Chapter 4

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.

4.1 Defining and Writing Sequences

We can define sequences mathematically as shown in Definition 4.1 . 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 4.1. *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 4.2. *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* $\{0, \ldots, 3\}$.

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 4.3 (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:*

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

Example 4.4. *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:* $\langle \texttt{'s}, \texttt{'e}, \texttt{'q} \rangle \equiv \texttt{``seq''}$.
- *An (integer $\times$ string) sequence:* $\langle (10, \texttt{``ten''}), (1, \texttt{``one''}), (2, \texttt{``two''})$
- *A (string sequence) sequence:* $\langle \langle \texttt{``a''} \rangle, \langle \texttt{``nested''}, \texttt{``sequence''}$
- *A function sequence, or more specifically a ($\mathbb{Z} \to \mathbb{Z}$) sequence:*

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

4.2 Computing with Sequences

Having defined sequences, we are now ready to start looking at operations on sequences that enable us to perform interesting computations

with them. ADT 4.5 shows the specification of an abstract data type for sequences using the mathematical definition of sequences shown in Definition 4.1. The ADT can be broadly divided into constructor functions that create sequences such as `empty`, `singleton`, and `tabulate`, and operations, such as `nth`, `map`. In addition to the operations specified here, we will consider `iter`, `iterh`, `reduce`, and `scan` later in this chapter.

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 sequences and the operations on them. Syntax 4.7 illustrates this syntax. In the rest of this section, we briefly explain the operations in the sequence ADT and their syntax.

Abstract Data Type 4.5 (Sequences). *For a value type α, the* **sequence data type** *is the type $\alpha\,\mathbb{S}$ consisting the set of all α sequences, and the following values and functions on $\alpha\,\mathbb{S}$:*

$$
\begin{array}{lcll}
\texttt{empty} & : & \alpha\,\mathbb{S} & = \{\} \\
\texttt{singleton}(v) & : & \alpha \to \alpha\,\mathbb{S} & = \{(0, v)\} \\
\texttt{tabulate}(f, n) & : & (\mathbb{N} \to \alpha) \times \mathbb{N} \to \alpha\,\mathbb{S} & = \{(i, f(i)) : 0 \le i < n\} \\
\texttt{length}(A) & : & \alpha\,\mathbb{S} \to \mathbb{N} & = |A| \\
\texttt{nth}(A, i) & : & \alpha\,\mathbb{S} \times \mathbb{N} \to (\alpha \cup \{\bot\}) & = \begin{cases} v & (i, v) \in A \\ \bot & \textit{otherwise} \end{cases} \\
\texttt{map}(f, A) & : & (\alpha \to \beta) \times \alpha\,\mathbb{S} \to \beta\,\mathbb{S} & = \{(i, f(v)) : (i, v) \in A\} \\
\texttt{subseq}(A, s, l) & : & \alpha\,\mathbb{S} \times \mathbb{N} \times \mathbb{N} \to \alpha\,\mathbb{S} & = \{(i - s, v) \\
 & & & \quad : (i, v) \in A \mid s \le i < s + l\} \\
\texttt{append}(A, B) & : & \alpha\,\mathbb{S} \times \alpha\,\mathbb{S} \to \alpha\,\mathbb{S} & = A \cup \{(i + |A|, v) : (i, v) \in B\} \\
\texttt{filter}(f, A) & : & (\alpha \to \mathbb{B}) \times \alpha\,\mathbb{S} \to \alpha\,\mathbb{S} & = \{(|\,\{(j, x) \in A \mid j < i \wedge f(x)\}\,|, v) \\
 & & & \quad : (i, v) \in A \mid f(v)\} \\
\texttt{flatten}(A) & : & \alpha\,\mathbb{S}\,\mathbb{S} \to \alpha\,\mathbb{S} & = \{(i + \sum_{(k,X) \in A, k < j} |X|, v) \\
 & & & \quad : (i, v) \in Y, (j, Y) \in A\} \\
\texttt{update}(A, (j, v)) & : & \alpha\,\mathbb{S} \times (\mathbb{N} \times \alpha) \to \alpha\,\mathbb{S} & = \{(i, x) : (i, v) \in A\}, \\
 & & & \quad x = \begin{cases} y & i = j \\ v & \textit{otherwise} \end{cases} \\
\texttt{inject}(A, P) & : & \alpha\,\mathbb{S} \times \mathbb{N} \times \alpha\,\mathbb{S} \to \alpha\,\mathbb{S} & = \{(i, x) : (i, v) \in A\}, \\
 & & & \quad x = \begin{cases} y & (j, i, y) \in P \\ v & \textit{otherwise} \end{cases}
\end{array}
$$

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

Remark 4.6 (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*

Syntax 4.7 (Syntax for Sequences)**.** *The table below defines the syntax for sequenced 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)
$\langle\, e : 0 \leq i < e_n\,\rangle$	$\equiv$	`tabulate` $(\textbf{fn}\ i \Rightarrow e\ e_n)$
$\lvert S\rvert$	$\equiv$	`length` (S)
$S[i]$	$\equiv$	`nth` (S)
$\langle\, e : p \in e_s\,\rangle$	$\equiv$	`map` $(\textbf{fn}\ p \Rightarrow e)\ e_s$
$\langle\, x \in e_s \mid e\,\rangle$	$\equiv$	`filter` $(\textbf{fn}\ x \Rightarrow e)\ e_s$
$A[e_l, \cdots, e'_n]$	$\equiv$	`subseq` $(A, e_l, e'_n - e_n + 1)$

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 4.8 (Tabulate)**.**

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

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:*

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

Example 4.9. *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 4.7.

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 *`map`* f S function that applies the function f to each element of S 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 4.10 (Map)**.**

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

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

$$\left\langle\, x^2 : x \in S \,\right\rangle$$

is equivalent to:

$$\mathtt{map}\ (\textbf{fn}\ x \Rightarrow x^2)\ S$$

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 ap-

plied at each location, instead of the sum. We will also see that *map* generalizes to arbitrary mappings, not just sequences.

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

```
map f S = tabulate (fn i ⇒ f(nth(S,i))) (length S)
```

or equivalently in our sequence notation as

```
map f S = ⟨ f(S[i]) : 0 ≤ i < |S| ⟩
```

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

It is often useful to subselect the elements from a sequences that satisfy some predicate. For example, in quick sort (Chapter 7) we will want all the elements that are less (or greater) than a pivot element. For this we can use the *filter* f S function that applies a Boolean function f to each element of S, and returns the sequence consisting exactly of those elements of $s \in S$ for which $f(s)$ returns true. These elements maintain their order.

Syntax 4.12 (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:

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

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 4.13. *Given the integer sequence* $S = \langle 9, 7, 13, 4, 11, 21 \rangle$, *and a function* `isPrime`$(v)$ *which checks if* v *is prime, the expression:*

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

is equivalent to:

$$\texttt{filter isPrime } S$$

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 `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 4.14 (Subsequence)**.**

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

It is also useful to put sequences together. The $\mathtt{append}(S_1, S_2)$ function appends the sequence S_2 after the sequence S_1. To append more than two sequences the $\mathtt{flatten}(S)$ function takes a sequence of sequences and flattens them—i.e. if the input is a sequence $S = \langle\, S_1, S_2, \ldots, S_n \,\rangle$ it appends all the S_i together one after the other.

Example 4.15. *We have:*

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

and

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

Updates. It is often convenient to update elements of a sequence. The function $\mathtt{update}(S, (i, v))$, updates location i of sequence S 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 $\texttt{inject}(S, P)$ 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 4.16. *Given the string sequence*

$$S = \langle \texttt{``the''}, \texttt{``cat''}, \texttt{``in''}, \texttt{``the''},$$

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

gives

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

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

$$\mathit{inject}(S, \langle (4, \texttt{``log''}), (1, \texttt{``dog''}), (6, \texttt{``}$$

gives

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

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* S.

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 S with all elements of the another sequence T, i.e., when computing the Cartesian product.

We use the following syntax to do this

$$\langle\, (x, y) : x \in S, y \in T \,\rangle\,.$$

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, the we assume that the resulting sequence is ordered by the natural generalization of the ordering of the sequences involved. For example with two sequences the element (s_1, t_1) comes before (s_2, t_2) if an only if in S, s_1 comes before s_2

or $s_1 = s_2$ and in T, t_1 comes before t_2 or $t_1 = t_2$.

Example 4.17. *Let* $S = \langle\, 0, 1 \,\rangle$ *and* $T = \langle\, a, b \,\rangle$.

$\langle\, (x, y) : x \in S, y \in T \,\rangle = \langle\, (0, \texttt{'a}), (0, \texttt{'b}), (1, \texttt{'a}),$

Question 4.18. *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 $ST' = \langle\, (0, T), (1, T) \,\rangle$, which can be computed by $ST' = \langle\, (x, T) : x \in S \,\rangle$. Now we can map over each element of the sequence ST' to obtain what we want, well more or less. Consider

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

It is not difficult to see that

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

Thus the only remaining issue is the nesting. Luckily, that is simple using our `flatten` function. Indeed it is easy to see that $ST = flattenST''$ is the Cartesian product that we sought after. Putting it all together, we can express the Cartesion product of S and T as follows:

$$\texttt{flatten}\ \langle\langle (x, y) : y \in z \rangle : (x, z) \in \langle (x, T) : x \in S$$

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

```
1 CartesianProduct =  fn (S,T) ⇒
2   flatten (map (fn (x,z) ⇒ map (fn y ⇒ (x,y)) z)
3                (map (fn x ⇒ (x,T)) S))
```

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 4.19. *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. of them:*

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

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

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

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.*

$$\langle\,(x,y) : x \in S, y \in T \mid \texttt{isEven}(x)\,\rangle$$

Example 4.21.

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

multiplies all pairs and evaluates to:

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

Example 4.22. *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.

The functions that we have described so far are summarized in Abstract Data type ADT 4.5. It relies on the definition of of sequences given in Definition 4.1 (i.e. mappings from inte-

gers to elements). We will add more functions later. The exact list of functions does not matter very much since many functions can be implemented with others. We already gave an example of how `map` can be implemented with `tabulate`. However, we have to be careful that the implementations are efficient. This sometimes depends on the cost model. For `map` implemented with `tabulate`, for example, the implementation is asymptotically efficient for array sequences but not for tree sequences.

4.3 Cost Specification

We now consider the cost specifications for the sequence ADT. We consider three cost specifications, which we refer to as the *array sequence*, *list 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. None of the three specifications we give dominates another.

The costs for ArraySequences is given in Cost Specification 4.3. 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`,

Cost Specification 4.23 (Array Sequences)**.**

	ArraySequence	
	Work	Span
$\texttt{length}(A)$	1	1
$\texttt{singleton}(v)$	1	1
$\texttt{nth}(A, i)$	1	1
$\texttt{map}\ f\ S$	$1 + \sum_{s \in S} W(f(s))$	$1 + \max_{s \in S} S(f(s))$
$\texttt{tabulate}\ f\ n$	$1 + \sum_{i=0}^{n} W(f(i))$	$1 + \max_{i=0}^{n} S(f(i))$
$\texttt{filter}\ p\ S$	$1 + \sum_{s \in S} W(p(s))$	$\log \lvert S \rvert + \max_{s \in S} S(p(s))$
$\texttt{subseq}(S, s, l)$	1	1
$\texttt{append}(S_1, S_2)$	$\lvert S_1 \rvert + \lvert S_2 \rvert$	1
$\texttt{flatten}(S)$	$\lvert\lvert S \rvert\rvert + \lvert S \rvert$	$\log \lvert S \rvert$
$\texttt{update}(P, S)$	$\lvert P \rvert + \lvert S \rvert$	1
$\texttt{inject}(S, P)$	$\lvert P \rvert + \lvert S \rvert$	1

The **array sequence** *cost specification.* $\lvert\lvert S \rvert\rvert = \sum_{X \in S} \lvert S \rvert$. *All entries are big-O.*

`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. We will justify this cost in the implementation section. 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 S since updating a single element should take 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 a later section we will define single-threaded array sequences that will allow us under certain re-

strictions to update a sequence in constant work.

Example 4.24. *As an example of* `map` *we have*

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

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

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

$$W(\langle\, x \in S \mid x < 27 \,\rangle) = O(1 + \sum_{i=0}^{|S|-1} O(1)) = O($$

$$S(\langle\, x \in S \mid x < 27 \,\rangle) = O(\log |S| + \max_{i=0}^{|S|-1} O(1)) =$$

Example 4.25. *As a more involved example we consider the code from Example 4.22:*

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

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*

Cost Specification 4.26 (Tree Sequences)**.**

	TreeSequence	
	Work	Span
`length`(S)	1	1
`singleton`(v)	1	1
`nth`(S, i)	$\log \lvert S \rvert$	$\log \lvert S \rvert$
`tabulate` f n	$1 + \sum_{i=0}^{n} W(f(i))$	$\log n + \max_{i=0}^{n} S(f(i))$
`map` f S	$1 + \sum_{s \in S} W(f(s))$	$\log \lvert S \rvert + \max_{s \in S} S(f(s))$
`filter` f S	$1 + \sum_{s \in S} W(f(s))$	$\log \lvert S \rvert + \max_{s \in S} S(f(s))$
`subseq`(S, s, l)	$\log(\lvert S \rvert)$	$\log(\lvert S \rvert)$
`append`(S_1, S_2)	$1 + \lvert \log(\lvert S_1 \rvert / \lvert S_2 \rvert) \rvert$	$1 + \lvert \log(\lvert S_1 \rvert / \lvert S_2 \rvert) \rvert$
`flatten`(S)	$\log(\lvert\lvert S \rvert\rvert) \lvert S \rvert$	$\log(\lvert\lvert S \rvert\rvert + \lvert S \rvert)$
`inject`(P, S)	$(\lvert P \rvert + \lvert S \rvert) \log \lvert S \rvert$	$\log(\lvert S \rvert + \lvert P \rvert)$

The **Tree Sequence** *cost specification. Again,* $\lvert\lvert S \rvert\rvert = \sum_{X \in S} \lvert S \rvert$ *and all entries are big-O.*

The costs for TreeSequences is given in Cost Specification 4.3. The first thing to notice is that the cost of the `nth` function is no longer constant. Instead it has logarithmic work and span. This is because TreeSequences 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 $O(\log |S|)$. Although `nth` does more work with TreeSequences, `append` does Instead of requiring linear work, the work

of `append` with TreeSequences 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 TreeSequences than within ArraySequences. More details on how these cost arise is given later.

Such a tradeoff between implementations, where some functions are cheaper with one cost specification and others with another, is common in data types. The user can decide to use an implementation that matches either specification, and this decisions should be based on which specification leads to better asymptotic performance for their algorithm. For example if making many calls to `nth` but no calls to `append`, then the user might want to use the ArraySequence specification, while if the

mostly use *append* and *update*, then TreeSequences might be better.

The work and span for other functions such *map*, *tabulate* and *filter* are approximately the same for ArraySequences and TreeSequences, except there is an extra logarithmic term in the span for *map* and *tabulate* in TreeSequences.

Exercise 4.27. *Earlier we showed how to implement* `map` *using* `tabulate`*. Decide whether this implementation preserves asymptotic costs for an ArraySequence, and then for TreeSequence.*

4.4 An Example: Binary Search

```
1  fun kthSmallest(A,B,k) =
2  % need base cases
3  let
4     val m_A = |A|/2
5     val m_B = |B|/2
6  in
7     case (k ≤ m_A + m_B, A[m_A] < B[m_B]) of
8        (false,false) ⇒ kthSmallest(A,B[m_a+1,|A|-1],k-m_a-1)
9     | (false,true) ⇒ kthSmallest(A[m_b+1,|B|-1],B,k-m_b-1)
10    | (true,false) ⇒ kthSmallest(A[0,m_a-1],B,k)
```

```
11    | (true,true) ⇒ kthSmallest(A,B[0,m_b-1],k)
12  end
```

4.5 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 4.28 (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 4.29.

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

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:

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

for work, since we just sum the cost of applying $\bmod$ to each j, and then add in 1, as given in Cost Specification **??** for *filter*.

Similarly we have for span:

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

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 4.30.

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

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

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

and the span is:

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

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 4.31. *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 4.32.

```
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 4.25. In particular generating each multiple takes constant work and span since it just a multiply. The the total work is proportional to 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 4.33.

```
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.

4.6 Iterate, Reduce and Scan

So far we have described functions that do something with each element of a sequence. Each calculation is independent. What if we want, for example to sum the elements of a sequence. We clearly cannot do this with a `map`, `tabulate`, or `filter`. Here we define three functions for working over the elements of a sequence: `iterate`, `reduce`, and `scan`. The first is sequential and the other two are parallel. We start with `iterate`.

Iterate.

it·er·ate (Merriam-Webster) :
to say or do again, or again and again

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 mean to start with an initial state and a sequence and on each step to 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 form

```
iterate f v S
```

where S is a sequence, v is an initial state, and f is a function mapping a state and an element

of S to a new state. As such `iterate` takes $|S|$ steps starting with the first element of S and ending with the last. For a sequence of length 5, for example, it is equivalent to :

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

If S is an α sequence and the states are of type β then f must have type $\beta \times \alpha \to \beta$ since it maps a state and an element to a new state.

Example 4.34.

`iterate '+' 0` $\langle 2,5,1,6 \rangle$

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

`iterate '-' 0` $\langle 2,5,1,6 \rangle$

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

The function can be implemented as:

Algorithm 4.35.

```
1 fun iterate f v S =
2 let
3     fun iter(v,i) =
4           if (i = |S|) then v
5           else iter(f(v, S[i]), i + 1)
6 in iter(v,0) end
```

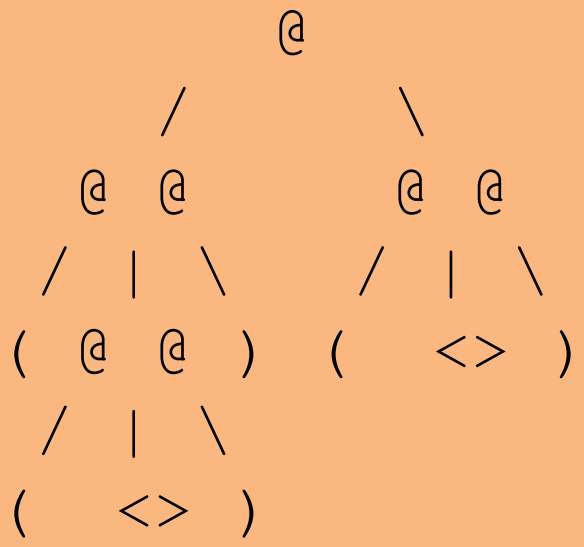
This algorithm uses recursion to go over the elements one by one.

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. We say a such 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 4.36. *The string* ``(())()'' *is matched since it can be decomposed as:*



where @ *indicates appending. The* ``())(()'', *however, is not matched.*

Problem 4.37 (Matched Parentheses)**.** *The* matched parenthesis *problem is given a string of parentheses to determined whether it is matched.*

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. Sequentially 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 a count and whenever we see a right parenthesis we decrement the count. 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. Proving this is left as an exercise, and the observation leads to the following algorithm:

Algorithm 4.38.

```
1  fun matchedParen(S) =
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) S) = Some(0)
10 end
```

The algorithm starts with the state (counter) $Some(0)$ 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.

Reduce.

re·duce (Merriam-Webster) :

1 a: to draw together or cause to converge : consolidate <reduce all the questions to one>

As we noted the *iterate* function is inherently sequential since it goes over the elements of a sequence iteratively one by one. However, for the example of summing a sequence of values, you might notice that it is possible to perform the sum in parallel. For example we might first pairwise add the element in odd positions of the sequence with their neighboring elements in even positions. This can then be repeated, as in the following diagram for *iterate '+' 0* $\langle 2,5,1,6 \rangle$

```
2 5 1 6
 7   7    step 1
   14     step 2
```

For a sequence with initial length n such a process will complete after $\lceil \log n \rceil$ steps.

So what is it about addition that allows us to do it in parallel. Well it is the fact that addition is associative. Recall that a binary function is associative if and only if $f(x, f(y, z)) = f(f(x, y), z)$ for all x, y, z (restricted to the type of the function, e.g. the integers). To take advantage of parallelism, the `reduce` function is defined in the same way as `iterate` but requires that the the function f is associative. It also requires that the initial state v is a left identity for f—a value v of type α is a left identity of f if $f(v, x) = x$ for all $x \in \alpha$. The associativity condition forces f to have type $(\alpha \times \alpha) \to \alpha$.

There are many functions that are associative. You probably already known that addition and multiplication are associative, with 0 and 1 as their (left) identities, respectively. Minimum and maximum are also associative with left identities ∞ and $-\infty$ respectively. The `append` function on sequences is also

associative, with left identity being the empty sequence. Set union and matrix multiply are associative, with the empty set and the set of all possible elements as the identity, respectively. There are many other functions that are associative.

Example 4.39.

reduce append ⟨``another'', ``way'', ``to'', ``flatten''⟩

would return ``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). You might asks why introduce another function. The reason is that *reduce* can be implemented to run in parallel while *iterate* cannot.

Scan.

scan (Merriam-Webster) :
to look at (something) carefully usually
in order to find someone or something

A function closely related to `reduce` is `scan`. It has the interface:

$$\texttt{scan}\ f\ I\ S : (\alpha \times \alpha \to \alpha) \to \alpha \to \alpha\ \texttt{seq} \to (\alpha\ \texttt{seq}$$

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 S, as well as the total sum of S. Hence the function is often called the *prefix sums* function (or problem). For a function f which is associative it can be defined as follows:

Algorithm 4.40.

```
1 fun scan f I S =
2   (⟨ reduce f I (S⟨0,...,l−1⟩ : 0 ≤ l < n)⟩
3     reduce f I S)
```

This uses our pseudocode notation and the $\langle\, \texttt{reduce}\ f\ I\ (\texttt{take}(S, i)) : i \in \langle\, 0, \ldots, n-1 \,\rangle \,\rangle$

indicates that for each i in the range from 0 to $n-1$ apply `reduce` to the first i elements of S. For example,

$$\begin{aligned}\texttt{scan}\ +\ 0\ \langle 2,1,3\rangle &= (\langle\ \texttt{reduce}\ +\ 0\ \langle\ \rangle,\ \texttt{redu}\\ &\qquad \texttt{reduce}\ +\ 0\ \langle 2,1,3\rangle)\\ &= (\langle 0,2,3\rangle, 6)\end{aligned}$$

Using a bunch of reduces, however, is not an efficient way to calculate the partial sums.

Exercise 4.41. *What is the work and span for the* `scan` *code shown above, assuming* f *takes constant work.*

We will soon see how to implement a scan with the following bounds:

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

assuming that the function f takes constant work. For now we will consider some useful applications of scans.

Note that the scan function takes the "sum" of the elements before the position i. Sometimes it is useful to include the value at position i. We therefore also will use a version of such an *inclusive scan*.

$$\texttt{scanI} \; + \; 0 \; \langle \, 2, 1, 3 \, \rangle = \langle \, 2, 3, 6 \, \rangle$$

This version does not return a second result since the total sum is already included in the last position.

4.6.1 The MCSS Problem Algorithm 5: Using Scan

Let's consider how we might use the scan function to solve the Maximum contiguous subsequence (MCSS) problem. Recall, this problem is given a sequence S to find:

$$\max_{0 \leq i \leq j \leq n} (\sum_{k=i}^{j-1} S_k) \; .$$

As a running example for this section consider the sequence

$$S = \langle\, 1, -2, 3, -1, 2, -3 \,\rangle \ .$$

What if we do an inclusive scan on our input S using addition? i.e.:

$$X = \texttt{scanI} \ + \ 0 \, S = \langle\, 1, -1, 2, 1, 3, 0 \,\rangle$$

Now for any j^{th} position consider all positions $i < j$. To calculate the sum from immediately after i to j all we have to do is return $X_j - X_i$. This difference represents the total sum of the subsequence from $i + 1$ to j since we are taking the sum up to j and then subtracting off the sum up to i. For example to calculate the sum between the -2 (location $i + 1 = 1$) and the 2 (location $i = 4$) we take $X_4 - X_0 = 3 - 1 = 2$, which is indeed the sum of the subsequence $\langle\, -2, 3, -1, 2 \,\rangle$.

Now consider how for each j we might calculate the maximum sum that starts at any $i \leq$

j and ends at j. Call it R_j. This can be calculated as follows:

$$\begin{aligned} R_j &= \max_{i=0}^{j} \sum_{k=i}^{j} S_k \\ &= \max_{i=0}^{j} (X_j - X_{i-1}) \\ &= X_j + \max_{i=0}^{j} (-X_{i-1}) \\ &= X_j + \max_{i=0}^{j-1} (-X_i) \\ &= X_j - \min_{i=0}^{j-1} X_i \end{aligned}$$

The last equality is because the maximum of a negative is the minimum of the positive. This indicates that all we need to know is X_j and the minimum previous $X_i, i < j$. This can be calculated with a scan using minimum as the binary combining function. Furthermore the result of this scan is the same for everyone, so we need to calculate it just once. The result of

the scan is:

$$(M, \_) = \texttt{scan}\ \min\ 0\ X = (\langle 0, 0, -1, -1, -1, -1 \rangle ,$$

and now we can calculate R:

$$R = \langle\, X_j - M_j : 0 \leq j < |S| \,\rangle = \langle 1, -1, 3, 2, 4, 1 \rangle \ .$$

You can verify that each of these represents the maximum contiguous subsequence sum ending at position j.

Finally, we want the maximum string ending at any position, which we can do with a reduce using $\max$. This gives 4 in our example.

Putting this all together we get the following very simple algorithm:

Algorithm 4.42 (Scan-based MCSS).

```
1 function MCSS(S) =
2 let
3    val X = scanI + 0 S
4    val (M,_) = can min 0 X
5    val Y = ⟨X_j − M_j : 0 ≤ j < |S|⟩
6 in
7    max(Y)
8 end
```

Given the costs for scan and the fact that addition and minimum take constant work, this algorithm has $O(n)$ work and $O(\log n)$ span.

4.6.2 Copy Scan

Previously, we used *scan* to compute partial sums to solve the maximum contiguous subsequence sum problem and to match parentheses. 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 α *option seq*. For example

$$\langle \mathit{None}, \mathit{Some}(7), \mathit{None}, \mathit{None}, \mathit{Some}(3), \mathit{None} \rangle$$

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

$$\langle \mathit{None}, \mathit{None}, \mathit{Some}(7), \mathit{Some}(7), \mathit{Some}(7), \mathit{Some}$$

Using a sequential loop or *iter* 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

$$\alpha\ option \times \alpha\ option \to \alpha\ option$$

How about

```
1 fun copy(a,b)
2   case b of
3     Some(_) ⇒ b
4   | None ⇒ a
```

What this function does is basically pass on its right argument if it is *Some* 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 $copy(x, copy(y, z)) = copy(copy(x, y), z)$ for all x, y and z. There are eight possibilities corresponding to each of x, y and z being either *Some* or *None*. For the cases that $z = Some(c)$ it is easy to verify that that either ordering returns z. For the cases that $z = None$ and $y = Some(b)$ one can verify that both orderings give y, for the cases that $y = z = None$ and $x = Some(a)$

they both return x, and for all being `None` either ordering returns `None`.

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

4.6.3 Contraction and Implementing Scan

Now let's consider how to implement `scan` efficiently and at the same time apply one of the algorithmic techniques from our toolbox of techniques: *contraction*. Throughout the following discussion we assume the work of the binary operator is $O(1)$. As described earlier a brute force method for calculating scans is to apply a reduce to all prefixes. This requires $O(n^2)$ work and is therefore not work-efficient since we can do it in $O(n)$ work sequentially.

Beyond the wonders of what it can do, a

surprising fact about `scan` is that it can be accomplished efficiently in parallel, although on the surface, the computation it carries out appears to be sequential in nature. At first glance, we might be inclined to believe that any efficient algorithms will have to keep a cumulative "sum," computing each output value by relying on the "sum" of the all values before it. It is this apparent dependency that makes *`scan`* so powerful. We often use *`scan`* when it seems we need a function that depends on the results of other elements in the sequence, for example, the copy scan above.

Suppose we are to run `plus_scan` (i.e. `scan (op +)`) on the sequence $\langle 2, 1, 3, 2, 2, 5, 4, 1 \rangle$. What we should get back is

$$(\langle 0, 2, 3, 6, 8, 10, 15, 19 \rangle, 20)$$

We will use this as a running example.

Divide and Conquer: We first consider a divide-and-conquer solution. We can do this by splitting the sequence in half, solving each half and then trying to put the results together. The question is how do we put the results together. In particular lets say the scans on the two halves return (S_l, t_l) and (S_r, t_r), which would be $(\langle\, 0, 2, 3, 6 \,\rangle$ and $(\langle\, 0, 2, 7, 11 \,\rangle\, , 12)$ in our example. Note that S_l already gives us the first half of the solution.

Question 4.43. *How do we get the second half?*

To get the second half, note that in calculating S_r in the second half we started with the identity instead of the sum of the first half, t_l. Therefore if we add the sum of the first half, t_l, to each element of S_r, we get the desired result. This leads to the following algorithm:

Algorithm 4.44 (Scan using divide and conquer).

```
1  function scan f I S =
2    case showt(S) of
3      EMPTY ⇒ (⟨ ⟩, I)
4    | ELT(v) ⇒ (⟨ I ⟩, v)
5    | NODE(L, R) ⇒ let
6          val ((S_l, t_l), (S_r, t_r)) = (scan f I L || scan f I R)
7          val X_r = ⟨ f(t_l, y) : y ∈ S_r ⟩
8        in
9          (append(S_l, X_r), t_l + t_r)
10       end
```

One caveat about this algorithm is that it only works if I is really the "identity" for f, i.e. $f(I, x) = x$, although it can be fixed to work in general.

We now consider the work and span for the algorithm. Note that the joining step requires a map to add t_l to each element of S_r, and then an append. Both these take $O(n)$ work and $O(1)$ span, where $n = |S|$. This leads to the following recurrences for the whole algorithm:

$$W(n) = 2W(n/2) + O(n) \in O(n \log n)$$

$$S(n) = S(n/2) + O(1) \in O(\log n)$$

Although this is much better than $O(n^2)$ work, we can do better.

Contraction: To compute `scan` in $O(n)$ work in parallel, we introduce a new inductive technique common in algorithms design: contraction. It is inductive in that such an algorithm involves solving a smaller instance of the same problem, much in the same spirit as a divide-and-conquer algorithm. But with contraction, there is only one subproblem. In particular, the contraction technique involves the following steps:

1. Contract the instance of the problem to a smaller instance (of the same sort).
2. Solve the smaller instance recursively.
3. Use the solution to help solve the original instance.

The contraction approach is a useful technique in algorithm design in general but for

various reasons it is more common in parallel algorithms than in sequential algorithms. This is usually because both the contraction and expansion steps can be done in parallel and the recursion only goes logarithmically deep because the problem size is shrunk by a constant fraction each time.

We'll demonstrate this technique first by applying it to a slightly simpler problem, `reduce`. To begin, we have to answer the following question: *How do we make the input instance smaller in a way that the solution on this smaller instance will benefit us in constructing the final solution?*

The idea is simple: We apply the combining function pairwise to adjacent elements of the input sequence and recursively run `reduce` on it. In this case, the third step is a "no-op"; it does nothing. For example on input sequence $\langle 2, 1, 3, 2, 2, 5, 4, 1 \rangle$ with addition, we would contract the sequence to $\langle 3, 5, 7, 5 \rangle$. Then we

would continue to contract recursively to get the final result. There is no expansion step.

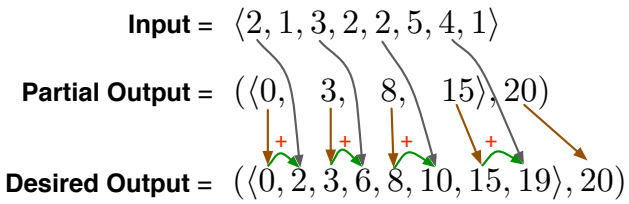
Thought Experiment II: How can we use the same idea to implement scan? What would be the result after the recursive call? In the example above it would be

$$(\langle\, 0, 3, 8, 15 \,\rangle\, , 20).$$

But notice, this sequence is every other element of the final scan sequence, together with the final sum—and this is enough information to produce the desired final output. This time, the third expansion step is needed to fill in the missing elements in the final scan sequence: Apply the combining function element-wise to the even elements of the input sequence and the results of the recursive call to `scan`.

To illustrate, the diagram below shows how to produce the final output sequence from the original sequence and the result of the recur-

sive call:



This leads to the following code. The algorithm we present works for when n is a power of two.

Algorithm 4.45 (Scan Using Contraction, for powers of 2).

```
1  function scanPow2 f i s =
2    case |s| of
3       0 ⇒ (⟨ ⟩, i)
4     | 1 ⇒ (⟨ i ⟩, s[0])
5     | n ⇒ let
6              val s' = ⟨ f(s[2i], s[2i + 1]) : 0 ≤ i < n/2 ⟩
7              val (r, t) = scanPow2 f i s'
8           in
9              (⟨ p_i : 0 ≤ i < n ⟩, t),   where p_i = { r[i/2]            even(i)
                                                      { f(r[i/2], s[i − 1])  otherwise.
10          end
```

4.7 Reduce Function

Recall that `reduce` function has the interface

$$\texttt{reduce}\ f\ I\ S : (\alpha \times \alpha \to \alpha) \to \alpha \to \alpha\ \texttt{seq} \to \alpha$$

When the combining function f is associative—i.e., $f(f(x,y),z) = f(x,f(y,z))$ for all x, y and z of type α—`reduce` returns the sum with respect to f of the input sequence S. It is the same result returned by `iter` f I S. The reason we include `reduce` is that it is parallel, whereas `iter` is strictly sequential. Note, though, `iter` can use a more general combining function with type: $\beta \times \alpha \rightarrow \beta$.

The results of `reduce` and `iter`, 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 combining functions that are non-associative, it is therefore important to specify the order that the combining function is applied to the elements of a sequence. This order is part of the specification of the ADT *Sequence*. 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.

For this reason, we define a specific combining tree. In particular we assume *reduce* is equivalent to the following code:

Algorithm 4.46 (Reduce definition)**.**

```
1  function reduce f I S =
2    let
3      function reduce'(S) =
4        case showt(S) of
5          ELT(v) ⇒ v
6        | NODE(L,R) ⇒ f(reduce(L),reduce(R))
7    in
8      case showt(S) of
9        NONE ⇒ I
10     | _ ⇒ f(I,reduce'(S))
11   end
```

Recall that `showt` splits S in half at the middle. If the length of S is odd, then the left "half" is one large than the right one.

4.7.1 Divide and Conquer with Reduce

Now, let's look back at divide-and-conquer algorithms you have encountered so far. Many of these algorithms have a "divide" step that simply splits the input sequence in half, proceed to solve the subproblems recursively, and continue with a "combine" step. This leads to the following structure where everything except what is in boxes is generic, and what is in boxes is specific to the particular algorithm.

```
1 fun myDandC(S) =
2   case showt(S) of
3     Empty ⇒ emptyVal
4   | Elt(v) ⇒ base(v)
5   | Node(L, R) ⇒ let
6         val (L′, R′) = (myDandC(L) || myDandC(R))
7     in
8       someMessyCombine(L′, R′)
9     end
```

Algorithms that fit this pattern can be implemented in one line using the sequence `reduce` function. Turning a divide-and-conquer algorithm into a reduce-based solution is as simple as invoking `reduce` with the following parameters:

```
reduce someMessyCombine emptyVal (map base S)
```

We will take a look two examples where `reduce` can be used to implement a relatively sophisticated divide-and-conquer algorithm. Both problems should be familiar to you.

Algorithm 4: MCSS Using Reduce.

The first example is the Maximum Contiguous Subsequence Sum problem from last lec-

ture. Given a sequence S of numbers, find the contiguous subsequence that has the largest sum—more formally:

$$\mathsf{mcss}(s) = \max\left\{\sum_{k=i}^{j} s_k \;:\; 1 \le i \le n, i \le j \le n\right\}.$$

Recall that the divide-and-conquer solution involved strengthening the problem so it returns four values from each recursive call on a sequence S: the desired result $\mathtt{mcss}(S)$, the maximum prefix sum of S, the maximum suffix sum of S, and the total sum of S. We will denote these as M, P, S, T, respectively. We refer to this strengthened problem as $\mathtt{mcss}'$. To solve $\mathtt{mcss}'$ we can then use the following implementations for `combine`, `base`, and `emptyVa`

fun `combine` $((M_L, P_L, S_L, T_L), (M_R, P_R, S_R, T_L)) =$
  $(\max(S_L + P_R, M_L, M_R),$
   $\max(P_L, T_L + P_R),$
   $\max(S_R, S_L + T_R),$
   $T_L + T_R)$
fun `base`$(v) = (v, v, v, v)$
fun `emptyVal` $= (-\infty, -\infty, -\infty, 0)$

and then solve the problem with:

```
fun mcss'(S) =
  reduce combine emptyVal (map base S)
```

Question 4.47. *Is the MCSS combine function described above associative?*

It turns out that the combine function for MCSS is associative, as we would expect for the binary function passed to `reduce`. Indeed this is true for the all the combine functions we have used in divide-and-conquer so far. To prove associativity of the MCSS combine function we could go through all the cases. However a more intuitive way to see it is to consider what $\texttt{combine}(A, \texttt{combine}(B, C))$ and $\texttt{combine}(\texttt{combine}(A, B), C)$ should return. In particular for A, B and C appearing in that order, both ways of associating the combines should return the overall maximum contiguous sum, the overall maximum prefix sum, the overall maximum suffix sum, and the overall sum. The divide-and-conquer algorithm would not be correct if this were not the case.

Stylistic Notes. We have just seen that we could spell out the divide-and-conquer steps in detail or condense our code into just a few lines that take advantage of the almighty `reduce`. *So which is preferable*, using the divide-and-conquer code or using reduce? We believe this is a matter of taste. Clearly, your reduce code will be (a bit) shorter, and for simple cases easy to write. But when the code is more complicated, the divide-and-conquer code is easier to read, and it exposes more clearly the inductive structure of the code and so is easier to prove correct.

Restriction. You should realize, however, that this pattern does not work in general for divide-and-conquer algorithms. In particular, it does not work for algorithms that do more than a simple split that partitions their input in two parts in the middle. For example, it cannot be used for implementing quick sort as the divide

step partitions the data with respect to a pivot. This step requires picking a pivot, and then filtering the data into elements less than, equal, and greater than the pivot. It also does not work for divide-and-conquer algorithms that split more than two ways, or make more than two recursive calls.

4.8 Analyzing the Costs of Higher Order Functions

In Section 1 we looked at using `reduce` to solve divide-and-conquer problems. In the MCSS problem the combining function f had $O(1)$ cost (i.e., both its work and span are constant). In that case the cost specifications of `reduce` on a sequence of length n is simply $O(n)$ (linear) work and $O(\log n)$ (logarithmic) span.

Question 4.48. *Does `reduce` have linear work and logarithmic span when the binary function passed to it does not have constant cost?*

Unfortunately when the function passed to reduce does not take constant work, then the work of the reduce is not necessarily linear. More generally when using a higher-order function that is passed a function f (or possibly multiple functions) one needs to consider the cost of f.

For `map` it is easy to find its costs based on the cost of the function applied:

$$W(\texttt{map } f\ S) = 1 + \sum_{s \in S} W(f(s))$$
$$S(\texttt{map } f\ S) = 1 + \max_{s \in S} S(f(s))$$

Tabulate is similar. But can we do the same for *reduce*?

Merge Sort. As an example, let's consider merge sort. As you have likely seen from previous courses you have taken, merge sort is a popular divide-and-conquer sorting algorithm with optimal work. It is based on a function `merge`

that takes two already sorted sequences and returns a sorted sequence containing all elements from both sequences. We can use our reduction technique for implementing divide-and-conquer algorithms to implement merge sort with a `reduce`. In particular, we can write a version of merge sort, which we refer to as reduceSort, as follows:

```
val combine = merge_<
val base = singleton
val emptyVal = empty
fun reduceSort(S) = reduce combine emptyVal (map base S)
```

where $\texttt{merge}_<$ is a merge function that uses an (abstract) comparison operator $<$. Note that merging is an associative function.

Assuming a constant work comparison function, two sequences S_1 and S_2 with lengths n_1 and n_2 can be merged with the following costs:

$$\begin{aligned} W(\texttt{merge}_<(S_1, S_2)) &= O(n_1 + n_2) \\ S(\texttt{merge}_<(S_1, S_2)) &= O(\log(n_1 + n_2)) \end{aligned}$$

Question 4.49. *What do you think the cost of* `reduceSort` *is?*

4.8.1 Reduce: Cost Specifications

We want to analyze the cost of `reduceSort`. Does the reduction order matter? As mentioned before, if the combining function is associative, which it is in this case, all reduction orders give the same answer so it seems like it should not matter.

To answer this question, let's consider the sequential reduction order that is used by *iter*, as given by the following code.

```
fun iterSort(S) =
  iterate merge< (empty) (map singleton S)
```

Since the merge is associative this is functionally the same as *reduceSort*, but will sequentially add the elements in one after the other. 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 $\texttt{merge}_<$ 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 4.50. *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.

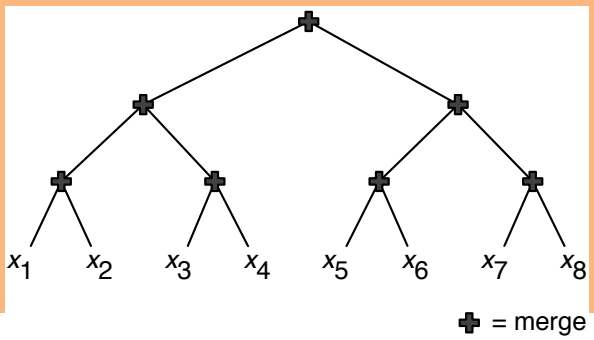
Question 4.51. *Can you think of a way to improve our bound by using a different reduction order with the merge function?*

In `iterSort`, the reduction tree is unbalanced. We can improve the cost by using a

balanced tree instead.

For ease of exposition, let's suppose that the length of our sequence is a power of 2, i.e., $|x| = 2^k$. Now we lay on top the input sequence a perfect binary tree[1] with 2^k leaves and merge according to the tree structure.

Example 4.52. *As an example, the merge sequence for $|x| = 2^3$ is shown below.*



What would the cost be if we use a perfect tree?

At the bottom level where the leaves are, there are $n = |x|$ nodes with constant cost each.

Stepping up one level, there are $n/2$ nodes,

[1] This is simply a binary tree in which every node either has exactly 2 children or is a leaf, and all leaves are at the same depth.

each corresponding to a `merge` call, each costing $c(1+1)$. In general, at level i (with $i = 0$ at the root), we have 2^i nodes where each node is a merge with input two sequences of length $n/2^{i+1}$.

Therefore, the work of such a balanced tree of $merge_<$'s is the familiar sum

$$\begin{aligned} &\leq \sum_{i=0}^{\log n} 2^i \cdot c\left(\frac{n}{2^{i+1}} + \frac{n}{2^{i+1}}\right) \\ &= \sum_{i=0}^{\log n} 2^i \cdot c\left(\frac{n}{2^i}\right) \end{aligned}$$

This sum, as you have seen before, evaluates to $O(n \log n)$.

Merge Sort. In fact, this algorithm is essentially the merge sort algorithm. We can use our reduction technique for implementing divide-and-conquer algorithms to implement merge sort with a `reduce`.

In particular, we can write a version of merge sort, which we refer to as reduceSort, as follows:

```
1 val combine = merge_<
2 val base = singleton
3 val emptyVal = empty
4 fun reduceSort(S) = reduce combine emptyVal (map base S)
```

where $merge_<$ is a merge function that uses an (abstract) comparison operator $<$.

Summary 4.53. *A brief summary of a few points.*

- *When applying a binary function in* `reduce`*, if the function is associative, the order of applications does not matter for the final result.*
- *When applying a binary function in* `reduce`*, the order of applications does matter when calculating the cost (work and span), regardless of whether the function is associative or not.*
- *Implementing a "reduce" with* `merge` *with a sequential order leads to insertion sort, while implementing with a balanced tree (parallel order) leads to merge sort.*

The cost of `reduce` in general. In general, how would we go about defining the cost of `reduce` with higher order functions. Given a reduction tree,

we'll first define $\mathcal{R}(\texttt{reduce}\ f\ \mathbb{I}\ S)$ as

$$\mathcal{R}(\texttt{reduce}\ f\ \mathbb{I}\ S) = \left\{\text{all function applications } f(a\right.$$

Following this definition, we can state the cost of `reduce` as follows:

$$W(\texttt{reduce}\ f\ \mathbb{I}\ S) = O\left(n + \sum_{f(a,b)\in\mathcal{R}(f\ \mathbb{I}\ S)} W(\right.$$

$$S(\texttt{reduce}\ f\ \mathbb{I}\ S) = O\left(\log n \max_{f(a,b)\in\mathcal{R}(f\ \mathbb{I}\ S)} S(f\right.$$

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 tree is at most $O(\log n)$ deep. Since each node in the tree has span at most $\max_{f(a,b)} S(f(a,$ any root-to-leaf path, including the "critical path," has at most $O(\log n \max_{f(a,b)} S(f(a,b))$ span.

This can be used, for example, to prove the following lemma:

Lemma 4.54. *For any combine function* $f : \alpha \times \alpha \to \alpha$ *and size function* $s : \alpha \to \mathbb{R}_+$, *if for any* x, y,

1. $s(f(x,y)) \leq s(x) + s(y)$ *and*
2. $W(f(x,y)) \leq c\,(s(x) + s(y))$ *for some constant* c,

then

$$W(\texttt{reduce}\ f\ \mathbb{I}\ S) = O\left(\log|S| \sum_{x \in S} (1 + s(x))\right).$$

Applying this lemma to the merge sort example, we have

$$W(\texttt{reduce}\,\texttt{merge}_<\ \langle\,\rangle\ \langle\langle a\rangle : a \in A\rangle) = O(|A|\,\mathrm{l}$$

4.9 Collect

Thus far we considered two very important functions on sequences, `scan` and `reduce`.

We now look a third function: `collect`.

Specification of Collect. Let's start with something that you may have heard of.

Question 4.55. *Do you know of key-value stores?*

The term key-value store often refers to a storage systems (which may in on disk or in-memory) that stores pairs of the form "key x value."

Question 4.56. *Can you think of a way of representing a key-value store using a data type that we know?*

We can use a sequence to represent such a store.

Example 4.57. *For example, we may have 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 4.58. *We can determine the classes taken by each student.*

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

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 interface is:

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

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.

4.10 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 $\texttt{update}\ (i, v)\ S$ 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 $\texttt{map}$ 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 the latest copy, since this is the only way we will be using an `stseq`. The interface and costs is as follows:

`fromSeq(S)` : α `seq` $\rightarrow$ α `stseq`

Converts from a regular sequence to a stseq.

`toSeq(ST)` : α `stseq` $\rightarrow$ α `seq`

Converts from a stseq to a regular sequence.

`nth ST` i : α `stseq` $\rightarrow$ `int` $\rightarrow$ α

Returns the i^{th} element of ST. Same as for seq.

`update` (i,v) ST : (`int` $\times$ α) $\rightarrow$ α `stseq` $\rightarrow$ α `stseq`

Replaces the i^{th} element of ST with v.

`inject` I ST : (`int` $\times$ α) `seq` $\rightarrow$ α `stseq` $\rightarrow$ α `stseq`

For each $(i, v) \in I$ replaces the i^{th} element of ST v

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 4.59.

```
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 con-

stant 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.

Syntax 4.60 (Sequences). *Translation from sequence notation to the sequence functions.*

S_i	`nth` S i
$\|S\|$	`length`(S)
$\langle\,\rangle$	`empty`$()$
$\langle\, v \,\rangle$	`singleton`(v)
$\langle\, i, \ldots, j \,\rangle$	`tabulate` (**fn** $k \Rightarrow i + k$) $(j - i + 1)$
$S\,\langle\, i, \ldots, j \,\rangle$	`subseq` S $(i, j - i + 1)$
$\langle\, e : p \in S \,\rangle$	`map` (**fn** $p \Rightarrow e$) S
$\langle\, e : 0 \leq i < n \,\rangle$	`tabulate` (**fn** $i \Rightarrow e$) n
$\langle\, p \in S \mid e \,\rangle$	`filter` (**fn** $p \Rightarrow e$) S
$\langle\, e_1 : p \in S \mid e_2 \,\rangle$	`map` (**fn** $p \Rightarrow e_1$) (`filter` (**fn** $p \Rightarrow e_2$) S)
$\langle\, e : p_1 \in S_1, p_2 \in S_2 \,\rangle$	`flatten`(`map` (**fn** $p_1 \Rightarrow$ `map` (**fn** $p_2 \Rightarrow e$) S_2) S_1)
$\langle\, e_1 : p_1 \in S_1, p_2 \in S_2 \mid e_2 \,\rangle$	`flatten`(`map` (**fn** $p_1 \Rightarrow \langle\, e_1 : p_2 \in S_2 \mid e_2 \,\rangle$) S_1)
$\sum_{p \in S} e$	`reduce add` 0 (`map` (**fn** $p \Rightarrow e$) S)
$\sum_{i=k}^{n} e$	`reduce add` 0 (`map` (**fn** $i \Rightarrow e$) $\langle\, k, \ldots, n \,\rangle$)

The $\sum$ can be replaced with min, max, $\cup$ *and* $\cap$ *with the presumed meanings, and by replacing* `add` *and* 0 *with the appropriate values.*

•