lecture notes 16 for more details).

Implement memoization on your lazy tree implementation. This requires modifying your `lt-left` and `lt-right` functions to store the results (that is, `((lbranch-left))` and `((lbranch-right))`, respectively) so that they are not re-computed on subsequent function calls. Depending on how you do the memoization, you may have to change some or all of `make-lt`, `lt-left`, and `lt-right`. So that we can see both your original and the memoized versions, create new functions `lt-leftm`, `lt-rightm`, and `make-ltm`, for the memoized versions.

Note that the memoized and non-memoized functions should be able to be used interchangeably. In other words, your memoized functions `lt-leftm` and `lt-rightm` should still construct valid lazy trees that could be accessed using `lt-left` and `lt-right`. This is similar to the lazy list implementation from lecture.

We've provided you with the function `print-n-levelsm`, which prints a list to depth `n` using the memoized `lt-leftm` and `lt-rightm`.

(d) [3 points] Write a function `(random-tree min max)` that generates a lazy binary tree with random values between `min` and `max`. In other words, the root of the tree should be a random value between `min` and `max`, the left tree should contain only values between `min` and the root value, and the right tree should contain only values between the root value and `max`.

To help you, we've defined the function `(random-float min max)`, which, as the name suggests, generates a random float between `min` and `max`. You may find the `my-round` function useful for making your trees a bit prettier.

Please write *two versions* of `random-tree`: one using your memoized lazy tree (`random-treem`), the other using the non-memoized lazy tree (`random-tree`). When testing, make sure to use

the appropriate print functions for each type of tree!

What, if any, are the differences between the two random trees you wrote? Please describe in one or two sentences, and copy/paste an example of each type of tree printed to depth of 2. (Hint: print the same tree more than once.)

2 Streams

(26 points total)

In this problem set, you'll get to explore interesting transformations on streams, ultimately using streams to find the solution to the equation

$$a^b = 165781607705330253724331770932092584302029519661579241709498594592363009092524474983564768737616980532466764253279737497762471076412550111177747942850514333041819023812651880087656665882027607$$

where a is prime and b is an integer. Now, you *could* just factor this, but that wouldn't be as much fun :).

We've defined given you a version of `lcons`, `lcar`, `lcdr` as well as a handful of useful functions for working with and creating streams. Most of these functions you've seen in lecture, but (re)familiarize yourself with them anyway.

2.1 Warm Up

Exercise 2.

6 points

(a) [4 points] Write the function, `first-n`, which takes a stream and returns a list of the first n elements of the stream. Make sure that your function is $O(n)$.

(b) [2 points] Write the function `reverse-first-n`, which should return the first n elements of the stream, but in reverse order. Again, make sure that your function is $O(n)$.

2.2 Don't Cross the Streams!

In lecture you saw that you can *zip* two streams a and b together so that the first element of a is combined with the first element of b , the second element of a is combined with the second element of b , etc. This is a simple way to combine streams. However, can also combine the first element of a with *every* element of b , the second element of a with *every* element of b , etc. This approach isn't quite as simple – we can't just start out by go through all the pairs with the first element of a because there are infinitely many of them! Fortunately, there is a clever trick we can use. Let's say the two streams we're combining are the alphabet (pretend that it is infinitely long for the moment). If we write the streams as the sides of

a table, we can think of every pair of elements as having “coordinates” in the grid, where those coordinates are really two elements in the streams. We can then start in the so-called “top-left” corner and visit each point in the grid by moving across increasing large diagonals. This is best expressed with a picture; we’ll make do with ASCII art.

```

      a   b   c   d   e   f   ...
a   1   3   6   10  15  22
      /   /   /   /   /
b   2   5   9   14  21
      /   /   /   /
c   4   8   13  20
      /   /   /
d   7   12  19
      /   /
e   11  18
      /
f   17
.
.
.

```

The numbers represent the order in which our new stream will list the pairs of elements from the old streams. In other words, the first 5 elements of our new stream will be $\{(a, a), (b, a), (a, b), (c, a), (b, b), \dots\}$. If we write out the x and y axes as the natural numbers, we notice that the numbers from the y axis are ordered $0, 1, 0, 2, 1, 0, 3, 2, 1, 0, 4, 3, \dots$ and the numbers from the right axis are ordered $0, 0, 1, 0, 1, 2, 0, 1, 2, 3, 0, 1, \dots$. So, we generate, using the functions you wrote above, streams of these forms and then zip them together.

Exercise 3.

6 points

This gives us `cross-streams`¹, a function that takes a function and two streams, and returns a single stream that combines every pair from the two streams. The function argument should take two arguments (elements from each of the two streams) and return a value that goes in the new stream – this is how the elements are “combined”, just like the function passed to `lzip`. We’ve given you the code for `cross-streams`.

Now let’s have some fun with it.

(a) [2 points] Define a stream, `rationals`, that contains every non-negative rational number (a rational number is just a number that can be written as the ratio of two integers). It’s okay if the stream contains numbers more than once. Now you have a stream that contains *every* rational number!

¹<http://www.youtube.com/watch?v=8jJ2WnRjzWs#t=0m58s>

(b) [2 points] Define a stream `2-coordinates` that contains all pairs of non-negative integer coordinates in a two-dimensional table. The coordinates should be represented as lists with two elements.

(c) [2 points] Define a stream `3-coordinates` that contains all triples of non-negative integer coordinates in a three-dimensional table. Again, the triples should be lists with three elements.

2.3 That Ridiculously Large Number

As alluded to before, the number in the first part of the section can be written as a^b where a is prime and b is an integer. Now you'll get to find a and b using streams.

Exercise 4.

14 points

(a) [4 points] Write a function `powers` which takes an argument x and returns a stream of powers of x , starting with 1. For example, `(powers 2)` should be $1, 2, 4, 8, \dots$. *Hint: Take a look at the way `nats` was defined in lecture.*

(b) [4 points] Write a function that takes a stream of streams (whoa!), and two integers x and y , that returns the x th element of the y th stream. We've given you a couple two dimensional streams to work with.

(c) [6 points] Write a function `2d-stream->stream` that takes a stream of streams and returns a stream of every element of every stream. This should not be very much code! Think of each stream as a row in a table and use a few of the functions above. Think for a second about what this implies. You can take all the elements in an infinite number of infinite streams and enumerate them!

(d) [0 points] (Not for credit) Finally, find a and b in the expression $a^b = \text{large-number}$ where a is prime, b is an integer, `large-number` is the number above (defined in the Scheme file).

3 Finishing Up

(1 point total)

Exercise 5.

1 point

(a) [1 point]

Please tell us how much time you spent on this problem set. You may use this space to add any thoughts or questions you want to send to your TF as well, if you so desire.