Data Structures: Part 2

The Sequence Abstraction

- The sequence abstraction presents a sequential view of a data structure
- All Clojure collections, and many data structures from Java, can be represented as sequences

Sequence functions

- Calling `seq` on a collection will generate a sequence in the form of a list:
  - `(seq [1 2 3]) => (1 2 3)`
- `cons` appends a value to a sequence:
  - `(cons 0 '(1 2 3)) => (0 1 2 3)`
- Similarly, `list*` appends multiple values to a sequence:
  - `(cons 0 0 '(1 2 3)) => (0 0 1 2 3)`

Note: These functions should be used with care as they convert all data structures to lists.

- `cons` is generally only used in place of the more general purpose `conj` when working with lazy-seqs, which we'll explain in a bit

Sequence functions

In addition to `conj` and `cons`, `first` and `next` form the core of the list processing paradigm.

- Called "car" and "cdr" in other Lisps
- First returns the first value of a sequence:
  - `(first [1 2 3]) => 1`
- Next returns the rest of the sequence:
  - `(next [1 2 3]) => [2 3]`

- Calling `next` on an empty sequence will return the empty list:
  - `(next []) => ()`
- `fnext` (or `second`), `ffirst`, and `nfirst` are compound functions of `first` and `next`
- `nth` provides indexed access to sequence, but in linear time:
  - `(nth [0 1 2 3 4] 3) => 2`

- Note: calling either `first` or `next` on an unordered sequence will return an error:
    - `(first {:foo 1, :bar 2}) => ...`
    - Instead, call `seq` on them first: `(first (seq {:foo 1, :bar 2})) => 1`

As we'll see in the next section, along with conditionals and recursion, one can use `first` and `next` to implement all higher order functions.

Lazy Sequences

Clojure also includes another kind of sequence: the "lazy-seq."

What is lazy (or non-strict) evaluation?

- Programming languages are interpreted in one of two ways: strict or lazy
- The most common type of strict evaluation is "call by value" whereas the most common type of lazy evaluation is "call by name" (we'll just use the terms "strict" and "lazy")

- Strict evaluation computes every part of an expression at once
- Lazy evaluation only computes them as they're needed by the process.
- Example of lazy evaluation: if-then-else
  - Depending on the outcome of the predicate, only one branch is ever evaluated
- Another example: dividing by zero
  - The following would trigger an error in a strict language, but not in a lazy one: `(* 0 (\ 1 0))`

Lazy Sequences

In Clojure lazy sequences can be generated by calling `lazy-seq` on any collection.

- Each element is wrapped in a "thunk": a function call that points to the next element
- Thunks become "realized" (i.e. evaluated) either when they're consumed by a function, or when the entire lazy-seq is converted to a strict sequence using `seq` or `sequence`
- The technical term for lazy sequences that haven't been realized is "weak head normal form" (WHNF)

Lazy Sequences

A common use case for lazy sequences is to produce infinite lists to pass as arguments to functions.

- `range` generates an infinite sequence of integers: `(range) => (0 1 2 3 4 5 ..)`
- `repeat` generates a lazy sequence of its argument: `(repeat "x") => ("x" "x" "x" "x" "x" ..)`

- `take` returns a lazy sequence of the first n items in a collection: `(take 5 (range)) => (0 1 2 3 4)`
- `drop` returns a lazy sequence of all *but* the first n items in a collection: `(drop 5 (range)) => (5 6 7 8 9 ..)`
- `concat` combines two collections into a lazy sequence: `(concat [1 2 3] [10 20 30]) => (1 2 3 10 20 30)`
- `rest` is the lazy version of `next`
  - Similar to using `cons` instead of `conj`, one should generally only use `rest` in order to preserve laziness

Note: empty sequences can either return `nil` if they're lazy or the empty list, `()`, if they're strict.

- This is major difference between `next` and `rest` as well as `sequence` and `seq`
- Can be a source of bugs when testing if a sequence is empty
- For this reason, it's generally best to just use `empty?`

Lazy Sequences

Lazy sequences as the output of functions:

- You'll also encounter lazy sequences as the output of higher order functions
- This can often be confusing when you really want the entire sequence output at once
- In this case, the output can be forced by wrapping the function in a call to `do`
- Similarly, when benchmarking code that uses lazy sequences it's crucial to realize them by calling `doall`

Key-Value Functions

Both maps and vectors can be used as associative structures.

Clojure provides a concise and consistent API for working with keyed values:

- `assoc` takes a key (or index for a vector) and a value and stores them in a collection:
  - `(assoc {} :year 2017) => {:year 2017}`
  - `(assoc [1 2 3] 0 "foo") => [foo 2 3]`

- `dissoc` takes a key and removes it from a collection:
  - `(dissoc {:year 2017 :month "may"} :may) => {:year 2017}`
- `get` is a faster way to access the value at a key (but *not* an index):
  - `(def cal {:year 2016 :month "june"})`
  - `(:month cal) => "june"`
  - `(get cal :month) => "june"`

- `update` takes a key and a function to apply to the matching value:
  - `(update cal :year inc) => {:year 2017, :month "june"}`
  - `(update [1 2 3] 0 * 2) => [2 2 3]`
- `assoc-in` and `update-in` work similarly, but take vectors of keys for working with nested collections:
  - `(assoc-in {:gregorian {:year 2017}} [:gregorian :month] "june")`
  - `=> {:gregorian {:year 2017, :month "june"}}`

- `merge` combines associative collections, using values from the last if keys overlap:
  - `(merge {:year 2016, :month "june"} {:year 2017, :day 3})`
  - `=> {:year 2017, :month "june", :day 3}`
- `merge-with` combines two associative collections by applying a function to all matching key-value pairs:
  - `(merge-with max {:year 2016, :month "june"} {:year 2017, :day 3})`
  - `=> {:year 2017, :month "june", :day 3}`

Mutation

Although Clojure emphasizes immutable state, mutation can occasionally be useful.

We'll cover two of the most common use cases.

Atoms

`atom` is used for mutable values where threadsafety is important.

- A common pattern using atoms is for global maps:
  - `(def store (atom {}))`
- Atoms can be turned into regular vars by dereferencing them:
  - `@store => {}`
  - `(deref store) => {}`

Atoms

- Atomic values are modified with `swap!`, which takes a function and an argument:
    - `(swap! store assoc :counter 0)`
    - `=> @atom => {:counter 0}`
    - `(swap! store update :counter inc)`
    - `=> @atom => {:counter 1}`

Transients

Transients are mutable versions of data structures created from persistent ones.

They have their own API and cannot be modified using functions for persistent collections.

Transients are most commonly used for better performance when a vector needs to be heavily modified.

Transients

- `transient` creates a new transient version of a collection
- `persistent!` creates a persistent collection from a transient
- Transients can be modified using `conj!`, `pop!`, `assoc!`, `dissoc!`, and `disj!`:

```
(loop [i 0 v (transient [])]
  (if (< i n)
    (recur (inc i) (conj! v i))
    (persistent! v))))
```

Transients

Note: creating a transient from a persistent collection or vice versa take linear time and thus should only be done once.