

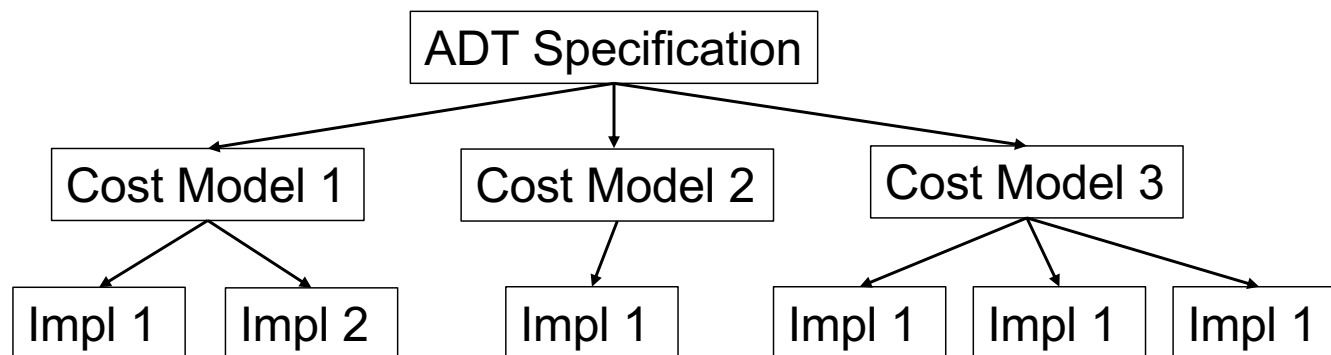
Parallel And Sequential Data Structures and Algorithms

Algorithms for Sequences

Learning Objectives

- Understanding the sequence ADT and how it can be implemented using arrays
- Implement core Sequence operations like **filter**, **flatten**, **collect**, and lexicographic **comparison** efficiently

Abstract Data Types (ADT)



Definitions

Definition (Sequence): Mathematically, a finite **sequence** of length n over elements of type T is a function $\{0, 1, \dots, n - 1\} \rightarrow T$

Interface (Sequence): A `sequence<T>` (with value type T) supports

- `nth(S : sequence<T>, i : int) -> T`:
returns the i^{th} element of the sequence S , abbreviated $S[i]$
- `length(S : sequence<T>) -> int`:
return the length of the sequence S , abbreviated $|S|$
- `subseq(S : sequence<T>, i : int, k : int) -> sequence<T>`:
returns a view of the subsequence of S starting at index i with length k

(Mostly) Full Interface

nth, size, empty, singleton

tabulate, map, zip

filter, flatten

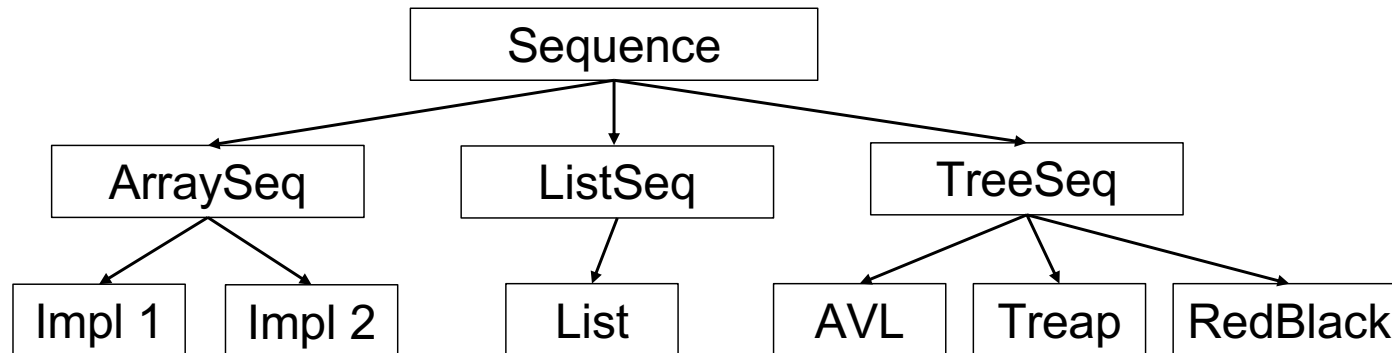
subseq, splitMid

reduce, scan

sort, merge

collect, collate (i.e., compare)

Abstract Data Types (ADT)



Arrays and Imperative Parallelism

Arrays: Building Block for Array Sequences

Definition (Array): An **array** is a contiguous fixed-size collection of elements of the same type that allows $O(1)$ access to the element at any position.

- We will assume that arrays support the following operations:
 - **allocate** $\langle T \rangle(n)$: allocate an array of length n of type T
 - **A[i]**: return the i^{th} element of A
 - **|A|**: return the length of A
 - **A[i] $\leftarrow$ x**: write the value x to the i^{th} element of A

Arrays are **mutable**! We need mutation to implement low-level primitives.

Definitions

Definition (Data Race): A **data race** occurs when there are two unsynchronized parallel operations on the same memory location, and at least one of the operations is a write.

- If two operations are both reads, this is not a data race.
- This can't happen in a purely functional program.
- In most programming languages, the presence of data race makes the behavior of the program **undefined**. We will therefore treat it as such in this class. **All data races are UB.**

Imperative Parallel Loops

Definition (Parallel For Loop): A parallel for loop executes its body for every value of its range in parallel.

```
parallel for i in 0...n-1:  
    e(i)
```

where $e(i)$ is some (impure) function of i

- This could cause a data race if $e(i)$ and $e(j)$ both read/write to the same memory location for different values of i and j .
- The work and span are: $W = \sum_i W(e(i)), \quad S = \max_i S(e(i))$

(Mostly) Full Interface Array Sequence Implementation

nth, size, empty, singleton

tabulate, map, zip

filter, flatten

subseq, splitMid

reduce, scan

sort, merge

collect, collate

Covered Today

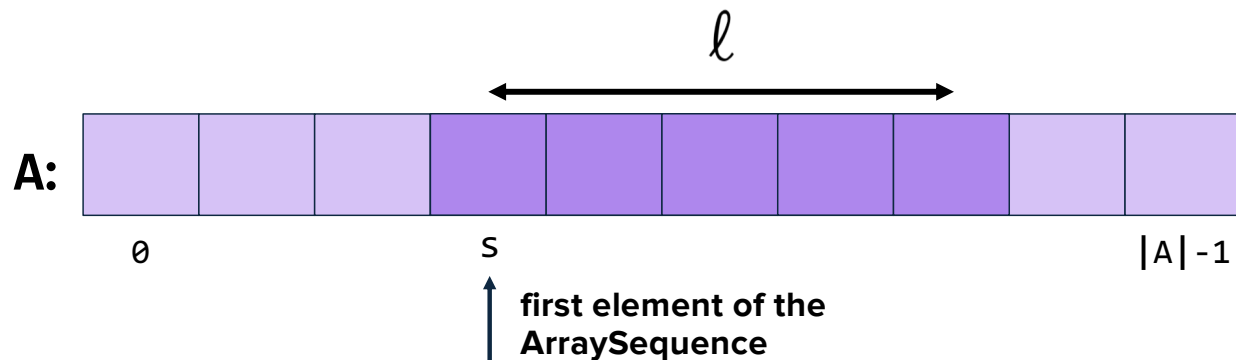
In previous lectures

Full cost specification for all operations is in the Documentation on the course website.

ArraySequence

ArraySequences

```
type sequence<T> = {  
  A : Array<T>  // actual Array A  
  s : int       // start index (initially 0)  
  ℓ : int       // length (initially |A|)  
}
```



nth and length

- `nth` and `length` are straightforward to implement from the definition of the `ArraySequence` data type

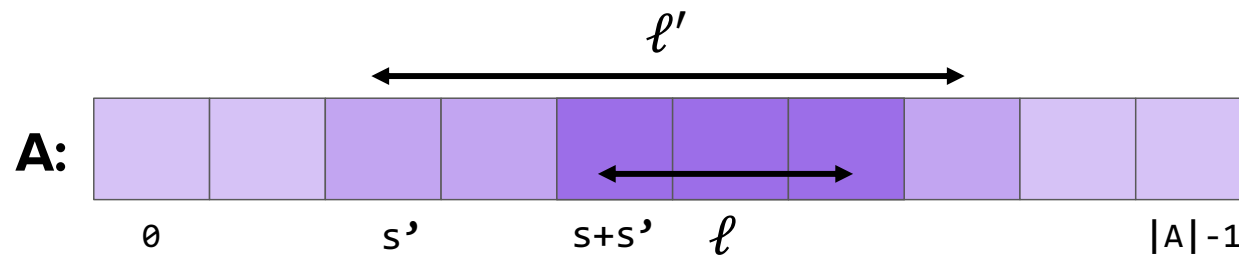
```
fun nth(S : sequence<T>, i : int) -> T:  
  (A, s, _) = S  
  return A[s+i]
```

```
fun length(S : sequence<T>) -> int:  
  (_, _, ℓ) = S  
  return ℓ
```

- Both have $O(1)$ cost (work and span)

subseq

```
fun subseq(S : sequence<T>, s : int,  $\ell$  : int) -> sequence<T>:  
  (A, s',  $\ell'$ ) = S  
  return (A, s+s',  $\ell$ )
```



tabulate

```
fun tabulate(f : (int -> T), n : int) -> sequence<T>:  
  R = allocate<T>(n)  
  parallel for i in 0...n-1:  
    R[i] ← f(i)  
  return (R, 0, n)
```

f(0)	f(1)	f(2)	f(3)	f(4)	f(5)	f(6)	f(7)	f(8)	f(9)
------	------	------	------	------	------	------	------	------	------

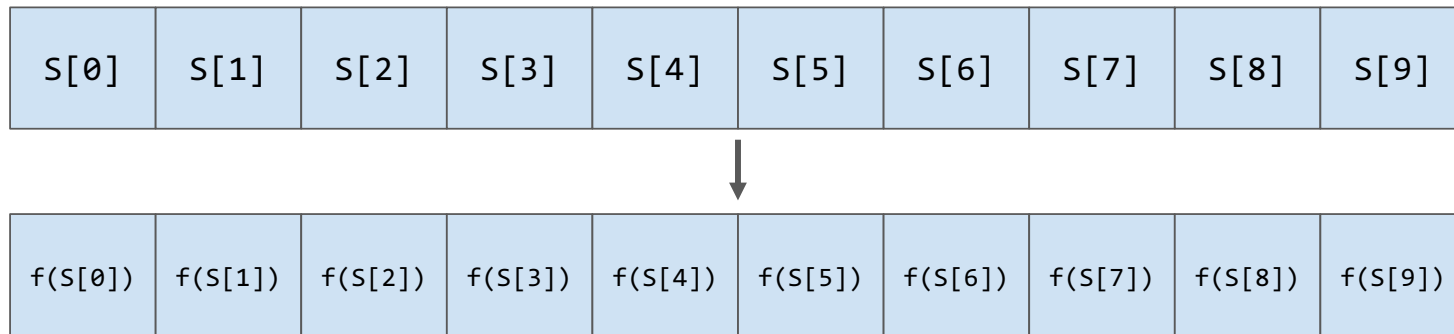
$$W = \sum_{0 \leq i < n} W(f(i))$$

Question: Is there a data race?

$$S = \max_{0 \leq i < n} S(f(i))$$

Answer: No! Each parallel execution of the loop body operates on a different element of R

map

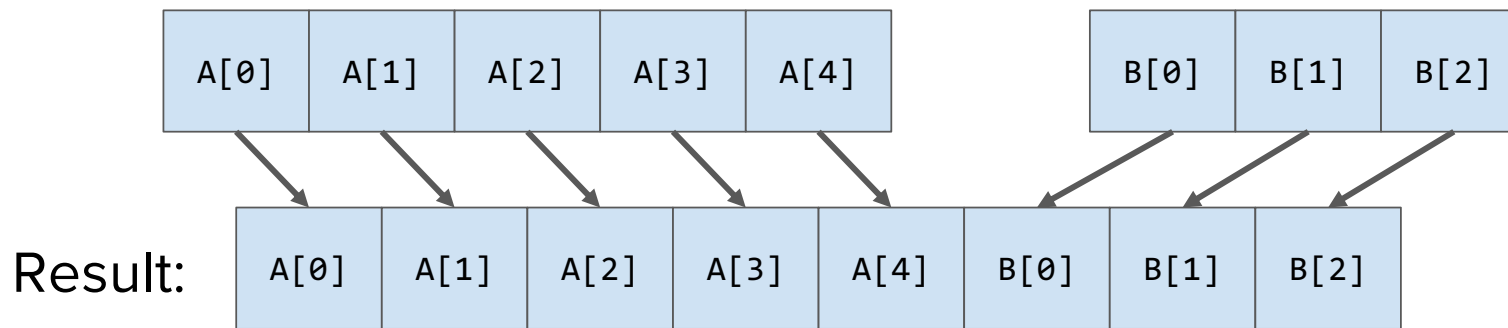


```
fun map(f : (T -> U), S : sequence<T>) -> sequence<U>:  
  return tabulate(fn i => f(S[i]), |S|)
```

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

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

append



```
fun append(A : sequence<T>, B : sequence<T>) -> sequence<T>:  
  return tabulate(fn i => (A[i] if i<|A| else B[i-|A|]), |A|+|B|)
```

$$W = O(|A| + |B|) \quad S = O(1)$$

filter

Definition (Filter):

`filter : (p : (T -> bool), S : sequence<T>) -> sequence<T>`

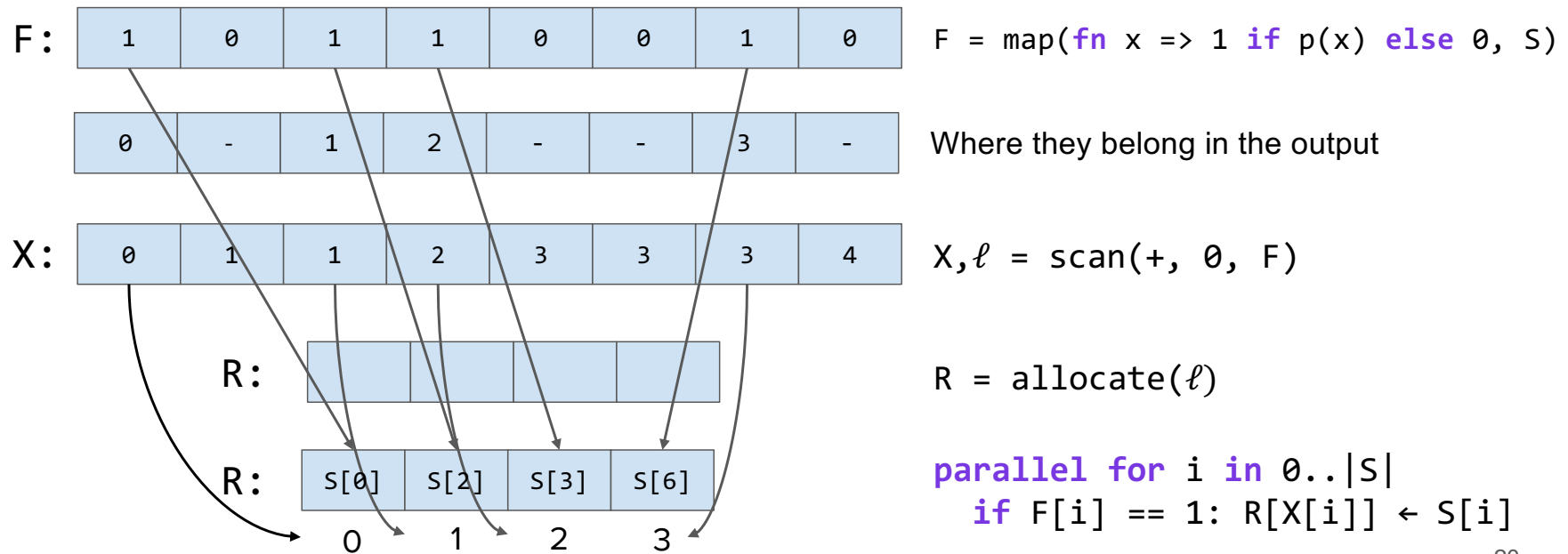
`filter(p, S)` returns a sequence consisting of the elements of `S` which satisfy the predicate `p`. The relative order of the elements is preserved.

Example:

`filter(fn x => x < 5, [7,1,3,11,7,2])` returns `[1, 3, 2]`

filter

Goal: implement filter in linear work and logarithmic span



filter implementation

```
fun filter(p : (T -> bool), S : sequence<T>) -> sequence<T>:
  F = map (fn x => 1 if p(x) else 0, S)  $O(\sum W(p(x)))$  work,  $O(\max S(p(x)))$  span
  X,  $\ell$  = scan(+, 0, F)  $O(|S|)$  work,  $O(\log |S|)$  span
  R = allocate<T>( $\ell$ )
  parallel for i in 0...|S|-1:  $O(|S|)$  work,  $O(1)$  span
    if F[i] == 1:
      R[X[i]]  $\leftarrow$  S[i]
  return (R, 0,  $\ell$ )
```

$$W = \sum_{x \in S} \mathcal{W}(p(x)) \qquad S = O(\log |S|) + \max_{x \in S} \mathcal{S}(p(x))$$

flatten

Definition (Flatten):

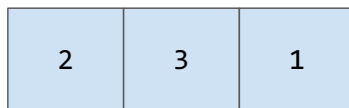
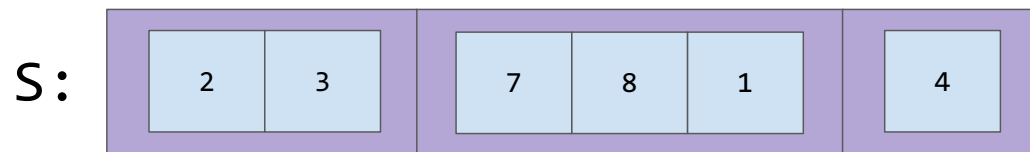
`flatten : (S : sequence<sequence<T>>) -> sequence<T>`

Given a nested sequence of sequences, return a sequence consisting of all the elements of the inner sequences in the same relative order.

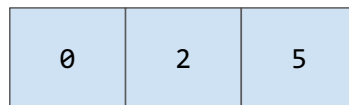
Example (Flatten):

`flatten([[2,3],[7,8,1],[4]])` returns `[2,3,7,8,1,4]`

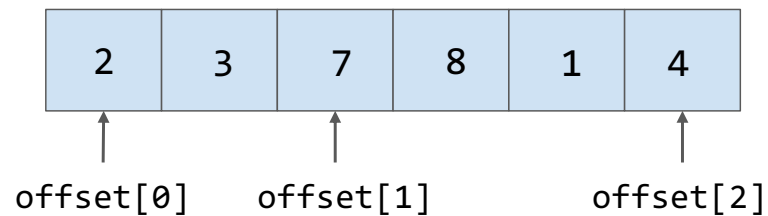
flatten



`L = map (fn x=> |x|, S) // length of inner sequences`



`offset, length = scan(+, 0, L)`



```
parallel for i in 0...|S|-1:  
  parallel for j in 0...L[i]-1:  
    R[offset[i]+j] ← S[i][j]
```

flatten implementation

```
fun flatten(S : sequence<sequence<T>>) -> sequence<T>:  
  L = map (fn x => |x|, S)            $O(|S|)$  work,  $O(\log |S|)$  span  
  offset, length = scan(plus, 0, L)   $O(|S|)$  work,  $O(\log |S|)$  span  
  R = allocate<T>(length)  
  parallel for i in 0...|S|-1:  
    parallel for j in 0...L[i]-1:     $O\left(|S| + \sum_{x \in S} |x|\right)$  work,  $O(1)$  span  
      R[offset[i]+j] ← S[i][j]  
  return (R, 0, length)
```

$$W = O\left(\sum_{x \in S} (1 + |x|)\right) \quad s = O(\log |S|)$$

collect

Definition (Collect):

`collect : (S : sequence<(K,V)>) -> sequence<(K, sequence<V>)>`

Given a sequence of key-value pairs, returns a **sorted** sequence of all distinct keys paired with a sequence of their corresponding values in their original relative order.

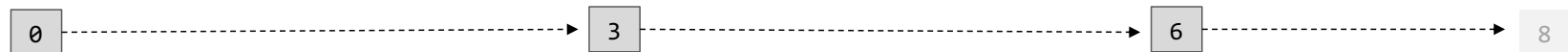
```
collect([(1, 'a'), (2, 'b'), (1, 'c'), (2, 'a'), (2, 'd')])  
  -> [(1, ['a', 'c']), (2, ['b', 'a', 'd'])]
```

collect

(2, 'a') (1, 'b') (1, 'a') (7, 'd') (2, 'e') (7, 'a') (2, 'b') (1, 'z')

↓ S' = sort(S) // by key only

(1, 'b') (1, 'a') (1, 'z') (2, 'a') (2, 'e') (2, 'b') (7, 'd') (7, 'a')



```
ids = filter(fn i => (i == 0 || i == |S| || S'[i-1][0] < S'[i][0]), [0, ..., |S|])
```

```
return tabulate(fn i => (S'[ids[i]][0],  
  map(fn p => p[1], subseq(S', ids[i], ids[i+1]-ids[i])), |ids|-1)
```

(1, ['b', 'a', 'z'])	(2, ['a', 'e', 'b'])	(7, ['d', 'a'])
----------------------	----------------------	-----------------

collect implementation

```
fun collect(S : sequence<(K,V)>) -> sequence<(K,sequence<V>>):  
    S' = sort(S) // stable sort by key only  
    ids = filter(fn i => (i == 0 || i == |S| || S'[i-1][0] < S'[i][0]), [0, ..., |S|])  
    return tabulate(fn i => (S'[ids[i]][0],  
                             map(fn p => p[1], subseq(S', ids[i], ids[i+1]-ids[i]))), |ids|-1)
```

- Cost of sort: $O(|S| \log |S|)$ work and $O(\log^2 |S|)$ span
- Cost of filter: $O(|S|)$ work and $O(\log |S|)$ span
- Cost of tabulate: $O(|ids|)$ work and $O(1)$ span ($|ids| \leq |S|$) +
 - Cost of map(s): $O(|S|)$ work and $O(1)$ span (each subseq is **disjoint**)

$$W = O(|S| \log |S|), \quad S = O(\log^2 |S|)$$

Lexicographic Comparison

Definition (Lexicographic Comparison):

$\text{compare} : ((T, T) \rightarrow \text{order}), \text{sequence}\langle T \rangle, \text{sequence}\langle T \rangle \rightarrow \text{order}$

$\text{compare}(f, s1, s2)$ returns the lexicographical comparison of sequences $s1$ and $s2$ using the comparison function $f : (T, T) \rightarrow T$

Examples (compare):

$[1, 5, 1, 2, 2] \text{ vs } [1, 5, 2, 1, 0] \rightarrow \text{LESS } (<)$
 $[1, 5, 2, 1, 0] \text{ vs } [1, 5, 2, 1] \rightarrow \text{GREATER } (>)$
 $["a", "b", "c"] \text{ vs } ["a", "b", "c"] \rightarrow \text{EQUAL } (=)$

compare

Question: Which element(s) dictate the result of the comparison?

$[1, 2, 3, 1, 5, 1, 2, 2]$
 $[1, 2, 3, 1, 5, 2, 1, 0]$ → LESS

$[1, 2, 3, 1, 5, 1, 2, 2]$
 $[1, 2, 3, 1, 5, 1, 1, 0]$ → GREATER

Observation: The result is based on the **first mismatching pair**

Finding the Mismatch

A: [1, 2, 3, 1, 5, 1, 2, 0, 1, 3, 5]
 B: [1, 2, 3, 1, 5, 2, 1, 0, 1, 4]

↓ comps = map(f, zip(A, B))

[=, =, =, =, =, <, >, =, =, >]

↓ How would we get this?
 Can we **propagate** the answer to the end?

[=, =, =, =, =, <, <, <, <, <]

= ↑ = ↑ = ↑ = ↑ = ↑ < ↑ < ↑ < ↑ < ↑ < ↑

Associative

```
fun copy_notequal(x, y):
    return (y if x == EQUAL else x)
```

reduce(copy_notequal, EQUAL, comps)

compare implementation

```
type order = LESS | EQUAL | GREATER

fun compare(f : (T,T)->order, a: sequence<T>, b: sequence<T>) -> order:
  comps = map(f, zip(a, b)) // compare corresponding pairs in a and b
  fun copy_notequal(x, y):
    return (y if x == EQUAL else x) // propagate leftmost non-EQUAL
  res = reduce(copy_notequal, EQUAL, comps)
  match res with:
    case EQUAL: return LESS if |a|<|b| else GREATER if |a|>|b| else EQUAL
    case _:      return res
```

$$W = O(\min(m, n)), \quad S = O(\log(\min(m, n)))$$

Summary

- Efficiently implementing sequences with arrays requires some imperative (non-functional) parallelism!
- Core sequence operations like **filter** and **flatten** can be implemented efficiently as applications of scan
- Useful algorithms build on top of earlier algorithms, e.g., **collect** as an application of **sort** and **filter**