Chapter 5

Sequences

A sequence is an ordered set, i.e., is a collection of elements that are totally ordered. We write a sequence by listing their elements from left to right according to their order demarcated by left and right angle brackets. For example, $\langle a, b, c \rangle$ is a sequence where a is the first, b is the second, and c is the third element. A sequence can be finite or (countably) infinite, as in $\langle 0, 1, 2, 3 \ldots \rangle$. In this course , however, we mostly consider finite sequences.

We use sequences to represent many different kinds of data, but it is important not to associate a sequence with a particular implementation—e.g., with an array of contiguous locations in the memory of the machine. Instead we will think of sequences abstractly in terms of their mathematical properties, and the functions they support. By thinking at this level of abstraction, we can use sequences without committing a particular implementation. This allows us for example to choose an implementation that suits our needs best. For example, we will consider three different implementations of sequences, one based on arrays, one based on linked-lists, and the third based on trees.

5.1 Defining Sequences

We can define sequences mathematically as shown in Definition 5.1 . As can be seen in the definition, we have a special term "ordered pairs", or simply as "pairs" for sequences with only two elements, because they arise frequently.

Definition 5.1. *An* **ordered pair** (a, b) *is a pair of elements in which the element on the left,* a*, is identified as the* **first** *entry, and the one on the right,* b*, as the* **second** *entry.*

An α **sequence** *is a mapping (function) from* $\mathbb{N}$ *to* α *with domain* $\{0, \ldots, n-1\}$ *for some* $n \in \mathbb{N}$ *(we use* $\mathbb{N}$ *to indicate the natural numbers, including zero). Traditionally sequences are indexed from* 1 *not* 0*, but being computer scientists, we violate the tradition here.*

This mathematical definition might seem pedantic, and is probably not necessary to go through in detail now, but it is useful for at least a couple reasons—it allows for a concise but yet precise definition of the semantics of the functions on sequences, and we will see, relating sequences to mappings will make a nice symmetry with the operations on mappings (also called tables or dictionaries). One thing to notice in the definition is that sequences are parametrized on the type (i.e. set of possible values) of their elements.

Example 5.2. *Let $A = \{0, 1, 2, 3\}$ and $B = \{a, b, c\}$. The relation*

$$R = \{(0, a), (1, b), (3, a)\}$$

is a function from A to B since each element only appears once on the left. It is, however, not a sequence since there is a gap in the domain.
The relation

$$Z = \{(1, b), (3, a), (2, a), (0, a)\}$$

from N to B is a sequence since it is a function with domain $\{0, \ldots, 3\}$.

5.2 Specification: A Sequence ADT

Having defined sequences, we are now ready to start looking at operations on sequences that enable us to perform interesting computations with them. ADT 5.3 shows the specification of an abstract data type for sequences using the mathematical definition of sequences shown in Definition 5.1. In our specification, we use a few syntactic conventions: for a sequence A, we write $A[i]$ to refer to the element of A at position i and $A[l...h]$ to refer to the subsequnce of A restricted to the position between l and h. The ADT can be broadly divided into several categories.

- Constructor functions that create sequences such as `empty`, `singleton`, and `tabulate`.
- Operations such as `nth` and `length` that return an element of the sequence or a particular property of it.
- Operations such as `tabulate`, `map`, `filter` that operate on each element of a sequence independently in parallel.
- Operations such as `append` and `flatten` that operate on sequences as a whole.
- Operations such as `iterate`, `reduce`, and `scan` that *aggregate* information over the elements of the sequence.

In this chapter, we present this ADT, give examples, and discuss their cost specification.

Abstract Data Type 5.3 (Sequences). *For a value type α, the* **sequence data type** *is the type $\mathbb{S}_\alpha$ consisting the set of all α sequences, and the following values and functions on $\mathbb{S}_\alpha$:*

$$
\begin{array}{lcl}
\texttt{empty} & : & \mathbb{S}_\alpha = \{\} \\
\texttt{singleton}(v) & : & \alpha \to \mathbb{S}_\alpha = \{(0, v)\} \\
\texttt{tabulate}(f, n) & : & (\mathbb{N} \to \alpha) \times \mathbb{N} \to \mathbb{S}_\alpha = \{(i, f(i)) : 0 \leq i < n\} \\
\texttt{length}(A) & : & \mathbb{S}_\alpha \to \mathbb{N} = |A| \\
\texttt{nth}(A, i) & : & \mathbb{S}_\alpha \times \mathbb{N} \to (\alpha \cup \{\bot\}) = \begin{cases} v & (i, v) \in A \\ \bot & \textit{otherwise} \end{cases} \\
\texttt{map}(f, A) & : & (\alpha \to \beta) \times \mathbb{S}_\alpha \to \mathbb{S}_\beta = \{(i, f(v)) : (i, v) \in A\} \\
\texttt{subseq}(A, s, l) & : & \mathbb{S}_\alpha \times \mathbb{N} \times \mathbb{N} \to \mathbb{S}_\alpha = \{(i - s, v) : (i, v) \in A \mid s \leq i < s + l\} \\
\texttt{append}(A, B) & : & \mathbb{S}_\alpha \times \mathbb{S}_\alpha \to \mathbb{S}_\alpha = A \cup \{(i + |A|, v) : (i, v) \in B\} \\
\texttt{filter}(f, A) & : & (\alpha \to \mathbb{B}) \times \mathbb{S}_\alpha \to \mathbb{S}_\alpha \\
 & = & \{(|\{(j, x) \in A \mid j < i \wedge f(x)\}|, v) : (i, v) \in A \mid f(v)\} \\
\texttt{flatten}(A) & : & \mathbb{S}_{\mathbb{S}_\alpha} \to \mathbb{S}_\alpha \\
 & = & \{(i + \sum_{(k,X) \in A, k<j} |X|, v) \ : (i, v) \in Y, (j, Y) \in A\} \\
\texttt{update}(A, (j, v)) & : & \mathbb{S}_\alpha \times (\mathbb{N} \times \alpha) \to \mathbb{S}_\alpha \\
 & = & \{(i, x) : (i, v) \in A\} \ \ x = \begin{cases} y & i = j \\ v & \textit{otherwise} \end{cases} \\
\texttt{inject}(A, P) & : & \mathbb{S}_\alpha \times \mathbb{S}_{\mathbb{N} \times \alpha} \to \mathbb{S}_\alpha \\
 & = & \{(i, x) : (i, v) \in A\}, \textit{where}, x = \begin{cases} y & (j, i, y) \in P \\ v & \textit{otherwise} \end{cases} \\
\texttt{iterate}\ f\ v\ A & : & (\alpha \times \beta \to \alpha) \to \alpha \to \mathbb{S}_\beta \to \alpha \\
 & = & \begin{cases} v & \textit{if } |A| = 0 \\ f(v, A[0]) & \textit{if } |A| = 1 \\ \texttt{iterate}\ f\ (f(v, A[0]))(A[1 \ldots |A| - 1]) & \textit{otherwise} \end{cases} \\
\texttt{reduce}\ f\ I\ A & : & (\alpha \times \alpha \to \alpha) \to \alpha \to \mathbb{S}_\alpha \to \alpha \\
 & = & \begin{cases} I & \textit{if } |A| = 0 \\ A[0] & \textit{if } |A| = 1 \\ f(A[0], A[1]) & \textit{if } |A| = 2 \\ f(\texttt{reduce}\ f\ I(A[0 \ldots \lceil |A|/2 \rceil]), & \\ \quad \texttt{reduce}\ f\ I(A[\lceil |A|/2 \rceil + 1, |A| - 1]) & \textit{otherwise} \end{cases} \\
\texttt{scan}\ f\ I\ A & : & (\alpha \times \alpha \to \alpha) \to \alpha \to \mathbb{S}_\alpha \to (\mathbb{S}_\alpha \times \alpha) \\
 & = & (\texttt{tabulate}\ (\texttt{fn}\ i \texttt{=>}\ \texttt{reduce}\ f\ I\ (A[0 \ldots i]))(|A| - 1), \\
 & & \ \texttt{reduce}\ f\ I\ A)
\end{array}
$$

where $\mathbb{B} = \{true, false\}$. The additional functions `iter`*,* `iterh`*,* `reduce`*, and* `scan` *are defined later.*

5.3 Syntax: Sequence Comprehension

Writing sequences as a set of integer value pairs becomes tiresome, so in this course we use triangle brackets to indicate sequences as defined by the following syntax.

Syntax 5.4 (Sequences)**.** *We use*

$$\langle\, s_0, s_1, \ldots, s_{n-1} \,\rangle$$

as shorthand for the sequence $\{(0, s_0), (1, s_1), \ldots, ((n-1), s_{n-1})\}$.
A character sequence is called a **string** *and we use the standard syntax of placing the characters between double quotes with no spaces or commas:*

$$\texttt{``}c_0c_1c_2c_{n-1}\texttt{''}.$$

Example 5.5. *Some example sequences:*

- *An integer sequence (or* $\mathbb{Z}$ *sequence):* $\langle\, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 \,\rangle$.
- *A character sequence, or a string:* $\langle\, \texttt{'s}, \texttt{'e}, \texttt{'q} \,\rangle \equiv$ `''seq''` .
- *An (integer* $\times$ *string) sequence:* $\langle\, (10,$ `''ten''`$), (1,$ `''one''`$), (2,$ `''two''`$) \,\rangle$.
- *A (string sequence) sequence:* $\langle\, \langle$ `''a''` $\rangle, \langle$ `''nested''`, `''sequence''` $\rangle \,\rangle$.
- *A function sequence, or more specifically a* $(\mathbb{Z} \to \mathbb{Z})$ *sequence:*

$$\langle\, (\textbf{fn}\ x \Rightarrow x^2), (\textbf{fn}\ y \Rightarrow y + 2), (\textbf{fn}\ x \Rightarrow x - 4) \,\rangle\ .$$

Sequences is one of the most prevalent ADT's used in this course , and more generally in computer science. For economy in writing, communication, and cognition, we have developed a syntax for our sequences ADT. Syntax 5.7 illustrates this syntax. In the rest chapter, we briefly explain the syntax for these operations as we describe them.

Remark 5.6 (Comprehensions)**.** *Notation such as* $\{x^2 : x \in A \mid \texttt{isPrime}(x)\}$ *in which one set is defined in terms of the elements of other sets, and conditions on them is referred to as a set* **comprehension**. *The example can be read as: the set of squares of the primes in the set* A*. Comprehensions are commonly used in mathematics, because of the wonderful economy of expression and "comprehension" that they offer.*

In this book we make heavy use of comprehension syntax. Specifically, the syntax for sequences is based on set comprehensions, taking of course the relatively close correspondence between sets and sequences. For brevity, the comprehension notation on sequences omits the ordering constraints but this is intuitively easy to see based on the usage. For completeness, we specify the ordering when defining the syntax.

Syntax based on set comprehensions is included in many programming languages either directly for sets (e.g. SETL), or for other collections of values such as lists, sequences, or mappings (e.g. Python, Haskell and Javascript). We should note, however, that the syntax is not uniform among the languages. Indeed even among texts on set theory in mathematics the syntax for set comprehensions varies significantly. In our usage, we try to be self consistent, but necessarily we are not always consistent with usage found elsewhere. To be precise we always view comprehensions as syntactic sugar for some specific function, and always define the translation between the two, as we do in Syntax 5.7).

5.4 Computing with Sequences

We describe the operations on sequences via examples

Empty and singleton. To construct a sequence, we can use the function `empty` which returns and empty sequence, and the function `singleton` which takes an element and returns a sequence containing that element.

Tabulate. To generate a larger sequence, we can use the `tabulate` function. Given a function f and a natural number n, `tabulate` f n generates a sequence of length n consisting of $\langle f(0), f(1), \ldots, f(n-1) \rangle$. Since, `tabulate` can apply the function f at each number independently, it can evaluate in parallel.

Syntax 5.7 (Syntax for Sequences)**.** *The table below defines the syntax for the sequence ADT used throughout this course . In the definition i is a variable ranging over natural numbers, x is a variable ranging over the elements of a sequence, e is a PML expression, e_n and e'_n are PML expressions whose values are natural numbers, e_s is a PML expression whose value is a sequence, p is a pattern that binds one or more variables.*

$\langle\,\rangle$	$\equiv$	`empty`
$\langle\, e\,\rangle$	$\equiv$	`singular` (e)
$\langle\, e : 0 \le i < e_n\,\rangle$	$\equiv$	`tabulate` (**fn** $i \Rightarrow e$) e_n
$\lvert S\rvert$	$\equiv$	`length` (S)
$S[i]$	$\equiv$	`nth` (S)
$\langle\, e : p \in e_s\,\rangle$	$\equiv$	`map` (**fn** $p \Rightarrow e$) e_s
$\langle\, x \in e_s \mid e\,\rangle$	$\equiv$	`filter` (**fn** $x \Rightarrow e$) e_s
$A[e_l, \cdots, e'_n]$	$\equiv$	`subseq` $(A, e_l, e'_n - e_n + 1)$
$\overline{\prod}^{f\ v}_{x \in A}\ g(x)$	$\equiv$	`iterate` $f\ v$ (`map` $g\ A$)
$\overline{\prod}^{f\ v}\ A$	$\equiv$	`iterate` $f\ v\ A$
$\overline{\sum}^{f\ I}_{x \in A}\ g(x)$	$\equiv$	`reduce` $f\ I$ (`map` $g\ A$)
$\overline{\sum}^{f\ I}\ A$	$\equiv$	`reduce` $f\ I\ A$
$\overline{\int}^{f\ I}_{x \in A}\ g(x)$	$\equiv$	`scan` $f\ I$ (`map` $g\ A$)
$\overline{\int}^{f\ I}\ A$	$\equiv$	`scan` $f\ I\ A$
$\overline{\oint}^{f\ I}_{x \in A}\ g(x)$	$\equiv$	`scani` $f\ I$ (`map` $g\ A$)
$\overline{\oint}^{f\ I}\ A$	$\equiv$	`scani` $f\ I\ A$

Syntax 5.8 (Tabulate)**.**

$$\langle\, e : 0 \le i < e_n\,\rangle \equiv \texttt{tabulate}\ (\textbf{fn}\ i \Rightarrow e)\ e_n$$

where e and e_n are expressions, the second evaluating to an integer, and i is a variable. We can also start at a number other than 0, as in:

$$\langle\, e : e_l \le i < e_h\,\rangle\ .$$

Example 5.9. *Given the Fibonacci function* $\texttt{fib}(i)$*, the expression:*

$$\langle\, \texttt{fib}(i) : 0 \leq i < 9 \,\rangle$$

is equivalent to:

$$\texttt{tabulate fib } 9$$

and when evaluated returns the sequence:

$$\langle\, 0, 1, 1, 2, 5, 8, 13, 21, 34 \,\rangle\ .$$

Length and indexing. The function `length` returns the length of a given sequence and the function `nth` returns the element of a sequence at a specified index. Of course the index demanded might be out of range if for example it is less than 0 or greater or equal to the length of the sequence. In this case, the function returns the special value $\perp$, as described before, which indicates an error (exception). The syntax for these function is more or less standard. as shown in Syntax 5.7.

Map. A common operation on sequences is doing something with every element of a sequence. For example we might want to add five to each element of a sequence. For this purpose we supply a $\texttt{map}\ f\ A$ function that applies the function f to each element of A returning a sequence of equal length with the results.

For mapping over sequences, we use special syntax inspired by the mathematical notation on sequences.

Syntax 5.10 (Map)**.**

$$\langle\, e : p \in e_s \,\rangle \equiv \texttt{map}\ (\textbf{fn}\ p \Rightarrow e)\ e_s$$

where e *and* e_s *are expressions, the second evaluating to a sequence, and* p *is a a pattern of variables (e.g.* x *or* (x, y)*).*

Example 5.11. *Given the integer sequence* $A = \langle\, 9, -1, 4, 11, 13, 2 \,\rangle$*, the expression:*

$$\langle\, x^2 : x \in A \,\rangle$$

is equivalent to:

$$\texttt{map}\ (\textbf{fn}\ x \Rightarrow x^2)\ A$$

and when evaluated returns the sequence:

$$\langle\, 81, 1, 16, 121, 169, 4 \,\rangle\ .$$

As with `tabulate`, in `map` the function f can be applied to all the elements of the sequence in parallel. As we will see in the cost model, this means the span of the function is the maximum of the spans of the function applied at each location, instead of the sum. We will also see that `map` generalizes to arbitrary mappings, not just sequences.

Question 5.12. *Can we implement `map` by using tabulate?*

The function `map` can easily be implemented using tabulated as follows:

$$\texttt{map}\ f\ A\ =\ \texttt{tabulate}\ (\textbf{fn}\ i\ \Rightarrow\ f(\texttt{nth}(A, i)))\ (\texttt{length}\ A)$$

or equivalently in our sequence notation as

$$\texttt{map}\ f\ A\ =\ \langle\, f(A[i]) : 0 \leq i < |A| \,\rangle$$

But, as we will see, this is not always an efficient way to implement `map`.

Filter. It is often useful to subselect the elements from a sequences that satisfy some predicate. For example, in quick sort (Chapter 9) we will want all the elements that are less (or greater) than a pivot element. For this we can use the $\texttt{filter}\ f\ A$ function that applies a Boolean function f to each element of A, and returns the sequence consisting exactly of those elements of $s \in A$ for which $f(s)$ returns true. These elements maintain their order.

Syntax 5.13 (Filter)**.**

$$\langle\, x \in e_s \mid e \,\rangle \equiv \texttt{filter}\ (\textbf{fn}\ x \Rightarrow e)\ e_s$$

It is important to note the distinction between the colon (:) and the bar (|) in the syntax. They can be used together, as in:

$$\langle\, e : x \in e_s \mid e_f \,\rangle \equiv \texttt{map}\ (\textbf{fn}\ x \Rightarrow e)\ (\texttt{filter}\ (\textbf{fn}\ x \Rightarrow e_f)\ e_s)\ .$$

What appears before the colon (if any) is an expression to apply each element of the sequence to generate the result, and what appears after the bar (if there is any) is an expression to apply to each element to decide whether to keep it.

Example 5.14. *Given the integer sequence* $A = \langle 9, 7, 13, 4, 11, 21 \rangle$*, and a function* $\texttt{isPrime}(v)$ *which checks if* v *is prime, the expression:*

$$\langle x \in A \mid \texttt{isPrime}(x) \rangle$$

is equivalent to:

$$\texttt{filter isPrime } A$$

and when evaluated returns the sequence:

$$\langle 7, 13, 11 \rangle \ .$$

As with `map` and `tabulate`, the function f in `filter` can be applied to the elements in parallel.

Subsequence, append, and flatten. It is often useful to extract a subsequence from a sequence. The $\texttt{subseq}(A, s, l)$ function extracts a contiguous subsequence starting at location s and with length l. If the subsequence is out of bounds of A, only the part within A is returned.

Syntax 5.15 (Subsequence)**.**

$$A[e_l, \cdots, e_h] \equiv \texttt{subseq}(A, e_l, e_h - e_l + 1)$$

It is also useful to put sequences together. The $\texttt{append}(A_1, A_2)$ function appends the sequence A_2 after the sequence A_1. To append more than two sequences the $\texttt{flatten}(A)$ function takes a sequence of sequences and flattens them—i.e. if the input is a sequence $A = \langle A_1, A_2, \ldots, A_n \rangle$ it appends all the A_i together one after the other.

Example 5.16. *We have:*

$$\texttt{append}(\langle \langle 1, 2, 3 \rangle, \langle 4, 5 \rangle) = \langle 1, 2, 3, 4, 5 \rangle$$

and

$$\texttt{flatten}(\langle \langle 1, 2, 3 \rangle, \langle 4 \rangle, \langle 5, 6 \rangle \rangle) = \langle 1, 2, 3, 4, 5, 6 \rangle \ .$$

Updates. It is often convenient to update elements of a sequence. The function $\texttt{update}(A, (i, v))$, updates location i of sequence A to contain the value v. If the location is out of range for the sequence, the function does nothing. It can be useful to update multiple elements at once. The function $\texttt{inject}(A, P)$ takes a sequence P of location-value pairs and updates each location with its associated value. If any locations are out of range, that pair does nothing. If multiple locations are the same, the rightmost one gets written.

Example 5.17. *Given the string sequence*

$$A = \langle \texttt{``the''}, \texttt{``cat''}, \texttt{``in''}, \texttt{``the''}, \texttt{``hat''} \rangle,$$

$$\texttt{update}(A, (1, \texttt{``bat''}))$$

gives

$$\langle \texttt{``the''}, \texttt{``bat''}, \texttt{``in''}, \texttt{``the''}, \texttt{``hat''} \rangle$$

since location 1 *is updated with* ``bat'', *and*

$$\texttt{inject}(A, \langle (4, \texttt{``log''}), (1, \texttt{``dog''}), (6, \texttt{``hog''}), (4, \texttt{``bog''}), (0, \texttt{``a''}) \rangle)$$

gives

$$\langle \texttt{``a''}, \texttt{``dog''}, \texttt{``in''}, \texttt{``the''}, \texttt{``bog''} \rangle$$

since location 0 *is updated with* ``a'', *location* 1 *with* ``dog'', *and location* 4 *with* ``bog'' *(it appears after* ``log'' *in the input sequence). The entry with location* 6 *is ignored since it is out of range for* A.

Operating on multiple sequences. In the examples we have considered thus far, we have operated on a single sequence, for example, when making a new sequence via the function `map`. In some cases, we need to consider multiple sequences. For example, we may want to form a sequence by pairing each element of one sequence A with all elements of the another sequence B, i.e., when computing the Cartesian product.

We use the following syntax to do this

$$\langle (x, y) : x \in A, y \in B \rangle .$$

Question 5.18. *What order should the resulting sequence be?*

An immediate question that arises when using binders that range over multiple sequences as in this example is what order should the resulting sequence be? Unless, there is a separate specification of ordering, then we assume that the resulting sequence is ordered by the natural generalization of the ordering of the sequences involved to multiple dimensions. For example with two sequences the element (s_1, t_1) comes before (s_2, t_2) if an only if in A, s_1 comes before s_2 or $s_1 = s_2$ and in B, t_1 comes before t_2 or $t_1 = t_2$.

Example 5.19. *Let* $A = \langle 0, 1 \rangle$ *and* $B = \langle a, b \rangle$.

$$\langle (x, y) : x \in A, y \in B \rangle = \langle (0, \texttt{'a}), (0, \texttt{'b}), (1, \texttt{'a}), (1, \texttt{'b}) \rangle .$$

Example 5.20.

$$\langle\, a \times b : a \in \langle\, 1, 2, 3 \,\rangle, b \in \langle\, 4, 5 \,\rangle \,\rangle$$

multiplies all pairs and evaluates to:

$$\langle\, 4, 5, 8, 10, 12, 15 \,\rangle$$

Question 5.21. *Can you express the Cartesian product example by using the single sequence syntax and the sequence functions that you have learned thus far in this chapter?*

While we presented the syntax for operating multiple-sequences as a separate syntax, it is actually expressible using the syntax that we have seen thus far. To see this consider sequence $AB' = \langle\, (0, B), (1, B) \,\rangle$, which can be computed by $AB' = \langle\, (x, B) : x \in A \,\rangle$. Now we can map over each element of the sequence AB' to obtain what we want, well more or less. Consider

$$AB'' = \langle\, \langle\, (x, y) : y \in z \,\rangle : (x, z) \in AB' \,\rangle\,.$$

It is not difficult to see that

$$AB'' = \langle\, \langle\, (0, \texttt{'a}), (0, \texttt{'b}) \,\rangle, \langle\, (1, \texttt{'a}), (1, \texttt{'b}) \,\rangle \,\rangle\,.$$

Thus the only remaining issue is the nesting. Luckily, that is simple using our `flatten` function. Indeed it is easy to see that $AB = \texttt{flatten}\ AB''$ is the Cartesian product that we sought after. Putting it all together, we can express the Cartesion product of A and B as follows:

$$\texttt{flatten}\ \langle\, \langle\, (x, y) : y \in z \,\rangle : (x, z) \in \langle\, (x, B) : x \in A \,\rangle \,\rangle\,.$$

Or equivalently, we can write the code for this expression in PML as follows:

```
1 CartesianProduct =  fn (A,B) =>
2   flatten (map (fn (x,z) => map (fn y => (x,y)) z)
3                (map (fn x => (x,B)) A))
```

This example shows the benefits of the sequence syntax defined so far.

Generalizing this syntax, we can allow essentially any expressions in place of the sequences and the expression being mapped, and also allow filtering over the bound variables.

Syntax 5.22 (Comprehensions for multiple sequences)**.** *Generalizing this syntax, we can allow essentially any expressions in place of the sequences and the expression being mapped, using expressions e, e_s, and e_t, and in fact any (finitely many) $m \in \mathbb{N}$, while also applying a filter e_f over all bound variables*

$$\langle\, e : x_1 \in e_1, x_2 \in e_2 \ldots, x_n \in e_n \mid e_f \,\rangle\,.$$

More generally, we can also allow variable binding involve ranges of natural numbers, as for example, can be used by `tabulate`. Specifically, $x_i \in e_i$ could be replaced by $e_l \leq i \leq e_h$, where e_l and e_h are expressions whose values are natural numbers and i is a variable.

Example 5.23. *Suppose that given two sequences A of naturals and B of letters, we wish to compute the a sequence that pairs each even element of A with all elements of B that are vowels. We can do this simply by adding the filtering predicate orthogonally as follows:*

$$\texttt{flatten}\,\langle\,\langle\,(x,y) : y \in z\,\rangle : (x,z) \in \langle\,(x,T) : x \in A \mid \texttt{isEven}(x)\,\rangle\ \texttt{and}\ \texttt{isVowel}(y)\,\rangle$$

where the self-explanatory predicate `isEven` *holds only when the argument is an even number, and* `isVowel` *holds only when the argument is a vowel.*
$\langle\,(x,y) : x \in A, y \in T \mid \texttt{isEven}(x)\,\rangle$

Example 5.24. *Let's say we want to generate all contiguous subsequences of a sequence A. Each sequence can start at any position $0 \leq i < |A|$, and end at any position $i \leq j < |A|$. We can do this with the following pseudocode:*

$$\langle\, A\,\langle\, i, \ldots, j\,\rangle : 0 \leq i < |A|, i \leq j < |A|\,\rangle\ ,$$

which is equivalent to:

```
flatten(tabulate
        (fn i ⇒ tabulate
                (fn l ⇒ subseq(A, i, l + 1))
                (length(A) − i))
        (length A))
```

Here we see that syntax based on comprehensions can be quite convenient.

Collect. The primitive `collect` is useful when elements of a sequence are "keyed", making it possible to associate data with some key. Such pairs consisting of a key and a value are sometimes called ***key-value*** pairs.

Example 5.25. *The following sequence shows a sequence of key-value pairs consisting of our students from last semester and the classes they take.*

```
val Data = ⟨(''jack sprat'', ''15-210''),
            (''jack sprat'', ''15-213''),
            (''mary contrary'', ''15-210''),
            (''mary contrary'', ''15-213''),
            (''mary contrary'', ''15-251''),
            (''peter piper'', ''15-150''),
            (''peter piper'', ''15-251''),
           ...⟩
```

Note that key-value pairs are intentionally asymmetric: they map a key to a value. This is fine because that is how we often like them to be.

But sometimes, we often want to put together all the values for a given key.

We refer to this function as a *collect*.

Example 5.26. *We can determine the classes taken by each student.*

```
classes = ⟨
      (''jack sprat'',⟨ ''15-210'', ''15-213'',...⟩),
      (''mary contrary'',⟨ ''15-210'', ''15-213'', ''15-251'',...⟩),
      (''peter piper'',⟨ ''15-210'', ''15-251'',...⟩),
      ...⟩
```

It is important to note that classes taken by each student are ordered in the same order as they are in the original sequence. Similarly, the keys in the resulting sequence have the same relative order as in the original sequence.

Collecting values together based on a key is very common in processing databases. In relational database languages such as SQL it is referred to as "Group by". More generally it has many applications and furthermore it is naturally parallel.

We will use the function `collect` for this purpose, and it is part of the sequence library. Its type signature is:

$$\texttt{collect} : (\alpha \times \alpha \to \textit{order}) \to (\alpha \times \beta)\ \textit{seq} \to (\alpha \times \beta\ \textit{seq})\ \textit{seq}$$

The first argument is a function for comparing keys of type α, and must define a total order over the keys.

The second argument is a sequence of key-value pairs.

The `collect` function collects all values that share the same key together into a sequence, ordering the values in the same order as their appearance in the original sequence.

5.5 Aggregation with Iteration

Iteration is a key concept in algorithm design, as well as many other areas of computer science. Iterative design, for example, is one of the most important concepts in designing good algorithms or programs—i.e. the idea of repeatedly improving and simplifying your algorithm or program. The term iteration implies that a sequence of steps is taken one after another, each taking the state from the previous step and updating it for the next step. It is therefore an inherently sequential concept.

In the context of sequences we use *iterate* to refer to computation that starts with an initial state and a sequence and on each step update- the state based on the next element of the sequence. The iteration therefore takes as many steps as the length of the sequence. More concretely the `iterate` function has the type signature

$$\texttt{iterate}\ (f\texttt{:}\ \alpha\ \star\ \beta\ \to\ \alpha)\ (v\texttt{:}\ \alpha)\ (A\texttt{:}\ \beta\ \texttt{sequence})\ \texttt{:}\ \alpha$$

where f is a function mapping a state and an element of A to a new state, v is the initial state, A is a sequence.

The function `iterate` computes its final result by computing a new state for each element of the function $v_0 = v, v_1 = f(v_0, A[0]), v_2 = f(v_1, A[1]), \ldots, v_n = f(v_{n-1}, A[n-1]$, where $n = |A|$ and the final result is v_n.

The type signature of `iterate` follows its structure. If the states have type α and A is an β sequence, then f has type $\alpha \times \beta \to \alpha$, and the result is of type α.

As with other operations on sequences, we use a two forms of special syntax for iteration.

Syntax 5.27 (Iteration)**.** *The first piece of syntax allows mapping, whereas the second one operation on the sequence directly.*

- *With mapping:* $\overline{\prod}^{f\ v}_{x \in A}\ g(x) \equiv \texttt{iterate}\ f\ v\ (\texttt{map}\ g\ A)$.
- *Direct:* $\overline{\prod}^{f\ v}\ A \equiv \texttt{iterate}\ f\ v\ A$

Example 5.28. *For a sequence of length* 5*, iteration computes its final result as*

$$f(f(f(f(f(v, A[0]), A[1]), A[2]), A[3]), A[4]).$$

Example 5.29.

$$\overline{\prod}^{'+'\ 0}\ \langle 2, 5, 1, 6 \rangle$$

returns 14 *since it starts with the integer state* 0 *and then one by one adds the integer elements* 2*,* 5*,* 1 *and* 6 *of* A *to the state. Similarly*

$$\overline{\prod}^{'-'\ 0}\ \langle 2, 5, 1, 6 \rangle$$

returns $(((0 - 2) - 5) - 1) - 6 = -14$.

Parentheses matching. As an example of how to use iteration, and specifically `iterate`, consider the problem of finding whether a string (sequence) of left and right parentheses is properly matched or nested.

Question 5.30. *You are likely familiar with what a properly matched strings are but how can we define such strings?*

We say a string is matched if it can be described recursively as

$$p = \langle\,\rangle \mid p\,p \mid \texttt{'('}\,p\,\texttt{')'},$$

where $\langle\,\rangle$ is the empty sequence, $p\,p$ indicates appending two strings of matched parentheses (recursively defined), and `'('` p `')'` indicates the string starting with `'('` followed by a matched string p followed by `')'`.

Example 5.31. *The string* `''(())(())''` *is matched.*
The `''())(()''`*, however, is not matched.*

Problem 5.32 (Parentheses matching)**.** *The* parentheses matching *problem asks determining whether a given a string of parentheses is matched.*

Question 5.33. *Can you think of a way of using* `iterate` *to solve this problem?*

There are a variety of algorithms for solving this problem. Here we go over a linear-work sequential algorithm based on `iterate`. In the next chapter we go over a divide-and-conquer algorithm that requires no more work asymptotically, but is highly parallel.

We can solve the problem by starting at the beginning of the sequence with a counter set to zero and iterating through the elements one by one. If we ever see a left parenthesis we increment the counter and whenever we see a right parenthesis we decrement the counter. A sequence of parentheses can only be matched if the count ends at 0 since being matched requires that there are an equal number of right and left parentheses. However ending with a count of 0 is not adequate since the string `'')(('` has count 0 but is obviously not matched. It also has to be the case that the count can never go below 0 during the iterations. and the observation leads to the following algorithm:

Algorithm 5.34.

```
1  fun mathParens(A) =
2  let
3     fun count(s,c) =
4        case (s,c) of
5           (None,_) ⇒ None
6        | (Some(n),)) ⇒ if (n = 0) then None else Some(n − 1)
7        | (Some(n),() ⇒ Some(n + 1)
8  in
9     (iterate count Some(0) A) = Some(0)
10 end
```

The algorithm starts with the state (counter) $\texttt{Some}(0)$ and increments or decrements the counter on a left and right parenthesis, respectively. If the iterations ever encounter a right parenthesis when the count is zero, this indicates the count will go below zero, and at this point the state is changed to `None`, which is propagated through the rest of the iterations to the result. Therefore at the end if the state is $\texttt{Some}(0)$ then the counter never went below zero and ended up at zero so the parentheses must be matched.

Question 5.35. *Consider the algorithm for summing the numbers in a sequence of natural numbers A using* `iterate`*, e.g.,*

$$\overline{\prod}^{'+'\ 0}\quad A\,.$$

This algorithm performs the plus operations in a particular order. Is this necessary?

Iteration is a powerful technique but it is sometimes too big of a hammer, especially when it is not needed. For example, when summing up the elements in a sequence you don't need to perform the addition operations in a particular order because addition operations are associative: they can be performed in any order desired. The iteration-based algorithm for computing the sum does not take advantage of this property, as it computes the sum by rolling over the sequence from left to right. Indeed as we will next, we can take advantage of the associativity of the addition operations to sum up the elements in a sequence in parallel.

Exercise 5.36 (Matching parentheses correctly)**.** *Prove that the* `iterate`*-based algorithm Algorithm 5.34 indeed solves the parentheses matching problem.*

5.6 Aggregation with Reduce

Reduction is a key concept in parallel algorithm design. The term "reduction" refers to a computation that repeatedly applies an associative binary operation to a collection of elements until the result is reduced to a single value. Let's recall the definition of an associative operation.

Definition 5.37. *A function $f : \alpha \to \alpha$ is associative if $f(f(x, y), z) = f(x, f(y, z))$ for all x, y and z of type α.*

What associativity implies is that if we are interested applying f to a number of values, the order in which the applications are performed does not matter. Note that associativity does not mean that you can reorder the arguments to a function (that would be commutativity) but it means that you can pick whatever order you want when performing the applications.

Many functions are associative. You probably already known that addition and multiplication on natural numbers are associative, with 0 and 1 as their (left) identities, respectively. Minimum and maximum are also associative with left identities ∞ and $-\infty$ respectively.

Question 5.38. *Is the* `append` *operation on sequences associative? If so, what is its identity element?*

The `append` function on sequences is also associative, with left identity being the empty sequence.

Question 5.39. *How about set union? If so, what is its identity element?*

Set union and matrix multiply are associative, with the empty set and the set of all possible elements as the identity, respectively.

Question 5.40. *Is floating point addition an associative operation?*

An important class of operations that are not associative include floating-point operations, which can cause significant loss of precision when performed in one order and not in another order.

In the context of sequences we use *reduce* to refer to computation that an associative binary operation to the elements of the sequence until the result is reduced to a single value. More concretely the `reduce` function has the type signature

$$\texttt{reduce}\ (f\texttt{:}\ \alpha\ \star\ \alpha \to \alpha)\ (I\texttt{:}\ \alpha)\ (A\texttt{:}\ \alpha\ \texttt{sequence})\texttt{:}\ \alpha$$

where f is an associative function, A is the sequence, and I is the identity element of f.

As with other operations on sequences, we use a special syntax for `reduce`

Syntax 5.41. *The first piece of syntax allows mapping, whereas the second one operation on the sequence directly.*

- *With mapping:* $\overline{\sum}_{x\in A}^{f\ I} g(x) \equiv \texttt{reduce}\ f\ I\ (\texttt{map}\ g\ A)$.
- *Direct:* $\overline{\sum}^{f\ I}\ A \equiv \texttt{reduce}\ f\ I\ A$

As shown in ADT 5.3, when applied to a sequence with a function `reduce` returns the "sum" with respect to f of the input sequence A. In fact if f is associative this sum in equal to iteration, i.e.,

$$\overline{\sum}_{x\in A}^{f\ I} x = \overline{\prod}_{x\in A}^{f\ I} x.$$

Example 5.42.

$$\overline{\sum}^{append\ \langle\rangle}\ \langle\ \texttt{``another''}, \texttt{``way''}, \texttt{``to''}, \texttt{``flatten''}\rangle$$

returns `''anotherwaytoflatten''`.

The function `reduce` is purely more restrictive than iterate since it is effectively the same function but with extra restrictions on its input (i.e. that f be associative, and I is a left identity).

Question 5.43. *Given that* `reduce` *and* `iterate` *are equivalent, why do we care to define another function?*

Even though the input-output behavior (extrinsic semantics) of `reduce` and `iterate` are identical for associative functions, their cost specifications (intrinsic semantics) differ significantly: unlike `iterate`, which is strictly sequential, `reduce` is parallel. In fact, as we will describe in Section 5.8, the span of `iterate` is linear span in the size of the input, whereas the span of `reduce` is logarithmic.

The results of `reduce` and `iterate`, however, may differ if the combining function is non-associative. In this case, the order in which the reduction is performed determines the result; because the function is non-associative, different orderings will lead to different answers. While we might try to apply reduce to only associative operations, unfortunately even some functions that seem to be associative are actually not. For instance, floating point addition and multiplication are not associative. In some languages integer addition is also not associative because of the possibility of overflow, which might raise an exception.

To properly deal with functions that are non-associative, it is therefore important to specify the order in which the function is applied to the elements of a sequence. This order is part of the specification of ADT 5.3. In this way, every (correct) implementation returns the same result when applying `reduce`; the results are deterministic regardless of what data structure and algorithm are used.

Exercise 5.44 (Equivalence of `reduce` and `iterate`)**.** *Prove that* `reduce` *and* `iterate` *are equivalent when the function being applied is associative.*

5.7 Aggregation with Scan

When we restrict ourselves to associative functions, the input-output behavior of the function `reduce` can be defined in terms of the `iterate`.

Question 5.45. *Assuming that we restrict ourselves to associative functions, can the input output behavior of* `iterate` *be defined in terms of* `reduce`*?*

But the reverse is not true: `iterate` cannot always be defined in terms of `reduce` because `iterate` can use the results of intermediate states computed on the prefixes of the sequence, whereas `reduce` cannot because such intermediate states are not available.

Question 5.46. *Can you give an example?*

For example, in our parenthesis matching algorithm (Algorithm 5.34), we used this property crucially by defining our function to propagate a mismatched parenthesis forward in the computation. In this section, we will describe a function called `scan` that gives us exactly this ability while also guaranteeing parallel execution.

Scan is another key concept in the design of parallel algorithms. Perhaps the easiest way to define this concept is in terms of reductions (Section 5.6): the term "scan" refers to a computation that reduces every prefix of a given sequence by repeatedly applying an associative binary operation. By a prefix of a sequence, we mean any subsequence of the sequence that starts at the beginning. Empty sequence is a prefix of any sequence.

Example 5.47. *All the prefixes of* $A = \langle 0, 1, 2 \rangle$ *are*

- $\langle\,\rangle$
- $\langle 0 \rangle$
- $\langle 0, 1 \rangle$
- $\langle 0, 1, 2, 3 \rangle$.

The `scan` function has the type signature

```
scan (f: α * α → α) (I: α) (A: α sequence) : (α sequence
* α)
```

where f is an associative function, A is the sequence, and I is the (left) identity element of f.

As with other operations on sequences, we use a special syntax for `scan`

Syntax 5.48. *The first piece of syntax allows mapping, whereas the second one operation on the sequence directly.*

- *With mapping:* $\overline{\int}^{f\ I}_{x \in A}\ g(x) \equiv$ `scan` $f\ I$ (`map` $g\ A$).
- *Direct:* $\overline{\int}^{x \in A\ f}\ IA \equiv$ `scan` $f\ I\ A$.

As specified in ADT 5.3, when applied to a sequence with a function `scan` returns the "sum" with respect to f of all prefixes of the input sequence A.

$$\texttt{scan}\ f\ I\ A = (\ \texttt{tabulate}\,(\texttt{fn}\,i \texttt{=>}\ \texttt{reduce}\ f\ I\ (\texttt{subseq}\ (A, 0, i)))(|A| - 1),\ \texttt{reduce}\ f\ I\ A).$$

As with `reduce`, when the function f is associative, the scan function returns the sum with respect to f of each prefix of the input sequence A, as well as the total sum of A. Hence the function is often called the ***prefix sums*** function (or problem). Using our syntax we can rewrite this definition more compactly as:

fun `scan` $f\ I\ A =$
$\left(\left\langle \overline{\sum}^{f\ I}\ (A\langle 0,\ldots,l-1\rangle) : 0 \le l < n \right\rangle, \overline{\sum}^{f\ I}\ A\right).$

Example 5.49. *The computation* $\overline{\int}^{+\ 0}\ \langle 2,1,3\rangle$ *can be written as follows.*

$$\overline{\int}^{+\ 0}\ \langle 2,1,3\rangle = \left(\left\langle \overline{\sum}^{+\ 0}\ \langle\,\rangle, \overline{\sum}^{+\ 0}\ \langle 2\rangle, \overline{\sum}^{+\ 0}\ \langle 2,1\rangle, \overline{\sum}^{+\ 0}\ \langle 2,1,3\rangle \right\rangle\right)$$

$$= (\langle 0,2,3\rangle, 6)$$

Note that when computing the result for position i, `scan` does not include the element of the input sequence at that position. It is sometimes useful to do so. To this end we define `scanI` ("I" stands for "inclusive") for this purpose.

Example 5.50. *The computation* $\overline{\oint}^{+\ 0}_{x\in\langle 2,1,3\rangle}\ x$ *can be written as follows.*

$$\overline{\oint}^{+\ 0}_{x\in\langle 2,1,3\rangle}\ x = \left\langle \overline{\sum}^{+\ 0}\ \langle 2\rangle, \overline{\sum}^{+\ 0}\ \langle 2,1\rangle, \overline{\sum}^{+\ 0}\ \langle 2,1,3\rangle \right\rangle$$

$$= \langle 2,3,6\rangle$$

We have seen that `reduce` is just a weaker version of `iterate`. Indeed it could be expressed just with `iterate`. We have now introduced another operation that can be expressed in terms of others, `reduce` and `tabulate` in particular.

Question 5.51. *Given that scan can be expressed just in terms of the operations that we have already seen, why do we need it?*

The reason for why we need this operation is again the cost. As we shall see, performing `reduce` repeatedly on every prefix is not work efficient. Remarkably `scan` can be implemented by performing essentially the same work and span of `reduce`.

Remark 5.52. *Experience with parallel programming shows that reduction and scan are powerful primitives that suffices to express many parallel algorithms on sequences. In some ways this is not surprising because as we described the scan primitive essentially gives us the ability to perform iteration, perhaps the most important sequential-algorithm design technique.*

Exercise 5.53 (Parentheses matching with scan)**.** *Give an algorithm for the parentheses matching problem using* `scan` *instead of* `iterate`*.*

Example: copy scan. We used `scan` to compute partial sums. Scan is also useful when you want pass information along the sequence. For example, suppose you have some "marked" elements that you would like to copy across to their right until they reach another marked element. One way to mark the elements is to use options.

That is, suppose you are given a sequence of type `int seq` consisting only of non-negative numbers. For example

$$\langle 0,\ 7,\ 0,\ 0,\ 3,\ 0 \rangle$$

and your goal is to return a sequence of the same length where each element receives the previous positive value. For the example:

$$\langle 0,\ 0,\ 7,\ 7,\ 7,\ 3 \rangle$$

Using a sequential loop or `iterate` would be easy. How would you do this with `scan`?

If we are going to use a `scan` directly, the combining function f must have type `int * int` $\rightarrow$ `int`. How about the following function:

```
1 fun copy(a, b)
2   if b > 0 then b
3   else a
```

What this function does is basically pass on its right argument if it is positive and otherwise it passes on the left argument. To be used in a scan it needs to be associative. In particular we need to show that $\texttt{copy}(x, \texttt{copy}(y, z)) = \texttt{copy}(\texttt{copy}(x, y), z)$ for all x, y and z. There are eight possibilities corresponding to each of x, y and z being either positive or not. For the cases where z is positive, it is easy to verify that that either ordering returns z. For the cases that $z = 0$ and y is positive, one can verify that both orderings give y, for the cases that both $y = z = 0$ and x is positive they both return x, and for all being zero, the ordering returns zero.

There are many other applications of scan in which more involved functions are used. One important case is to simulate a finite state automaton.

5.8 Cost Specification

We now consider the cost specifications for the sequence ADT. We consider two cost specifications, which we refer to as the *array sequence*, and *tree sequence* cost specifications. The names roughly indicate the class of implementation that can achieve these cost bounds, but there might be many specific implementations that match the bounds. For examples for the tree-sequence specification there are many types of trees that might be used. To use the cost bounds, you don't need to know the specifics of how these implementations work. The reason to have more than one specification is that in different usages different specifications are better. We say that one cost specification ***dominates*** another if for every function its asymptotic costs are no higher.

Of the two specifications we consider, none dominates another. There are however trade-offs, some function are cheaper in one specification and others are cheaper in the other specification. Such a trade-offs are common in data types and should be considered when selecting which cost specification to use. The user can decide to use the specification the cost of the algorithm of interest under each specification. For example, as you will see soon, if an algorithm makes many calls to `nth` but no calls to `append`, then the user might want to use the array-sequence specification (and an implementation consistent with it) rather than the tree-sequence specification, while if the mostly use `append` and `update`, then tree-sequence specification might be better. After the user decides the specification to use, what remains is to select the implementation that matches the specification, which can include additional considerations.

For the cost bounds, we consider the aggregation operations separately, because the cost of such operations depend on the nature of the aggregation performed as specified by the function used in the aggregation.

Non-aggregation functions. The costs for array sequences is given in Cost Specification 5.8. The first thing to notice is that in the cost specification the function `nth` takes constant work and span. This is because in an array we can access an arbitrary element in constant time. The work and span for `singleton`, and `length` are also constant, which should not be surprising. For the three functions `tabulate`, `map`, and `filter` the work includes the sum of the work of applying f at each position, as we would expect from the definition of work. We also have to add 1 for the overhead of applying each function. In all three functions it is possible to apply the function f in parallel since there is no dependence among the positions. Therefore the span of the functions is the maximum of the span of applying f at each position. For `map` and `tabulate` there is again a constant overhead, but for `filter` there is a logarithmic overhead. We will see why when we cover the implementation, but it has to do with packing the remaining elements into contiguous locations in an array.

The `subseq` function has constant work. The `append`, `flatten`, `update` and `inject` functions all require work proportional to the length of the sequences they are working on, but can be implemented in parallel so the span is at most logarithmic in the length. It might see surprising that `update` takes work proportional to the size of the input sequence A since updating a single element should take constant work. The reason is that the interface is purely functional

Cost Specification 5.54 (Array Sequences)**.** *We define the* **array-sequence** *cost specification as follows.*

	Array Sequence	
	Work	Span
$\texttt{length}(A)$	1	1
$\texttt{singleton}(v)$	1	1
$\texttt{nth}(A, i)$	1	1
$\texttt{map}\ f\ A$	$1 + \sum_{x \in A} W(f(x))$	$1 + \max_{x \in A} S(f(x))$
$\texttt{tabulate}\ f\ n$	$1 + \sum_{i=0}^{n} W(f(i))$	$1 + \max_{i=0}^{n} S(f(i))$
$\texttt{filter}\ p\ A$	$1 + \sum_{x \in A} W(p(x))$	$\log \lvert A \rvert + \max_{x \in A} S(p(x))$
$\texttt{subseq}(A, s, l)$	1	1
$\texttt{append}(A_1, A_2)$	$1 + \lvert A_1 \rvert + \lvert A_2 \rvert$	1
$\texttt{flatten}(A)$	$1 + \sum_{x \in A} \lvert A \rvert + \lvert A \rvert$	$1 + \log \lvert A \rvert$
$\texttt{update}(P, A)$	$1 + \lvert P \rvert + \lvert A \rvert$	1
$\texttt{inject}(A, P)$	$1 + \lvert P \rvert + \lvert A \rvert$	1
$\texttt{collect}\ f\ A$	$1 + W(f) \cdot \lvert A \rvert \log \lvert A \rvert$	$1 + S(f) \cdot \log^2 \lvert A \rvert$

so that the input sequence needs to be copied–we are not allowed to update the old copy. Later, we will define single-threaded array sequences that will allow us under certain restrictions to update a sequence in constant work (Section 5.10). The primary cost in implementing collect is a sorting step sorts the sequence based on the keys. The work of collect is therefore work and span of (comparison) sorting with the specified sorting function. Since the comparison function can take non-constant time, its work and span appears as a parameter in both.

Example 5.55. *As an example of* `tabulate` *and* `map`*, we have*

$$\begin{aligned} W(\langle\, i^2 : 0 \leq i < n \,\rangle) &= O(1 + \sum_{0=1}^{n-1} O(1)) = O(n) \\ S(\langle\, i^2 : 0 \leq i < n \,\rangle) &= O(1 + \max_{i=0}^{n-1} O(1)) = O(1) \end{aligned}$$

given that the work, and hence span, for i^2 is $O(1)$. As an example of `filter` *we have:*

$$\begin{aligned} W(\langle\, x : x \in A \mid x < 27 \,\rangle) &= O(1 + \sum_{i=0}^{|A|-1} O(1)) = O(|A|) \\ S(\langle\, x : x \in A \mid x < 27 \,\rangle) &= O(\log |A| + \max_{i=0}^{|A|-1} O(1)) = O(\log |A|) \end{aligned}$$

Example 5.56. *As a more involved example we consider the code from Example 5.24:*

$$e = \langle\, A \langle\, i, \ldots, j \,\rangle : 0 \leq i < |A|, i \leq j < |A| \,\rangle\ ,$$

which extracts all contiguous subsequences from the sequence A. Recall that the notations is equivalent to a nested `tabulate` *first over the indices i, and then inside over the indices j. The results are then* `flattened`*. The nesting of the tabulates allows all the calls to $A\langle\, i, \ldots, j \,\rangle$ (i.e.,* `subseq`*) to run in parallel. Let $n = |A|$. There are a total of*

$$\sum_{i=1}^{n} i = n(n+1)/2 = O(n^2)$$

contiguous subsequences and hence that many calls to `subseq`*. Each call takes constant work and span according to the cost specifications. The overall work for the subsequences is therefore $O(n^2)$. The overall span is $O(1)$ since the span of the inner* `tabulate` *is the maximum over the spans of the* `subseqs`*, which is $O(1)$, and the outer* `tabulate` *is the maximum over those, which is again $O(1)$.*
The `flatten` *at the end requires $O(n^2)$ work and $O(\log n)$ span , since $||A|| = n(n+1)/2 = O(n^2)$, and $|A| = n$. We therefore have in total that*

$$\begin{aligned} W(e) &= O(|A|^2) \\ S(e) &= O(\log |A|) \end{aligned}$$

The costs for tree sequences is given in Cost Specification 5.8. The first thing to notice is that the cost of the `nth` function is no longer constant. Instead it has logarithmic work and span.

Cost Specification 5.57 (Tree Sequences). *We define the* **tree-sequence** *cost specification as follows.*

	Tree Sequence	
	Work	Span
`length`(A)	1	1
`singleton`(v)	1	1
`nth`(A, i)	$\log \lvert A \rvert$	$\log \lvert A \rvert$
`tabulate` f n	$1 + \sum_{i=0}^{n} W(f(i))$	$1 + \log n + \max_{i=0}^{n} S(f(i))$
`map` f A	$1 + \sum_{x \in A} W(f(x))$	$1 + \log \lvert A \rvert + \max_{x \in A} S(f(x))$
`filter` f A	$1 + \sum_{x \in A} W(f(x))$	$1 + \log \lvert A \rvert + \max_{x \in A} S(f(x))$
`subseq`(A, s, l)	$1 + \log(\lvert A \rvert)$	$1 + \log(\lvert A \rvert)$
`append`(A_1, A_2)	$1 + \lvert \log(\lvert A_1 \rvert / \lvert A_2 \rvert) \rvert$	$1 + \lvert \log(\lvert A_1 \rvert / \lvert A_2 \rvert) \rvert$
`flatten`(A)	$1 + \lvert A \rvert \log \left(\sum_{x \in A} \lvert A \rvert \right)$	$1 + \log(\lvert A \rvert + \sum_{x \in A} \lvert x \rvert)$
`inject`(P, A)	$1 + (\lvert P \rvert + \lvert A \rvert) \log \lvert A \rvert$	$1 + \log(\lvert A \rvert + \lvert P \rvert)$
`collect`$f A$	$1 + W(f) \cdot \lvert A \rvert \log \lvert A \rvert$	$1 + S(f) \cdot \log^2 \lvert A \rvert$

This is because tree sequences use a balanced tree, and require following a path from the root to a leaf to find the nth element. Such a path has length $O(\log |A|)$. Although `nth` does more work with tree sequences, `append` does Instead of requiring linear work, the work of `append` with tree sequences is proportional to the log of the ratio of the size of the larger sequence to the size of the smaller one smaller one. For example if the two sequences are the same size, then `append` takes $O(1)$ work. On the other hand if one is length n and the other 1, then the work is $O(\log n)$. The work of `update` is also less with tree sequences than within array sequences. More details on how these cost arise is given later.

The work and span for other functions such `map`, `tabulate` and `filter` are approximately the same for array sequences and tree sequences, except there is an extra logarithmic term in the span for `map` and `tabulate` in tree sequences. The cost of `collect` is the same in both cases.

Exercise 5.58 (Cost of `map` via `tabulate`)**.** *Earlier we showed how to implement* `map` *using* `tabulate`*. Decide whether this implementation preserves asymptotic costs for an array sequence, and then for tree sequence.*

Exercise 5.59 (Cost of collect)**.** *Describe how to implement collect in a way that is consistent with the specified costs.*

Cost of `iterate`**.** The cost of aggregating functions are somewhat more difficult to specify because they depend on the functions supplied.

Example 5.60. *Consider appending the following sequence of strings using* `iterate`*:*

$$\overline{\prod}^{\texttt{append}\ \texttt{``''}}\ \langle\ \texttt{``abc''}, \texttt{``d''}, \texttt{``e''}, \texttt{``f''}\rangle.$$

If we only count the work of `append` *operations using the array-sequence specification, we obtain a total work of* 18*, because the following* `append` *operations are performed.*

1. `append`(``abc'', ``d''), *work* = $4 + 1 = 5$.
2. `append`(``abcd'', ``e''), *work* = $5 + 1 = 6$.
3. `append`(``abcde'', ``f''), *work* = $6 + 1 = 7$.

Consider now appending the following sequence of strings using `iterate`*:*

$$\overline{\prod}^{\texttt{append}\ \texttt{``''}}\ x.\langle\ \texttt{``a''}, \texttt{``b''}, \texttt{``c''}, \texttt{``def''}\rangle$$

If we only count the work of `append` *operations using the array-sequence specification, we obtain a total work of* 14*, because the following* `append` *operations are performed.*

1. `append`(``a'', ``b''), *work* = $2 + 1 = 3$.
2. `append`(``ab'', ``c''), *work* = $3 + 1 = 4$.
3. `append`(``abc'', ``def''), *work* = $6 + 1 = 7$.

As shown by the example, iterating over two sequences of the same length differ (only when their internal structure have also essentially the same length up to a permutation), because the cost depends on the intermediate values generated during computation. Such dependency on the order in which functions are applied can make it difficult to reason about cost abstractly without knowing more details about the implementation because the intermediate values, which are not obvious from the specification, effect cost. Fortunately, the specification of the sequence ADT as shown in ADT 5.3 does in fact provide the right level of abstraction.

Cost Specification 5.61 (Cost specification for `iterate`)**.** *Consider evaluation of* `iterate` $f\ v\ A$) *and let* $\mathcal{R}$(`iterate` $f\ v\ A$) *denote the set of calls to* $f(\cdot,\cdot)$ *performed along with the arguments.*
For both the array-sequence and tree-sequence specification

$$W(\texttt{iterate}\ f\ v\ A) \;=\; O\left(|A| + \textstyle\sum_{f(a,b)\in\mathcal{R}(\texttt{iterate}\ f\ v\ A)} W(f(a,b))\right)$$
$$S(\texttt{iterate}\ f\ v\ A) \;=\; O\left(|A| + \textstyle\sum_{f(a,b)\in\mathcal{R}(\texttt{iterate}\ f\ v\ A)} S(f(a,b))\right)$$

As another interesting example, let's suppose that we have a $\texttt{merge}_<$ for merging two sequences based on a comparison function between the elements of the sequence.

Question 5.62. *Can you see how to sort a sequence by using such a function?*

We can use such a function sort a sequence as follows.

fun `iterSort`$(A) = \overline{\prod}^{\texttt{merge}_<\ \langle\,\rangle} \quad A$

For example, on input $x = \langle\, x_1, x_2, \ldots, x_n \,\rangle$, the algorithm will first merge $\langle\,\rangle$ and $\langle\, x_1 \,\rangle$, then merge in $\langle\, x_2 \,\rangle$, then $\langle\, x_3 \,\rangle$, etc.

With this order $merge$ is called when its left argument is a sequence of varying size between 1 and $n-1$, while its right argument is always a singleton sequence. The final merge combines $(n-1)$-element with 1-element sequences, the second to last merge combines $(n-2)$-element with 1-element sequences, so on so forth. Therefore, the total work for an input sequence S of length n is

$$W(\texttt{iterSort}\ S) \;\leq\; \sum_{i=1}^{n-1} c\cdot(1+i) \;\in\; O(n^2)$$

since merge on sequences of lengths n_1 and n_2 has $O(n_1+n_2)$ work.

Question 5.63. *Can you see what algorithm* `iterSort` *implements?*

Using this reduction order the algorithm is effectively working from the front to the rear, "inserting" each element into a sorted prefix where it is placed at the correct location to maintain the sorted order. This corresponds to the well-known insertion sort.

We can also analyze the span of `iterSort`. Since we iterate adding in each element after the previous, there is no parallelism between merges, but there is parallelism within a merge. We can calculate the span as

$$S(\texttt{iterSort}\ x) \;\leq\; \sum_{i=1}^{n-1} c\cdot\log(1+i) \;\in\; O(n\log n)$$

since merge on sequences of lengths n_1 and n_2 has $O(\log(n_1 + n_2))$ span. This means our algorithm does have a reasonable amount of parallelism, $W(n)/S(n) = O(n/\log(n))$, but the real problem is that it does much too much work.

Cost of reduce. We have seen that cost of `iterate` depends on the intermediate computations.

Question 5.64. *Does the same dependency exist in* `reduce`*?*

The same problem arises in `reduce`. In fact, with reduce, the problem is a bit more complicated. Recall that with `reduce`, we noted that the result of the computation is not affected by the order in which the associative function is applied and in fact is the same as that of performing the same computation with `iterate`. The cost, however, is highly dependent on the order, as shown by the example below.

Example 5.65. *Consider appending the following sequence of strings using* `reduce`*:*

$$\overline{\sum}^{\texttt{append}\ \texttt{''}\ \texttt{''}}\ x.\langle\ \texttt{''abc''}, \texttt{''d''}, \texttt{''e''}, \texttt{''f''}\rangle$$

Suppose that we perform the append operations in left-to-right order and count their work using the array-sequence specification. We obtain a total work of 18*, because the following* `append` *operations are performed.*

1. `append`(`''abc''`, `''d''`), *work* = $4 + 1 = 5$.
2. `append`(`''abcd''`, `''e''`), *work* = $5 + 1 = 6$.
3. `append`(`''abcde''`, `''f''`), *work* = $6 + 1 = 7$.

Consider now performing the same computation by performing the `append` *operations from right to left. We obtain a total cost of* 11*, because the following* `append` *operations are performed.*

1. `append`(`''e''`, `''f''`), *work* = $2 + 1 = 3$.
2. `append`(`''d''`, `''ef''`), *work* = $3 + 1 = 4$.
3. `append`(`''abc''`, `''def''`), *work* = $6 + 1 = 7$.

Once again, however, the specification of the sequence ADT as shown in ADT 5.3 does in fact provide the right level of abstraction.

Cost Specification 5.66 (Cost specification for `reduce`)**.** *Consider evaluation of* $\texttt{reduce}\ f\ I\ A)$ *and let* $\mathcal{R}(\texttt{reduce}\ f\ v\ A)$ *denote the set of calls to* $f(\cdot,\cdot)$ *performed along with the arguments.*
For both the array-sequence and tree-sequence specification

$$\begin{aligned} W(\texttt{reduce}\ f\ v\ A) &= O\left(|A| + \textstyle\sum_{f(a,b)\in\mathcal{R}(\texttt{reduce}\ f\ v\ A)} W(f(a,b))\right) \\ S(\texttt{reduce}\ f\ v\ A) &= O\left(\log|A| \cdot \max_{f(a,b)\in\mathcal{R}(\texttt{reduce}\ f\ v\ A)} S(f(a,b))\right) \end{aligned}$$

The work bound is simply the total work performed, which we obtain by summing across all combine functions. The span bound is more interesting. The $\log n$ term expresses the fact that the recursion tree in the specification of `reduce` (ADT 5.3) is at most $O(\log n)$ deep. Since each node in the recursion tree has span at most $\max_{f(a,b)} S(f(a,b)$, any root-to-leaf path, has at most $O(\log n \max_{f(a,b)} S(f(a,b))$ span.

Exercise 5.67. *Recall the* $\texttt{merge}_<$ *function that we considered above.*

- *Prove that* $\texttt{merge}_<$ *is associative.*
- *Show that we can use* $\texttt{merge}_<$ *and* `reduce` *to implement a sorting algorithm.*
- *Analyze the work and span of the resulting algorithm.*
- *What algorithm does this algorithm resemble?*

Cost of scan. As in `iterate` and `reduce` the cost specification of `scan` depends on the intermediate results. But such dependency is a bit more tricky that can be represented by our ADT specification. For `scan`, we will stop at giving a cost specification by assuming that the function that we are scanning with performs $O(1)$ work and span.

Cost Specification 5.68 (Cost specification for `scan`)**.** *Consider evaluation of* $\texttt{scan}\ f\ I\ A)$*. For both the array-sequence and tree-sequence specification*

$$\begin{aligned} W(\texttt{scan}\ f\ I\ A) &= O(|A|) \\ S(\texttt{scan}\ f\ I\ A) &= O(\log|A|). \end{aligned}$$

5.9 An Example: Primes

We now give some more involved examples of how to use sequences and analyze and compare costs. We are of course interested in analyzing both the work and the span. As usual, the ratio

of the two will give us the parallelism of the algorithm. The examples we use are all algorithms for the the following problem:

Problem 5.69 (Primes)**.** *The* primes *problem is given an integer n to find all prime numbers up to, and including n.*

Recall that a integer i is a prime if it has no positive divisors other than 1 and itself. We note that if an integer n is not prime, then it must have a divisor that is at most $\lceil \sqrt{n} \rceil$ since for any $i \times j = n$, either i or j has to be less than or equal to $\lceil \sqrt{n} \rceil$. We therefore can check if n is a prime by checking whether any j, $2 \leq j \leq \lceil \sqrt{n} \rceil$ is a divisor of n. This can be checked as follows:

Algorithm 5.70.

$$\texttt{isPrime}(n) \;=\; (|\langle 2 \leq j \leq \lceil \sqrt{n} \rceil \mid n \bmod j = 0 \rangle| = 0)$$

The length of the sequence is simply the number of j that divide n. If that is zero, then n is a prime. We now consider the work and span of this function based on the array sequence cost specification. The function runs a tabulate to generate a sequence of length $\lceil \sqrt{n} \rceil$ and then filters it. We note that the work for evaluating $n \bmod j = 0$ is constant for each j. With this we can write down the equation:

$$W(|\langle 2 \leq j \leq \lceil \sqrt{n} \rceil \mid i \bmod j = 0 \rangle|) = O\left(1 + \sum_{j=2}^{\lceil \sqrt{n} \rceil} O(1)\right) = O(\sqrt{n})$$

for work, since we just sum the cost of applying $\bmod$ to each j, and then add in 1, as given in Cost Specification 5.8 for `filter`. Similarly we have for span:

$$W(|\langle 2 \leq j \leq \lceil \sqrt{n} \rceil \mid i \bmod j = 0 \rangle|) = O\left(\log(\sqrt{n}) + \max_{j=2}^{\lceil \sqrt{n} \rceil} O(1)\right) = O(\log n)$$

Now that we can determine if an integer is a prime, we can just check by brute force for all integers between 2 and n if they are primes. This leads to the following algorithm.

Algorithm 5.71.

$$\texttt{primes}(n) \;=\; \langle 1 \leq i \leq n \mid \texttt{isPrime}(i) \rangle$$

Again we use Array Sequence to analyze its work and span. We have:

$$\begin{aligned} W(\langle\, 2 \le i \le n \mid \texttt{isPrime}(i)\,\rangle) &= O\left(\sum_{i=2}^{n} W(\texttt{isPrime(i)})\right) \\ &= O(\left(\sum_{i=2}^{n} O(\sqrt{i})\right) \\ &= O(n^{3/2}) \end{aligned}$$

and the span is:

$$\begin{aligned} S(\langle\, 2 \le i \le n \mid \texttt{isPrime}(i)\,\rangle) &= O\left(\log n + \max_{i=2}^{n} S(\texttt{isPrime(i)})\right) \\ &= O\left(\log n + \max_{i=2}^{n} O(\log i)\right) \\ &= O(\log n) \end{aligned}$$

The parallelism is hence $P(n) = W(n)/S(n) = n^{3/2}/\log n$. This is plenty of parallelism. The work for the algorithm, however, can be improved significantly. The observation is that an integer j close to $\lceil\sqrt{n}\,\rceil$ only divide a very small percentage of the integers between 2 and n. In particular 1 out of about $\lceil\sqrt{n}\,\rceil$ of them. It is therefore wasteful to check every integer to see if j divides it. Instead we can generate all multiples of j up to n. In fact we can do this for every $2 \le j < \lceil\sqrt{n}\,\rceil$, and knock out all of their multiples.

Question 5.72. *How do we knock out the multiples?*

To do this we can start with a Boolean sequence of length n with all true values, and then write a false into all the multiples of the js. All the writes can be done in parallel using an `inject`. This can be implemented with the following algorithm:

Algorithm 5.73.

```
function primes(n) =
  let
    val sieves = ⟨(i × j, false) : 2 ≤ i ≤ ⌈√n⌉, 1 ≤ j ≤ ⌈n/i⌉⟩
    val R = inject({true : 0 ≤ i ≤ n}, sieves)
  in
    ⟨i : 2 ≤ i ≤ n | R[i]⟩
  end
```

The work and span for calculating `sieves` is similar to the analysis for finding all subsequences in Example 5.56. In particular generating each multiple takes constant work and span since it just a multiply. The the total number of such sieves, i.e.

the length of `sieves`, which we analyze below. The span is $O(\log n)$ because of the flatten implied by the syntax. The work of `inject` is also proportional to the number of sieves, and its span is constant. The work of the filter (Line 6) is proportional to n, and the span is $O(\log n)$. Therefore the total work is proportional to the length of `sieves`, which is larger than n, and the total span is $O(\log n)$.

To calculate the number of sieves (length of `sieves`) we can add up the number of multiples each j from 2 to $\lceil \sqrt{n} \rceil$ have. This gives:

$$\begin{aligned} |\texttt{sieves}| &= \sum_{i=2}^{\lceil \sqrt{n} \rceil} \left\lceil \frac{n}{i} \right\rceil \\ &\leq (n+1) \sum_{i=2}^{\lceil \sqrt{n} \rceil} \frac{1}{i} \\ &= (n+1) H(\lceil \sqrt{n} \rceil) \\ &\leq (n+2) \ln n^{1/2} \\ &= \frac{n+2}{2} \ln n \end{aligned}$$

Here $H(n)$ is the n^{th} harmonic number, which is known to be bounded below by $\ln n$ and above by $\ln n + 1$. We therefore have: $W(n) = O(n \log n)$ and $S(n) = O(\log n)$. This is a significant improvement in the work.

The work can actually be improved by noticing that j is not a prime we do not have to use its multiples. This is because one of its divisors will include all its multiples. For example we need not consider the multiples of 6 since all multiples of 6 are also multiples of 2 and of 3. The question is how do we generate just the primes less than $\lceil \sqrt{n} \rceil$ for the filters. Well this can be done recursively, giving the following algorithm.

Algorithm 5.74.

```
function primes(n) =
if (n < 2) then ⟨ ⟩
else let
    val P = primes(⌈√n⌉)
    val sieves = ⟨(p × i, false) : p ∈ P, 1 ≤ i ≤ ⌈n/p⌉⟩
    val R = inject({true : 0 ≤ i ≤ n}, sieves)
in
    ⟨i : 2 ≤ i ≤ n | R[i]⟩
end
```

We leave the analysis of this algorithm as an exercise, but we state without justification that it has $O(n \log \log n)$ work and $O(\log n)$ span.

5.10 Single-Threaded Array Sequences

In this course we will be using purely functional code because it is safe for parallelism and enables higher-order design of algorithms by use of higher-order functions. It is also easier to reason about formally, and is just cool. For many algorithms using the purely functional version makes no difference in the asymptotic work bounds—for example quickSort and mergeSort use $\Theta(n \log n)$ work (expected case for quickSort) whether purely functional or imperative. However, in some cases purely functional implementations lead to up to a $O(\log n)$ factor of additional work. To avoid this we will slightly cheat in this class and allow for benign "effect" under the hood in exactly one ADT, described in this section. These effects do not affect the observable values (you can't observe them by looking at results), but they do affect cost analysis—and if you sneak a peak at our implementation, you will see some side effects.

The issue has to do with updating positions in a sequence. In an imperative language updating a single position can be done in "constant time". In the functional setting we are not allowed to change the existing sequence, everything is persistent. This means that for a sequence of length n an update can either be done in $\Theta(n)$ work with an arraySequence (the whole sequence has to be copied before the update) or $\Theta(\log n)$ work with a treeSequence (an update involves traversing the path of a tree to a leaf). In fact you might have noticed that our sequence interface does not even supply a function for updating a single position. The reason is both to discourage sequential computation, but also because it would be expensive.

Consider a function `update` (i, v) S that updates sequence S at location i with value v returning the new sequence. This function would have cost $\Theta(|S|)$ in the arraySequence cost specification. Someone might be tempted to write a sequential loop using this function. For example for a function $f : \alpha -> \alpha$, a `map` function can be implemented as follows:

```
fun map f S =
  iter (fn ((i, S'), v) => (i + 1, update (i, f(v)) S'))
       (0, S)
       S
```

This code iterates over S with i going from 0 to $n - 1$ and at each position i updates the value S_i with $f(S_i)$. The problem with this code is that even if f has constant work, with an `arraySequence` this will do $\Theta(|S|^2)$ total work since every update will do $\Theta(|S|)$ work. By using a `treeSequence` implementation we can reduce the work to $\Theta(|S| \log |S|)$ but that is still a factor of $\Theta(\log |S|)$ off of what we would like.

In the class we sometimes do need to update either a single element or a small number of elements of a sequence. We therefore introduce an ADT we refer to as a *Single Threaded Sequence* (`stseq`). Although the interface for this ADT is quite straightforward, the cost specification is somewhat tricky. To define the cost specification we need to distinguish between the latest "copy" of an instance of an `stseq`, and earlier copies. Basically whenever we update a sequence we create a new "copy", and the old "copy" is still around due to the persistence in functional languages. The cost specification is going to give different costs for updating the latest copy and old copies. Here we will only define the cost for updating and accessing the

latest copy, since this is the only way we will be using an `stseq`. The interface and costs is as follows:

	Work	Span
`fromSeq(S) : α seq → α stseq` Converts from a regular sequence to a stseq.	$O(\|S\|)$	$O(1)$
`toSeq(ST) : α stseq → α seq` Converts from a stseq to a regular sequence.	$O(\|S\|)$	$O(1)$
`nth ST` i `: α stseq → int → α` Returns the i^{th} element of ST. Same as for seq.	$O(1)$	$O(1)$
`update` (i,v) ST `: (int × α) → α stseq → α stseq` Replaces the i^{th} element of ST with v.	$O(1)$	$O(1)$
`inject` I ST `: (int × α) seq → α stseq → α stseq` For each $(i,v) \in I$ replaces the i^{th} element of ST with v.	$O(\|I\|)$	$O(1)$

An `stseq` is basically a sequence but with very little functionality. Other than converting to and from sequences, the only functions are to read from a position of the sequence (`nth`), update a position of the sequence (`update`) or update multiple positions in the sequence (`inject`). To use other functions from the sequence library, one needs to covert an `stseq` back to a sequence (using `toSeq`).

In the cost specification the work for both `nth` and `update` is $O(1)$, which is about as good as we can get. Again, however, this is only when S is the latest version of a sequence (i.e. noone else has updated it). The work for `inject` is proportional to the number of updates. It can be viewed as a parallel version of `update`.

Now with an `stseq` we can implement our map as follows:

Algorithm 5.75.

```
1 fun map f S =
2 let
3   val S' = StSeq.fromSeq(S)
4   val R = iter (fn ((i, S''), v) ⇒ (i + 1, StSeq.update(i, f(v)) S''))
5                (0, S')
6                S
7 in
8   StSeq.toSeq(R)
9 end
```

This implementation first converts the input sequence to an `stseq`, then updates each element of the `stseq`, and finally converts back to a sequence. Since each update takes constant work,

and assuming the function f takes constant work, the overall work is $O(n)$. The span is also $O(n)$ since `iter` is completely sequential. This is therefore not a good way to implement `map` but it does illustrate that the work of multiple updates can be reduced from $\Theta(n^2)$ on array sequences or $O(n \log n)$ on tree sequences to $O(n)$ using an `stseq`.

Implementing Single Threaded Sequences. You might be curious about how single threaded sequences can be implemented so they act purely functional but match the cost specification. Here we will just briefly outline the idea.

The trick is to keep two copies of the sequence (the original and the current copy) and additionally to keep a "change log". The change log is a linked list storing all the updates made to the original sequence. When converting from a sequence to an `stseq` the sequence is copied to make a second identical copy (the current copy), and an empty change log is created. A different representation is now used for the latest version and old versions of an `stseq`. In the latest version we keep both copies (original and current) as well as the change log. In the old versions we only keep the original copy and the change log. Lets consider what is needed to update either the current or an old version. To update the current version we modify the current copy in place with a side effect (non functionally), and add the change to the change log. We also take the previous version and mark it as an old version removing its current copy. When updating an old version we just add the update to its change log. Updating the current version requires side effects since it needs to update the current copy in place, and also has to modify the old version to mark it as old and remove its current copy.

Either updating the current version or an old version takes constant work. The problem is the cost of `nth`. When operating on the current version we can just look up the value in the current copy, which is up to date. When operating on an old version, however, we have to go back to the original copy and then check all the changes in the change log to see if any have modified the location we are asking about. This can be expensive. This is why updating and reading the current version is cheap ($O(1)$ work) while working with an old version is expensive.

In this course we will use `stseq`s for some graph algorithms, including breadth-first search (BFS) and depth-first search (DFS), and for hash tables.

.