Contents

1 Preamble **2**

2 Signature **3**

3 Documentation **5**
- 3.1 Constructing a Sequence 6
- 3.2 Deconstructing a Sequence 7
- 3.3 Simple Transformations 8
- 3.4 Combinators and Higher-Order Functions 9
- 3.5 Indexing-Related Functions 11
- 3.6 Sorting and Searching 13

4 Views **14**
- 4.1 List Views 14
- 4.2 Tree Views 14

5 Thinking About Cost **15**

6 Cost Graphs **17**

1 Preamble

The type `Seq.t` 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. In this document, we describe the cost of array-based sequences.

2 Signature

```
signature SEQUENCE =
sig

  type 'a t
  type 'a seq = 'a t     (* abstract *)

  exception Range of string


  (* Constructing a Sequence *)

  val empty : unit -> 'a seq
  val singleton : 'a -> 'a seq
  val tabulate : (int -> 'a) -> int -> 'a seq
  val fromList : 'a list -> 'a seq


  (* Deconstructing a Sequence *)

  val nth : 'a seq -> int -> 'a
  val null : 'a seq -> bool
  val length : 'a seq -> int
  val toList : 'a seq -> 'a list
  val toString : ('a -> string) -> 'a seq -> string
  val equal : ('a * 'a -> bool) -> 'a seq * 'a seq -> bool


  (* Simple Transformations *)

  val rev : 'a seq -> 'a seq
  val append : 'a seq * 'a seq -> 'a seq
  val flatten : 'a seq seq -> 'a seq
  val cons : 'a -> 'a seq -> 'a seq


  (* Combinators and Higher-Order Functions *)

  val filter : ('a -> bool) -> 'a seq -> 'a seq
  val map : ('a -> 'b) -> 'a seq -> 'b seq
  val reduce : ('a * 'a -> 'a) -> 'a -> 'a seq -> 'a
  val reduce1 : ('a * 'a -> 'a) -> 'a seq -> 'a
  val mapreduce : ('a -> 'b) -> 'b -> ('b * 'b -> 'b) -> 'a seq -> '
   b
  val zip : ('a seq * 'b seq) -> ('a * 'b) seq
  val zipWith : ('a * 'b -> 'c) -> 'a seq * 'b seq -> 'c seq


  (* Indexing-Related Functions *)
```

```
  val enum : 'a seq -> (int * 'a) seq
  val mapIdx : (int * 'a -> 'b) -> 'a seq -> 'b seq
  val update : ('a seq * (int * 'a)) -> 'a seq
  val inject : 'a seq * (int * 'a) seq -> 'a seq

  val subseq : 'a seq -> int * int -> 'a seq
  val take : 'a seq -> int -> 'a seq
  val drop : 'a seq -> int -> 'a seq
  val split : 'a seq -> int -> 'a seq * 'a seq


  (* Sorting and Searching *)

  val sort : ('a * 'a -> order) -> 'a seq -> 'a seq
  val merge : ('a * 'a -> order) -> 'a seq * 'a seq -> 'a seq
  val search : ('a * 'a -> order) -> 'a -> 'a seq -> int option


  (* Views *)

  datatype 'a lview = Nil | Cons of 'a * 'a seq

  val showl : 'a seq -> 'a lview
  val hidel : 'a lview -> 'a seq

  datatype 'a tview = Empty | Leaf of 'a | Node of 'a seq * 'a seq

  val showt : 'a seq -> 'a tview
  val hidet : 'a tview -> 'a seq

end
```

3 Documentation

If unspecified, we assume that all functions that are given as arguments (such as the `g` in `reduce g`) have $O(1)$ work and span. In order to analyze the runtime of sequence functions when this is not the case, we need to analyze the corresponding cost graphs.

Constraint: Whenever you use these sequence functions, please make sure you meet the specified preconditions.

If you do not meet the precondition for a function, it may not behave as expected or meet the cost bounds stated below.

Definition 3.1 (Associative)**.** Fix some type `t`. We say a function `g : t * t -> t` is *associative* if for all `a`, `b`, and `c` of type `t`:

$$\texttt{g (g (a,b),c)} \cong \texttt{g (a,g (b,c))}$$

Definition 3.2 (Identity)**.** Fix some type `t`. Given a function `g : t * t -> t`, we say `z` is the *identity* for `g` if for all `x : t`:

$$\texttt{g (x,z)} = \texttt{g (z,x)} = \texttt{x}$$

3.1 Constructing a Sequence

```
empty : unit -> 'a seq
```

ENSURES: `empty ()` evaluates to the empty sequence (the sequence of length zero).

Work: $O(1)$, Span: $O(1)$.

```
singleton : 'a -> 'a seq
```

ENSURES: `singleton x` evaluates to a sequence of length 1 whose only element is `x`.

Work: $O(1)$, Span: $O(1)$.

```
tabulate : (int -> 'a) -> int -> 'a seq
```

REQUIRES: For all $0 \leq$ `i` $<$ `n`, `f i` is valuable.

ENSURES: `tabulate f n` evaluates to a sequence `S` of length `n`, where the `i`th element of `S` is equal to `f i`.

Note that indices are zero-indexed.

Raises `Range` if `n` is less than zero.

Work: $O(\mathtt{n})$, Span: $O(1)$, with constant-time `f`.

```
fromList : 'a list -> 'a seq
```

ENSURES: `fromList L` returns a sequence consisting of the elements of `L`, preserving order. This function is intended primarily for debugging purposes.

Work: $O(|\mathtt{L}|)$, Span: $O(|\mathtt{L}|)$.

3.2 Deconstructing a Sequence

```
nth : 'a seq -> int -> 'a
```

ENSURES: `nth S i` evaluates to the i^{th} element (zero-indexed) of `S`. Raises `Range` if `i` is negative or greater than or equal to `length S`.

Work: $O(1)$, Span: $O(1)$.

```
null : 'a seq -> bool
```

ENSURES: `null S` evaluates to `true` if `S` is an empty sequence, and `false` otherwise.

Work: $O(1)$, Span: $O(1)$.

```
length : 'a seq -> int
```

ENSURES: `length S` (often written as $|S|$) evaluates to the number of items in `S`.

Work: $O(1)$, Span: $O(1)$.

```
toList : 'a seq -> 'a list
```

ENSURES: `toList S` returns a list consisting of the elements of `S`, preserving order. This function is intended primarily for debugging purposes.

Work: $O(|S|)$, Span: $O(|S|)$.

```
toString : ('a -> string) -> 'a seq -> string
```

REQUIRES: `ts x` evaluates to a value for all elements `x` of `S` (e.g. if `ts` is total).

ENSURES: `toString ts S` evaluates to a string representation of `S`, using the function `ts` to convert each element of `S` into a string.

Work: $O(|S|)$, Span: $O(\log|S|)$.

```
equal : ('a * 'a -> bool) -> 'a seq * 'a seq -> bool
```

REQUIRES: `eq` is total.

ENSURES: `equal eq (S1,S2)` returns whether or not the two sequences are equal according to the equality function `eq`.

Work: $O(\min(|S1|,|S2|))$, Span: $O(\log(\min(|S1|,|S2|)))$.

3.3 Simple Transformations

`rev : 'a seq -> 'a seq`

ENSURES: `rev S` returns the sequence containing the elements of `S` in reverse order.

Work: $O(|\texttt{S}|)$, Span: $O(1)$.

`append : 'a seq * 'a seq -> 'a seq`

ENSURES: `append (S1,S2)` evaluates to a sequence of length $|\texttt{S1}| + |\texttt{S2}|$ whose first $|\texttt{S1}|$ elements are the sequence `S1` and whose last $|\texttt{S2}|$ elements are the sequence `S2`.

Work: $O(|\texttt{S1}| + |\texttt{S2}|)$, Span: $O(1)$.

`flatten : 'a seq seq -> 'a seq`

ENSURES: `flatten S` flattens a sequence of sequences down to a single sequence (similar to `List.concat` for lists).

Work: $O\left(|\texttt{S}| + \sum_{\texttt{s} \in \texttt{S}} |\texttt{s}|\right)$, Span: $O(\log |\texttt{S}|)$.

`cons : 'a -> 'a seq -> 'a seq`

ENSURES: If the length of `S` is `n`, `cons x S` evaluates to a sequence of length `n+1` whose first item is `x` and whose remaining `n` items are exactly the sequence `S`.

Constraint: Beware of using `cons`. When overused, it often leads to a sequential coding style with little opportunity for parallelism, defeating the purpose of using sequences. See if there's a way to use other sequence functions to achieve your objective.

Work: $O(|\texttt{S}|)$, Span: $O(1)$.

3.4 Combinators and Higher-Order Functions

```
filter : ('a -> bool) -> 'a seq -> 'a seq
```

REQUIRES: `p x` evaluates to a value for all elements `x` of `S` (e.g. if `p` is total)

ENSURES: `filter p S` evaluates to a sequence containing all of the elements `x` of `S` such that `p x` $\Longrightarrow$ `true`, preserving element order.

Work $\sum_{x \in \mathtt{S}} W_{\mathtt{p}}(x)$, Span $O(\log |\mathtt{S}|) + \max_{x \in \mathtt{S}} S_{\mathtt{p}}(x)$.

Work $O(|\mathtt{S}|)$, Span $O(\log |\mathtt{S}|)$, with constant-time `p`.

```
map : ('a -> 'b) -> 'a seq -> 'b seq
```

REQUIRES: `f x` evaluates to a value for all elements `x` of `S` (e.g. if `f` is total).

ENSURES: `map f S` $\Longrightarrow$ `S'` such that $|\mathtt{S}| = |\mathtt{S'}|$ and for all $0 \le \mathtt{i} < |\mathtt{S'}|$, `nth S' i` $\cong$ `f (nth S i)`.

Work $\sum_{\mathtt{x} \in \mathtt{S}} W_{\mathtt{f}}(\mathtt{x})$, Span $\max_{x \in \mathtt{S}} S_{\mathtt{f}}(\mathtt{x})$.

Work $O(|\mathtt{S}|)$, Span $O(1)$, with constant-time `f`.

```
reduce : ('a * 'a -> 'a) -> 'a -> 'a seq -> 'a
```

REQUIRES:

- `g` is total and associative

In other courses, like 15-210, they likely require that `z` is an identity for `g`. However, this is not a requirement for our implementation of `reduce`.

ENSURES: `reduce g z S` uses the function `g` to combine the elements of `S` using `z` as a base case (analogous to `foldr g z L` for lists, but with a less-general type).

Work: $O(|\mathtt{S}|)$, Span: $O(\log |\mathtt{S}|)$, with constant-time `g`.

```
reduce1 : ('a * 'a -> 'a) -> 'a seq -> 'a
```

REQUIRES: `g` is total and associative.

ENSURES: `reduce1 g S` uses the function `g` to combine the elements of `S`. If `S` is a singleton sequence, the sequence element is returned. Raises `Range` if `S` is empty.

Work: $O(|\mathtt{S}|)$, Span: $O(\log |\mathtt{S}|)$, with constant-time `g`.

```
mapreduce : ('a -> 'b) -> 'b -> ('b * 'b -> 'b) -> 'a seq -> 'b
```

REQUIRES: `g` and `f` meet the preconditions of `reduce` and `map`, respectively.

ENSURES: `mapreduce f z g S` $\cong$ `reduce g z (map f S)`

Work: $O(|\texttt{S}|)$, Span: $O(\log|\texttt{S}|)$, with constant-time `g` and `f`.

```
zip : 'a seq * 'b seq -> ('a * 'b) seq
```

ENSURES: `zip (S1,S2)` evaluates to a sequence of length min(`length S1`, `length S2`) whose $\texttt{i}^{\text{th}}$ element is the pair of the $\texttt{i}^{\text{th}}$ element of `S1` and the $\texttt{i}^{\text{th}}$ element of `S2`.

Work: $O(\min(|\texttt{S1}|,|\texttt{S2}|))$, Span: $O(1)$.

```
zipWith : ('a * 'b -> 'c) -> 'a seq * 'b seq -> 'c seq
```

ENSURES: `zipWith f (S1,S2)` $\cong$ `map f (zip (S1,S2))`.

Work $\sum_{i=0}^{min(|\texttt{S1}|,\texttt{S2}|)-1} W_{\texttt{f}}(\texttt{S1}[i],\texttt{S2}[i])$, Span $\max_{i=0}^{min(|\texttt{S1}|,\texttt{S2}|)-1} S_{\texttt{f}}(\texttt{S1}[i],\texttt{S2}[i])$.

Work $O(\min(|\texttt{S1}|,|\texttt{S2}|))$, Span $O(1)$, with constant-time `f`.

3.5 Indexing-Related Functions

`enum : 'a seq -> (int * 'a) seq`

ENSURES: `enum S` evaluates to a sequence such that for each index $0 \leq$ `i` $<$ `length S`, the `i`$^{\text{th}}$ index of the result is `(i,nth S i)`.

Work: $O(|\mathtt{S}|)$, Span: $O(1)$.

`mapIdx : (int * 'a -> 'b) -> 'a seq -> 'b seq`

ENSURES: `mapIdx f S` $\cong$ `map f (enum S)`.

Work $\sum_{\mathtt{x}\in\mathtt{S}} W_{\mathtt{f}}(\mathtt{x})$, Span $\max_{x\in\mathtt{S}} S_{\mathtt{f}}(\mathtt{x})$.

Work $O(|\mathtt{S}|)$, Span $O(1)$, with constant-time `f`.

`update : 'a seq * (int * 'a) -> 'a seq`

ENSURES: `update (S,(i,x))` returns a sequence identical to `S` but with the `i`$^{\text{th}}$ element (0-indexed) now `x` if $0 \leq$ `i` $<$ `length(S)`, and raises `Range` otherwise.

Work: $O(|\mathtt{S}|)$, Span: $O(1)$.

`inject : 'a seq * (int * 'a) seq -> 'a seq`

ENSURES: `inject (S,U)` evaluates to a sequence where for each `(i,x)` in `U`, the `i`$^{\text{th}}$ element of `S` is replaced with `x`. If there are multiple elements at the same index, one is chosen nondeterministically. If any indices are out of bounds, raises `Range`.

Work: $O(|\mathtt{S}| + |\mathtt{U}|)$, Span: $O(1)$.

`subseq : 'a seq -> int * int -> 'a seq`

ENSURES: `subseq S (i,l)` takes the subsequence of `S` with length `l` starting at index `i`. If any indices are out of bounds, raises `Range`.

Work: $O(1)$, Span: $O(1)$.

`take : 'a seq -> int -> 'a seq`

ENSURES: `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.

Work: $O(1)$, Span: $O(1)$.

`drop : 'a seq -> int -> 'a seq`

ENSURES: `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.

Work: $O(1)$, Span: $O(1)$.

`split : 'a seq -> int -> 'a seq * 'a seq`

ENSURES: `split S i` evaluates to a pair of sequences `(S1,S2)` such that `S1` has length `i` and `append (S1,S2)` $\cong$ `s` if $0 \leq$ `i` $\leq$ `length S`, and raises `Range` otherwise.

Work: $O(1)$, Span: $O(1)$.

3.6 Sorting and Searching

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

REQUIRES: `cmp` is total.

ENSURES: `sort cmp S` returns a permutation of `S` that is sorted according to `cmp`. The sort is stable: elements that are considered equal by `cmp` remain in the same order they were in `S`.

Work: $O(|\texttt{S}| \log |\texttt{S}|)$, Span: $O\left(\log^2 |\texttt{S}|\right)$, with constant-time `cmp`.

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

REQUIRES:

- `S1` and `S2` are both `cmp`-sorted.
- `cmp` is total.

ENSURES: `merge cmp (S1,S2)` returns a sorted permutation of `append (S1,S2)` .

Work: $O(|\texttt{S1}| + |\texttt{S2}|)$, Span: $O(\log(|\texttt{S1}| + |\texttt{S2}|))$, with constant-time `cmp`.

```
search : ('a * 'a -> order) -> 'a -> 'a seq -> int option
```

REQUIRES:

- `cmp` is total.
- `S` is `cmp`-sorted.

ENSURES: `search cmp x S` $\Longrightarrow$ `SOME i` where `i` is the first index in `S` satisfying `cmp (nth i S, x)` $\cong$ `EQUAL` or `NONE` if no such index exists.

Work: $O(\log |\texttt{S}|)$, Span: $O(\log |\texttt{S}|)$, with constant-time `cmp`.

4 Views

4.1 List Views

Recall that list operations have bad parallel complexity, whereas the corresponding sequence operations are much better.

However, sometimes you want to write a ***sequential*** algorithm (e.g., because the inputs aren't very big, or because no good parallel algorithms are known for the problem). Given the sequence interface so far, it is difficult to decompose a sequence as "either empty, or a cons with a head and a tail." You *could* write this jank codemonkey's "length induction instead of structural induction"...

```
case Seq.length s of
  0 =>
| _ => ... uses (Seq.hd s) and (Seq.tl s) ...
```

But nah. We can solve this problem using a ***view***. We'll put an appropriate datatype in the signature, along with corresponding functions that convert sequences to and from this datatype. This allows us to pattern-match on an abstract type, while keeping the actual representation abstract. These definitions enable viewing a sequence like a list:

```
datatype 'a lview = Nil | Cons of 'a * 'a seq

val showl : 'a seq -> 'a lview
val hidel : 'a lview -> 'a seq
```

Note the invariant:

$$\texttt{showl (hidel v)} \Longrightarrow \texttt{v}$$

Because the datatype definition is in the signature, the constructors `Nil` and `Cons` can be used outside the abstraction boundary. The `showl` and `hidel` functions convert between sequences and list views. The following is an example of using this view to perform list-like pattern matching:

```
case Seq.showl s of
  Seq.Nil => ... (* Nil case *)
| Seq.Cons (x,s') => ... uses x and s' ... (* Cons case *)
```

Note that the second argument to `Cons` is another `'a seq`, ***not*** an `lview`. Thus, `showl` lets you do one level of pattern matching at a time: you can write patterns like `Seq.Cons (x,xs)` but not `Seq.Cons (x,Seq.Nil)` (to match a sequence with exactly one element).

We have also provided `hidel`, which converts a view back to a sequence—`Seq.hidel (Seq.Cons (x,xs))` is equivalent to `Seq.cons(x,xs)` and `Seq.hidel Seq.Nil` is equivalent to `Seq.empty()`.

4.2 Tree Views

The analogous `'a tview`, `showt`, and `hidet` are provided in the signature.

5 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:

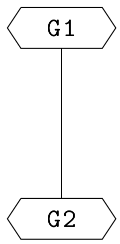
1. The code itself must not bake in a schedule.
2. You must be able to reason about the *behavior* of your code independently of the schedule.
3. You must be able to reason about the *time complexity* of your code independently of the schedule.

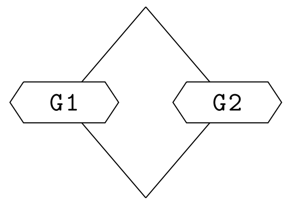
Our central tool for avoiding baking in a schedule is *functional programming*. First, we focus on bulk operations on big collections which do not specify a particular order in which the operations on each element are performed. For example, sequences come with an operation `map` that is specified by saying that the value of `map f <x1,x2,...,xn>` 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 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, 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.

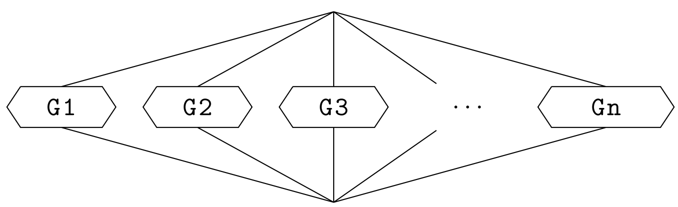
Cost Semantics Reminder

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:





Depicted on the left is *sequential combination*: the graph formed by putting an edge from the sink of `G1` to the source of `G2`. The other, *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.

6 Cost Graphs

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, such as `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.

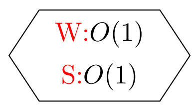
Note: $\langle x_1, ..., x_n \rangle$ is **not** SML syntax. It is mathematical syntax which we will use to represent a sequence value.

Length

Behavior of `length`:

$$\texttt{length}\ \langle x_1, ..., x_n \rangle \cong \texttt{n}$$

Cost Graph for `length s`:



As a consequence,

Work: $O(1)$

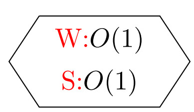
Span: $O(1)$

Nth

Behavior of `nth`:

$$\texttt{nth}\ \langle x_0, ..., x_{n-1} \rangle\ \texttt{i} \cong x_i \text{ if } 0 \leq \texttt{i} < n \text{ or raises } \texttt{Range} \text{ otherwise}$$

Cost Graph for `nth s i`:



As a consequence,

Work: $O(1)$

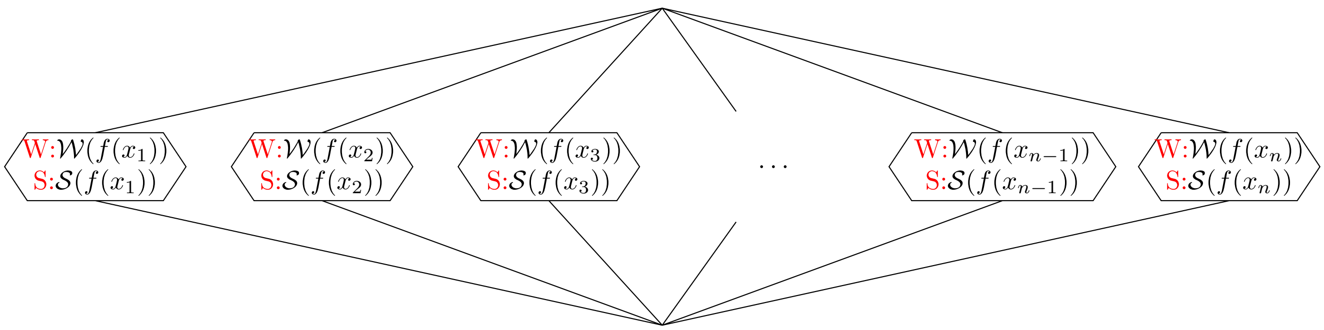
Span: $O(1)$

Map

Behavior of `map`:

$$\texttt{map f } \langle x_1, ..., x_n \rangle \cong \langle \texttt{f } x_1, ..., \texttt{f } x_n \rangle$$

Cost Graph for `map f s`:



where `s` is $\langle x_1, ..., x_n \rangle$.
As a consequence,

Work:

$$\sum_{x \in s} \mathcal{W}(f(x))$$

Span:

$$\max_{x \in s} \mathcal{S}(f(x))$$

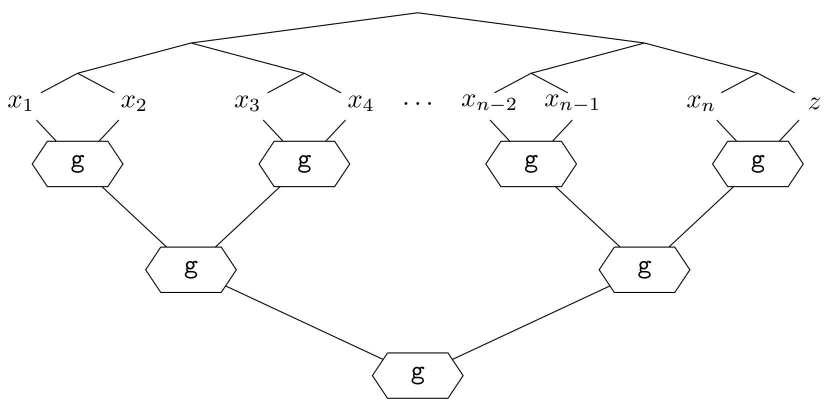
Reduce

The behavior of `reduce` is

$$\texttt{reduce g z S} \cong \texttt{List.foldr g z (toList S)}$$

That is, `reduce` applies its argument function `g` between every pair of elements in the sequence, using `z` as a rightmost base case.

Cost Graph for `reduce g z` $\langle x_1, \ldots x_n \rangle$:



Observe from the above graph that, – if we assume that `g` is $O(1)$ – then `reduce` has the following time bounds.

Work: $O(n)$

Span: $O(\log n)$

where n is the length of the sequence. If `g` does not have constant work and span, then refer directly to the cost graph above.

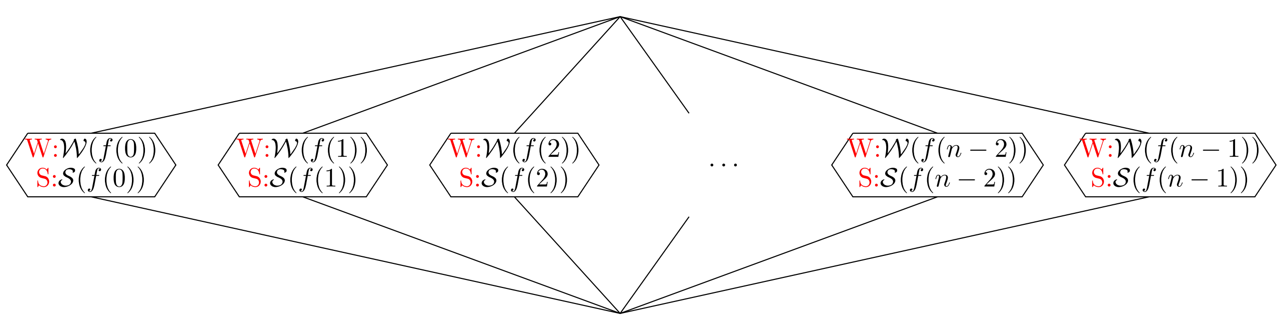
Tabulate

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.
Behavior of `tabulate f n`:

$$\texttt{tabulate f n} \cong \langle v_0, ..., v_{n-1} \rangle$$

$$\begin{aligned} \texttt{where} \qquad & \texttt{f 0} \cong v_0 \\ & \texttt{f 1} \cong v_1 \\ & \vdots \\ & \texttt{f (n-1)} \cong v_{n-1} \end{aligned}$$

Cost Graph for `tabulate f n`:



As a consequence,

Work:

$$\sum_{i=0}^{n-1} \mathcal{W}(f(i))$$

Span:

$$\max_{i=0}^{n-1} \mathcal{S}(f(i))$$