Higher Order Functions

Higher Order Functions

Functional programming is powerful because functions can take other functions as input. We refer to these as "higher order functions."

These functions allow the programmer to transform entire data structures in one compound expression, while encapsulating the implementation details.

For example, Google's search algorithm uses `map` and `reduce` to process extremely large data sets in parallel using large-scale distributed systems.

Higher Order Functions

Clojure includes many higher order functions in the core language and, rather than explicit recursion, these form the building blocks of most programs.

You're probably already familiar with many functions we'll cover here as they're increasingly included in imperative languages.

Very basic recursive implementations of several are included to motivate understanding, but note these examples greatly simply the actual implementations.

Transducers

Higher order functions in Clojure are implemented as "transducers," a concept from stream processing.

What does this mean?

- Most common higher order functions are variadic: they can take anywhere from zero to an unlimited number of collections as input
- They can take compound expressions and anonymous functions as the input
- The output is lazy by default

- All of these properties reduce incidental complexity for the programmer...
- As well as unnecessary space complexity when transforming immutable data structures
- When zero collections are provided, they return just the transformation itself that can be passed to other functions or used to process wire data
- Working with transducers (and even writing your own) can be extremely powerful, but is rather advanced. We'll just stick to using built-in ones

Map

`map` applies a function to the elements of one or more collections in order and returns the results as a lazy sequence. If the collections differ in length, the input of longer ones is discarded.

```
(defn map
  [f coll]
  (if (empty? coll)
      nil
      (conj (f (first coll))
            (map f (rest coll)))))
```

Note: if one wants to take and return vectors, use `mapv` instead. To flatten result use `mapcat`.

Map

Examples:

```
(map inc [1 2 3 4 5]) => (2 3 4 5 6)
```

```
(map + [1 2 3] [4 5 6]) => (5 7 9)
```

```
(map #(str "Hello " % "!" )
     ["Ada" "Grace" "Barbara"])
=> ("Hello Ada!" "Hello Grace!" "Hello Barbara!")
```

```
(map #(vector (first %) (* 2 (second %)))
     {:a 1 :b 2 :c 3})
=> ([:a 2] [:b 4] [:c 6])
```

Reduce

Applies a function to one or more sequential arguments in a collection, building up the result.

- Called "fold" or "accumulate" in other languages
- For optimal performance, uses Java Iterators on collections that implement the Iterable interface
- Otherwise uses first/next recursion

```
[f coll)
(if (empty? coll)
    nil
    (f (first coll)
       (reduce f (rest coll)))))
```

Reduce

Examples:

```
(reduce + [1 2 3 4 5]) => 15
(reduce + 0 (range 10)) => 45
```

```
(reduce #(cons %2 %1) [1 2 3] [4 5 6])
=> (6 5 4 1 2 3)
```

```
(defn fact [x]
  (reduce *' (range 1 (inc x))))
```

Notice an initial value can be supplied as the second argument. To reduce associative coll, use `reduce-kv`.

Filter

Takes a predicate and a collection and returns a lazy sequence of the items for which the predicate returns true.

```
[pred coll]
(cond
  (empty? coll) nil
  (pred (first coll)) (conj (first coll)
                            (filter pred (rest coll)))
  :else (filter pred (rest coll))))
```

Note: to take and return vectors, use `filterv`.

Filter

Examples:

```
(filter even? (range 10))
=> (0 2 4 6 8)
```

```
(def #(not (empty? %)) [[1 2 3] [] [1] []])
=> ([1 2 3] [1])
```

```
(filter #(= (count %) 1) ["a" "b" "foo" "bar" ""])
=> ("a" "b")
```

```
(filter (comp #{2 3} last) {:x 1 :y 2 :z 3})
=> ([:y 2] [:z 3])
```

Apply

Passes the elements in a collection to a variadic function as arguments.

Examples:

```
(apply str [1 2 3]) => "123"
```

```
(apply max [1 2 3]) => 3
```

```
(apply + 1 2 '(3 4)) => 10
;;note: additional arguments before the collection
```

For

Called "list comprehension" in Haskell and Python. Syntax is same as set-builder notation.

- Takes a vector of bindings, followed by keyword modifiers (`:let`, `:when`, `:while`), and output form
- Iterates through all elements in the bindings according to the modifiers and returns a lazy sequence of the results

For

Examples:

```
(for [x ['a 'b 'c] y [1 2]] [x y])`
=> ([a 1] [a 2] [b 1] [b 2] [c 1] [c 2])
```

```
(for [x (range 1 6)
      :let [y (* x x)
            z (* x x x)]]
=> ([1 1 1] [2 4 8] [3 9 27] [4 16 64])
```

```
(for [x (range 3) y (range 3) :when (not= x y)] [x y])
=> ([0 1] [0 2] [1 0] [1 2] [2 0] [2 1])
(for [x (range 3) y (range 3) :while (not= x y)] [x y])
=> ([1 0] [2 0] [2 1])
```

DoSeq

Takes a vector of bindings and a body of expressions and repeatedly executes the body with the bindings as arguments, returning `nil`. Examples:

```
(doseq [x [-1 0 1]
        y [1  2 3]]
  (print (* x y)))
=> -1 -2 -3 0 0 0 1 2 3 nil
```

```
(doseq [x (range 6)
        :when (odd? x)
        :let [y (* x x)] ]
  (print [x y]))
=> [1 1] [3 9] [5 25] nil
```

Run!

Applies a function to each element in one collection (using reduce), generally for the purpose of side-effects.

- Referred to as "for-each" or "iterate" in other languages.

```
(defn run!
   [f coll]
   (if (empty? coll)
       nil
      (do
        (f (first coll))
        (for-each f (next coll)))))
```

Run!

Examples:

```
(run! println (range 5))
=> 0
   1
   2
   3
   4
```

```
;; drawing line segments to canvas in ClojureScript
(run!
  (fn [segment]
    (.lineTo ctx (start-segment segment))
    (.lineTo ctx (end-segment segment)))
  segment-list)
```