15-150 Summer 2017
Homework 08

Out: Wednesday, 2 November 1936
Due: **Thursday**, 10 November 2016

0 Introduction

This homework will focus on exceptions, structures, signatures, and functors.

0.1 Getting The Homework Assignment

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

0.2 Submitting The Homework Assignment

Code submissions will be handled through Autolab, at

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

Written submissions will be handled through Gradescope, at

`https://gradescope.com`

To submit your code, run `make` from the `hw/08` directory (that contains a `code` folder. This should produce a file `handin.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 `handin.tar` file via the "Handin your work" link.

To submit your written solutions (which *must* be in PDF form), log in to Gradescope and submit your PDF to the assignment **Homework 08**. If you are unable to do this, make a Piazza post or find a TA to help.

The Autolab handin script does some basic checks on your submission: making sure file names are correct; making sure that no files are missing; making sure your code compiles cleanly; making sure your functions have the right types. In addition, the script tests each function that you submit individually using public tests. These public tests are in no way

comprehensive, and will in most cases consist of a simple base case test. In order to receive full points, you must pass private tests that will not be run until after the submission deadline has passed, so you will still have to write your own tests to ensure correctness.

The script indicates a correct submission via a score on the Autolab website. If all files are present, your submission will receive a score of "0.0" in the "files" category. If your code compiles, it will receive a score of "0.0" in the "compile" category. If files are missing or the code does not compile, you will receive a highly negative score for the corresponding category. For each function, you will receive a score that represents the number of public tests passed for that function. If you do not receive full points for the public tests for a particular function, you will receive a score of zero for that function.

Your `hw08.sml` file 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.

0.3 Due Date

This assignment is due on **Thursday**, 10 November 2016. Remember that you ***do not get grace days this semester***, so we will not be accepting late submissions without a documented, university-approved excuse.

0.4 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:

```
(*  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
```

0.5 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:

```
$ smlnj
Standard ML of New Jersey v110.78 [built: Fri Jul 17 17:45:05 2015]
- 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. Repeat as necessary

You can also do the following to load your cm file:

```
> smlnj -m sources.cm
```

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.

0.5.1 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.

1 Definitely-Not-Sequential Sequences

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 `src/sequence/sequencereference.pdf` in your git repository (for convenience, we have also included this documentation in Appendix C of this handout). 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 $O(\log n)$.

1.1 Append

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

Task 1.1 (5 pts). 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`, `Seq.flatten`, and `Seq.toList`. That's cheating.

Task 1.2 (2 pts). 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 1.3 (2 pts). 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.

1.2 SeqExists

Recall the function `List.exists : ('a -> bool) -> 'a list -> bool`, which determines whether an element of the list satisfies the given predicate. You will write an analogous function for sequences:

Task 1.4 (4 pts). Implement the function

```
seqExists : ('a -> bool) -> 'a Seq.seq -> bool
```

to determine if the sequence has an element that satisfies the given predicate. `seqExists p S` should have $O(|S|)$ work and $O(\log |S|)$ span.

Note: if you're really cool, you can make `seqExists` have $O(1)$ span.

1.3 Transpose

Recall the function `transpose` from Homework 5:

```
transpose [[1,2,3],
           [4,5,6]]
  ==>
[[1,4],
 [2,5],
 [3,6]]
```

Task 1.5 (4 pts). Implement the function

```
transpose (s : 'a Seq.seq Seq.seq) : 'a Seq.seq Seq.seq
```

that transposes a sequence of sequences. You may assume that s is rectangular, with dimensions $m \times n$, where $m, n > 0$. Your solution should have $O(m \times n)$ work and $O(1)$ span.

2 Roller Coaster

You are the first rider on the newest roller coaster at Kennywood. Every second, your cart either climbs up on the track or drops down. You want to find the biggest drop that you'll experience while riding the roller coaster. More specifically, given a sequence of integers `s`, you wish to find the maximal $s_i - s_j, j > i$. 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 biggestDrop (s : int Seq.seq) : int =
    let fun diff (start, stops) = Seq.map (fn stop => start - stop) stops
        val diffs = Seq.map diff (withSuffixes s)
    in maxAll diffs end
```

After your inaugural ride, 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 stopping for a few Potato Patch Fries, 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 2.1 (2 pts). 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 2.2 (2 pts). 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.

Task 2.3 (2 pts). 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.

Task 2.4 (2 pts). 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.

Task 2.5 (3 pts). Give a tight O-bound for the work of `biggestDrop s`, in terms of the length of `s`. Briefly explain why your answer is correct.

Task 2.6 (3 pts). Give a tight O-bound for the span of `biggestDrop s`, in terms of the length of `s`. Briefly explain why your answer is correct.

3 Rain Fall

Vijay loves tending to crops, and is super excited to be hosting Rohan on his farm for a rural 150 TA get together. However, Rohan has ulterior motives. He is secretly in a puddlehunt, and wants to know where the rainfall will collect on Vijay's farm. He will need a speedy function to find the puddles without Vijay noticing. He has given you an $n \times n$ two-dimensional grid of integers, or an `int Seq.seq Seq.seq`, of land elevations across the 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 puddles, where we define a puddle 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 Rohan's input sequence cannot be puddles. 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 puddle is at $(1, 1)$ and 0 is less 2, 4, 5, and 7 so the index $(1, 1)$ is a puddle.

Task 3.1 (10 pts). Help Rohan 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 puddle positions. Your puddle 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 `isPuddle` in `sequences.sml` for determining if an index is an edge or puddle, respectively.

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

Task 3.2 (2 pts). 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 3.3 (2 pts). 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.

Appendix A: Turing Machine Structures/Functors

Name	What	Source	Description
`FindFirstOne`	structure: `TURING_MACHINE`	Homework 8	Flips first 1 and accepts; rejects if there are no 1s
`MkNegation`	functor: `TURING_MACHINE` to `TURING_MACHINE`	Task 2.1	Produces the logical negation of a given Turing machine
`DreadFirstOne`	structure: `TURING_MACHINE`	Task 2.2	Flips first 1 and rejects; accepts if there are no 1s
`EveryOtherDouble`	structure: `DOUBLE_MACHINE`	Provided	Accepts if every odd-indexed or every even-indexed symbol is 1.
`MkDouble`	functor: `DOUBLE_MACHINE` to `TURING_MACHINE`	Task 2.3	Produces a Turing machine from a double machine
`EveryOther`	structure: `TURING_MACHINE`	Task 2.4	See `EveryOtherDouble`

Appendix B: The Lib Structure

Function	Description
`Lib.charToSymbol : char -> Tape.symbol`	Converts characters `#"1"` and `#"0"` to `Tape.symbol`s
`Lib.cellToChar : Tape.cell -> char`	Converts `Tape.cell`s to `#"1"`, `#"0"`, and `#"_"`
`Lib.toInput : string -> Tape.symbol list`	Uses `charToSymbol` to turn a string into input for a Turing machine.
`Lib.fromOutput : Tape.cell list -> string`	Uses `cellToChar` to turn Turing machine output into a string.
`Lib.wrap : (Tape.symbol list -> bool * Tape.cell list) -> (string -> string)`	Uses `toInput` and `fromOutput` to turn a Turing machine `simulate` function into a function that takes/returns strings.

Appendix C: Sequence Reference

Please cite this document if you use it in your homework.

The type `Seq.seq` represents sequences. Sequences are *parallel collections*: ordered collections of things, with parallelism-friendly operations on them. Don't think of sequences as being implemented by lists or trees (though you could implement them as such); think of them as a new built-in type with only the operations we're about to describe. The differences between sequences and lists or trees is the cost of the operations, which we specify below.

4 The Signature

- `Seq.length : 'a Seq.seq -> int`
  `Seq.length s` evaluates to the number of items in `s`.
- `Seq.empty : unit -> 'a Seq.seq`
  `Seq.empty ()` evaluates to the sequence of length zero.
- `Seq.singleton : 'a -> 'a Seq.seq`
  `Seq.singleton x` evaluates to a sequence of length 1 where the only item is `x`.
- `Seq.append : 'a Seq.seq * 'a Seq.seq -> 'a Seq.seq`
  If `s1` has length l_1 and `s2` has length l_2, `Seq.append` evaluates to a sequence with length $l_1 + l_2$ whose first l_1 items are the sequence `s1` and whose last l_2 items are the sequence `s2`.
- `Seq.tabulate : (int -> 'a) -> int -> 'a Seq.seq`
  `Seq.tabulate f n` evaluates to a sequence `s` with length `n` where the i^{th} item of `s` is the result of evaluating `(f i)`. This is zero-indexed. `Seq.tabulate f i` raises `Range` if `n` is less than zero.
- `Seq.nth : 'a Seq.seq -> int -> 'a`
  `nth s i` evaluates to the i^{th} item in `s`. This is zero-indexed. `Seq.nth s i` will raise `Range` if `i` is negative or greater than `(Seq.length s)-1`.
- `Seq.filter : ('a -> bool) -> 'a Seq.seq -> 'a Seq.seq`
  `Seq.filter p s` evaluates to a sequence that contains all of the elements of `s` for which `p` evaluates to `true`, preserving element order.
- `Seq.map : ('a -> 'b) -> 'a Seq.seq -> 'b Seq.seq`
  `Seq.map f s` maps `f` over the sequence `s`. That is to say, it evaluates to a sequence `s'` such that `s` and `s'` have the same length and the i^{th} item in `s'` is the result of applying `f` to the i^{th} item of `s`.

- `Seq.reduce : (('a * 'a) -> 'a) -> 'a -> 'a Seq.seq -> 'a`
  `Seq.reduce c b s` combines all of the items in `s` pairwise with `c` using `b` as the base case, respecting the order of items as they appear in `s`. `c` must be associative. (Frequently one requires `b` to be an identity for `c`, but we sometimes allow more general `b`.)

- `Seq.reduce1 : (('a * 'a) -> 'a) -> 'a Seq.seq -> 'a`
  `Seq.reduce1 c s` combines all of the items in `s` pairwise with `c`, again respecting the order of items in `s`. The sequence `s` must be nonempty and the operation `c` must be associative.

- `Seq.mapreduce : ('a -> 'b) -> 'b -> ('b * 'b -> 'b) -> 'a Seq.seq -> 'b`
  `Seq.mapreduce l e n s` is equivalent to `Seq.reduce n e (Seq.map l s)`.

- `Seq.toString : ('a -> string) -> 'a Seq.seq -> string`
  `Seq.toString ts s` evaluates to a string representation of `s` by using `ts` to convert each item in `s` to a `string`.

- `Seq.repeat : int -> 'a -> 'a Seq.seq`
  `Seq.repeat n x` evaluates to a sequence consisting of exactly `n`-many copies of `x`.

- `Seq.flatten : 'a Seq.seq Seq.seq -> 'a Seq.seq`
  `Seq.flatten ss` is equivalent to `reduce append (empty ()) ss`

- `Seq.zip : ('a Seq.seq * 'b Seq.seq) -> ('a * 'b) Seq.seq`
  `Seq.zip (s1,s2)` evaluates to a sequence whose n^{th} item is the pair of the n^{th} item of `s1` and the n^{th} item of `s2`.

- `Seq.split : 'a Seq.seq -> int -> 'a Seq.seq * 'a Seq.seq`
  If `s` has at least `i` elements, `Seq.split s i` evaluates to a pair of sequences `(s1,s2)` where `s1` has length `i` and `append s1 s2` is the same as `s`. Otherwise it raises `Range`.

- `Seq.take : 'a Seq.seq -> int -> 'a Seq.seq`
  `Seq.take s i` evaluates to the sequence containing exactly the first `i` elements of `s` if $0 \leq$ `i` $\leq$ `length s`, and raises `Range` otherwise.

- `Seq.drop : 'a Seq.seq -> int -> 'a Seq.seq`
  `Seq.drop s i` evaluates to the sequence containing all but the first `i` elements of `s` if $0 \leq$ `i` $\leq$ `length s`, and raises `Range` otherwise.

- `Seq.cons : 'a -> 'a Seq.seq -> 'a Seq.seq`
  If the length of `xs` is `l`, `Seq.cons x xs` evaluates to a sequence of length `l+1` whose first item is `x` and whose remaining `l` items are exactly the sequence `xs`.
  (**NOTE:** Please do not use `Seq.cons` unless it is specifically suggested. If it is overused, it can lead to a sequential style that is somewhat against the spirit of sequences.)

- `Seq.update : 'a Seq.seq * int * 'a -> 'a Seq.seq`
  `Seq.update (s, i, x)` returns a sequence identical to `s` but with the `i`th element (0-indexed) now `x`. Index `i` must satisfy $0 \leq$ `i` $<$ `length(s)`.

- `Seq.toList : 'a seq -> 'a list`
  `Seq.toList s` returns a list consisting of the elements of `s`, preserving order. This function is intended primarily for debugging purposes.

- `Seq.fromList : 'a list -> 'a seq`
  `Seq.fromList L` returns a sequence consisting of the elements of `L`, preserving order. This function is intended primarily for debugging purposes.

5 Cost Bounds

Unfortunately, there is no known way of stating time complexity of a higher order function, such as map, itself abstractly in the function—there is no theory of asymptotic analysis for higher-order functions. Therefore, this chart **assumes that all functions that are given as arguments take constant time.** To consider the cost if these functions do not take constant time, we will need to go back to the cost graphs and expand them to include the additional work and span.

Function	Work	Span
`Seq.length S`	$O(1)$	$O(1)$
`Seq.empty ()`	$O(1)$	$O(1)$
`Seq.singleton x`	$O(1)$	$O(1)$
`Seq.append (S1, S2)`	$O(\|S1\| + \|S2\|)$	$O(1)$
`Seq.tabulate f n`	$O(n)$	$O(1)$
`Seq.nth i S`	$O(1)$	$O(1)$
`Seq.filter p S`	$O(\|S\|)$	$O(\log \|S\|)$
`Seq.map f S`	$O(\|S\|)$	$O(1)$
`Seq.reduce c b S`	$O(\|S\|)$	$O(\log \|S\|))$
`Seq.reduce1 c b S`	$O(\|S\|)$	$O(\log \|S\|))$
`Seq.mapreduce l e n S`	$O(\|S\|)$	$O(\log \|S\|))$
`Seq.toString ts S`	$O(\|S\|)$	$O(\log \|S\|)$
`Seq.repeat n x`	$O(n)$	$O(1)$
`Seq.flatten S`	$O(\|S\| + \max_{s \in S} \|s\|)$	$O(\log \|S\|)$
`Seq.zip (S1,S2)`	$O(\min(\|S1\|, \|S2\|)$	$O(1)$
`Seq.split i S`	$O(\|S\|)$	$O(1)$
`Seq.take i S`	$O(i)$	$O(1)$
`Seq.drop i S`	$O(\|S\| - i)$	$O(1)$
`Seq.cons x S`	$O(\|S\|)$	$O(1)$
`Seq.update (S, i, x)`	$O(\|S\|)$	$O(1)$
`Seq.toList S`	$O(\|S\|)$	$O(\|S\|)$
`Seq.fromList L`	$O(\|L\|)$	$O(\|L\|)$

6 Detailed Cost Bounds

Intuitively, these sequence operations do the same thing as the operations on lists that you are familiar with. However, they have different time complexity than the list functions: First, sequences admit constant-time access to elements—`nth` takes constant time. Second, sequences have better parallel complexity—many operations, `map` act on each element of the sequence in parallel.

For each function, we (1) describe its behavior abstractly and (2) give a cost graph.

Length

The behavior of `length` is

```
length <x1 , ... , xn> == n
```

The cost graph for `length s` is

As a consequence, `length` has $O(1)$ work/span.

Nth

Sequences provide constant-time access to elements. Abstractly, we define the behavior of `nth` as

```
nth <x0 , ... , xn-1> i == xi if 0 <= i < n
                              or raises Range if i>=n
```

Here `<x1 , ... , xn>` is *not* SML syntax, but mathematical syntax for a sequence value.

The cost graph for `nth s i` is

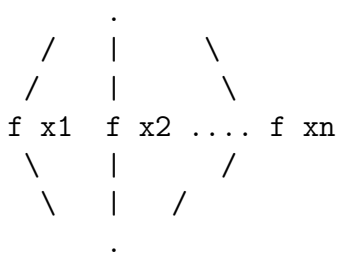
As a consequence, `nth` has $O(1)$ work/span.

Map

The the behavior of `map f <x1,...xn>` is

```
map f <x1 , ... , xn> ==> <f x1,...,f xn>
```

Each function application may be evaluated in parallel. This is represented by the cost graph



where we write `f x1`, etc. for the cost graphs associated with these expressions.

As a consequence, if `f` takes constant time, then `map f` has $O(n)$ work and $O(1)$ span.

Reduce

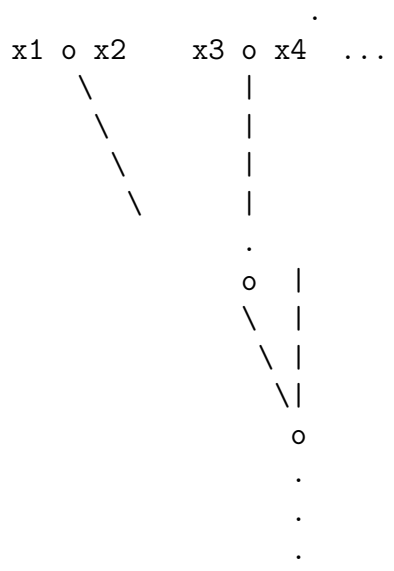
The behavior of `reduce` is

$$\texttt{reduce} \odot\ b\ \langle x_1, \ldots x_n \rangle = x_1 \odot x_2 \odot \ldots \odot x_n$$

That is, `reduce` applied its argument function (which we write here as infix $\odot$) between every pair of elements in the sequence.

However, the right-hand side is ambiguous, because we have not parenthesized it. There are two options: First, we could assume the function $\odot$ is associative, with unit b, in which case these all mean the same thing. However, there are some useful non-associative operations (e.g. floating point). So the second option is to specify a particular parenthesization $(x_1 \odot (x_2 \odot \ldots \odot (x_n \odot b)) \ldots)$ or $(\ldots (x_1 \odot x_2) \odot \ldots \odot x_n)$ or the balanced tree $((x_1 \odot x_2) \odot (x_3 \odot x_4)) \odot \ldots$ (with b filled in at the end if the sequence has odd length). Unless we say otherwise, you can assume the balanced tree.

The cost graph for $\texttt{reduce} \odot\ b\ \langle x_1, \ldots x_n \rangle$ is

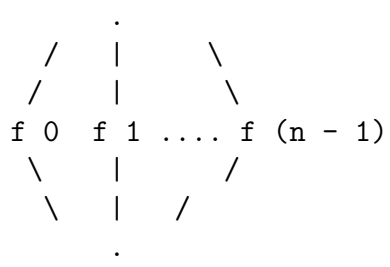


(with the last pair being either `xn-1 o xn` or `xn o b` depending on the parity of the length). That is, it is the graph of the balanced parenthesization described above. Consequently, if $\odot$ takes constant time (e.g. `op+`) then the graph has size (work) $O(n)$ and critical path length (span) $O(\log n)$. Reduce does not have constant span, because later applications of $\odot$ depend on the values of earlier ones.

Tabulate The way of introducing a sequence is *tabulate*, which constructs a sequence from a function that gives you the element at each position, from 0 up to a specified bound:

```
tabulate f n ==> <v0 , ... , v_(n-1)>
   if f 0 ==> v0
      f 1 ==> v1
      ...
      f (n-1) ==> v_(n-1)
```

The cost graph (and therefore time complexities) for tabulate is analogous to the graph for `map`:



Note that with `nth` and `tabulate` you can write very index-y array-like code. Use this sparingly: it's hard to read! E.g. never write

```
Seq.tabulate (fn i => ... nth s i ...)
             (Seq.length s)
```

if the function doesn't otherwise mention `i`: you're reimplementing `map` in a hard-to-read way!

```
Seq.map (fn x => ... x ...) s
```

7 Thinking About Cost

Let's think about the big picture of parallelism. Parallelism is relevant to situations where many things can be done at once—e.g. using the multiple cores in multi-processor machine, or the many machines in a cluster. Overall, the goal of parallel programming is to describe computation in such a way that it can make use of this ability to do work on multiple processors simultaneously. At the lowest level, this means deciding, at each moment in time,

what to do on each processor. This is limited by the data dependencies in a problem or a program. For example, evaluating `(1 + 2) + (3 + 4)` takes three units of work, one for each addition, but you cannot do the outer addition until you have done the inner two. So even with three processors, you cannot perform the calculation in fewer than two timesteps. That is, the expression has work 3 but span 2.

The approach to parallelism that we're advocating in this class is based on raising the level of abstraction at which you can think, by *separating the specification of what work there is to be done from the schedule that maps it onto processors.* As much as possible, you, the programmer, worry about specifying what work there is to do, and the compiler takes care of scheduling it onto processors. Three things are necessary to make this separation of concerns work:

1. The code itself must not bake in a schedule.
2. You must be able to reason about the *behavior* of your code independently of the schedule.
3. You must be able to reason about the *time complexity* of your code independently of the schedule.

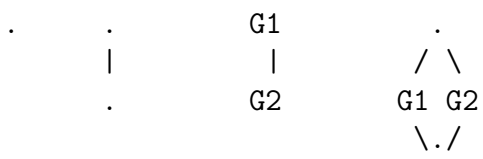
Our central tool for avoiding baking in a schedule is *functional programming.* First, we focus on bulk operations on big collections which do not specify a particular order in which the operations on each element are performed. For example, today, we will talk about *sequences*, which come with an operation `map f <x1,x2,...,xn>` that is specified by saying that its value is the sequence `<f x1, f x2, ... f xn>`. This specifies the data dependencies (to calculate `map`, you need to calculate `f x1` ...) without specifying a particular schedule. You can implement the same computation with a loop, saying "do `f x1`, then do `f x2`,...", but this is inlining a particular schedule into the code—which is bad, because it gratuitously throws away opportunities for parallelism. Second, functional programming focuses on pure, mathematical functions, which are evaluated by calculation. This limits the dependence of one chunk of work on another to what it is obvious from the data-flow in the program. For example, when you `map` a function `f` across a sequence, evaluating `f` on the first element has no influence on the value of `f` on the second element, etc.—this is not the case for imperative programming, where one call to `f` might influence another via memory updates. It is in general undecidable to take an imperative program and notice notice, after the fact, that what you really meant by that loop was a bulk operation on a collection, or that this particular piece of code really defines a mathematical function.

So why are we teaching you this style of parallel programming? There are two reasons: First, even if you have to get into more of the gritty details of scheduling to get your code to run fast today, it's good to be able to think about problems at a high level first, and then figure out the details. If you're writing some code for an internship this summer using a low-level parallelism interface, it can be useful to first think about the abstract algorithm—what are the dependencies between tasks? what can possibly be done in parallel?—and then figure out the details. You can use parallel functional programming to design algorithms,

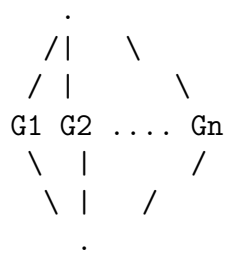
and then translate them down to whatever interface you need. Second, it's our thesis that eventually this kind of parallel programming will be practical and common: as language implementations improve, and computers get more and more cores, this kind of programming will become possible and even necessary. You're going to be writing programs for a long time, and we're trying to teach you tools that will be useful years down the road.

7.1 Cost Semantics Reminder

A cost graph is a form of *series-parallel graphs*. A series-parallel graph is a directed graph (we always draw a cost graph so that the edges point down the page) with a designated source node (no edges in) and sink node (no edges out), formed by two operations called sequential and parallel composition. The particular series-parallel graphs we need are of the following form:



The first is a graph with one node; the second is a graph with two nodes with one edge between them. The third, *sequential combination*, is the graph formed by putting an edge from the sink of `G1` to the source of `G2`. The fourth, *parallel combination*, is the graph formed by adding a new source and sink, and adding edges from the source to the source of each of `G1` and `G2`, and from the sinks of each of them to the new sink. We also need an n-ary parallel combination of graphs `G1 ... Gn`



The *work* of a cost graph is the number of nodes. The *span* is the length of the longest path, which we may refer to as the *critical path*, or the *diameter*. We will associate a cost graph with each closed program, and define the work/span of a program to be the work/span of its cost graph.

These graphs model *fork-join parallelism*: a computation forks into various subcomputations that are run in parallel, but these come back together at a well-defined joint point. These forks and joins are well-nested, in the sense that the join associated with a later fork precedes the join associated with an earlier fork.