Chapter 6

Sequences

A sequence is an ordered set, i.e., is a collection of elements that are totally ordered. Computer scientists use sequence data structures such as arrays and lists to represent many different sorts of data. In this chapter, we specify a sequence abstract data type and consider several cost specifications for it.

6.1 Definitions

Mathematically, a sequence is an enumerated collection, or an ordered set. As with a set, a sequence has ***elements***. The ***length*** of the sequence is the number of elements in the sequence. A sequence can be finite or infinite; in this book, we only consider finite ones. As an ordered set, sequences allow for repetition: an element can appear at multiple positions. The position of an element is called its ***rank*** or its ***index***. Traditionally, the first element of the sequence is given rank 1, but, being computer scientists, we start at 0. We define sequences mathematically as a function, whose domain is natural numbers.

Definition 6.1. An α ***sequence*** is a mapping (function) from $\mathbb{N}$ to α with domain $\{0, \ldots, n-1\}$ for some $n \in \mathbb{N}$.

This mathematical definition might seem pedantic 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 creates a symmetry with the abstract data types such as tables or dictionaries for representing mappings. An important point to notice in the definition is that sequences are parametrized by the type (i.e., set of possible values) of their elements.

Example 6.2. Let $A = \{0, 1, 2, 3\}$ and $B = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$. The relation

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

is a function from A to B with domain $\{0, 1, 3\}$ 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

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

from A to B is a sequence because its domain matches A. The first element of the sequence is $\mathtt{a}$ and thus has rank 0. The second element is $\mathtt{b}$ and has rank 1. The length of the sequence is 4.

Mathematicians use a special notation for writing sequences. We shall also use such a notation, writing for example the sequence $\{(1, \mathtt{b}), (3, \mathtt{a}), (2, \mathtt{a}), (0, \mathtt{a})\}$ as $\langle\, \mathtt{a}, \mathtt{b}, \mathtt{a}, \mathtt{a} \,\rangle$.

Syntax 6.3. [Sequences and Indexing] The notation $\langle\, a_0, a_1, \ldots, a_{n-1} \,\rangle$ is shorthand for the sequence $\{(0, a_0), (1, a_1), \ldots, ((n-1), a_{n-1})\}$. For a sequence a, we write $a[i]$ to refer to the element of a at position i and $a[l, \ldots, h]$ to refer to the subsequence of a restricted to the position between l and h.

Since they occur frequently, we use special notation and terminology for sequences with two elements and sequences of characters.

- An ***ordered pair*** (x, y) is a pair of elements in which the element on the left, x, is identified as the ***first*** entry, and the one on the right, y, as the ***second*** entry.
- We refer to a sequence of characters as a ***string***, and use the standard double-quote syntax for them, e.g., "$c_0c_1c_2 \ldots c_{n-1}$" is a string consisting of the n characters $c_0 \ldots c_{n-1}$.

Example 6.4. Some example sequences follow.

- Consider the integer sequence (or $\mathbb{Z}$ sequence)
  $a = \langle\, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 \,\rangle$.
  We have $a[0] = 2$, $a[2] = 5$, and $a[1, \ldots, 4] = \langle\, 3, 5, 7, 11 \,\rangle$.
- A character sequence, or a string: $\langle\,$`'s'`, `'e'`, `'q'`$\,\rangle \equiv$ `"seq"` .
- An (integer ** 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:

$$\langle\, (\lambda\, x \,.\, x^2), (\lambda\, y \,.\, y + 2), (\lambda\, x \,.\, x - 4) \,\rangle\ .$$

6.2 The Sequence Abstract Data Type

ADT 6.5 shows the interface of an abstract data type for sequences. Sequences are one of the most prevalent ADT's used in this book, and more generally in computing. For economy in writing, communication, and cognition, we adopt a small extension of the standard mathematical notation on sequences, called ***sequence comprehensions***. Syntax 6.6 shows this notation on sequences. In the rest of this section, we describe the semantics of sequences and the comprehension-based syntax. Throughout, we use e and its variants to for SPARC expression. When specifying the semantics of operation, we rely on the mathematical definition of sequences as a partial function whose domain is natural numbers.

The sequence ADT can be broadly divided into several categories.

- Operations such as `nth` and `length` that return an element of the sequence or a particular property of it.
- Constructors that create sequences such as `empty`, `singleton`, and `tabulate`.
- Operations such as `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 `update` and `inject` that updates the elements of a sequence.
- Operations such as `iterate`, `reduce`, and `scan` that ***aggregate*** information over the elements of the sequence.

Length and indexing. Given a sequence a, `length` a, also written $|a|$, returns the length of a. The function `nth` returns the element of a sequence at a specified index, e.g. `nth` a 2, written $a[2]$, returns the element of a with rank 2. If the element demanded is out of range, the behavior is undefined and leads to an error.

Abstract Data Type 6.5. [Sequences] For a value type α, the ***sequence data type*** is the type $\mathbb{S}_\alpha$ consisting of the set of all α sequences, and the following values and functions on $\mathbb{S}_\alpha$. In the specifiction $\mathbb{B} = \{\texttt{true}, \texttt{false}\}$ and $\mathcal{O} = \{\mathit{less}, \mathit{greater}, \mathit{equal}\}$.

`length`	:	$\mathbb{S}_\alpha \to \mathbb{N}$
`nth`	:	$\mathbb{S}_\alpha \to \mathbb{N} \to \alpha$
`empty`	:	$\mathbb{S}_\alpha$
`singleton`	:	$\alpha \to \mathbb{S}_\alpha$
`tabulate`	:	$(\mathbb{N} \to \alpha) \to \mathbb{N} \to \mathbb{S}_\alpha$
`map`	:	$(\alpha \to \beta) \to \mathbb{S}_\alpha \to \mathbb{S}_\beta$
`subseq`	:	$\mathbb{S}_\alpha \to \mathbb{N} \to \mathbb{N} \to \mathbb{S}_\alpha$
`append`	:	$\mathbb{S}_\alpha \to \mathbb{S}_\alpha \to \mathbb{S}_\alpha$
`filter`	:	$(\alpha \to \mathbb{B}) \to \mathbb{S}_\alpha \to \mathbb{S}_\alpha$
`flatten`	:	$\mathbb{S}_{\mathbb{S}_\alpha} \to \mathbb{S}_\alpha$
`update`	:	$\mathbb{S}_\alpha \to (\mathbb{N} \times \alpha) \to \mathbb{S}_\alpha$
`inject`	:	$\mathbb{S}_\alpha \to \mathbb{S}_{\mathbb{N}\times\alpha} \to \mathbb{S}_\alpha$
`isEmpty`	:	$\mathbb{S}_\alpha \to \mathbb{B}$
`isSingleton`	:	$\mathbb{S}_\alpha \to \mathbb{B}$
`iterate`	:	$(\alpha \times \beta \to \alpha) \to \alpha \to \mathbb{S}_\beta \to \alpha$
`reduce`	:	$(\alpha \times \alpha \to \alpha) \to \alpha \to \mathbb{S}_\alpha \to \alpha$
`scan`	:	$(\alpha \times \alpha \to \alpha) \to \alpha \to \mathbb{S}_\alpha \to (\mathbb{S}_\alpha \times \alpha)$
`collect`	:	$(\alpha \times \alpha \to \mathcal{O}) \to \mathbb{S}_{\alpha\times\beta} \to \mathbb{S}_{\alpha\times\mathbb{S}_\beta}$

Empty and singleton. The value `empty` is the empty sequence, $\langle\,\rangle$. The function `singleton` takes an element and returns a sequence containing that element, e.g., `singleton` 1 evaluates to $\langle\, 1 \,\rangle$.

Tabulate. The function `tabulate` takes a function f and an natural number n and produces a sequence of length n by applying f at each position. The function f can be applied to each element in parallel. We specify `tabulate` as follows

$$\texttt{tabulate}\;(f : (\mathbb{N} \to \alpha))\;(n : \mathbb{N}) : \mathbb{S}_\alpha = \langle\, f(0), f(1), \ldots, f(n-1) \,\rangle\,.$$

We use the following syntax for `tabulate` operation

$$\langle\, e : 0 \le i < e_n \,\rangle \equiv \texttt{tabulate}\;(\lambda\, i \,.\, e)\; e_n,$$

where e and e_n are expressions, the second evaluating to an integer, and i is a variable. More generally, we can also start at any other, as in:

$$\langle\, e : e_j \le i < e_j \,\rangle\,.$$

Syntax 6.6. [Syntax for Sequences] The table below defines the syntax for the sequence ADT used throughout this book . 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 SPARC expression, e_n and e'_n are SPARC expressions whose values are natural numbers, e_s is a SPARC expression whose value is a sequence, p is a SPARC pattern that binds one or more variables.

$\lvert e_s \rvert$	$\equiv$	$\texttt{length}\,(e_s)$
$e_s[i]$	$\equiv$	$\texttt{nth}\,(e_s)$
$\langle\,\rangle$	$\equiv$	$\texttt{empty}$
$\langle\, e\,\rangle$	$\equiv$	$\texttt{singleton}\,(e)$
$\langle\, e : 0 \le i < e_n \,\rangle$	$\equiv$	$\texttt{tabulate}\,(\lambda\, i\,.\,e)\; e_n$
$\langle\, e : p \in e_s \,\rangle$	$\equiv$	$\texttt{map}\,(\lambda\, p\,.\,e)\; e_s$
$\langle\, x \in e_s \mid e \,\rangle$	$\equiv$	$\texttt{filter}\,(\lambda\, x\,.\,e)\; e_s$
$A[e_l, \cdots, e'_n]$	$\equiv$	$\texttt{subseq}\,(A, e_l, e'_n - e_l + 1)$
$e_s \mathbin{++} e'_s$	$\equiv$	$\texttt{append}\; e_s\; e'_s$

Map. A common operation on sequences is to apply some computation to each element of a sequence. For example we might want to add five to each element of a sequence. For this purpose, we can use the operation map, which takes a function f and a sequence a and applies the function f to each element of a returning a sequence of equal length with the results. We can specify the behavior of map as follows

$$\texttt{map}\;(f : \alpha \to \beta)\;(a : \mathbb{S}_\alpha) : \mathbb{S}_\beta = \{(i, f\;x) : (i, x) \in a\}$$

or equivalently as

$$\texttt{map}\;(f : \alpha \to \beta)\;\langle\, a_1, \ldots, a_{n-1} \,\rangle : \mathbb{S}_\alpha) : \mathbb{S}_\beta = \langle\, f(a_1), \ldots, f(a_{n-1}) \,\rangle\,.$$

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 use the following syntax for the map function

$$\langle\, e : p \in e_s \,\rangle \equiv \texttt{map}\;(\lambda\, p\,.\,e)\; e_s,$$

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)).

Filter. To filter out elements from a given sequence, we can use the function filter. The function takes a Boolean function f and a sequence a as arguments and applies f to each element of a, returning the sequence consisting exactly of those elements of $s \in a$ for which $f(s)$

returns true, and maintaining the order of the elements returned. We can specify the behavior of `filter` as follows

$$\texttt{filter}\,(f : \alpha \to \mathbb{B})\,(a : \mathbb{S}_\alpha) : \mathbb{S}_\alpha = \{(|\{(j,y) \in a \mid j < i \wedge f\ y\}|, x) : (i,x) \in a \mid f\ x\}.$$

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

We use the following syntax for the `filter` function

$$\langle\, x \in e_s \mid e \,\rangle \equiv \texttt{filter}\,(\lambda\, x \,.\, e)\; e_s,$$

where e and e_s are expressions. In the syntax, note the distinction between the colon (:) and the bar (|). We use the colon to draw elements from a sequence for mapping and we use the bar to select the elements that we wish to filter. We can use them together, as in:

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

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

Example 6.7. Given the function `fib` i, which returns the i^{th} Fibonacci number, the expression:

$$a = \langle\, \texttt{fib}\, i : 0 \le i < 9 \,\rangle$$

is equivalent to

$$a = \texttt{tabulate fib } 9.$$

When evaluated, it returns the sequence

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

The expression

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

is equivalent to

$$\texttt{map}\ (\lambda\, x \,.\, x^2)\ a.$$

When evaluated it returns the sequence:

$$\langle\, 0, 1, 1, 4, 25, 64, 169, 441, 1156 \,\rangle\,.$$

Given the function `isPrime` x which checks if x is prime, the expression

$$\langle\, x \ :\ x \in a \mid \texttt{isPrime}\ x \,\rangle$$

is equivalent to

$$\texttt{filter isPrime}\ a.$$

When evaluated, it returns the sequence $\langle\, 2, 5, 13 \,\rangle$.

Subsequences. The $\texttt{subseq}(a, i, j)$ function extracts a contiguous subsequence starting at location i and with length j. If the subsequence is out of bounds of a, only the part within a is returned. We can specify `subseq` as follows

$$\texttt{subseq}\ (a : \mathbb{S}_\alpha)\ (i : \mathbb{N})\ (j : \mathbb{N}) : \mathbb{S}_\alpha = \{(k - i, x) : (k, x) \in a \mid i \le k < i + j\}.$$

We use the following syntax for denoting subsequences

$$a[e_i, \ldots, e_j] \equiv \texttt{subseq}(a, e_i, e_j - e_i + 1).$$

As we shall see in the rest of this book, many algorithms operate inductively on a sequence by splitting the sequence into parts, consisting for example, of the first element and the rest, a.k.a., the ***head*** and the ***tail***, or the first half or the second half. We could define functions such as `splitHead`, `splitMid`, `take`, and `drop` for these purposes. Since all of these are trivially expressible in terms of subsequences, we omit their discussion for simplicity.

Append and flatten. For constructing large sequences from smaller ones, the sequence ADT provides the functions `append` and `flatten`. The function `append` (a, b) appends the sequence b after the sequence a. More precisely, we can specify `append` as follows

$$\texttt{append}\,(a : \mathbb{S}_\alpha)\;(b : \mathbb{S}_\alpha) : \mathbb{S}_\alpha = a \cup \{(i + |a|, x) : (i, x) \in b\}$$

We write $a \mathbin{++} b$ as a short form for `append` $a\ b$.

To append more than two sequences the `flatten` a function takes a sequence of sequences and flattens them. If the input is a sequence $a = \langle\, a_1, a_2, \ldots, a_n \,\rangle$ it appends all the a_i's. We can specify `flatten` more precisely as follows

$$\texttt{flatten}\,(a : \mathbb{S}_{\mathbb{S}_\alpha}) : \mathbb{S}_\alpha \qquad = \{(i + \sum_{(k,c)\in a, k<j} |c|, x) \ : (i, x) \in b, (j, b) \in a\}.$$

Example 6.8. The `append` operation $\langle\, 1, 2, 3 \,\rangle \mathbin{++} \langle\, 4, 5 \,\rangle$ yields the sequence $\langle\, 1, 2, 3, 4, 5 \,\rangle$.

The `flatten` operation `flatten` $\langle\, \langle\, 1, 2, 3 \,\rangle, \langle\, 4 \,\rangle, \langle\, 5, 6 \,\rangle \,\rangle$ yields the sequence $\langle\, 1, 2, 3, 4, 5, 6 \,\rangle$.

Updates and injections. The function `update`$(a, (i, x))$, updates location i of sequence a to contain the value x. If the location is out of range for the sequence, the function returns the input sequence unchanged. We can specify `update` as follows

$$\begin{aligned}\texttt{update}\ \ &(a : \mathbb{S}_\alpha)\;(i : \mathbb{N}, x : \alpha) : \mathbb{S}_\alpha \\ &= \begin{cases} \{(j, y) : (j, y) \in a \mid j \neq i\} \cup \{(i, x)\} & \text{if } 0 \leq i < |a| \\ a & \text{otherwise.} \end{cases}\end{aligned}$$

To update multiple positions at once, we can use `inject`. The function `inject`(a, b) takes a sequence b 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, one of the updates take effect.

In the case of duplicates in the update sequence b, i.e., multiple updates to the same position, we leave it unspecified which update takes effect. The operation `inject` may thus treat duplicate updates non-deterministically.

Example 6.9. Given the string sequence

$$a = \langle \texttt{"the"}, \texttt{"cat"}, \texttt{"in"}, \texttt{"the"}, \texttt{"hat"} \rangle,$$

$$\texttt{update}\ a\ (1, \texttt{"rabbit"})$$

magically yields

$$\langle \texttt{"the"}, \texttt{"rabbit"}, \texttt{"in"}, \texttt{"the"}, \texttt{"hat"} \rangle$$

since location 1 is updated with `"rabbit"`. The expression

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

could yield

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

since location 0 is updated with `"a"`, location 1 with `"dog"`, and location 4 with `"bog"`. It could also yield

$$\langle \texttt{"a"}, \texttt{"dog"}, \texttt{"in"}, \texttt{"the"}, \texttt{"log"} \rangle$$

The entry with location 6 is ignored since it is out of range for a.

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. Given a sequence of key-value pairs, we might want to ***collect*** together all the values for a given key. 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.

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

$$\texttt{collect} : (cmp : \alpha \times \alpha \to \mathcal{O}) \to (a : \mathbb{S}_{\alpha \times \beta}) \to \mathbb{S}_{\alpha \times \mathbb{S}_\beta}.$$

Here the "order set" $\mathcal{O} = \{\texttt{less}, \texttt{equal}, \texttt{greater}\}$.

The first argument cmp is a function for comparing keys of type α, and must define a total order over the keys. The second argument a is a sequence of key-value pairs. The `collect` function collects all values in a that share the same key together into a sequence, ordering the values in the same order as their appearance in the original sequence.

Example 6.10. The following sequence shows a sequence of key-value pairs consisting of our students from last semester and the classes they take.

$$
\begin{array}{lll}
kv & = \langle & (\texttt{"jack"}, \texttt{"15210"}), (\texttt{"jack"}, \texttt{"15213"}) \\
 & & (\texttt{"mary"}, \texttt{"15210"}), (\texttt{"mary"}, \texttt{"15213"}), (\texttt{"mary"}, \texttt{"15251"}), \\
 & & (\texttt{"peter"}, \texttt{"15150"}), (\texttt{"peter"}, \texttt{"15251"}), \\
 & & \ldots\rangle.
\end{array}
$$

We can determine the classes taken by each student by using `collect`~*cmp*, where *cmp* is a comparison function for strings

$$
\begin{array}{lll}
\texttt{collect}\ cmp\ kv & = \langle & (\texttt{"jack"}, \langle \texttt{"15210"}, \texttt{"jack"}, \texttt{"15213"}, \ldots\rangle) \\
 & & (\texttt{"mary"}, \langle \texttt{"15210"}, \texttt{"15213"}), \texttt{"15251"}, \ldots\rangle), \\
 & & (\texttt{"peter"}, \langle \texttt{"15150"}, \texttt{"15251"}, \ldots\rangle), \\
 & & \ldots\rangle.
\end{array}
$$

Note that the output sequence is ordered based on the first instance of their key in the input sequences. Similarly, the order of the classes taken by each student are the same as in the input sequence.

Checking for empty and singularity sequences. Many algorithms need to distinguish trivial sequences such as empty sequences and singular sequences, which contain only one element, from other non-trivial sequences. The functions `isEmpty` and `isSingular` can be used to check for this purpose; the functions return respectively return `true` if the sequence is empty or singular and return `false` otherwise.

Aggregation with `iterate`. Iteration is a key concept in computing, and specifically in algorithm design. Iteration involves a sequence of steps, taken one after another, where each step transforms the state from the previous step. Iteration is therefore an inherently sequential process. The function `iterate` can be used to create a computation that iterates over a sequence while accumulating information. It starts with an initial state and a sequence, and on each step updates the state based on the next element of the sequence. The function `iterate` has the type signature

$$\texttt{iterate}\ (f\texttt{:}\ \alpha \times \beta \to \alpha)\ (x\texttt{:}\ \alpha)\ (a\texttt{:}\ \mathbb{S}_\beta)\ \texttt{:}\ \alpha$$

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

We can define the semantics of `iterate` inductively as follows.

$$
\texttt{iterate}\ f\ x\ a = \begin{cases} x & \text{if } |a| = 0 \\ f(x, a[0]) & \text{if } |a| = 1 \\ \texttt{iterate}\ f\ (f(x, a[0]))(a[1, \ldots, |a| - 1]) & \text{otherwise.} \end{cases}
$$

The function `iterate` computes its final result by computing a new state for each element of the function

$$\begin{aligned} x_0 &= x \\ x_1 &= f(x_0, a[0]) \\ x_2 &= f(x_1, a[1]) \\ &\vdots \\ x_n &= f(x_{n-1}, a[n-1]), \end{aligned}$$

where $n = |a|$ and the final result is x_n.

We can similarly define a variant, `iteratePrefixes`, which takes the same arguments but returns all the intermediate values computed as a sequence, i.e., $\langle\, x_0, x_1, \ldots, x_{n-1} \,\rangle$.

As an application of iteration, consider the problem of finding whether a string (sequence) of left and right parentheses is properly matched or nested. 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 `')'`.

Problem 6.11. [Parentheses matching] The *parentheses matching* problem requires determining whether a given a string of parentheses is matched.

There are a several algorithms for solving this problem. Here we consider a linear-work sequential algorithm based on `iterate`. In Chapter 8, we present a divide-and-conquer algorithm that requires no more work asymptotically, but has a low span. Using iteration, 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. This observation leads to the following algorithm.

Algorithm 6.12.

```
matchParens $a$ =
let
  count $(s, x)$ =
    case $(s, x)$
    | (None, _) => None
    | (Some $n$, ')') => if ($n$ = 0) then None else Some ($n$ - 1)
    | (Some $n$, '(') => Some ($n$ + 1)
in
   (iterate count (Some 0) $a$) = Some 0
end
```

The algorithm starts with the state 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 Some 0 then the counter never went below zero and ended up at zero so the parentheses must be matched.

Iteration is a powerful technique but can be too big of a hammer, especially when used unnecessarily. For example, when summing the elements in a sequence, we don't need to perform the addition operations in a particular order because addition operations are associative and thus they can be performed in any order desired. The iteration-based algorithm for computing the sum does not take advantage of this property, computing instead the sum in a left-to-right order. As we will see next, we can use the associativity of the addition operations to sum up the elements of a sequence in parallel.

Aggregation with `reduce`. Reduction is a key concept in the design of parallel algorithms. 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.

Recall that associative operations are defined as operations that allow commuting the order of operations.

Definition 6.13. 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.

Associativity implies that when applying f to some 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).

Many functions are associative. For example, addition and multiplication on natural numbers are associative, with 0 and 1 as their identities, respectively. Minimum and maximum are also associative with identities ∞ and $-\infty$ respectively. The `append` function on sequences is associative, with identity being the empty sequence. The union operation on sets is associative, with the empty set as the identity. An important class of operations that are not associative is floating-point operations. These operations are typically not associative because performing a set of operations in different orders can lead to different results because of loss of precision.

In the sequence ADT, we use the function `reduce` to perform a reduction over a sequence by applying an associative binary operation to the elements of the sequence until the result is reduced to a single value. The operation function has the type signature

$$\texttt{reduce}\ (f\texttt{:}\ \alpha \times \alpha \to \alpha)\ (id\texttt{:}\ \alpha)\ (a : \mathbb{S}_\alpha)\ \texttt{:}\ \alpha$$

where f is an associative function, a is the sequence, and id is the ***left identity*** of f, i.e., $f(id, x) = x$ for all $x \in \alpha$.

When applied to an input sequence with a function f, `reduce` returns the "sum" with respect to f of the input sequence. In fact if f is associative this sum in equal to iteration. We can define the behavior of `reduce` inductively as follows

$$\texttt{reduce}\ f\ id\ a = \begin{cases} id & \text{if } |a| = 0 \\ a[0] & \text{if } |a| = 1 \\ f\Big(\texttt{reduce}\ f\ id\ (a[0,\ldots,\lfloor\frac{|a|}{2}\rfloor - 1]), & \\ \quad \texttt{reduce}\ f\ id\ (a[\lfloor\frac{|a|}{2}\rfloor,\ldots,|a|-1]\Big) & \text{otherwise.} \end{cases}$$

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

$$\texttt{reduce}\ f\ id\ a = \texttt{iterate}\ f\ id\ a.$$

Example 6.14. The expression
`reduce append ⟨ ⟩  ⟨"another","way","to","flatten"⟩`
evaluates to
`"anotherwaytoflatten"`.

Even though the input-output behavior of `reduce` and `iterate` may match, their cost specifications differ: unlike `iterate`, which is strictly sequential, `reduce` is parallel. In fact, as we will describe in Section 6.4, the span of `iterate` is linear in the size of the input, whereas the span of `reduce` is logarithmic.

The results of `reduce` and `iterate` may differ when 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 may lead to different answers. To deal properly with functions that are non-associative, the specification above therefore makes precise the order in which the argument function `f` is applied. For instance, when reducing with floating point addition or multiplication, we will need to take the order of operations into account. Note that every (correct) implementation of `reduce` must return the same result: the result is deterministic regardless of the specifics of the algorithm used in the implementation.

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`. 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. For example, in our parenthesis matching algorithm (Algorithm 6.12), we used this property crucially by defining our function to propagate a mismatched parenthesis forward in the computation. We now describe a function called `scan` that allows using the results of intermediate computations and also does so in parallel.

The term "scan" refers to a computation that reduces every prefix of a given sequence by repeatedly applying an associative binary operation. The `scan` function has the type signature

$$\texttt{scan}\ (f\texttt{:}\ \alpha\ \times\ \alpha\ \rightarrow\ \alpha)\ \ (id\texttt{:}\ \alpha)\ \ (a : \mathbb{S}_\alpha)\ \texttt{:}\ (\mathbb{S}_\alpha\ \times\ \alpha)\texttt{,}$$

where f is an associative function, a is the sequence, and id is the left identity element of f. The expression $\texttt{scan} f a$ returns the "sum" with respect to f of all prefixes of the input sequence a. For this reason, the `scan` operation is sometimes called the ***prefix sums*** operation. We can define `scan` inductively as follows

$$\texttt{scan}\ f\ id\ a = (\langle\ \texttt{reduce}\ f\ id\ a[0 \ldots i] : 0 \leq i < |a|\ \rangle\ ,\ \texttt{reduce}\ f\ id\ a).$$

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 6.15. Consider the sequence $a = \langle\, 0, 1, 2 \,\rangle$. The prefixes of a are

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

The prefixes of a sequence are all the subsequences of the sequence that starts at its beginning. Empty sequence is a prefix of any sequence. The computation `scan` $+\ 0\ \langle\, 0, 1, 2 \,\rangle$ can be written as

$$\begin{array}{lcl} \texttt{scan} + 0\ \langle\, 0,1,2 \,\rangle & = & (\quad \langle \quad \texttt{reduce} + 0\ \langle\, \,\rangle\,, \\ & & \qquad\quad \texttt{reduce} + 0\ \langle\, 0 \,\rangle\,, \\ & & \qquad\quad \texttt{reduce} + 0\ \langle\, 0,1 \,\rangle \\ & & \qquad \rangle\,, \\ & & \qquad\quad \texttt{reduce} + 0\ \langle\, 0,1,2 \,\rangle \\ & &) \\ & = & \qquad\quad (\langle\, 0,0,1 \,\rangle\,, 3). \end{array}$$

The computation `scanI` $+\ 0\ \langle\, 0, 1, 2 \,\rangle$ can be written as

$$\begin{array}{lcl} \texttt{scanI} + 0\ \langle\, 0,1,2 \,\rangle & = & \langle \quad \texttt{reduce} + 0\ \langle\, 0 \,\rangle\,, \\ & & \qquad \texttt{reduce} + 0\ \langle\, 0,1 \,\rangle\,, \\ & & \qquad \texttt{reduce} + 0\ \langle\, 0,1,2, \,\rangle \\ & & \rangle \\ & = & \quad \langle\, 0,1,3 \,\rangle\,. \end{array}$$

Since `scan` can be specified in terms of reduce, one might be tempted to argue that it is redundant. In fact, it is not: 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`.

Example 6.16. [Copy scan] Scan is useful when we 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. For example, suppose that we are given a sequence of type $\mathbb{S}_{\mathbb{N}}$ consisting only of natural numbers and asked to return a sequence of the same length where each element receives the previous positive value. For the example, for input $\langle\, 0,\ 7,\ 0,\ 0,\ 3,\ 0\,\rangle$, the result should be $\langle\, 0,\ 0,\ 7,\ 7,\ 7,\ 3\,\rangle$.

Using a sequential loop or `iterate` would be easy. To solve this problem using `scan` we need a combining function f. Consider the function

$$\texttt{skipZero}\ (x,y)\ =\ \textbf{if}\ \ y > 0\ \ \textbf{then}\ \ y\ \ \textbf{else}\ \ x\,.$$

The function passes its right argument if it is positive, 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 for all x, y and z, we have

$$\texttt{skipZero}(x, \texttt{skipZero}(y,z)) = \texttt{skipZero}(\texttt{skipZero}(x,y),z).$$

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, it is likewise easy to verify that both orderings give y. Finally, 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.

To use `skipZero` as part of the scan operation, we need to find its left identity. We can see that for any natural number y

$$\texttt{skipZero}\ (0,y)\ =\ \ y,$$

and that for any natural number x

$$\texttt{skipZero}\ (x,0)\ =\ x.$$

Thus 0 is the left identity for `skipZero`.

Remark 6.17. Experience in parallel computing shows that `reduce` and `scan` are powerful primitives that suffice to express many parallel algorithms on sequences. In some ways this is not surprising, because the operations allow using two important algorithm-design techniques: `reduce` operation allows expressing divide-and-conquer algorithms and the `scan` operation allows expressing a iterative algorithms.

6.3 Sequence Comprehensions

Notation such as $\{x^2 : x \in a \mid \texttt{isPrime}\ a\}$ in which one set is defined in terms of the elements of other sets, and conditions on them is referred to as a ***set comprehensions***. 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 economy of expression and "comprehension" that they offer. In this book we use the comprehension syntax for a similar reason: they allow expressing algorithms in clear, concise notation. For example, the syntax for sequences shown in Syntax 6.6, as well as sets which will see later in Chapter 13, are based on set comprehensions.

In the examples shown thus far, we have mostly used sequences in "flat" fashion, without nesting sequence primitives within each other. In this section, we consider more uses of sequence comprehensions that involve nesting. As our first example, suppose that we wish to create a sequence consisting of points in two dimensional space (x, y) whose coordinates are natural numbers that satisfy the conditions that $0 \le x \le n-1$ and $1 \le y \le n$. For example, for $n = 3$, we would like to construct the sequence $\langle\,(0,1),(0,2),(1,1),(1,2),(2,1),(2,2)\,\rangle$. The code below shows one way to generate such a sequence.

```
points2D n =
  flatten (tabulate (λx.tabulate (λy.(x,y+1)) n) n)
```

The algorithm first generates a sequence of the form

$$\begin{array}{ll}\langle & \langle\,(0,1),(0,2),\ldots,(0,n)\,\rangle \\ & \langle\,(1,1),(1,2),\ldots,(1,n)\,\rangle \\ & \vdots \\ & \langle\,(n-1,1),(n-1,2),\ldots,(n-1,n)\,\rangle \\ \rangle & ,\end{array}$$

and then concatenates the inner sequences by using `flatten`.

Using our sequence comprehension notation, we can express the same code more succinctly as

```
points2D n =
  flatten $\langle\,\langle\,(x,y) : 1 \le y \le n\,\rangle : 0 \le x \le n-1\,\rangle$.
```

We simplify this notation a bit more and write the same algorithm as

```
points2D n =
   $\langle\,(x,y) : 1 \le y \le n, 0 \le x \le n-1\,\rangle$.
```

In this notation, we allow the expression to define the members of a sequence by using multiple variables. Note that there is an implicit `flatten` in such notation. We will have to remember this point, as we analyze algorithms that range over multiple sequences.

The notation generalizes to arbitrary levels of nesting. For example, we may want to add one more dimension to point sequences by considering points in three dimensional space, (x, y, z), whose coordinates are natural numbers that satisfy the conditions that $0 \leq x \leq n-1$, $1 \leq y \leq n$, $2 \leq z \leq n+1$. We can write the code for this by nesting the sequences in three levels as follows.

```
points3D n =
  flatten $\langle \langle \langle (x,y,z) : 2 \leq z \leq n+1 \rangle : 1 \leq y \leq n \rangle : 0 \leq x \leq n-1 \rangle$
```

or more succinctly as

```
points3D n =
  $\langle (x,y,z) : 2 \leq z \leq n+1, 1 \leq y \leq n, 0 \leq x \leq n-1 \rangle$.
```

We can also nest other sequence operations. For example, suppose that we wish to compute the Cartesian product of two sequences, e.g., given $a = \langle 1, 2 \rangle$, and
$b = \langle 3.0, 4.0, 5.0 \rangle$.
The Cartesian product, $a \times b = \langle (1, 3.0), (1, 4.0), (1, 5.0), (2, 3.0), (2, 4.0), (2, 5.0) \rangle$. We can write the code for a function for computing as

```
CartesianProduct $(a,b)$ =
  flatten (map (λ x.map (λ y.(x,y)) b) a).
```

or equivalently as

```
CartesianProduct $(a,b)$ =
  $\langle (x,y) : x \in a, y \in b \rangle$.
```

Note that the resulting sequence is ordered by the natural lexicographic generalization of the ordering of all the sequences involved.

In general, we can `tabulate`, `map` and `filter` over any number of sequences.

Syntax 6.18. [Comprehensions for multiple sequences] We can sample from any finitely many sequences and compute an expression in terms of their elements, while also filtering the elements using finitely many expressions:

$$\left\langle\, e : x_1 \in e_1, x_2 \in e_2 \ldots, x_n \in e_n \mid e'_1, e'_2 \ldots e'_m \,\right\rangle.$$

We can also allow variable binding involving ranges of natural numbers, as for example, can be used by `tabulate`. Specifically, $x_i \in e_i$ could be replaced by $e_j \leq i \leq e_k$, where e_j and e_k are expressions whose values are natural numbers and i is a variable.

Example 6.19. Given sequences a of natural numbers and b of letters of the alphabet, we wish to compute the sequence that pairs each even element of a with all elements of b that are vowels. We can writes this simply by adding the filtering predicates `isEven`, which holds for even numbers, and `isVowel`, which holds for vowels.

$$\langle\, (x,y) : x \in a, y \in b \mid \texttt{isEven } x,\ \texttt{isVowel } y \,\rangle.$$

Example 6.20. Let's say we want to generate all contiguous subsequences of a sequence a. Each sequence can start at any position $0 \le i < |a|$, and end at any position $i \le j < |a|$. We can do this with the following pseudocode

$$\langle\, a\,\langle\, i, \ldots, j \,\rangle : 0 \le i < |a|, i \le j < |a| \,\rangle\ ,$$

which is equivalent to

$$\texttt{flatten (tabulate (}\boldsymbol{\lambda}\, i\,\textbf{.}\,\texttt{tabulate (}\boldsymbol{\lambda}\, j\,\textbf{.}\,\texttt{a}[i, \ldots, i+j]\texttt{)}\ |a| - i - 1\texttt{)}\ |a|\texttt{)}.$$

This example shows again that comprehensions can be quite convenient.

Remark 6.21. [Comprehensions] 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 6.6).

6.4 Cost Specification

So far in this chapter, we have only specified the behavior of the operations in the sequence ADT. In this section, we consider several different cost specifications for the sequence ADT that are implemented by using arrays, trees, and lists. The cost specifications indicate the cost for the class of implementations that can achieve these cost bounds, keeping in mind that there can be many specific implementations that match the bounds. For example, for the tree-sequence specification, an implementations can use one of many balanced binary tree data structures available. To apply the cost bounds, we don't need to know the details of how these implementations work. Cost specifications can thus be viewed as an abstraction over implementation details that do not matter for the purposes of the algorithm.

Since there usually are many ways to implement an ADT specification, there can be multiple cost specifications for an ADT. 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 but there are trade-offs: while some operations may be cheaper in one specification and other operations may be more expensive.

Such trade-offs are common and should be considered when selecting which cost specification to use. When designing an algorithm, our goal would be to choose the specification that minimizes the cost for the algorithm. For example, as we will see soon, if an algorithm makes many calls to `nth` but no calls to `append`, then we would use the array-sequence specification rather than the tree-sequence specification. Conversely, if the algorithm mostly uses `append` and `update`, then tree-sequence specification would be better. After we decide the specification to use, what remains is to select the implementation that matches the specification, which can include additional considerations.

When presenting the cost bounds, we consider the aggregation operations separately, because the cost of such operations depend on the nature of the aggregation performed.

6.4.1 Array Sequences

Cost Specification 6.22 shows the costs for array sequences. Simple operations such as `length` as well as `isEmpty` and `singleton` and `isSingleton` all require constant work and span. Since arrays support random access to any element in constant time, the function `nth` takes constant work and span. For the three operations `tabulate`, `map`, and `filter` the work includes the sum of the work of applying f at each position, as well as an additional unit cost, for the operation itself. In all three operations it is possible to apply the function f in parallel since there is no dependency among the positions. Therefore the span of the functions is the maximum of the span of applying f at each position. The operations `map` and `tabulate` incur an additional unit overhead in the span, but for `filter` the overhead is logarithmic, because `filter` requires compacting or packing the chosen elements contiguously into the result array.

The operation `subseq` has constant work and span. The operation `append` requires work proportional to the length of the sequences they are working on and can be implemented in constant span. The operation `flatten` generalizes `append`, requiring work proportional to the total length of the sequences flattened, and can be implemented in parallel in logarithmic span in the number of sequences flattened. The operations `update` and `inject` both require work proportional to the length of the sequences they are working on, but can be implemented in constant span. It might seem surprising that `update` takes work proportional to the size of the input sequence a, since updating a single element should require constant work. The reason is that the interface is purely functional so that the input sequence needs to be copied–we are not allowed to update the old copy. In Section 6.6, we describe single-threaded array sequences that will allow us under certain restrictions to update a sequence in constant work. The primary cost in implementing `collect` is a sorting step that sorts the sequence based on the keys. The work and span of collect is therefore determined by the work and span of (comparison) sorting with the specified comparison function f.

Cost Specification 6.22. [Array Sequences] We specify the ***array-sequence*** costs as follows. The notation $\mathcal{T}(-)$ refer to the trace of the corresponding operation. The specification for `scan` assumes that f has constant work and span.

Operation	Work	Span
`length` a `singleton` x `isSingleton` x `isEmpty` x	1	1
`nth` $a\ i$	1	1
`tabulate` $f\ n$	$1 + \sum_{i=0}^{n} W\left(f(i)\right)$	$1 + \max_{i=0}^{n} S\left(f(i)\right)$
`map` $f\ a$	$1 + \sum_{x \in a} W\left(f(x)\right)$	$1 + \max_{x \in a} S\left(f(x)\right)$
`filter` $f\ a$	$1 + \sum_{x \in a} W\left(p(x)\right)$	$\log\lvert a\rvert + \max_{x \in a} S\left(f(x)\right)$
`subseq` $a\ (i, j)$	1	1
`append` $a\ b$	$1 + \lvert a\rvert + \lvert b\rvert$	1
`flatten` a	$1 + \lvert a\rvert + \sum_{x \in a} \lvert x\rvert$	$1 + \log\lvert a\rvert$
`update` $a\ (i, x)$	$1 + \lvert a\rvert$	1
`inject` $a\ b$	$1 + \lvert a\rvert + \lvert b\rvert$	1
`collect` $f\ a$	$1 + W\left(f\right) \cdot \lvert a\rvert \log\lvert a\rvert$	$1 + S\left(f\right) \cdot \log^2\lvert a\rvert$
`iterate` $f\ x\ a$	$1 + \sum_{f(y,z) \in \mathcal{T}(-)} W\left(f(y,z)\right)$	$1 + \sum_{f(y,z) \in \mathcal{T}(-)} S\left(f(y,z)\right)$
`reduce` $f\ x\ a$	$1 + \sum_{f(y,z) \in \mathcal{T}(-)} W\left(f(y,z)\right)$	$\log\lvert a\rvert \cdot \max_{f(y,z) \in \mathcal{T}(-)} S\left(f(y,z)\right)$
`scan` $f\ a$	$\lvert a\rvert$	$\log\lvert a\rvert$

Example 6.23. As an example of tabulate and map, we have

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

since the work and span for i^2 is $O\left(1\right)$.

As an example of filter, we have

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

Example 6.24. Consider the code from Example 6.20:

$$e = \langle\, a[i,\ldots,j] : 0 \leq i < |a|, i \leq j < |a| \,\rangle\,,$$

which extracts all contiguous subsequences from the sequence a. Recall that the notation is equivalent to a nested tabulate first over the indices i, and then inside over the indices j. The results are then flatten'ed. The nesting of tabulate's allows all the calls to $a[i,\ldots,j]$ (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 of which has constant work and span according to the cost specifications. The work of the nested tabulate's and the subseq's is therefore $O(n^2)$. The span of the inner tabulate is maximum over the span of the inner subseq's, which is $O\left(1\right)$. The span of the outer tabulate is the maximum over the inner tabulate's, which is again $O\left(1\right)$. The flatten at the end requires $O(n^2)$ work and $O(\lg n)$ span, because $||a|| = n(n+1)/2 = O(n^2)$, and $|a| = n$. The total work and span are therefore

$$\begin{aligned} W\left(e\right) &= O(|a|^2), \text{ and} \\ S\left(e\right) &= O(\lg|a|). \end{aligned}$$

The cost of aggregation operations, iterate, reduce, and scan are somewhat more difficult to specify because they depend on the functions supplied as arguments and more specifically on the intermediate values computed during evaluation.

Cost of `iterate`. The cost of `iterate` depends not only on the arguments but also the intermediate values computed during evaluation. Example 6.25 shows an example where the length of the sequence being iterated over and that of the results are the same but the costs differ due to intermediate values.

Example 6.25. Consider appending the following sequence of strings using `iterate`:

$$\texttt{iterate append " "}\ \langle \texttt{"abc"}, \texttt{"d"}, \texttt{"e"}, \texttt{"f"} \rangle.$$

If we only count the work of `append` operations performed during evaluation, we obtain a total work of 19, because the following `append` operations are performed

1. `"abc"` ++ `"d"` (work 5),
2. `"abcd"` ++ `"e"` (work 6), and
3. `"abcde"` ++ `"f"` (work 7).

Consider now appending the following sequence of strings, which is a permutation of the previous, using `iterate`:

$$\texttt{iterate append " "}\ \langle \texttt{"d"}, \texttt{"e"}, \texttt{"f"}, \texttt{"abc"} \rangle$$

If we only count the work of `append` operations using the array-sequence specification, we obtain a total work of 15, because the following `append` operations are performed

1. `"d"` ++ `"e"` (work 3),
2. `"de"` ++ `"f"`, (work 4) and
3. `"def"` ++ `"abc"` (work 7).

In summary, we have used iteration over two sequences both with 4 elements and obtained different costs even though the sequences are permutations of each other (their elements have the same length). This is because the total cost depends on the intermediate values generated during computation.

To specify its cost, we will consider the intermediate values from the specification of `iterate`, reproduced here for convenience.

$$\texttt{iterate}\ f\ x\ a = \begin{cases} x & \text{if } |a| = 0 \\ f(x, a[0]) & \text{if } |a| = 1 \\ \texttt{iterate}\ f\ (f(x, a[0]))(a[1, \ldots, |a|-1]) & \text{otherwise.} \end{cases}$$

Consider evaluation of `iterate` $f\ v\ A$ and let $\mathcal{T}(\texttt{iterate}\ f\ v\ A)$ denote the set of calls to $f(\cdot,\cdot)$ performed along with the arguments, as defined by the specification above. We refer to this set of function calls as the ***trace*** of `iterate`. We define the cost of `iterate` as the sum of these calls.

Cost Specification 6.26. [Cost for `iterate`] Consider evaluation of `iterate` $f\ v\ a$ and let $\mathcal{T}(\texttt{iterate}\ f\ v\ a)$ denote the set of calls (trace) to $f(\cdot,\cdot)$ performed along with the arguments. The work and span are

$$\begin{aligned} W\left(\texttt{iterate}\ f\ x\ a\right) &= O\left(1 + \sum\nolimits_{f(y,z) \in \mathcal{T}(\texttt{iterate}\ f\ x\ a)} W\left(f(y,z)\right)\right), \text{ and} \\ S\left(\texttt{iterate}\ f\ x\ a\right) &= O\left(1 + \sum\nolimits_{f(y,z) \in \mathcal{T}(\texttt{iterate}\ f\ x\ a)} S\left(f(y,z)\right)\right). \end{aligned}$$

As an interesting example, consider the function `mergeOne` $a\ x$ for merging a sequence a with the singleton sequence $\langle\, x\,\rangle$ by using an assumed comparison function. The function performs $O(n)$ work in $O(\lg n)$ span, where n is the total number of elements in the output sequence. We can use the `mergeOne` function to sort a sequence via iteration as follows

$$\texttt{iterSort}\ a\ =\ \texttt{iterate mergeOne}\ \langle\,\rangle\ a.$$

For example, on input $a = \langle\, 2, 1, 0\,\rangle$, `iterSort` first merges $\langle\,\rangle$ and $\langle\, 2\,\rangle$, then merges the result $\langle\, 2\,\rangle$ with $\langle\, 1\,\rangle$, then merges the resulting sequence $\langle\, 1, 2\,\rangle$ with $\langle\, 0\,\rangle$ to obtain the final result $\langle\, 0, 1, 2\,\rangle$.

The trace for `iterSort` with an input sequence of length n consists of a set of calls to `mergeOne`, where the first argument is a sequence of sizes varying from 1 to $n-1$, while its right argument is always a singleton sequence. For example, the final `mergeOne` merges the first $(n-1)$ elements with the last element, the second to last `mergeOne` merges the first $(n-2)$ elements with the second to last element, and so on. Therefore, the total work for an input sequence a of length n is

$$W\left(\texttt{iterSort}\ a\right) \ \leq\ \sum\nolimits_{i=1}^{n-1} c \cdot (1+i) \ = O(n^2).$$

Using the trace, 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 `mergeOne`, whose span is is logarithmic. We can calculate the total span as

$$S\left(\texttt{iterSort}\ a\right) \ \leq\ \sum\nolimits_{i=1}^{n-1} c \cdot \lg\left(1+i\right) \ = O(n \lg n).$$

Since average parallelism, $W\left(n\right)/S\left(n\right) = O(n/\lg n)$, we see that the algorithm has a reasonable amount of parallelism. Unfortunately, it does much too much work.

Using this reduction order the algorithm is effectively working from the front to the rear, using `mergeOne` to "insert" each element into a sorted prefix where it is placed at the correct location to maintain the sorted order. The algorithm thus implements the well-known insertion sort.

Cost of `reduce`. 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 of `reduce`, however, depends on the order in which the operations are performed, as shown by Example 6.27.

Example 6.27. Consider appending the following code

`reduce append " " ⟨"abc","d","e","f"⟩.`

Suppose performing append operations in left-to-right order and count their work using the array-sequence specification. The total work is 19, because the following `append` operations are performed

1. `"abc"` ++ `"d"` (work 5),
2. `"abcd"` ++ `"e"` (work 6), and
3. `"abcde"` ++ `"f"` (work 7).

Consider now performing the `append` operations from right to left order. We obtain a total cost of 15, because the following `append` operations are performed

1. `"e"` ++ `"f"` (work 3),
2. `"d"` ++ `"ef"`, (work 4) and
3. `"abc"` ++ `"def"` (work 7).

To specify the cost of reduce, we therefore consider its trace based on its specification as gives in Section 6.2, reproduced below for convenience.

$$\texttt{reduce}\ f\ id\ a = \begin{cases} id & \text{if } |a| = 0 \\ a[0] & \text{if } |a| = 1 \\ f\Big(\texttt{reduce}\ f\ id\ (a[0,\ldots,\lfloor\frac{|a|}{2}\rfloor - 1]), \\ \quad \texttt{reduce}\ f\ id\ (a[\lfloor\frac{|a|}{2}\rfloor,\ldots,|a|-1]\Big) & \text{otherwise.} \end{cases}$$

Cost Specification 6.28. [Cost for `reduce`] Consider evaluation of `reduce` $f\ x\ a$ and let $\mathcal{T}(\texttt{reduce}\ f\ x\ a)$ denote the set of calls to $f(\cdot,\cdot)$ performed along with the arguments. The work and span are defined as

$$\begin{aligned} W\left(\texttt{reduce}\ f\ x\ a\right) &= O\left(1 + \textstyle\sum_{f(y,z)\in\mathcal{T}(\texttt{reduce}\ f\ x\ a)} W\left(f(y,z)\right)\right), \text{ and} \\ S\left(\texttt{reduce}\ f\ x\ a\right) &= O\left(\lg|a| \cdot \max_{f(y,z)\in\mathcal{T}(\texttt{reduce}\ f\ x\ a)} S\left(f(y,z)\right)\right). \end{aligned}$$

The work bound is simply the total work performed, which we obtain by summing across all combine functions, plus one for the `reduce`. The span bound is more interesting. The $\lg|a|$ term expresses the fact that the recursion tree in the specification of `reduce` (ADT 6.5) is at most $O(\lg|a|)$ deep. Since each node in the recursion tree has span at most $\max_{f(y,z)} S\,(f(y,z))$, any root-to-leaf path, has at most $O(\lg|a| \cdot \max_{f(a,b)} S\,(f(a,b)))$ span.

Cost of `scan`. As in `iterate` and `reduce` the cost specification of `scan` depends on the intermediate results. But the dependency is more complex than 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 6.29. [Cost for `scan`] Consider evaluation of `scan` $f\ x\ a$). For both the array-sequence and tree-sequence specification

$$\begin{array}{lcl} W\,(\texttt{scan}\ f\ x\ a) & = & O(|a|) \\ S\,(\texttt{scan}\ f\ x\ a)) & = & O(\lg|a|). \end{array}$$

6.4.2 Tree Sequences

The costs for tree sequences is given in Cost Specification 6.30. The specification represents the cost for a class of implementations that use a balanced tree to represent the sequence. The cost of each operation is similar to the array-based specification, and many are exactly the same, i.e., `length`, `singleton`, `isSingleton`, `isEmpty`, `collect`, `iterate`, `reduce`, and `scan`.

There are also differences. The work and span of the operation `nth` is logarithmic, as opposed to being constant. This is because in balanced-tree based implementation, the operation must follow a path from the root to a leaf to find the desired element element. For a sequence a, such a path has length $O(\lg|a|)$. Although `nth` does more work with tree sequences, `append` does not. Instead of requiring linear work, the work of `append` with tree sequences is proportional to the logarithm 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 $O(\lg n)$. The work of `update` is also less with tree sequences than within array sequences.

The work for operations `map` and `tabulate` are the same as those for array sequences; their span incurs an extra logarithmic overhead. The work and span of `filter` are the same for both.

6.4.3 List Sequences

The costs for tree sequences is given in Cost Specification 6.31. The specification represents the cost for a class of implementations that use (linked) lists to represent the sequence. The

Cost Specification 6.30. [Tree Sequences]
We specify the ***tree-sequence*** costs as follows. The notation $\mathcal{T}(-)$ refer to the trace of the corresponding operation. The specification for `scan` assumes that f has constant work and span.

Operation	Work	Span
`length` a	1	1
`singleton` x	1	1
`isSingleton` x	1	1
`isEmpty` x	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, i, j)	$1 + \log(\lvert a\rvert)$	$1 + \log(\lvert a\rvert)$
`append` $a\ b$	$1 + \lvert\log(\lvert a\rvert/\lvert b\rvert)\rvert$	$1 + \lvert\log(\lvert a\rvert/\lvert b\rvert)\rvert$
`flatten` a	$1 + \lvert a\rvert \log\left(\sum_{x \in a} \lvert x\rvert\right)$	$1 + \log(\lvert a\rvert + \sum_{x \in a} \lvert x\rvert)$
`inject` $a\ b$	$1 + (\lvert a\rvert + \lvert b\rvert)\log\lvert a\rvert$	$1 + \log(\lvert a\rvert + \lvert b\rvert)$
`collect` $f\ a$	$1 + W(f) \cdot \lvert a\rvert \log\lvert a\rvert$	$1 + S(f) \cdot \log^2\lvert a\rvert$
`iterate` $f\ x\ a$	$1 + \sum_{f(y,z) \in \mathcal{T}(-)} W(f(y,z))$	$1 + \sum_{f(y,z) \in \mathcal{T}(-)} S(f(y,z))$
`reduce` $f\ x\ a$	$1 + \sum_{f(y,z) \in \mathcal{T}(-)} W(f(y,z))$	$\log\lvert a\rvert \cdot \max_{f(y,z) \in \mathcal{T}(-)} S(f(y,z))$
`scan` $f\ a$	$\lvert a\rvert$	$\log\lvert a\rvert$

determining cost in list-based implementations is the sequential nature of the representation: accessing the element at position i requires traversing the list from the head to i, which leads to $O(i)$ work and span. List-based implementations therefore expose hardly any parallelism. Their main advantage is that they require quick access to the ***head*** and the ***tail*** of the sequence, which are defined as the first element and the suffix of the sequence that starts at the second element respectively.

The work of each operation is similar to the array-based specification. Since the data structure mostly serial, the span of each operation is essentially the same as that of its work, except that the total is taken over the spans of its components. The work and span of `subseq` operation depends on the beginning position of the subsequence, because list-based representation can share their suffixes.

Remark 6.32. Since they are serial, list-based sequences are usually ineffective for parallel algorithm design.

6.5 An Example: Primes

We now give some more involved examples of how to use sequences and analyze work and span. As usual, the ratio of work and span gives us the parallelism of the algorithm. As our example, we consider the problem of finding prime numbers, more precisely defined as follows.

Problem 6.33. [Primes] The ***primes*** problem requires finding all prime numbers less than a given natural number n.

Recall that a natural number n is a prime if it has exactly two distinct divisors 1 and itself. For example, the number 1 is not prime, but 2, 3, 7, and 9967 are. If n is not prime, then it has a divisor that is at most $\sqrt{n}$ since for any $i \times j = n$, either i or j has to be less than or equal to $\sqrt{n}$. We therefore can check if n is a prime by checking whether any i, $2 \leq i \leq \sqrt{n}$ is a divisor of n. We can write such an algorithm using sequences as follows. For simplicity we assume throughout that $n \geq 2$.

Algorithm 6.34.

```
isPrime $n$ =
  **let**
    $all = \langle\, n \bmod i : 1 \leq i \leq \lfloor \sqrt{n} \rfloor \,\rangle$
    $divisors = \langle\, x : x \in all \mid x = 0 \,\rangle$
  **in**
    $|divisors| = 1$
  **end**
```

Cost Specification 6.31. [List Sequences] We specify the ***array-sequence*** costs as follows. The notation $\mathcal{T}(-)$ refer to the trace of the corresponding operation. The specification for `scan` assumes that f has constant work and span.

Operation	Work	Span
`length` a `singleton` x `isSingleton` x `isEmpty` x	1	1
`nth` $a\ i$	i	i
`tabulate` $f\ n$	$1 + \sum_{i=0}^{n} W(f(i))$	$1 + \sum_{i=0}^{n} S(f(i))$
`map` $f\ a$	$1 + \sum_{x \in a} W(f(x))$	$1 + \sum_{x \in a} S(f(x))$
`filter` $f\ a$	$1 + \sum_{x \in a} W(p(x))$	$1 + \sum_{x \in a} S(p(x))$
`subseq` $a\ (i,j)$	$1 + i$	$1 + i$
`append` $a\ b$	$1 + \|a\|$	$1 + \|a\|$
`flatten` a	$1 + \|a\| + \sum_{x \in a} \|x\|$	$1 + \|a\| + \sum_{x \in a} \|x\|$
`update` $a\ (i,x)$	$1 + \|a\|$	$1 + \|a\|$
`inject` $a\ b$	$1 + \|a\| + \|b\|$	$1 + \|a\| + \|b\|$
`collect` $f\ a$	$1 + W(f) \cdot \|a\| \lg \|a\|$	$1 + S(f) \cdot \|a\| \lg \|a\|$
`iterate` $f\ x\ a$	$1 + \sum_{f(y,z) \in \mathcal{T}(-)} W(f(y,z))$	$1 + \sum_{f(y,z) \in \mathcal{T}(-)} S(f(y,z))$
`reduce` $f\ x\ a$	$1 + \sum_{f(y,z) \in \mathcal{T}(-)} W(f(y,z))$	$1 + \sum_{f(y,z) \in \mathcal{T}(-)} S(f(y,z))$
`scan` $f\ a$	$\|a\|$	$\|a\|$

Let's calculate the work and span of this algorithm based on the array sequence cost specification. The algorithm constructs a sequence of length $\lfloor\sqrt{n}\rfloor$ and then filters it. Since the work for computing $i \bmod n$ and checking that a value is zero $x = 0$ is constant, based on the array-sequence costs, we can write work as

$$W_{\texttt{isPrime}}\left(n\right) = O\left(1 + \sum_{i=1}^{\lfloor\sqrt{n}\rfloor} O\left(1\right)\right) = O\left(\sqrt{n}\right).$$

Similarly we have for span:

$$S_{\texttt{isPrime}}\left(n\right) = O\left(\lg\sqrt{n} + \max_{i=1}^{\lfloor\sqrt{n}\rfloor} O\left(1\right)\right) = O\left(\lg n\right).$$

The $\lg\sqrt{n}$ additive terms comes from the cost specification for `filter`.

Since parallelism is the ratio of work to span, the algorithm `isPrime` has parallelism of

$$O\left(\frac{\sqrt{n}}{\lg\sqrt{n}}\right).$$

This is not an abundant amount of parallelism but still plenty for all practical purposes, because work is quite small.

Now that we can test for primality of an number, we can solve the primes problem by testing the numbers up to n. We can write the code for such a brute-force algorithm as follows.

Algorithm 6.35.

```
primesBF $n$ =
  **let**
    $all$ = $\langle\, i : 1 < i < n \,\rangle$
    $primes$ = $\langle\, x : x \in all \mid$ isPrime $x \,\rangle$
  **in**
    $primes$
  **end**
```

Let's analyze work and span, again us array sequences. Constructing the sequence *all* using `tabulate` requires linear work. The work of filtering through it is the sum of the work of the calls to `isPrime`; thus we have

$$\begin{aligned} W_{\texttt{primesBF}}\left(n\right) &= O\left(\sum_{i=2}^{n-1} 1 + W_{\texttt{isPrime}}\left(i\right)\right) \\ &= O\left(\sum_{i=2}^{n-1} 1 + \sqrt{i}\right) \\ &= O\left(n^{3/2}\right). \end{aligned}$$

Similarly, the span is dominated by the maximum of the span of calls to `isPrime` and a logarithmic additive term.

$$\begin{aligned} S_{\texttt{primesBF}}(n) = \; &= O\left(\lg n + \max_{i=2}^{n} S_{\texttt{isPrime}}(i)\right) \\ &= O\left(\lg n + \max_{i=2}^{n} \lg i\right) \\ &= O(\lg n) \end{aligned}$$

The parallelism is hence

$$\frac{W_{\texttt{primesBF}}(n)}{S_{\texttt{primesBF}}(n)} = n^{3/2}/\lg n.$$

This is plenty of parallelism but comes at the expense of a large amount of work. We can improve the work for the algorithm significantly. Intuitively, we can see that the algorithm does a lot of redundant work, because it repeatedly performs primality checks for the same numbers. To check whether a number m is prime, the algorithm checks its divisors, it then checks essentially the same divisors for multiples of m, such as $2m, 3m, \ldots$, which largely overlap, because if a number divides m, it also divides its multiples.

We can eliminate this redundancy by more actively eliminating numbers that are composites. The basic idea is to create a collection of composite numbers up to n and use this as a "sieve." Generating such a sieve is easy: we just have to include for any number $i \leq \sqrt{n}$, its multiples of up to $\frac{n}{i}$. Having generated the sieve, what remains is to run the numbers up to n through the sieve. To do this in parallel, we can use `inject`. The idea is to do construct the sieve as a length-n sequence of the Boolean value `true`, and then update the sequence by writing `false` into all positions that correspond to composite numbers. The remaining `true` values indicate the prime numbers. The code for this algorithm is shown below.

Algorithm 6.36.

```
primeSieve $n$ =
  **let**
    $cs = \langle i * j : 0 \leq i \leq \lfloor \sqrt{n} \rfloor, 2 \leq j \leq n/i \rangle$
    $sieve = \langle (x, \texttt{false}) : x \in cs \rangle$
    $all = \langle \texttt{true} : i \leq 0 < n \rangle$
    $isPrime =$ inject $all$ $sieve$
    $primes = \langle i : 2 \leq i < n \mid isPrime[i] = true \rangle$
  **in**
    $primes$
  **end**
```

The work and span for calculating `primeSieve` is similar to the analysis for finding all subsequences in Example 6.24. We shall consider the phases of the algorithm and show that the work and span are functions of the total number of composites which we denote by m.

- Generating each composite takes constant work and span since it just a multiply. The work for generating the sequence of composites is linear in the total number of such composites m. The span is $O(\lg m)$ because of the nested sequence and the implied `tabulate`. Constructing the sieve requires linear work in its length, which is m, and constant span.

- The work of `inject` is also proportional to the length of `sieve`, m, and its span is constant.

- The work for computing `primes`, using `tabulate` and `filter` is proportional to n, and the span is $O(\lg n)$.

Therefore the total work is proportional to the number composites m, which is larger than n, and the total span is $O(\lg n + m)$. To calculate m, we can add up the number of multiples each i from 2 to $\lceil \sqrt{n} \rceil$ have, i.e.,

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

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

$$W_{\texttt{primeSieve}}(n) = O(n \lg n), \text{ and } S_{\texttt{primeSieve}}(n) = O(\lg n).$$

This is a significant improvement in the work.

The work can actually be further improved by noticing that when computing the composites, we only need to consider the multiples of prime numbers. For example we don't need to consider the multiples of 6 since all multiples of 6 are also multiples of 2 and of 3. Based on this observation, it is possible to improve the work to $O(n \lg \lg n)$ without changing the span. We leave this as an exercise.

Remark 6.37. The algorithm for computing primes described here dates back to antiquity and attributed to Eratosthenes of Cyrene, a Greek mathematician.

6.6 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 $f : \alpha \to \alpha$, a `map` function can be implemented as follows:

```
fun map $f$ $S$ =
  iterate $(\lambda\,((i, S'), v).\,(i+1,$ 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` $S : \alpha$ seq $\rightarrow \alpha$ stseq Converts from a regular sequence to a stseq.	$O(\|S\|)$	$O(1)$
`toSeq` $ST : \alpha$ stseq $\rightarrow \alpha$ seq Converts from a stseq to a regular sequence.	$O(\|S\|)$	$O(1)$
`nth` $ST\ i : \alpha$ stseq $\rightarrow$ int $\rightarrow \alpha$ Returns the i^{th} element of ST. Same as for seq.	$O(1)$	$O(1)$
`update` $ST\ (i, v) : \alpha$ stseq $\rightarrow$ (int $\times\ \alpha$) $\rightarrow \alpha$ stseq Replaces the i^{th} element of ST with v.	$O(1)$	$O(1)$
`inject` $ST\ I : \alpha$ stseq $\rightarrow$ (int ** α) seq $\rightarrow \alpha$ 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 6.38.

```
map $f$ $S$ =
  **let**
    $S'$ = StSeq.fromSeq $S$
    $R$ =  iterate ($\boldsymbol{\lambda}((i, S''), v).(i+1,$ StSeq.update $S''$ $(i, f(v)))$)
                   $(0, S')$
                   $S$
  **in**
    StSeq.toSeq $R$
  **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. Let's 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 by 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.

6.7 Problems

6-1 Subsequences
Recall that for a sequence a, $a[i, \ldots, j]$ is the subsequence starting at position i and ending at position j. Can you compute $a[i, \ldots, j]$ using tabulate?

6-2 Matching parentheses correctly
Prove that Algorithm 6.12, which uses `iterate`, solves the parentheses matching problem.

6-3 Equivalence of `reduce` and `iterate`
Prove that `reduce` and `iterate` are equivalent when the function being applied is associative.

6-4 Parentheses matching with scan
Give an algorithm for the parentheses matching problem using `scan` instead of `iterate`.

6-5 Map and tabulate
Can we implement `map` by using `tabulate`? What is the cost of your implementation using array and tree sequences?

6-6 Cost of `collect`
Describe how to implement `collect` in a way that is consistent with the specified costs.

6-7 Map and tabulate
We have seen in this chapter that `map` can be implemented using `tabulate`. Is this implementation consistent with the array-sequence and the tree-sequence cost specifications?

Exercise 6.39. Recall the `mergeOne` function that we considered in Section 6.4.

- Prove that `mergeOne` is associative.
- Show that we can use `mergeOne` and `reduce` to implement a sorting algorithm.
- Analyze the work and span of the resulting algorithm.
- What algorithm does this algorithm resemble?

6-8 Primes Revisited
Describe an algorithm for computing prime numbers up to n in $O(\lg \lg n)$ work and $O(\lg n)$ span.