BST as Maps & Zippers

Zippers are defined for
- XML
- vectors
- seq

What about other structures?

{:left {:value 5 :left nil :right nil} :value 10 :right {:value 15 :left nil :right nil}}

Can define zippers on other types
Making New Zippers

(zipper branch? children make-node root)

branch?
   One argument - node
   Returns true if node can have children

children
   One argument - node
   Returns sequence of the node's children

make-node
   Two arguments - Existing node, seq of children
   Returns new node from the children

Root
   Root of the structure
Zipper for BST as a map

{:left {:value 5 :left nil :right nil} :value 10 :right nil}

branch?
  map?

children
  (defn tree->children
    [map]
    [(:value map) (:left map) (:right map)])

make-node
  (defn children->tree
    [sequence]
    {:value (first sequence)
     :left (second sequence)
     :right (last sequence)"

Order has to match that in tree->children
Using the Zipper

(def map-tree {:left {:value 5 :left nil :right nil} :value 10 :right {:value 15 :left nil :right nil}})

(def map-zipper (zip/zipper map? tree->children children->tree map-tree))

(-> map-zipper
   zip/down
   zip/right
   zip/node) {value 5, :left nil, :right nil}
Doing insert in BST as map

(defn bst-map-insert
  [tree x]
  (bst-zipper-insert
   (bst-zipper-insert
     (zip/zipper map? tree->children children->tree tree)
     x)))
Notice the repeat

(zip/zipper map? tree->children children->tree tree)

Once we figure out the needed functions would like to forget about it

(defn bst-map-zipper
  [tree-map]
  (zip/zipper map? tree->children children->tree tree tree-map)
Shorter Way - partial

(defn bst-map-zipper (partial zip/zipper map? tree->children children children->tree)

(partial f arg1 arg2 … argk)

f - function with n > k arguments
arg1 arg2 … argk - first k arguments of f
Return function that needs n - k arguments
Examples

```clojure
(def hundred-times (partial * 100))
(hundred-times 5) 500

(hundred-times 5 4) 2000

(reduce + (take-while (partial > 1000) (iterate inc 0))) 499500

(def to-english (partial clojure.pprint/cl-format nil "~@(~@[~R~]~^ ~A.~)"))

(to-english 123456)

"One hundred twenty-three thousand, four hundred fifty-six"
```
Currying

Currying
Multi-argument function -> chain of single-argument functions

adder(a, b c) {a + b + c;}

addA = adder.curry();
addB = addA(2);
addC = addB(3);
answer = addC(4);
Manipulating Functions
juxt

Combines a set of functions
Returns vector applying each function to input

```
(def basic-math (juxt + - * /))
(basic-math 2 5)                    [7 -3 10 2/5]
```

```
(def split-collection (juxt take drop))
(split-collection 4 (range 9))      [(0 1 2 3) (4 5 6 7 8)]
```
juxt in Sorting

((juxt :last :first) {:last "Adams" :first "Zak"})

["Adams" "Zak"]

(sort-by (juxt :last :first) [{:last "Adams" :first "Zak"}
   {:last "Zen" :first "Alan"}
   {:last "Smith" :first "Alan"}])

({:last "Adams", :first "Zak"}
 {:last "Smith", :first "Alan"}
 {:last "Zen", :first "Alan"})

(sort-by (juxt :first :last) [{:last "Adams" :first "Zak"}
   {:last "Zen" :first "Alan"}
   {:last "Smith" :first "Alan"}])

({:last "Smith", :first "Alan"}
 {:last "Zen", :first "Alan"}
 {:last "Adams", :first "Zak"})
comp

Takes a sequence of functions
Composes the functions

((comp str +) 8 8 8)  
"24"

(def fourth (comp first rest rest rest))

(fourth [:a :b :c :d :e])  
:d
Given $n$ can we produce

$$(\text{comp first rest rest rest ... rest})$$

where we have $n - 1$ rest's?
Yes We Can!

(defn fnth
  [n]
  (apply comp
    (cons first
      (take (dec n) (repeat rest))))

((fnth 1) [:a :b :c :d :e])    :a
((fnth 3) [:a :b :c :d :e])    :c
How does this work?

(repeat rest)          infinite lazy sequence of rest

(take (dec n) (repeat rest))  '(rest rest ... rest) ; n-1 rest's

(cons first
  (take (dec n) (repeat rest)))  '(first rest rest ... rest)

(apply comp
  (cons first
    (take (dec n) (repeat rest))))  (comp first rest rest ... rest)
memoize

(memoize f)

    Caches results of function f
    Uses cached value next time f is called with same arguments

(defn adder
    [x]
    (println "adder" x)
    (inc x))

(def adder-memoized (memoize adder))

(adder-memoized 1)   prints 1, returns 2
(adder-memoized 1)   returns 2
(adder-memoized 2)   prints 2, returns 3
(adder-memoized 1)   returns 2
memoize - Cache Size

Cache is a map

Contains return values for each different set of input arguments
Delay

Suspends execution of code until delay is dereferenced
Caches result
Second time dereferenced returns cached result
Thread safe

(def wait (delay (println "do it now") (+ 1 2)))

@wait          prints "do it now", returns 3
@wait          returns 3
realized?

Returns true if a value has been produced for a promise, delay, future or lazy sequence.

(def wait (delay (println "do it now") (+ 1 2)))

( realized? wait ) false
@wait prints "do it now", returns 3
( realized? wait ) true
@wait returns 3
Example - Proxy for Expensive Operation

(defn fetch-page
  [url]
  {:url url
   :contents (delay (slurp url))})

(def result (fetch-page "http://www.eli.sdsu.edu/index.html"))

(:contents result)  ; Delay@2fcc470c: :pending
(realized? (:contents result))     false
@(:contents result) "<!DOCTYPE html>
<html lang="en">
 ..."
@ and deref

@(:contents result)

(deref (:contents result))

They do the same thing
Future

Computes body on another thread
Use @ or deref to get answer
@, deref blocks until computation is done

(def long-calculation (future (apply + (range 1e8))))
@long-calculation
Future & Delay in ending program

When you end your program there will be a 1 minute delay if you used future

End your program with (shutdown-agents)

(def long-calculation (future (apply + (range 1e8))))

@long-calculation

(shutdown-agents)
(shutdown-agents) & REPL

(shutdown-agents) shuts down your REPL
deref with Timeout

(deref (future (Thread/sleep 5000) :done!))
1000
    :impatient!)
;= :impatient!
Future & Zipper Example

Count the number of items in RSS feed with given title

Get RSS xml from server
Parse the XML
Get the title of each article in the feed
Match titles of articles with title looking for
Count the matches
RSS

Rich Site Summary
Really Simple Syndication

XML document

Contains meta-data & content of website
blogs
news sites
<rss version="2.0">
  <channel>
    <title>RSS Title</title>
    <description>This is an example of an RSS feed</description>
    <link>http://www.example.com/main.html</link>
    <lastBuildDate>Mon, 06 Sep 2010 00:01:00 +0000</lastBuildDate>
    <pubDate>Sun, 06 Sep 2009 16:20:00 +0000</pubDate>
    <ttl>1800</ttl>
    <item>
      <title>Example entry</title>
      <description>Here is some text containing an interesting description.</description>
      <link>http://www.example.com/blog/post/1</link>
      <guid>7bd204c6-1655-4c27-aeee-53f933c5395f</guid>
      <pubDate>Sun, 06 Sep 2009 16:20:00 +0000</pubDate>
    </item>
  </channel>
</rss>
<?xml version="1.0" encoding="utf-8"?>
<rss version="2.0" xmlns:content="http://purl.org/rss/1.0/modules/content/"
  <channel>
    <title>CS596 Course Portal RSS feed</title>
    <link>http://bismarck.sdsu.edu/CoursePortal</link>
    <description>Changes and additions to the Course Portal</description>
    <item>
      <title>Grades</title>
      <description>Grades posted</description>
      <pubDate>Wed, 01 Oct 2014 20:06:33 +0000</pubDate>
      <link>http://bismarck.sdsu.edu/CoursePortal</link>
      <guid>http://bismarck.sdsu.edu/CoursePortal?news=4471</guid>
    </item>
  </channel>
</rss>
clojure.xml/parse

(parse s)
Parses and load source s
Source - File, InputStream, URI string
Returns tree of xml/element struct-map

(clojure.xml/parse "http://bismarck.sdsu.edu/CoursePortalFeed.rss?class=cs596")

{:tag :rss, :attrs
Using a Zipper

(ns basiclectures.basic-language.feed-example
  (:require (clojure [xml :as xml]))
  (:require (clojure [zip :as zip])))

(defn feed->zipper
  [uri-str]
  (->> (xml/parse uri-str)
       zip/xml-zip)

(feed->zipper "http://bismarck.sdsu.edu/CoursePortalFeed.rss?class=cs596")
### RSS verses Atom

<table>
<thead>
<tr>
<th>RSS 2.0</th>
<th>Atom</th>
</tr>
</thead>
<tbody>
<tr>
<td>channel</td>
<td>Top level tag</td>
</tr>
<tr>
<td>item</td>
<td>Tag for article</td>
</tr>
<tr>
<td>title</td>
<td>Tag for article title</td>
</tr>
</tbody>
</table>

Has one more nesting level

There are other differences but these are the ones that will affect us now
Normalizing both to Common structure

```clojure
(ns basiclectures.basic-language.feed-example
 (:require (clojure [xml :as xml]))
 (:require (clojure [zip :as zip])))

(defn is-atom?
  [feed]
  (= :feed (:tag (first feed)))))

(defn normalize
  [feed]
  (if (is-atom? feed)
      feed
      (zip/down feed)))
```
Getting all Articles from a Feed

(ns basiclectures.basic-language.feed-example
 (:require (clojure [xml :as xml]))
 (:require (clojure [zip :as zip])))

(defn feed-children
 [uri-str]
 (->> uri-str
 feed->zipper
 normalize
 zip/children
 (filter (comp #{:item :entry} :tag))))
Example

(feed-children "http://bismarck.sdsu.edu/CoursePortalFeed.rss?class=cs596")

{:tag :item, :attrs nil,  
:content [{:tag :title, :attrs nil,  
:content ["Grades"]}  
{:tag :description, :attrs nil, :content ["Grades posted"]}  
{:tag :pubDate, :attrs nil, :content ["Wed, 01 Oct 2014 20:06:33 +0000"]}  
{:tag :link, :attrs nil, :content ["http://bismarck.sdsu.edu/CoursePortal"]}  
Getting the title of single article

(defn title
  [entry]
  (some->> entry
    :content
    (some #(when (= :title (:tag %)) %))
    :content
    first))
(defn string-contains
[string pattern]
(let [lower-case-string (.toLowerCase string)
      lower-case-pattern (.toLowerCase pattern)
      index (.indexOf lower-case-string lower-case-pattern)]
  (> index -1 )))
Counting the matching articles

(defn count-text-task
  [extractor text feed]
  (->> (feed-children feed)
       (map extractor)
       (filter #(string-contains % text))
       count))

(def count-title-task (partial count-text-task title))

(count-title-task "Grade"
  "http://bismarck.sdsu.edu/CoursePortalFeed.rss?class=cs596")
Ebola news

(count-title-task
  "Ebola"
  "http://news.google.com/news?pz=1&cf=all&ned=us&hl=en&output=rss")

But there are a lot of different news feeds
Using future to fetch in parallel

(def news-feeds
  #{"http://news.yahoo.com/rss/
    "http://feeds.nbcnews.com/feeds/worldnews"

(let [results (for [feed news-feeds]
       (future (count-title-task "Ebola" feed)))]
  (reduce + (map deref results)))

19
(ns basiclectures.basic-language.feed-example
 (:require (clojure [xml :as xml]))
 (:require (clojure [zip :as zip])))

(defn feed->zipper
 [uri-str]
 (->> (xml/parse uri-str)
      zip/xml-zip))

(defn is-atom?
 [feed]
 (= :feed (:tag (first feed)))))

(defn normalize
 [feed]
 (if (is-atom? feed)
 feed
 (zip/down feed)))

(defn feed-children
 [uri-str]
 (->> uri-str
      feed->zipper
      normalize
      zip/children
      (filter (comp #{:item :entry} :tag))))

(defn title
 [entry]
 (some->> entry
      :content
      (some #(when (= :title (:tag %)) %))
      :content
      first))

(defn string-contains
 [string pattern]
 (let [lower-case-string (.toLowerCase string)
       lower-case-pattern (.toLowerCase pattern)
       index (.indexOf lower-case-string lower-case-pattern)
       (> index -1 )))

(defn count-text-task
 [extractor text feed]
 (->> (feed-children feed)
      (map extractor)
      (filter #(string-contains % text))
      count))

(def count-title-task (partial count-text-task title))

(def news-feeds #"http://news.yahoo.com/rss/"
   "http://feeds.nbcnews.com/feeds/worldnews"
   "http://news.google.com/news?pz=1&cf=all&ned=us&hl=en&output=rss")

(let [results (for [feed news-feeds]
                  (future (count-title-task "Ebola" feed)))]
  (reduce + (map deref results)))