15-150 Fall 2013
Homework 09

Out: 30 October 2013
Due: 5 November 2013, 23:59 EST

1 Introduction

This homework will build your experience with functors and sequences. As a reminder, functors allow us to produce more extensible code with less redundancy. The first half of this homework will guide you through the implementation of an extensible serialization framework. The second half of this homework stresses analysis of operations on sequences, and how they improve upon those on lists. As you will see in lecture, sequences provide similar functionality to lists, but with greater potential for parallelization.

1.1 Getting The Homework Assignment

The starter files for the homework assignment have been distributed through our `git` repository as usual.

1.2 Submitting The Homework Assignment

Submissions will be handled through Autolab, at

`https://autolab.cs.cmu.edu`

In preparation for submission, your `hw/09` directory should contain a file named exactly `hw09.pdf` containing your written solutions to the homework.

To submit your solutions, run `make` from the `hw/09` directory (that contains a `code` folder and a file `hw09.pdf`). This should produce a file `hw09.tar`, containing the files that should be handed in for this homework assignment. Open the Autolab web site, find the page for this assignment, and submit your `hw09.tar` file via the "Handin your work" link.

The Autolab handin script does some basic checks on your submission: making sure that the file names are correct; making sure that no files are missing; making sure that your PDF is valid; making sure that your code compiles cleanly. Note that the handin script is *not* a grading script—a timely submission that passes the handin script will be graded, but will not necessarily receive full credit. You can view the results of the handin script by clicking the number (usually either 0.0 or 1.0) corresponding to the “check” section of your latest handin on the “Handin History” page. If this number is 0.0, your submission failed the check script; if it is 1.0, it passed.

Remember that your written solutions must be submitted in PDF format—we do not accept MS Word files or other formats.

Your `marshal.sml` and `sequences.sml` files must contain all the code that you want to have graded for this assignment, and must compile cleanly. If you have a function that happens to be named the same as one of the required functions but does not have the required type, it will not be graded.

1.3 Due Date

This assignment is due on 5 November 2013, 23:59 EST. Remember that you may use a maximum of one late day per assignment, and that you are allowed a total of three late days for the semester.

1.4 Methodology

You must use the five step methodology discussed in class for writing functions, for **every** function you write in this assignment. Recall the five step methodology:

1. In the first line of comments, write the name and type of the function.
2. In the second line of comments, specify via a `REQUIRES` clause any assumptions about the arguments passed to the function.
3. In the third line of comments, specify via an `ENSURES` clause what the function computes (what it returns).
4. Implement the function.
5. Provide testcases, generally in the format

```
val <return value> = <function> <argument value>
```

For example, for the factorial function presented in lecture:

```
(*  fact : int -> int
 *  REQUIRES:  n >= 0
 *  ENSURES: fact(n) ==> n!
*)

fun fact (0 : int) : int = 1
  | fact (n : int) : int = n * fact(n-1)

(* Tests: *)

val 1 = fact 0
val 720 = fact 6
```

1.5 The SML/NJ Build System

We will be using several SML files in this assignment. In order to avoid tedious and error-prone sequences of `use` commands, the authors of the SML/NJ compiler wrote a program that will load and compile programs whose file names are given in a text file. The structure `CM` has a function

```
val make: string -> unit
```

`make` reads a file usually named `sources.cm` with the following form:

```
Group is

$/basis.sml
file1.sml
file2.sml
file3.sml
...
```

Loading your code using the REPL is simple. Launch SML in the directory containing your work, and then:

```
$ sml
Standard ML of New Jersey v110.69 [built: Wed Apr 29 12:25:34 2009]
- CM.make "sources.cm";
[autoloading]
[library $smlnj/cm/cm.cm is stable]
[library $smlnj/internal/cm-sig-lib.cm is stable]
...
```

Simply call

```
CM.make "sources.cm";
```

at the REPL whenever you change your code instead of a `use` command like in previous assignments. The compilation manager offers a better interface to the command line. There is less typing and less of an issue with name shadowing between iterations of your code. In short, on this assignment, the development cycle will be:

1. Edit your source files.
2. At the REPL, type

   ```
   CM.make "sources.cm";
   ```

3. Fix errors and debug.
4. If done, consider doing 251 homework; else go to 1.

Be warned that `CM.make` will make a directory in the current working directory called `.cm`. This is populated with metadata needed to work out compilation dependencies, but can become quite large. The `.cm` directory can safely be deleted at the completion of this assignment.

It's sometimes the case that the metadata in the `.cm` directory gets in to an inconsistent state—if you run `CM.make` with different versions of SML in the same directory, for example. This often produces bizarre error messages. When that happens, it's also safe to delete the `.cm` directory and compile again from scratch.

1.5.1 Emphatic Warning

CM will not return cleanly if any of the files listed in the sources have no code in them. Because we want you to learn how to write modules from scratch, we have handed out a few files that are empty except for a few place holder comments. That means that there are a few files in the `sources.cm` we handed out that are commented out, so that when you first get your tarball `CM.make "sources.cm"` will work cleanly.

You must uncomment these lines as you progress through the assignment! If you forget, it will look like your code compiles cleanly even though it almost certainly doesn't.

2 Marshalling

2.1 Introduction

One common programming task is *marshalling*: transforming data into a form where it can be written out and later read back in. This is useful if you want data to persist across different runs of your program by storing it in a file, or if you want to send data across a network.[1] It's easy to write a string to a file or to send a string over the network, so we won't take this problem any farther than producing a string from data and vice versa.

In this problem, you will define a small marshalling library, focused on capturing the notion of data that can be marshalled with the typeclass

```
signature MARSHAL =
sig
    type t

    (* invariant: For all v, s: read (write v ^ s) == SOME (v , s) *)
    val write : t -> string
    val read : string -> (t * string) option
end
```

That is: a type `t` may be marshalled only if it supports `read` and `write` operations that convert it to and from a `string`, which interact appropriately. "Appropriately" means that if you read from a string whose prefix was constructed by `write`, then `read` returns the value that was written, along with any suffix. More formally:

For all values `v:t` and `s:string`, `read (write v ^ s) = SOME (v, s)`

For example, given `M : MARSHAL`, if a web server sends a string produced by `M.write v` to a browser, and then the browser calls `M.read` on that string, the spec says that the client will recover the value the server intended. Note that this spec allows `read` to have arbitrary behavior when applied to a string that does not have a prefix produced by `write`—e.g., we assume that the string is not corrupted during transmission.

2.2 Utility Structure

To minimize the amount of tedious parsing code you need to write, we've provided an implementation of the following signature in the handout code. Be sure to understand these functions and use them when you can: they should make your code a lot cleaner.

[1]Check out http://en.wikipedia.org/wiki/Marshalling_(computer_science), http://en.wikipedia.org/wiki/Serialization for a lot more information.

This signature lives in `util.sig`. We have provided an implementation for you in `util.sml`. You should understand the signature and freely use the functions the structure ascribing to it provides, but you do not need to understand their implementation.[2]

```
signature UTIL =
sig
  (* peelOff (s1,s2) = SOME s' if s2 = s1 ^ s'
   *                 = NONE otherwise
   *
   * Ex:
   *   peelOff ("a","a")  = SOME("")
   *   peelOff ("a","ab") = SOME("b")
   *   peelOff ("a","c")  = NONE
   *)
  val peelOff : string * string -> string option

  (* peelInt s = SOME (i,s') if the longest non-empty prefix of s
   *                comprised only of an optional leading #"~" and following digits
   *                parses to the integer i, and s = i ^ s'
   *           = NONE otherwise
   *
   * Ex:
   *   peelInt "55hello"  = SOME(55,"hello")
   *   peelInt "~55hello" = SOME(~55,"hello")
   *   peelInt "-55hello" = NONE
   *   peelInt "hello55"  = NONE
   *   peelInt "12~100"   = SOME(12,"~100")
   *)
  val peelInt : string -> (int * string) option
end
```

2.3 Marshalling Booleans

Here is an example structure that demonstrates one of many possible ways to marshal the type `bool`.[3] In `write`, we choose to marshal the value `true` as the string `"(TRUE)"` and the value `false` as the string `"(FALSE)"`.

To try to read back one of these values from a string `s`, we first try to peel off `"(TRUE)"` from `s`. If that succeeds, we return the value `true` and whatever is left over; if it fails, we

[2]This signature is provided in `util.sig` and implemented in a module `Util` ascribing to `UTIL` provided in `util.sml`.

[3]This particular implementation is provided in `marshal.sml`; you should feel free to experiment with it to use it for testing.

try to peel off `"(FALSE)"`. If this succeeds, return the value `false` and any leftovers. If this fails, having now failed overall, we return `NONE`.

2.4 Integers, Pairs, and Lists—Oh, my!

Your task is to write marshallers for integers, pairs, and lists.

2.4.1 One Possible Strategy

To marshal values of a type, consider how many constructors that type has and how many arguments each of them takes. To keep track of the tree-structure of an expression, you will need to be able to distinguish between constructors and between different instances of the same constructor.

One way to do this is to represent a value constructed with the constructor `con` and arguments `A1` through `An` as

```
(con A1 A2 ...  An)
```

If you consider the type `bool` to be defined as

```
datatype bool = true | false
```

then this is exactly what we did above: the type is given by two nullary constructors and nothing else.

2.4.2 Tasks

Submit your solutions for the following tasks in `marshal.sml`.

Task 2.1 (10%). Implement a structure `MarshalInt` that ascribes to `MARSHAL` and defines mashalling for the type `int`.

Task 2.2 (10%). Prove the following theorem of correctness for integer marshalling:

For all values `v:int` and `s:string`, `read (write v ^ s) = SOME (v, s)`

You may assume the following lemmas (but cite them when you use them):

Lemma 1: For all `i : int`, all non-digit `c : char`, all `s : string`, `peelInt (Int.toString i ^ "c" ^ s) = SOME(i, "c" ^ s)`

Lemma 2: For all `s1 : string, s2 : string`, `peelOff s1 (s1 ^ s2) = SOME (s2)`

Lemma 3: `^` is associative.

You may also assume `Int.toString` and `^` are total (but again, cite this when used).

Solution 2.2 **Need to show:** `read(write v ^ s) = SOME(v, s)` for all `v : int` and `s : int`

Proof:
Consider some `v : int` and `s : string`

```
  read(write v ^ s)

 = read(Int.toString v ^ "." ^ s)            [Step, Lemma 3]

 = case (Util.peelInt(Int.toString v ^ "." ^ s)) of
        SOME (i, s) =>
         (case Util.peelOff (".", s) ...)
      | NONE => NONE          [Step, Totality of Int.toString and ^, v and s are values]

 = case SOME(v, "." ^ s) of
        SOME (i, s) =>
         (case Util.peelOff (".", s) ...)
      | NONE => NONE          [Lemma 1, Ref. trans.]

 = case Util.peelOff (".", "." ^ s) of
        SOME x => SOME (v, x)
      | NONE => NONE          [Step, bind i to v]

 = case SOME s of
        SOME x => SOME (v, x)
      | NONE => NONE          [Lemma 2, Ref. trans.]

 = SOME (v,s)         [Step]
```

`v :int` and `s :string` were arbitrary.
Thus we have shown `read(write v ^ s) = SOME(v, s)` for all `v :int` and `s :int`

Task 2.3 (10%). The signature

```
signature MARSHALPAIR =
sig
```

```
  structure M1 : MARSHAL
  structure M2 : MARSHAL
end
```

packages together two modules that can be marshaled. [4] Implement a functor

```
functor MarshalPair (P : MARSHALPAIR) : MARSHAL
```

that implements marshalling for the type `P.M1.t * P.M2.t`. You may assume that `P.M1` and `P.M2` both obey the above invariant, but you may not make any other assumptions about them.

Task 2.4 (10%). Implement a functor

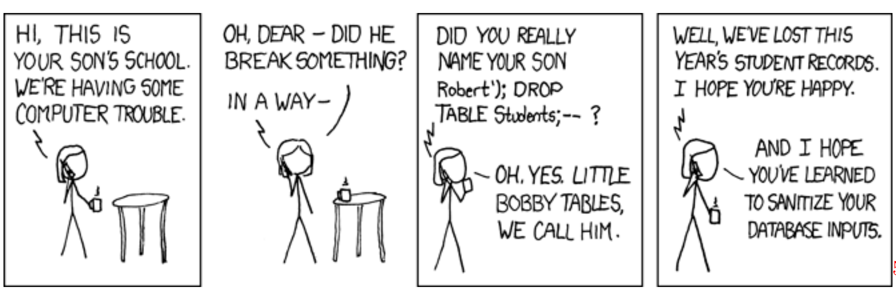
```
functor MarshalList (S : MARSHAL) : MARSHAL
```

that defines marshaling for the type `S.t list`. You may assume that `S` obeys the marshaling invariant above, but you may not assume anything else about `S`.

Task 2.5 (5%). Define marshalling instances for the following types using only the modules above.

1. `int list list` with a structure named `ILL` ascribing to `MARSHAL`.
2. `(int list) * bool` with a structure named `ILSB` ascribing to `MARSHAL`.
3. `(int * bool) list` with a structure named `ISBL` ascribing to `MARSHAL`.
4. `(int * (bool list)) list` with a structure named `ISBLL` ascribing to `MARSHAL`.

Task 2.6 (Extra Credit).Write a structure `MarshalString` ascribing to `MARSHAL` that defines marshalling for the type `string`. Hint:



[5]

[4] This signature is provided in `marshalpair.sig`.

[5] http://xkcd.com/327/

3 Sequence Basics

For the remainder of the problems in this homework, you will be using the sequence library. We have provided you a sequence implementation which you will be using for this and the remaining homeworks. For complete documentation see `handout/sequencereference.pdf` in your git repository. You should take some time now to read the handout and familiarize yourself with the functions in the sequence library and their cost bounds.

Now, for the following tasks you will work with and analyze functions on sequences.

Here is an example of how to perform an analysis on functions using sequences:

```
fun sum (s : int seq) : int = reduce (op +) 0 s
fun count (s : int seq seq) : int = sum (map sum s)
```

`sum` takes an integer sequence and adds up all the numbers in it using `reduce`, just like we did with folds for lists and trees. `count` sums up all the numbers in a sequence of sequences, by (1) summing each individual sequence and then (2) summing the sequence that results.

(Note : We could rewrite `count` with `mapreduce`, so it takes only one pass, we will be talking more about `mapreduce` later).

Example Task Suppose all sequences, inner and outer, have length n. Give a tight O-bound for the span of `count`. Briefly explain why your answer is correct.

Solution :
Let `count` be on a sequence of n sequences, each of length n.
This requires $O(\log n)$ span :

`sum s` is implemented using `reduce` with constant-time arguments, and thus has $O(\log n)$ span, where n is the length of s. Each call to `sum` inside the the `map` doesn't contribute anything, and both the inner and outer `sum`s are on sequences of length n, and therefore have $O(\log n)$ span. The total span is the sum of the inner span and the outer span, because of the data dependency: the outer additions happen after the inner sum have been computed. The sum of $\log n$ and $\log n$ is still $O(\log n)$, so the total span is $O(\log n)$.

3.1 Append

As we know, appending two lists can be expensive, and is unfortunately non-parallelizable. What about appending two sequences?

Task 3.1 (6%). Provide a non-recursive implementation of `Seq.append` called `myAppend` in the file `sequences.sml`. Your solution may use any of the above elements of the Sequence Library except for `Seq.append` and `Seq.flatten`. That's cheating.

Task 3.2 (2%). Give a tight O-bound for the work of `myAppend`. Make sure you explicitly state what quantities you are analyzing the work in terms of. Briefly explain why your answer is correct.

Task 3.3 (2%). Give a tight O-bound for the span of `myAppend`. Make sure you explicitly state what quantities you are analyzing the span in terms of. Briefly explain why your answer is correct.

Solution 3.3 The function argument to tabulate has constant work/span, because $<$, `Seq.length`, `Seq.nth` all have constant work/span.

So let $n1$ be `length(s1)` and $n2$ be `length(s2)`.

$W(n1, n2)$ is $O(n1 + n2)$ because tabulate has work linear on the length of the produced sequence, assuming the function is constant.

$S(n1, n2)$ is $O(1)$ because the span of tabulate f n is just the span of f, which is constant in this case.

3.2 Two-Largest

One classic problem on lists is finding the largest elements in the list. In the following tasks, you will implement a function which finds the two largest elements of a sequence.

Task 3.4 (3%). Implement the function

```
max4 : ('a * 'a -> order) -> (('a * 'a) * ('a * 'a)) -> ('a * 'a)
```

such that for a comparison function `cmp` and two pairs of elements `(a, b)` and `(c, d)`, `max4 cmp ((a, b), (c, d))` returns the two largest elements of the four. Order does not matter. For example,

```
val (3, 4) = max4 Int.compare ((1, 4), (3, 2))
val (5, 6) = max4 Int.compare ((3, 4), (5, 6))
```

For your convenience, we have implemented the `maxL` function which you may find useful for `max4`. `maxL : ('a * 'a -> order) -> 'a list -> ('a * 'a list)` takes a comparison function `cmp` and a non-empty list `L` and returns the largest element of `L` along with a list `L'` which is all elements of `L` excluding the max.

Task 3.5 (7%). Implement the function

```
twoLargest : ('a * 'a -> order) -> ('a * 'a) -> 'a seq -> ('a * 'a)
```

such that for a comparison function `cmp`, a pair of elements `(a, b)` and a sequence `s`, `twoLargest cmp (a, b) s` returns the two largest elements amongst all the elements of `s` and `(a, b)`. Again, order does not matter. For example:

```
(* s = <1, 5, 8, 2>
   s' = <"b"> where s and s' are sequences *)
val (5, 8) = twoLargest Int.compare (0, 0) s
val ("b", "c") = twoLargest String.compare ("a", "c") s'
```

Note: your solution **must** have $O(n)$ work and $O(\log n)$ span where n is the size of the input sequence (you may assume that the comparison function has $O(1)$ work and span). To achieve the time bounds, make sure to look at the functions available to you in the sequence library. You may also find `max4` useful for implementing `twoLargest`.

4 Kittens on Stairs

A cute kitten is trying to climb some stairs![6] Every second, it climbs up to a higher stair or tumbles adorably down to a lower stair. You want to congratulate it for doing so well, so you want to find the farthest climb that the kitten made up the stairs. More specifically, given a sequence of integers `s`, you wish to find the maximal $s_i - s_j, i > j$. Since SML is almost as wonderful as a kitten, you write the following code:

```
fun suffixes (s : 'a Seq.seq) : ('a Seq.seq) Seq.seq =
  Seq.tabulate (fn x => Seq.drop (x + 1) s) (Seq.length s)

val SOME(minInt) = Int.minInt
val maxS : int Seq.seq -> int = Seq.reduce Int.max minInt
fun maxAll (s : int Seq.seq Seq.seq) : int =
  maxS (Seq.map maxS s)

fun withSuffixes (t : int Seq.seq) : (int * int Seq.seq) Seq.seq =
  Seq.zip (t, suffixes t)

fun mostSteps (s : int Seq.seq) : int =
    let fun diff (start, stops) = Seq.map (fn stop => stop - start) stops
        val diffs = Seq.map diff (withSuffixes s)
    in maxAll diffs end
```

After using your new functions and congratulating the kitten, you realize that the work of `suffixes s`, in terms of the length of `s`, is in $O(n^2)$, where n is the length of `s`. After watching a few cat videos, you realize that the span of `suffixes s` is in $O(1)$. Then you realize your brain is telling you to get back to your homework.

Task 4.1 (2%). Give a tight O-bound for the work of `withSuffixes s`, in terms of the length of `s`. Briefly explain why your answer is correct.

Task 4.2 (2%). Give a tight O-bound for the span of `withSuffixes s`, in terms of the length of `s`. Briefly explain why your answer is correct.

Solution 4.2 $W(n)$ is $O(n^2)$. There's $O(n^2)$ work for `suffixes`, and `Seq.zip` is cheaper at $O(n)$.

$S(n)$ is $O(1)$. `Seq.zip` and `suffixes` are both constant-span.

Task 4.3 (3%). Give a tight O-bound for the work of

$$\texttt{maxAll}\langle\langle x_1^1, \ldots, x_{k_1}^1\rangle, \ldots, \langle x_1^n, \ldots, x_{k_n}^n\rangle\rangle$$

(i.e. the i^{th} inner sequence has length k_i and the outer sequence of sequences has length n) in terms of $k_1, \ldots, k_n$ and n. Briefly explain why your answer is correct.

[6] Relevant: http://www.youtube.com/watch?v=D4BJrdrBi0Y

Task 4.4 (3%). Give a tight O-bound for the span of

$$\texttt{maxAll}\langle\langle x_1^1, \ldots, x_{k_1}^1\rangle, \ldots, \langle x_1^n, \ldots, x_{k_n}^n\rangle\rangle$$

in terms of $k_1, \ldots, k_n$ and n. Briefly explain why your answer is correct.

> **Solution 4.4** $W(n)$ is $O(n+\sum_{i=1}^{n} k_i)$. `maxS` is linear time since it's a `Seq.reduce` of a constant-time function. Then the work of mapping it across the input sequence is the sum of the work on the elements plus another $O(n)$ term for iterating over the sequence. Then the final `maxS` takes another $O(n)$ time since it's linear.
>
> $S(n)$ is $O(\lg n + \max_{i\in[n]}(\lg k_n))$. `maxS` takes logarithmic time since it's a reduce of a constant-time function. In `maxAll` we must do the `Seq.map` first, and the time it takes for `Seq.map` is a constant plus the longest time it takes for any single element, giving $O(1 + \max_{i\in[n]}(\lg k_n))$ span. Likewise the second call gives an additional $O(\lg n)$ span (which subsumes the constant).

Task 4.5 (3%). Give a tight O-bound for the work of `mostSteps s`, in terms of the length of `s`. Briefly explain why your answer is correct.

Task 4.6 (3%). Give a tight O-bound for the span of `mostSteps s`, in terms of the length of `s`. Briefly explain why your answer is correct.

> **Solution 4.6** $W(n)$ is $O(n^2)$. First `withSuffixes` gives $O(n^2)$ work. Then the nested map gives us $O(n^2)$ work since the function in the inner map is constant-time and the inner sequences have sizes that sum to $O(n^2)$. Plugging in, `maxAll` gives $O(n + n^2)$ or equivalently $O(n^2)$ work, for a total of $O(n^2)$.
>
> $S(n)$ is $O(\lg n)$. `withSuffixes` and the nested `Seq.map` of a constant-time function are both constant span. Then since the longest inner sequence is length n, `maxAll` gives span $O(\lg n + \lg n)$, which is $O(\lg n)$.

5 Rainfall

Old MacDonald had a farm, and on that farm he would like to know where to expect rainfall to collect. He has given you an $n \times n$ two-dimensional grid of integers, or an `int Seq.seq Seq.seq`, of land elevations across his farm. A position on a farm is defined as pair of indices (i, j) corresponding to the j-th entry in the i-th row. For example, the sequence $\langle\langle 1, 2, 3\rangle, \langle 4, 0, 5\rangle, \langle 6, 7, 8\rangle\rangle$ has the entry 1 at position $(0, 0)$ and looks like

```
1 2 3
4 0 5
6 7 8
```

Your task is to produce a sequence of sinks, where we define a sink to be a position with elevation lower than the four points surrounding it (up, down, left, right). You do not need to consider diagonals. Any points on the edges of Old MacDonald's input sequence cannot be sinks. You may assume that the sequence is made up of unique heights.

In the above example, `rainfall <<1, 2, 3>, <4, 0, 5>, <6, 7, 8>> = <(1, 1)>` because the only possible sink is at $(1, 1)$ and 0 is less 2, 4, 5, and 7 so the index $(1, 1)$ is a sink.

Task 5.1 (13%). Help Old MacDonald by implementing the following function in `sequences.sml`:

```
fun rainfall : int Seq.seq Seq.seq -> (int * int) Seq.seq
```

which takes an $n \times n$ sequence of heights and returns a sequence of sink positions. Your sink sequence must be ordered by *index* (e.g. $(1, 1)$ precedes $(2, 0)$). Your solution should be **non-recursive** and also use functions in the **sequence library**.

Feel free to use any of the testing helper functions or farms provided in the `SequenceHelper` structure when testing your code!

Hint 1: You may find it useful to implement the two helper functions `atEdge` and `isSink` in `sequences.sml` for determining if an index is an edge or sink, respectively.

Hint 2: Make sure you've taken a good look at the sequence library at your disposal!

Task 5.2 (3%). Give a tight O-bound for the work of `rainfall s`, in terms of n where `s` is an $n \times n$ sequence. Briefly explain why your answer is correct.

Task 5.3 (3%). Give a tight O-bound for the span of `rainfall s`, in terms of n where `s` is $n \times n$. Briefly explain why your answer is correct.

Solution 5.3 Let n be the length of the input sequence.

`atEdge` and `isSink` both have constant work/span, because `Seq.length` and `Seq.nth` both have constant work/span.

The tabulate has $O(n^2)$ work and $O(1)$ span, because it produces a sequence of length n^2 with a constant time function.

The filter has $O(n^2)$ work and $O(\log(n^2)) = O(\log(n))$ span, as it operates on a sequence of length n^2 with a constant time function.

Summing the work/span from each of these computations gives us the total work/span of `rainfall`: $O(n^2)$ and $O(\log(n))$, respectively.