Recitation 1

Scan

1.1 Announcements

- *SkylineLab* has been released, and is due **Friday afternoon**. It's worth 125 points.
- *BignumLab* will be released on Friday.

1.2 What is `scan`?

In the `SEQUENCE` library, there is a symmetry among certain aggregation functions:

Sequential	Parallel
`iterate`	`reduce`
`iteratePrefixes`	`scan`
`iteratePrefixesIncl`	`scanIncl`

We can see this symmetry in their types...

`iterate`	$(\beta * \alpha \to \beta) \to \beta \to \alpha$ `seq` $\to \beta$
`reduce`	$(\alpha * \alpha \to \alpha) \to \alpha \to \alpha$ `seq` $\to \alpha$
`iteratePrefixes`	$(\beta * \alpha \to \beta) \to \beta \to \alpha$ `seq` $\to \beta$ `seq` $* \beta$
`scan`	$(\alpha * \alpha \to \alpha) \to \alpha \to \alpha$ `seq` $\to \alpha$ `seq` $* \alpha$
`iteratePrefixesIncl`	$(\beta * \alpha \to \beta) \to \beta \to \alpha$ `seq` $\to \beta$ `seq`
`scanIncl`	$(\alpha * \alpha \to \alpha) \to \alpha \to \alpha$ `seq` $\to \alpha$ `seq`

...as well as their output behavior: each of the parallel functions has output identical to its sequential analog under the condition that the first two arguments are an *associative function* and a corresponding *identity*, respectively.

Definition 1.1. *A function f is associative if for every x, y, z,*

$$f(f(x, y), z) = f(x, f(y, z)).$$

Definition 1.2. *A value b is an identity of a binary function f if for every x,*

$$f(b, x) = x = f(x, b).$$

So, for now, you can think of `scan` as a magical function which performs iteration of an associative function in parallel. If the function is constant-time, then an application of scan has linear work and logarithmic span.

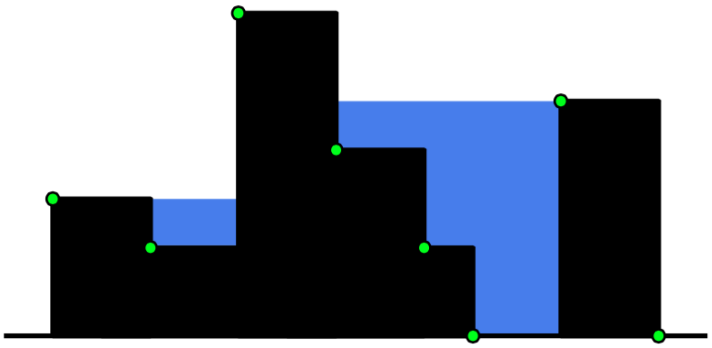
Remark 1.3. *In reality, we can relax the constraint on the identity. It only needs to be an identity for the values encountered during the execution of the `reduce`, `scan`, or `scanIncl`. For example, if S is a sequence of non-negative integers, then (`scan Int.max 0` S) will still be logically equivalent to (`iteratePrefixes Int.max 0` S), despite the fact that, in general, 0 is not an identity for `Int.max`.*

1.3 Skyline-Fill

For this example, we'll use the same conventions given in *SkylineLab*:

- Skylines are sequences of points (x, y) sorted by x-coordinate,
- all x-coordinates are unique and non-negative, and
- all y-coordinates (heights) are non-negative.

Imagine pouring water on a skyline. How much water can it hold?



Task 1.4. *Implement the function*

```
val fill : (int * int) Seq.t → int
```

where (`fill S`) returns the area of water which can fill the skyline S. Your implementation should have $O(|S|)$ work and $O(\log |S|)$ span.

We can approach this problem by conceptually splitting the skyline up into columns, determining the quantity of water that each column holds, and finally summing these quantities. We'll have one column per adjacent pair of points in the skyline.

For a given segment with height h and width w, if it has a skyscraper to its left with height ℓ and another to its right with height r, then the area of water we can store above this segment is $w \cdot \max(0, \min(\ell, r) - h)$. But how do we determine ℓ and r?

Notice that ℓ and r are simply the max of all heights that come before and after the current segment, respectively. The former is the max of a prefix; the latter, the max of a suffix. `scan` with `Int.max` can directly give us maxes of prefixes, but we'll have to reverse the sequence and run it again to handle suffixes.

Algorithm 1.5. *Parallel Skyline-fill.*

```
1  fun fill S =
2    let
3      fun ithSeg i =
4        let val ((x1, h), (x2, _)) = (Seq.nth S i, Seq.nth S (i+1))
5        in (x2 - x1, h)
6        end
7      val segments = Seq.tabulate ithSeg (Seq.length S - 1)
8
9      val hs = Seq.map (fn (w, h) => h) segments
10     val (lhs, _) = Seq.scan Int.max 0 hs
11     val (revrhs, _) = Seq.scan Int.max 0 (Seq.rev hs)
12     val rhs = Seq.rev revrhs
13
14     fun zip3With f (A, B, C) =
15       let fun f' (x, (y, z)) = f(x, y, z)
16       in Seq.map f' (Seq.zip (A, Seq.zip (B, C)))
17       end
18     fun columnPour (l, (w, h), r) =
19       w * Int.max (0, Int.min (l, r) - h)
20     val columns = zip3With columnPour (lhs, segments, rhs)
21   in
22     Seq.reduce op+ 0 columns
23   end
```

This algorithm meets the required cost bounds:

- Lines 7,9,12,20: $O(|S|)$ work and $O(1)$ span.
- Lines 10,11,22: $O(|S|)$ work and $O(\log |S|)$ span.

1.4 A Group at Dinner

A group of n friends sit around a circular table at a restaurant. Some of them know what they want to order; some of them don't. The ones who don't know what to order decide to pick the same thing as the person on their left.

Task 1.6. *Implement the function*

```
val groupOrder : (int → α option) → int → α Seq.t
```

where (`groupOrder f n`*) returns the sequence of orders of a group of n people. $f(i)$ is either the preferred order of the i^{th} person, or* `NONE` *if they don't know what they want. Assume the people are labeled 0 to $n-1$ counter-clockwise, and that at least one person originally knows what they want to order. Your implementation should have $O(n)$ work and $O(\log n)$ span.*

We can begin with the sequence of `option`s indicating preferred orders – $\langle f(i) : 0 \leq i < n \rangle$ – but this sequence has "missing pieces" (`NONE`s) for each person who originally didn't know what they wanted. The key to the puzzle is using `scan` (or actually `scanIncl`) to "copy" information from left to right. The necessary function is

```
fun copy (a, b) =
  case b of
    NONE ⇒ a
  | SOME _ ⇒ b
```

This function is in fact associative, but we leave proving this as an exercise to the reader. We can use `NONE` as its identity.

After copying across the sequence, some values at the front might still be `NONE`. Since these should really wrap around from the end of the sequence (the table is circular!), we can just replace them with the last order. This is guaranteed to be a `SOME`, because we know that at least one person in the group originally knew what they wanted to order.

Algorithm 1.7. *An example of copy-scan.*

```
1  fun groupOrder f n =
2    let
3      fun copy (a,b) =
4        case b of
5          NONE ⇒ a
6        | SOME _ ⇒ b
7
8      val init = Seq.tabulate f n
9      val copiedRight = Seq.scanIncl copy NONE init
10     val SOME lastOrder = Seq.nth copiedRight (n − 1)
11   in
12     Seq.map (fn SOME x ⇒ x | NONE ⇒ lastOrder) copiedRight
13   end
```

This algorithm meets the required cost bounds:

- Line 8,12: $O(n)$ work and $O(1)$ span.
- Line 9: $O(n)$ work and $O(\log n)$ span.
- Line 10: $O(1)$ work and span.

1.5 Bonus Exercises

Exercise 1.8. *Implement* `parenMatch` *(from the previous recitation) using* `scan` *such that it has linear work and logarithmic span. Try adapting the iterative approach.*

Exercise 1.9. *Implement* `parenDist` *(from* ParenLab*) using* `scan` *such that it has linear work and logarithmic span.*

Exercise 1.10. *Did you know that you can calculate the first n Fibonacci numbers in $O(n)$ time and $O(\log n)$ span? We claim that if we extend the Fibonacci sequence as so...*

$$\begin{aligned} F_{-1} &= 1 \\ F_0 &= 0 \\ F_1 &= 1 \\ &\vdots \\ F_n &= F_{n-1} + F_{n-2} \end{aligned}$$

...that the following holds for $n \geq 0$ (easily provable via induction):

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}$$

Using this fact, implement a function

```
val fibs : int → int Seq.t
```

which returns the first n Fibonacci numbers in $O(n)$ work and $O(\log n)$ span. Use `scan` *to compute prefixes of matrix multiplications.*

Exercise 1.11. *Implement a function*

```
val parenPairs : Paren.t Seq.t → (int * int) Seq.t
```

where (`parenPairs` *S) returns a sequence of index-pairs where each pair contains the index of a parenthesis as well as its matching partner. For example the input* `(())()` *should yield some permutation of $\langle(0,3),(4,5),(1,2)\rangle$. Your implementation should have $O(|S| \log |S|)$ work and $O(\log^2 |S|)$ span.*
Hint: try marking each parenthesis with how many other parentheses are enclosing it. You might also need to `sort` *at some point...*

.