Parallel Functional Programming, continued

COS 326
Andrew Appel
Princeton University
The parallel sequence abstraction is powerful:

- tabulate
- nth
- length
- map
- split
- treeview
- scan
  - used to implement prefix-sum
  - clever 2-phase implementation
  - used to implement filters
- sorting
PARALLEL COLLECTIONS IN THE "REAL WORLD"
If Google wants to index all the web pages (or images or gmys or google docs or ...) in the world, they have a lot of work to do

• Same with Facebook for all the facebook pages/entries
• Same with Twitter
• Same with Amazon
• Same with ...

Many of these tasks come down to map, filter, fold, reduce, scan
Do the reading . . .

Chapter 2, “Search Engine Indexing”

(On reserve for this course, available at blackboard.princeton.edu)

(Read also Chapter 3, “Page Rank” so you can appreciate what you were doing in Assignment 5 . . .)
Indexing the web

Step 1: web crawling
- make a big set of key-value pairs
  key: URL  value: contents of web page (HTML)
  (how many? 130,000,000,000,000 that’s 130 trillion)

Step 2: indexing
  “Invert”
  key: contents  value: URL
or more precisely,
  key: word  value: set of (URL, position-in-page) pairs

Step 3: search
  3,800,000 searches per minute, 2,000,000,000,000 per year
Google MapReduce (2004): a fault tolerant, massively parallel functional programming paradigm

– based on our friends "map" and "reduce"
– Hadoop is the open-source variant
– Database people complain that they have been doing it for a while
  • ... but it was hard to define

Fun stats circa 2012:

– Big clusters are ~4000 nodes
– Facebook had 100 PB in Hadoop
– TritonSort (UCSD) sorts 900GB/minute on a 52-node, 800-disk hadoop cluster
• Map-reduce operates over collections of key-value pairs
  – millions of files (eg: web pages) drawn from the file system and parsed in parallel by many machines
• The map-reduce engine is parameterized by 3 functions, which roughly speaking do this:

map : key1 * value1 -> (key2 * value2) list
combine : key2 * (value2 list) -> value2 option
reduce : key2 * (value2 list) -> key3 * (value3 list)

optional – often used to compress data before transfer from a mapper machine to a reducer machine
Architecture

Input Data

Mapper

Local Storage

Reducer

Mapper

Local Storage

Reducer

Mapper

Local Storage

Combine

Output Data

Map

Shuffle/Sort

Reduce
Iterative Jobs are Common

Input Data → Mapper → Reducer → Output Data
Input Data → Mapper → Reducer → Output Data
Input Data → Mapper → Reducer → Output Data
Input Data → Mapper → Reducer → Output Data

Working Set
Worker → Worker

The Control Plane

User Program

Controller

Worker

Input Data

Worker

Input Data

Worker

Input Data
Jobs, Tasks and Attempts

- A single *job* is split into many *tasks.*
- Each *task* may include many calls to map and reduce.
- *Workers* are long-running processes that are assigned many tasks.
- Multiple workers may *attempt* the same task:
  - each invocation of the same task is called an attempt.
  - the first worker to finish "wins".
- Why have multiple machines attempt the same task?
  - machines will fail:
    - approximately speaking: 5% of high-end disks fail/year.
    - if you have 1000 machines: 1 failure per week.
    - repeated failures become the common case.
  - machines can partially fail or be slow for some reason:
    - reducers can't start until *all* mappers complete.
Flow of Information

Worker
- Heartbeats
- Tasks to start
- Completed

Controller

User Program
- Job config.
- OK

Completed

Tasks to start

Heartbeats
A Modern Software Stack

Workload Manager
High-level scripting language

Cluster Node
Cluster Node
Cluster Node
Cluster Node

Distributed Filesystem
Distributed Execution Engine
Key-value store
High-level scripting language
Workload Manager
Hadoop interfaces:

```java
interface Mapper<K1,V1,K2,V2> {
    public void map (K1 key,
                      V1 value,
                      OutputCollector<K2,V2> output)
                      ...
}
```

```java
interface Reducer<K2,V2,K3,V3> {
    public void reduce (K2 key,
                        Iterator<V2> values,
                        OutputCollector<K3,V3> output)
                        ...
}
```
class WordCountMap implements Map {
    public void map(DocID key, List<String> values, OutputCollector<String, Integer> output) {
        for (String s : values)
            output.collect(s, 1);
    }
}

class WordCountReduce {
    public void reduce(String key, Iterator<Integer> values, OutputCollector<String, Integer> output) {
        int count = 0;
        for (int v : values)
            count += 1;
        output.collect(key, count)
    }
}
PLEASE RELAX
AND FOR THE SAKE OF HYGIENE,
WIPE THE
JAVA CODE OFF YOUR BRAIN
ASSIGNMENT #7: IMPLEMENTING AND USING PARALLEL COLLECTIONS
End goal: develop a system for efficiently computing US population queries by geographic region
Assignment 7

Libraries build layered concurrency abstractions

message passing

Operating System kernel (fork, pipe, ...)

Applications (your assignment)

Library

Library

Library

Library

Core  Core  Hardware  Core  Core

web index

gerographic queries

parallel sequences

futures

unix processes
### map-reduce API for Assignment 7

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Work</th>
<th>Span</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tabulate</code> (f: <code>int</code> -&gt; <code>α</code>) (n: <code>int</code>) : <code>α</code> seq</td>
<td>Create seq of length n, element i holds f(i)</td>
<td>n</td>
<td>1</td>
</tr>
<tr>
<td><code>seq_of_array</code> : <code>α</code> array -&gt; <code>α</code> seq</td>
<td>Create a sequence from an array</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><code>array_of_seq</code> : <code>α</code> seq -&gt; <code>α</code> array</td>
<td>Create an array from a sequence</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><code>iter</code> (f: <code>α</code> -&gt; <code>unit</code>): <code>α</code> seq -&gt; <code>unit</code></td>
<td>Applying f on each element in order.</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td><code>length</code> : <code>α</code> seq -&gt; <code>int</code></td>
<td>Return the length of the sequence</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><code>empty</code> : <code>unit</code> -&gt; <code>α</code> seq</td>
<td>Return the empty sequence</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><code>cons</code> : <code>α</code> -&gt; <code>α</code> seq -&gt; <code>α</code> seq</td>
<td>cons a new element on the beginning</td>
<td>n</td>
<td>1</td>
</tr>
<tr>
<td><code>singleton</code> : <code>α</code> -&gt; <code>α</code> seq</td>
<td>Return the sequence with a single element</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><code>append</code> : <code>α</code> seq -&gt; <code>α</code> seq -&gt; <code>α</code> seq</td>
<td>(nondestructively) concatenate two sequences</td>
<td>m+n</td>
<td>1</td>
</tr>
<tr>
<td><code>nth</code> : <code>α</code> seq -&gt; <code>int</code> -&gt; <code>α</code></td>
<td>Get the nth value in the sequence. Indexing is zero-based.</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><code>map</code> (f: <code>α</code> -&gt; <code>'b</code>) -&gt; <code>α</code> seq -&gt; <code>α</code> seq</td>
<td>Map the function f over a sequence</td>
<td>n</td>
<td>1</td>
</tr>
<tr>
<td><code>reduce</code> (f: <code>α</code> -&gt; <code>α</code> -&gt; <code>α</code>) (base: <code>α</code>): <code>α</code> seq -&gt; <code>α</code></td>
<td>Fold a function f over the sequence. f must be associative, and base must be the unit for f.</td>
<td>n</td>
<td>log n</td>
</tr>
<tr>
<td><code>mapreduce</code> : <code>(α-&gt;β)-&gt;(β-&gt;β-&gt;β)-&gt;β</code> -&gt; <code>α</code> seq -&gt; <code>β</code></td>
<td>Combine the map and reduce functions.</td>
<td>n</td>
<td>log n</td>
</tr>
<tr>
<td><code>flatten</code> : <code>α</code> seq seq -&gt; <code>α</code> seq</td>
<td>flatten <code>[[a0;a1]; [a2;a3]]</code> = <code>[a0;a1;a2;a3]</code></td>
<td>n</td>
<td>log n</td>
</tr>
<tr>
<td><code>repeat</code> (x: <code>α</code>) (n: <code>int</code>) : <code>α</code> seq</td>
<td>repeat x 4 = <code>[x;x;x;x]</code></td>
<td>n</td>
<td>1</td>
</tr>
<tr>
<td><code>zip</code> : <code>α</code> seq * <code>β</code> seq -&gt; <code>(α * </code>β`) seq</td>
<td><code>zip</code> <code>[a0;a1] [b0;b1;b2]</code> = <code>[(a0,b0);(a1,b1)]</code></td>
<td>n</td>
<td>1</td>
</tr>
<tr>
<td><code>split</code> : <code>α</code> seq -&gt; <code>int</code> -&gt; <code>α</code> seq * <code>α</code> seq</td>
<td><code>split</code> <code>[a0;a1;a2;a3]</code> 1 = <code>([a0],[a1;a2;a3])</code></td>
<td>1 ?</td>
<td>1</td>
</tr>
<tr>
<td><code>scan</code> : <code>(α-&gt;α-&gt;α)</code> -&gt; <code>α</code> -&gt; <code>α</code> seq -&gt; <code>α</code> seq</td>
<td><code>scan f b</code> <code>[a0;a1;a2;...]</code> = <code>[f b a0; f (f b a0) a1; f (f (f b a0) a1) a2; ...]</code></td>
<td>n</td>
<td>log n</td>
</tr>
</tbody>
</table>
IMPLEMENTATION OF PARALLEL SEQUENCES
Processes

Processes separate address spaces (shared readable data, no write-sharing)

communication channel (pipe)

process 1

process 2

separate address spaces
(shared readable data, no write-sharing)
**Need-to-know Info**

- Processes are managed by your operating system
- Share time executing on available cores
- Processes have separate address spaces so communication occurs by:
  - serializing data (converting complex data to a sequence of bits)
  - writing data to a buffer
  - reading data out of the buffer on the other side
  - deserializing the data
- Cost is relative to the amount of data transferred
  - minimizing data transfers is an important performance consideration
Unix (Linux) pipe(), fork(), exec()

(Standard Unix, C-language calling sequences)

```c
int pipe(int fd[2]);
```

(now can read from file-descriptor \(fd[0]\), write to \(fd[1]\))

```c
int fork(void)
```

creates a new OS process;

in child, returns 0; in parent, returns process id of child.)

```c
int execve(char *filename, char *argv[], char *envp[])
```

(overwrite this process with a new execution of filename(argv);

if execve returns at all, then it must have failed)
Typical use of pipe, fork, exec

What you write at the shell prompt

```bash
cat foo | grep abc
```

What the shell does (simplified)

```c
int fd[2]; int pid1, pid2;
pipe (fd);
pid1 = fork();
if (pid1) { /* in the parent */
    close(fd[0]); close(1); dup2(fd[1],1); close(fd[1]);
    exec("/bin/cat", "foo");
} else { /* in the child */
    close(fd[1]); close(0); dup2(fd[0],0); close(fd[0]);
    exec("/bin/grep", "abc")
}
```

One learns this in COS 217

fd 0 – standard in
fd 1 – standard out
Typical use of pipe, fork, exec

What you write at the shell prompt

cat foo | grep abc

What the shell does (simplified)

```c
int fd[2]; int pid1, pid2;
pipe (fd);
pid1 = fork();
if (pid1) { /* in the parent */
    close(fd[0]); close(1); dup2(fd[0], 1);
    exec("/bin/cat", "foo");
} else { /* in the child */
    close(fd[1]); close(0); dup2(fd[1], 0);
    exec("/bin/grep", "abc")
}
```

One learns this in COS 217

pipe is a beautiful functional abstraction, isn't it?

It hides all this garbage so I don't have to think about it!!
create a child process using `fork : unit -> int`

- creates two processes; identical except for the return value of fork()

**Parent process**

```ocaml
let x = fork () in
```

**Child process**

```ocaml
let x = fork () in
```

**Standard use:**

```
match fork () with
| 0 -> ... child process code ...
| pid -> ... parent process code ...
```

copies of data are made when either parent or child writes to the data
Interprocess Communication via Pipes

- A pipe is a first-in, first-out queue
- Data (a sequence of bytes) may be written on one end of the pipe and read out the other
  - writes block after the underlying buffer is filled but not yet read
  - reads block until data appears to be read
  - bad idea to read and write the same pipe in the same process!

- Creating a pipe:
  - pipe : unit -> file descr * file descr
**Futures via Processes**

Future interface:

- Type: 'a future
- Value:
  - `future : ('a -> 'b) -> 'a -> 'b future`
  - `force : 'a future -> 'a`

Future `f x` runs `f x` in a child process.

Result of `f x` serialized and sent through a pipe back to the parent.

Child process:

- `f x`

Parent process:

- `force future`
Futures via Processes

future interface

```plaintext
type 'a future
val future : ('a -> 'b) -> 'a -> 'b future
val force : 'a future -> 'a
```

type 'a future = {
  fd : file_descr;
  pid: int
}

- pipe endpoint read by parent
- process id of the child
Futures via Processes

future interface

```ocaml
type 'a future
val future : ('a -> 'b) -> 'a -> 'b future
val force : 'a future -> 'a
```

type 'a future = {
  fd : file_descr;
  pid: int
}

```ocaml
let future (f: 'a -> 'b) (x: 'a) : 'b future =
  let (fin, fout) = pipe () in
  match fork () with
  | 0 -> (close fin;
      let oc = out_channel_of_descr fout in
      Marshal.to_channel oc (f x) [Marshal.Closures];
      Pervasives.exit 0 )
  | cid -> (close fout;
      {fd=fin; pid=cid} )
```
Futures via Processes

future interface

type 'a future
val future : ('a -> 'b) -> 'a -> 'b future
val force : 'a future -> 'a

type 'a future = {
  fd : file_descr;
  pid: int
}

let future (f: 'a -> 'b) (x: 'a): 'b future =
  let (fin, fout) = pipe () in
  match fork () with
  | 0 -> (close fin;
           let oc = out_channel_of_descr fout in
           Marshal.to_channel oc (f x) [Marshal.Closures];
           Pervasives.exit 0 )
  | cid -> (close fout;
            {fd=fin; pid=cid} )
Futures via Processes

future interface

type 'a future
val future : ('a -> 'b) -> 'a -> 'b future
val force : 'a future -> 'a

let future (f: 'a -> 'b) (x: 'a) : 'b future =
let (fin, fout) = pipe () in
match fork () with
| 0 -> (close fin;
let oc = out_channel_of_descr fout in
Marshal.to_channel oc (f x) [Marshal.Closures];
Pervasives.exit 0 )
| cid -> (close fout;
{fd=fin; pid=cid} )

type 'a future = {
  fd : file_descr;
pid: int
}
future interface

```ocaml
type 'a future
val future : ('a -> 'b) -> 'a -> 'b future
val force : 'a future -> 'a

let future (f: 'a -> 'b) (x: 'a) : 'b future =
  let (fin, fout) = pipe () in
  match fork () with
  | 0 -> (close fin;
    let oc = out_channel_of_descr fout in
    Marshal.to_channel oc (f x) [Marshal.Closures;
    Pervasives.exit 0 )
  | cid -> (close fout;
    {fd=fin; pid=cid} )
```

parent completes routine immediately;
keeping the future data structure around
to force later

```ocaml
type 'a future = {
  fd : file_descr;
  pid: int
}
```
Futures via Processes

future interface

type 'a future
val future : ('a -> 'b) -> 'a -> 'b future
val force : 'a future -> 'a

let future (f: 'a -> 'b) (x: 'a) : 'b future =
  let (fin, fout) = pipe () in
  match fork () with
  | 0 -> (close fin;
              let oc = out_channel_of_descr fout in
              Marshal.to_channel oc (f x) [Marshal.Closures];
              Pervasives.exit 0 )
  | cid -> (close fout;
              {fd=fin; pid=cid} )

child executes the future function

type 'a future = {
f d : file_descr;
   pid: int
}
Futures via Processes

future interface

```ocaml
type 'a future
val future : ('a -> 'b) -> 'a -> 'b future
val force : 'a future -> 'a
```

```ocaml
let future (f: 'a -> 'b) (x: 'a) : 'b future =
  let (fin, fout) = pipe () in
  match fork () with
  | 0 -> (
    close fin;
    let oc = out_channel_of_descr fout in
    Marshal.to_channel oc (f x) [Marshal.Closures];
    Pervasives.exit 0 )
  | cid -> (
    close fout;
    {fd=fin; pid=cid} )
```

then marshalls the results, sending them over the pipe

... and then terminates, its job complete
Module **Marshal**

```ocaml
module Marshal: sig .. end
```

Marshaling of data structures.

This module provides functions to encode arbitrary data structures as sequences of bytes, which can then be written on a file or sent over a pipe or network connection. The bytes can then be read back later, possibly in another process, and decoded back into a data structure. The format for the byte sequences is compatible across all machines for a given version of OCaml.

Warning: marshaling is currently not type-safe. The type of marshaled data is not transmitted along the value of the data, making it impossible to check that the data read back possesses the type expected by the context. In particular, the result type of the `Marshal.from_*` functions is given as `'_a`, but this is misleading: the returned OCaml value does not possess type `'_a` for all `'_a`; it has one, unique type which cannot be determined at compile-type. The programmer should explicitly give the expected type of the returned value, using the following syntax:

- `(Marshal.from_channel chan : type)`. Anything can happen at run-time if the object in the file does not belong to the given type.
Futures via Processes

future interface

```ocaml
type 'a future
val future : ('a -> 'b) -> 'a -> 'b future
val force : 'a future -> 'a
```

type 'a future = {
  fd : file_descr;
  pid: int
}

let force (f: 'a future) : 'a =
  let ic = in_channel_of_descr f.fd in
  let res = ((Marshals.from_channel ic) : 'a) in
  close f.fd;
  match waitpid [] f.pid with
  | (_,WEXITED 0) -> res
  | _ -> failwith "process failed to terminate in force"

reads the data from the future's pipe

closes the file descriptor
future interface

```ocaml
type 'a future
val future : ('a -> 'b) -> 'a -> 'b future
val force : 'a future -> 'a
```

```ocaml
let force (f: 'a future) : 'a =
  let ic = in_channel_of_descr f.fd in
  let res = ((Marshal.from_channel ic) : 'a) in
  close f.fd;
  match waitpid [] f.pid with
  | (_,WEXITED 0) -> res
  | _ -> failwith "process failed to terminate in force"
```

wait until child terminates; prevents "fork bomb" (other techniques could be used here)
Costs of “fork”

- Futures enable a rather simple communication pattern:

  ![Diagram showing parent and worker processes communicating]

- But the cost of starting up a process and communicating data back and forth is high

  Unix “fork” system call copies the entire address space into the child process. That includes all the closures and heap data structures in your entire program!

  Operating system does it *lazily*, using virtual-memory paging.

- That means this pattern: if (fork()) {parent...} else {exec();} does not pay a price, does no copying

But the pattern on the previous slides has no “exec();” call.
Another problem with “fork”

```ocaml
let future (f: 'a -> 'b) (x: 'a) : 'b future =
let (fin, fout) = pipe () in
match fork () with
| 0 -> (close fin;
    let oc = out_channel_of_descr fout in
    Marshal.to_channel oc (f x) [Marshal.Closures];
    Pervasives.exit 0 )
| cid -> (close fout; {fd=fin; pid=cid} )
```

Parent process and child process must share memory!

- This is possible on two different cores of the same multicore chip
- Sometimes possible with two chips on the same circuit board.
- **Not scalable to massive parallelism in the data center!**
Message Passing

- Futures enable a rather simple communication pattern:

  ![Diagram of parent and worker processes with message passing](image)

  But the cost of starting up a process and communicating data back and forth is high

- Instead: spawn 1 worker and have it do many tasks
  - (the implementation of futures could be optimized to reuse 1 process for many futures)
Message Passing

- Instead: spawn 1 worker and have it do many tasks
  - (the implementation of futures could be optimized to reuse 1 process for many futures)

Also: when creating the worker (with “fork”), don’t send data at the same time! No need to share memory; the “fork” can be remote on another machine (in the data center).
More remarks about

ASSIGNMENT 7
module type S = sig
  type 'a t
  val tabulate : (int -> 'a) -> int -> 'a t
  val seq_of_array : 'a array -> 'a t
  val array_of_seq : 'a t -> 'a array
  val iter : ('a -> unit) -> 'a t -> unit
  val length : 'a t -> int
  val empty : unit -> 'a t
  val cons : 'a -> 'a t -> 'a t
  val singleton : 'a -> 'a t
  val append : 'a t -> 'a t -> 'a t
  val nth : 'a t -> int -> 'a
  val map : ('a -> 'b) -> 'a t -> 'b t
  val map_reduce : ('a -> 'b -> 'b) -> ('b -> 'b) -> 'a t -> 'b
  val reduce : ('a -> 'a -> 'a) -> 'a t -> 'a
  val flatten : 'a t t -> 'a t
  val repeat : 'a -> int -> 'a t
  val zip : ('a t * 'b t) -> ('a * 'b) t
  val split : 'a t -> int -> 'a t
  val scan : ('a -> 'a -> 'a) -> 'a -> 'a t
end

module ArraySeq : S = struct
  type 'a t = 'a array
  let length = Array.length
  let empty () = Array.init 0 (fun _ -> raise (Invalid_argument ""))
  let singleton x = Array.make 1 x
  let append = Array.append
  let cons (x:'a) (s:'a t) = append (singleton x) s
  let tabulate f n = Array.init n f
  let nth = Array.get
  let map = Array.map
  let map_reduce = ...
  let reduce = ...
  let flatten = ...
  let repeat = ...
  let zip = ...
  let split = ...
  let scan = ...
  end
Work/Span estimation

module type S = sig
  type 'a t
  val tabulate : (int -> 'a) -> int -> 'a t
  val seq_of_array : 'a array -> 'a t
  val array_of_seq : 'a t -> 'a array
  val iter : ('a -> unit) -> 'a t -> unit
  val length : 'a t -> int
  val empty : unit -> 'a t
  val cons : 'a -> 'a t -> 'a t
  val singleton : 'a -> 'a t
  val append : 'a t -> 'a t -> 'a t
  val nth : 'a t -> int -> 'a
  val map : ('a -> 'b) -> 'a t -> 'b t
  val map_reduce : ('a -> 'b) -> ('b -> 'b -> 'b) -> 'b -> 'a t -> 'b
  val reduce : ('a -> 'a -> 'a) -> ('a -> 'a) -> 'a t
  val flatten : 'a t t -> 'a t
  val repeat : 'a -> int -> 'a t
  val zip : ('a t * 'b t) -> ('a * 'b) t
  val split : 'a t -> int -> 'a t * 'a t
  val scan : ('a -> 'a -> 'a) -> ('a -> 'a) -> 'a t -> 'a t
end

module Accounting (M: S) : SCount =
  struct
  let work = ref 0
  let span = ref 0
  let reporting name f x = ...
  module SM = struct
    type 'a t = 'a M.t
    let tabulate f n = (cost n 1;
      let s = !span in
      let smax = ref s in
      let z = M.tabulate (fun x -> let y = f x in
        smax := max (!smax) (!span);
        span := s; y) n
      in span := !smax; z)
    let length a = (cost 1 1; M.length a)
    let append a b = (cost (M.length a + M.length b) 1;
      M.append a b)
    ...
  end
end
How to use it

Open Sequence
module A = Accounting(ArraySeq)
module M = A.SM

let s1 = M.seq_of_array [|1;2;3;4;5|]
let f (s: int M.seq) = M.map (fun i -> i+1) s
let s2 = A.reporting "test1" f s1
let r = Array.to_list (M.array_of_seq s2)

(* Prints: *)

test1 work=5 span=1

r : int list = [2;3;4;5;6]
### map-reduce API for Assignment 7

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Work</th>
<th>Span</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tabulate (f: int-&gt;α) (n: int) : α seq</code></td>
<td>Create seq of length n, element i holds f(i)</td>
<td>n</td>
<td>1</td>
</tr>
<tr>
<td><code>seq_of_array: α array -&gt; α seq</code></td>
<td>Create a sequence from an array</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><code>array_of_seq: α seq -&gt; α array</code></td>
<td>Create an array from a sequence</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><code>iter (f: α -&gt; unit): α seq -&gt; unit</code></td>
<td>Applying f on each element in order.</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td><code>length: α seq -&gt; int</code></td>
<td>Return the length of the sequence</td>
<td>1</td>
<td>n</td>
</tr>
<tr>
<td><code>empty: unit -&gt; α seq</code></td>
<td>Return the empty sequence</td>
<td>1</td>
<td>n</td>
</tr>
<tr>
<td><code>cons: α -&gt; α seq -&gt; α seq</code></td>
<td>cons a new element on the beginning</td>
<td>n</td>
<td>1</td>
</tr>
<tr>
<td><code>singleton: α -&gt; α seq</code></td>
<td>Return the sequence with a single element</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><code>append: α seq -&gt; α seq -&gt; α seq</code></td>
<td>(nondestructively) concatenate two sequences</td>
<td>m+n</td>
<td>1</td>
</tr>
<tr>
<td><code>nth: α seq -&gt; int -&gt; α</code></td>
<td>Get the nth value in the sequence. Indexing is zero-based.</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><code>map (f: α -&gt; 'b) -&gt; α seq -&gt; α seq</code></td>
<td>Map the function f over a sequence</td>
<td>n</td>
<td>1</td>
</tr>
<tr>
<td><code>reduce (f: α -&gt; α -&gt; α) (base: α): α seq -&gt; α</code></td>
<td>Fold a function f over the sequence. f must be associative, and base must be the unit for f.</td>
<td>n</td>
<td>log n</td>
</tr>
<tr>
<td><code>mapreduce: (α-&gt;β)-&gt;(β-&gt;β-&gt;β)-&gt; β -&gt; α seq -&gt; β</code></td>
<td>Combine the map and reduce functions.</td>
<td>n</td>
<td>log n</td>
</tr>
<tr>
<td><code>flatten: α seq seq -&gt; α seq</code></td>
<td>flatten <code>[[a0;a1]; [a2;a3]] = [a0;a1;a2;a3]</code></td>
<td>n</td>
<td>log n</td>
</tr>
<tr>
<td><code>repeat (x: α) (n: int) : α seq</code></td>
<td>repeat <code>x 4 = [x;x;x;x]</code></td>
<td>n</td>
<td>1</td>
</tr>
<tr>
<td><code>zip: (α seq * β seq) -&gt; (α * β) seq</code></td>
<td>zip <code>[a0;a1] [b0;b1;b2] = [(a0,b0);(a1,b1)]</code></td>
<td>n</td>
<td>1</td>
</tr>
<tr>
<td><code>split: α seq -&gt; int -&gt; α seq * α seq</code></td>
<td>split <code>[a0;a1;a2;a3] 1= ([a0],[a1;a2;a3])</code></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><code>scan: (α-&gt;α-&gt;α) -&gt; α -&gt; α seq -&gt; α seq</code></td>
<td>scan <code>f b [a0;a1;a2;...] = [f b a0; f (f b a0) a1; f (f (f b a0) a1) a2; ...]</code></td>
<td>n</td>
<td>log n</td>
</tr>
</tbody>
</table>
Discussion

How to use these operators to make an inverted index?

key: URL value: contents of web page (HTML) sequence of words

key: word value: sequence of (URL, position-in-seq) pairs