Clojure Reference Sheet

Everything is a list!

- ◇ Functions are first-class values; variables and functions share the same namespace —"Clojure is a Lisp-1 Language".
- ◇ To find out more about `name` execute `(doc name)`!

For these notes, I followed the delightful Clojure for the Brave and True; I suggest starting with ClojureScript Koans: An interactive question-answer approach where you type up (super small) code to progress ;-)

Primitive Data Structures

Use `def` to bind a name to a value —this' like CommonLisp's `setq`.

Clojure's data structures are all immutable; that cannot be changed in place.

Numbers Integers, floats, ratios: `(printf "%s and %s and %s" -93 1.2 1/5)`

Symbols Atomic literals, `'hello`. They're like numbers: *They are what they are!* Unlike strings, the idea is to treat these *literally* rather than operate on their 'contents'. They're *first-class names!*

Closely related, are **keywords**, such as `:hello`. In Clojure, these are *not* symbols since they always evaluate to themselves and cannot act as names.

It's a *semantic* difference: With `eval`, the symbol `'hello` will have whatever value the *name* `hello` has; whereas `:hello` yields `:hello` as its value.

Strings Enclosed in double-quotes; use `str` to catenate a bunch of lists.

Maps/Hashes/Dictionaries/property-lists These structures allow you to associate keys/words/properties with values/definitions. *Extremely versatile!*

- ◇ The empty map is `{}`;
- ◇ The map sending $\texttt{key}_i$ to $\texttt{value}_i$ is $\{\texttt{:key}_0\ \texttt{value}_0\ \ldots\ \texttt{:key}_n\ \texttt{value}_n\}$.

```
(def m {:name {:first "Bob" :last "Abe"} :age 10 :act +})
(m :age) ;; ⇒ 10
(:last (:name m)) ;; ⇒ "Abe"
(apply (get m :act) [1 2]) ;; ⇒ 3

;; Keys can be anything
({3 40} 3) ;; ⇒ 40
```

- ◇ If `m` is a map, use `(:key m)` to get the value associated with `:key`.
- ◇ Since maps are essentially functions, you can use function application syntax: `(m key)`.
- ◇ Trying to get the value associated to a key *not in* a map will yield `nil`.

  Three equivalent ways to get the value of `k` in map `m` if it is in `m`, otherwise return default value `d` —which is optional and defaults to `nil`.

  `(get m :k d)`, `(m :k d)`, `(:k m d)`

  In particular, observe that keywords `:k` act as functions that lookup a key in a map —symbols also act as look-up functions.
- ◇ Starting from the empty map, one can 'conjoin' new values: `(conj {k v} m)` is the new map that at key `k` yields `v` and at all other keys behaves like `m`.
- ◇ Alternatively, use $(\texttt{hash-map :key}_0\ \texttt{value}_0\ \ldots\ \texttt{:key}_n\ \texttt{value}_n)$ to make a map and `(get m key)` to get the value associated with key `key`.

Vectors (*Append friendly!*) These are like maps but numbers are used to access the values. E.g., $(\texttt{get}\ [\texttt{x}_0\ \texttt{x}_1\ \ldots\ \texttt{x}_n]\ \texttt{i}) = \texttt{x}_i$ for $\texttt{i} : \texttt{0..n}$.

- ◇ Vectors can also be created with $(\texttt{vector}\ \texttt{x}_0\ \texttt{x}_1\ \ldots\ \texttt{x}_n)$.
- ◇ Use `conj` to 'conjoin' a new element to the *end* of a vector: $(\texttt{conj}\ \texttt{x}_{n+1}\ [\texttt{x}_0\ \ldots\ \texttt{x}_n]) = [\texttt{x}_0\ \ldots\ \texttt{x}_n\ \texttt{x}_{n+1}]$.

Lists (*Prepend friendly!*) These are like vectors but `conj` adds elements to the *start* of a list.

- ◇ `get` doesn't work with lists; use $(\texttt{nth}\ '(\texttt{x}_0\ \ldots\ \texttt{x}_n)\ \texttt{i}) = \texttt{x}_i$.
- ◇ Lists can also be created with $(\texttt{list}\ \texttt{x}_0\ \texttt{x}_1\ \ldots\ \texttt{x}_n)$.

For lists, `conj` is also known as `cons` since it *constructs* lists by adding elements at the start. The operation to add elements at the end is known as *snoc*, the reverse reading of *cons*; as such, Clojure has both (cons) lists and snoc lists (vectors) as primitive data structures.

Sets Collections of unique values, $\#\{\texttt{x}_0\ \ldots\ \texttt{x}_n\}$ —duplication is ignored in that

`(conj xs x)` is just `xs` if `x` is already a member of set `x`.

- ◇ Sets can also be created with $(\texttt{hash-set}\ \texttt{x}_0\ \texttt{x}_1\ \ldots\ \texttt{x}_n)$.
- ◇ `(contains? xs x)` indicates whether `x` is an element of set `xs`.
- ◇ Sets are essentially maps where the values are just themselves: $\#\{\texttt{x}_0\ \ldots\ \texttt{x}_n\} = \{\texttt{x}_0\ \texttt{x}_0\ \ldots\ \texttt{x}_n\ \texttt{x}_n\}$; so you can use any of the three above ways to `get` values.

Unlike many other Lisps, in Clojure maps, vectors, lists, and sets can (*optionally*) have their arguments separated by *commas*! E.g., `(= '(1 2 3) '(1, 2, 3) '(1 2, 3))` is true.

Sequence & Collection Methods

All of Clojure's data structures —strings, vectors, maps, lists, and sets— take part in both the sequence abstraction and the collection abstraction.

The sequence abstraction focuses on the elements, whereas the collection abstraction focuses on the structure as a whole.

The Collection Abstraction

- ◇ `count` ⇒ How many elements does the collection have?
- ◇ `empty?` ⇒ Does it have any elements?
- ◇ `every?` ⇒ Does every element in the collection satisfy a given predicate?
- ◇ `into` ⇒ Insert all the elements from the second collection *into* the first.
  - ○ E.g., `(into #{} xs)` converts `xs` into a set, whereas `(into {:a 1} xs)` converts a list `xs` of *key-value vector pairs* into a map but `:a` will map to whatever it maps to in `xs`, if any, or 1 otherwise.

  This is like `concat` but it does a collection conversion: The result is the same collection type as the first argument.
- ◇ `(conj xs x)` ⇒ Insert `x` into the collection `xs`.
  - ○ For vectors, insert at the *end*.
  - ○ For lists, insert at the *beginning*.

- For sets, insert if it's not already in `xs`.
- For maps, `x` has the shape `{:key value}`, and we update the value of `:key` in `xs` to now be associated with `value`.

This is like `cons` but it *preserves* the collection type —whereas `cons` forces it to be a list: `(cons x xs) = (cons x (seq xs))` where `(seq xs) = (into '() xs)` is discussed below.

`(conj xs x) = (into {} (cons x xs))` for `xs` a map.

`conj` and `into` are essentially the same function, except one takes a varadic number of arguments whereas the other takes a formal sequence for the second argument:

$(\texttt{conj xs } y_0\ y_1\ \ldots\ y_n) = (\texttt{into xs (list } y_0\ y_1\ \ldots\ y_n))$

In summary, every collection can be formed from the empty collection — `'(), [], {}, #{}` — and using `into` to shove new elements *into* it —alternatively, *conjoining* new elements with `conj`. Interestingly, `(into '() xs)` reverses a list `xs` but does nothing to a vector `xs` —since `conj` prepends for lists and appends for vectors.

The Sequence Abstraction

Sequences implement `first`, `rest`, `cons` and so may use the sequence operations `map`, `reduce`, `filter`, `distinct`, `group-by`, ...

Using `first` and `rest` we can always obtain a list from any sequence type; the method to do so is called `seq` and it's characterised by: `(seq xs) = (cons (first xs) (seq (rest xs)))`. For instance, for maps, $(\texttt{seq \{key}_0\ \texttt{val}_0\ \ldots\ \texttt{key}_n\ \texttt{val}_n\texttt{\}}) = \texttt{'([key}_0\ \texttt{val}_0\texttt{]}\ \ldots\ \texttt{[key}_n\ \texttt{val}_n\texttt{])}$. Moreover, whenever a sequence is expected, `seq` is called —e.g., when `map` is called. This is why `map` always returns a list; e.g., `(map #(* 3 (second %)) {:a 1 :b 3 :c 5}) = '(3 9 15)`. Use `into` —discussed above— to convert to a different sequence type.

Below are a few examples *shown using lists*, but they work with the other data-structures too, such as hashmaps!

- `(range start end) = (list start (+ start 1) (+ start 2) ... (dec end))`.

  `start` may be omitted, defaulting to 0.
- $(\texttt{concat '(}x_0\ \ldots\ x_k\texttt{) '(}x_k\ x_{k+1}\ \ldots\ x_n\texttt{)) = '(}x_0\ \ldots\ x_n\texttt{)}$.
- `(some p xs) = (p` x_k`)` where x_k is the first element to satisfy predicate `p`; or nil otherwise. (*Linear Search*)

  `(some #(and (p %) %) xs)` = x_k is how to get the actual element that satisfies the predicate `p`, if any.
- `map` zips its lists arguments together along a given "zipping function" `f`.

  $(\texttt{(nth (map f xs}_0\ \texttt{xs}_1\ \ldots\ \texttt{xs}_n\texttt{) i)}$
  $= \texttt{(f (nth xs}_0\ \texttt{i) (nth xs}_1\ \texttt{i) ... (nth xs}_n\ \texttt{i))}$

  For `f` being `vector`, the `map` takes some lists ("rows") and produces a list of lists ("columns") —think "matrix transpose" or "tupling".

  Recall that `:key` act as functions extracting the values associated with key `:key`; whence, `(map :key ms)` projects the value of `:key` from each map in `ms`.
- `reduce` replaces every (implicit) `cons/conj` with a new binary operation.

  $(\texttt{(reduce } \oplus\ \texttt{e '(}x_0\ x_1\ \ldots\ x_n\texttt{)) = (}\oplus\ (\oplus\ (\oplus\ (\oplus\ \texttt{e }x_0)\ x_1)\ \ldots)\ x_n)$

  The initial value `e` is optional and may be omitted.
- $\texttt{take k '(}x_0\ x_1\ \ldots\ x_n\texttt{)) = '(}x_0\ x_1\ \ldots\ x_{k-1}\texttt{)}$
- $\texttt{drop k '(}x_0\ x_1\ \ldots\ x_n\texttt{)) = '(}x_k\ x_{k+1}\ \ldots\ x_n\texttt{)}$
- $\texttt{take-while p '(}x_0\ x_1\ \ldots\ x_n\texttt{)) = '(}x_0\ x_1\ \ldots\ x_{k-1}\texttt{)}$ *where* x_k is the first element to not satisfy the predicate `p`.
- $\texttt{drop-while p '(}x_0\ x_1\ \ldots\ x_n\texttt{)) = '(}x_k\ x_{k+1}\ \ldots\ x_n\texttt{)}$ *where* x_k is the first element to not satisfy the predicate `p`.
- Extensionality: `xs = (concat (take-while p xs) (drop-while p xs))`
- `(filter p xs)` is the *largest* subsequence of `xs` whose elements all satisfy predicate `p`; unlike `take-while`, it process *all* of `xs` rather than stopping at the first value that falsifies the predicate `p`.
- `sort` sorts a list in ascending order.
- `(sort-by f xs)` sorts the elements of `xs` according to the order '$\sqsubset$' defined by `x` $\sqsubset$ `y` $\equiv$ `f x < f y`. E.g., `(sort-by count xss)` sorts the sequence of sequences `xss` according to their length.

There is also *list comprehensions* `(for [x xs] body) = (map (fn [x] body) xs)`; more generally, these are "nested for-loops":

$$\begin{aligned}&(\texttt{for [}x_0\ \texttt{xs}_0\ \ldots\ x_n\ \texttt{xs}_n\ \texttt{:let }lc\ \texttt{:when }wc\texttt{] body})\\ =\ &(\texttt{map (fn [[}x_0\ x_1\ \ldots\ x_n\texttt{]] (let }lc\ \texttt{body))}\\ &\quad\texttt{(filter (fn [[}x_0\ x_1\ \ldots\ x_n\texttt{]] (let }lc\ \texttt{(when }wc\ \texttt{[}x_0\ \ldots\ x_n\texttt{])))}\\ &\qquad\texttt{(cartesian-product xs}_0\ \texttt{xs}_1\ \ldots\ \texttt{xs}_n\texttt{)))}\end{aligned}$$

Where a supposed `cartesian-product` function essentially behaves as $\texttt{(for [}x_0\ \texttt{xs}_0\ \ldots\ x_n\ \texttt{xs}_n\texttt{] [}x_0\ \ldots\ x_n\texttt{])}$ —i.e., it returns all vectors `vs` where $\texttt{(nth vs i)}$ is an element of $\texttt{xs}_i$.

Laziness: "I *thunk*, therefore I'm done!"

The `map` (and `filter`) function is *lazy*: The i^{th}-element of `(map f xs)` is computed *only* when it is actually needed. E.g., `(first (map f xs))` is `(f (first xs))` and so the rest of the map is not evaluated at all.

Likewise, `(def result (map f xs))` is evaluated nearly instantaneously regardless of how big `xs` may be: The `map` is computed as elements of `result` are accessed. If you try to access or "think" of `(nth result i)` then if it is already computed —i.e., we have already "thunk" it— then we return that value, otherwise, we compute it and return it.

More accurately, Clojure *chunks* its computations: When an element is requested, it will compute a few elements after it as well since you're likely to request them as well. From below, it can be seen that it computes the next 30 elements.

```
(def result (map #(do (Thread/sleep 100) (* 2 %)) (range 0 100)))
(time (nth result 1))  ;; ⇒ 2;  Elapsed time: 3270.204959 msecs
(time (nth result 30)) ;; ⇒ 60; Elapsed time: 0.051002 msecs (Neato!)
(time (nth result 33)) ;; ⇒ 66; Elapsed time: 3293.824322 msecs
```

Accessing the first element takes ~3200 milliseconds since Clojure prepared the next ~30 elements in case we want to access them next; e.g., in the next line we access the 30th element almost instantaneously. After that, we try to access an element not yet computed and it and the next ~30 after it are computed.

Note:

- ◇ `(Thread/sleep 1000)` ⇒ Sleep/pause for 1second
- ◇ `(time e)` ⇒ Evaluate `e` and return its value *along* with a print to standard output indicating how long it took to evaluate `e`.

```
(+ 2 (time (+ 1 2))) ;; ⇒ 5 (Elapsed time: 0.022374 msecs)
```

Warning: Unused = Unevaluated!

```
;; Since the map's result in unused, it is not evaluated!
(do (map print (range 0 100)) (println "bye")) ;; Prints' "bye" only
;; Use "mapv" which is eager!
(do (mapv print (range 0 100)) (println "bye")) ;; Prints' "01...100bye" only
```

Infinite sequences: `(repeat x)` is the infinite sequence that returns `x` at every index: `(nth (repeat x) i) = x` for any $i \geq 0$.

Evaluating `(repeat x)` will *take forever* since it's an infinite list; instead use `(take n (repeat x))` to get a finite list of length `n`.

Likewise, `(repeatedly f)` generates an infinite sequence from the nullary function `f`.

```
(take 3 (repeatedly (fn [] (rand-int 10)))) ;; ⇒ '(5 9 6)
(take 2 (repeat "n")) ;; ⇒ '("n" "n")
```

You can also use `lazy-seq` to treat a sequence lazily.

```
(defn evens
  ([]  (evens 0))
  ([n] (cons (* 2 n) (lazy-seq (evens (inc n))))))

(take 3 (evens)) ;; ⇒ '(0 2 4)
```

You can memoize a function `f` so that if you've already 'thunk' it at input `a` then anytime `(f a)` you get the value *immediately!*

```
(defn slow [x] (Thread/sleep 1000) x)
(def fast (memoize slow))
(time (fast 1)) ;; ⇒ "Think it!" Elapsed time: 1000.529943 msecs
(time (fast 1)) ;; ⇒ "Already thunk it!" Elapsed time: 0.068761 msecs
```

Conditionals

Booleans: `true` and `false`

- ◇ (Deep structural) equality: `(= x y)`.
- ◇ `nil` indicates *no value*.
  - ○ (*Warning!* Unlike other Lisps, `nil` ≠ `false` and `nil` ≠ `'()`!)
- ◇ Use `nil?` to check if a value is `nil`.
- ◇ Comparisons: As expected; e.g., `(<= x y)` denotes $x \leq y$.

Regarding Boolean operations —such as `and, or, if`— both `nil` and `false` denote *logically false values*; everything else denotes logical truth: `(boolean x)` returns true exactly when `x` is truthy —i.e., it converts things to Booleans.

- ◇ `or, and` returns the first truthy / fasley value if any, or the last value otherwise.
- ◇ `if` takes **at-most 3** arguments: `(if condition thenExpr optionalElseExpr)`
  - ○ If you want to perform **multiple expressions**, use `do` —this is `progn` in CommonLisp.
  - ○ Note: `(if x y) = (if x y nil)`; better: `(when c thenBlock) = (if c (do thenBlock))`.
  - ○ `(if xs ...)` means "if xs is non-fasley then ..." is akin to C style idioms on linked lists. E.g., `(if 9 2 4) = 2`.

Avoid nested if-then-else clauses by using a `cond` statement –a (lazy) generalisation of switch statements: It sequentially evaluates the expressions $\texttt{test}_i$ and performs only the action of the first true test; yielding `nil` when no tests are true. Below we use the keyword `:else` to *simulate* a 'default case'—indeed, any non-falsey value would have sufficed.

```
(cond
  test0 expr0
  test1 expr1
   .
   .
   .
  :else defaultExpr)) ;; optional
```

Replacing `cond` by `case x` results in an *exhaustive* case-analysis: If `x` is the literal expression $\texttt{test}_i$ then yield $\texttt{expr}_i$, if no match happens, crash —to use a default, just have a default value as the final expression, no need to precede it by anything. A hybrid of `cond` and `case` is `condp`.

Block of Code `do` and control flow `and, or`

Use the `do` function to treat multiple expressions as a single expression. E.g.,

```
(do (println "hello")
    (def x  (if (< 2 3) 'two-less-than-3))
    (println (format "%s" x))
    23) ;; Return value of the "do" block
```

This' like curly-braces in C or Java. The difference is that the last expression is considered the 'return value' of the block.

- ◇ Lazy conjunction and disjunction —`and, or`— can be thought of as *control flow* first and Boolean operations second:

  `(and` $\texttt{s}_0$ `...` $\texttt{s}_n$ `e)` ⇒ when all x_i are non-falsey, do e
  `(or` $\texttt{s}_0$ `...` $\texttt{s}_n$ `e)` ⇒ when no x_i is falsey, do e
- ◇ That is, `and` is the *maybe monad*: Perform multiple statements but stop when any of them fails, returns falsey.
- ◇ Likewise, `or` performs multiple statements until one of them succeeds, returns non-falsey:
- ◇ We can coerce a statement $\texttt{s}_i$ to returning non-falsey as so: `(do` $\texttt{s}_i$ `true)`. Likewise, coerce falsey by `(do` $\texttt{s}_i$ `nil)`.

Functions

Functions are (*unexceptional*) data:

```
;; Add numbers
(+ 1 2 3) ;; ⇒ 6

;; Apply a function to each number
(map inc [2 4 6]) ;; ⇒ (3 5 7)
```

Function invocation: `(f` $\texttt{x}_0$ $\texttt{x}_1$ `...` $\texttt{x}_n$`)`. E.g., `(+ 3 4)` or `(print "hello")`.

- Clojure is *strict*: In a function call, all arguments are evaluated **before** the function is executed. Besides function calls, there are *special forms* which are 'special' since they don't always evaluate all of their operands; e.g., `if`, `when`, and macros such as `def`. There are also 'lazy collection' types.
- Only prefix invocations means we can use `-,+,*` in *names* since `(f+*- a b)` is parsed as applying function `f+*-` to arguments `a` and `b`.
- Function definition:

```
(defn my-function
  "This functions performs task ..."      ;; documentation, optional
  [arg0 arg1 & more]                      ;; header, signature
  (str arg0 arg1 (nth more 3)))           ;; body, instructions
```

  - The **return value** of the function is the result of the last expression executed.
  - The documentation string may indicate the return type, among other things.

```
(doc my-function) ;; ⇒ See signature and docstring
```

  - In function definition, use `&` to make a function take *any extra* number of arguments; it must come at the end and it is treated as a list in the function body. This is known as a *rest parameter*.

```
(my-function "x" "y" "a" "b" "c" "d") ;; ⇒ "xyd"
```

Functions also support **arity overloading**, which act as a way to support default arguments.

```
(defn doit
  ([x y z]
    (+ x y z))
  ([x y]
    ;; default value for z
    (doit x y 666))
  ([x]
    ;; Completely different behaviour
    ;; when y,z omitted
    (* 7 x)))
```

```
(doit 1 2 3) ;; ⇒ 6
(doit 1 2)   ;; ⇒ 669
(doit 1)     ;; ⇒ 7
```

Instead of dispatching/choosing function definitional clauses according to arity, some language also allow dispatch according to argument type; Clojure goes further by *dispatching along the result of an arbitrary function of the arguments.* These functions are called *multimethods*; they are defined by `defmulti` which takes their name and the dispatch function, then each overloaded method is defined independently using `defmethod` which takes the multimethod name, the dispatch value, and the function body.

```
;; "doit" is an overloaded method, dispatched along
;; the ":kind" value of the 1st arg, being a hashmap,
;; AND along the length of the second argument, being a sequence.
;; That is, the dispatch value is a vector of length 2.
(defmulti  doit (fn [x y] [(:kind x) (count y)]))

;; Here are three boring implementations, along the dispatch value
(defmethod doit [:fruit  5] [x y] :one)
(defmethod doit [:fruit  4] [x y] :two)
(defmethod doit [:veggie 4] [x y] :three)

;; Optional default method when nothing matches
(defmethod doit :default [x y] x)

;; Example calls
(doit {:kind :fruit  :age 10} "hola!") ;; ⇒ :one
(doit {:kind :veggie :age 10} "hola!") ;; ⇒ the 1st arg
      ;; ⇒ Without the default method, this crashes!
```

Multimethods are another way to define ordinary functions, and so can be used wherever a function is used. Such functions could be defined using `condp` to dispatch along the appropriate implementation; but this is more difficult to maintain as more alternatives are added, whereas the `defmethod` approach is a nice *separation of concerns* —also known as open-closed principle.

You can also *destructure* parameters —lists and maps are discussed below.

```
(defn doit2
  [ [x y &zs] ;; The first argument is a list/vector consisting
              ;; of at least 2 values, say x and y, then
              ;; followed by the list/vector zs

    {w :w}    ;; The second argument is a map;
  ]           ;; Let w denote the value of key :w in the map.

  (+ y (or w 1))) ;; In-case :w is not in the map, use 1 as default.

(doit2 [1 2 3 4] {:v 5 :w 6}) ;; ⇒ 7
;; (doit2 [1] {:v 5 :w 6}) ;; ⇒ Error: First argument too short!
```

Anonymous functions: `(fn [arg0 ... argk] bodyHere)`; as usual, you can also destructure arguments and have rest parameters.

```
;; make then later invoke
(def my-f (fn [x y] (+ x y)))
(my-f 1 2) ;; ⇒ 3

;; make and immediately invoke
((fn [x y] (+ x y)) 1 2) ;; ⇒ 3
```

```
;; Super terse notation
(map #(* % 2) [1 2 3]) ;; ⇒ [2 4 6]
```

The anonymous function `#(...%i...)` is a function where `%i` refers to the i-th argument; `%` is equivalent to `%1` and `%&` refers to the rest parameter.

Keyword/named arguments = rest parameters + (hash)maps + destructuring.

```
(defn f [a & {:keys [b c d] :or {c 7 d 11}}] (list a b c d))
```

Huh? `f` has a required argument, `a`, followed by a rest parameter (`&`), that we destructure into a hashmap `{...}`, whose `:keys` *contain at least* `b, c, d` —`:or`, if the key `c` is not there then use 7 as *default*, and likewise use 11 as default for `d`.

```
;; (f)                    ⇒ Error, "a" is a required positional argument!
(f 2) ;;                  ⇒ '(2 nil 7 11)
(f 2 :d 4) ;;             ⇒ '(2 nil 7 4)
(f 2 :c 5 :b 3 :d 7) ;;   ⇒ '(2 3 5 7)
(f 2 :ignored 99 :whatever "ok") ;; ⇒ '(2 nil 7 11)
```

Higher order functions

- `apply` unpacks a sequence so it can be passed to a function that expects a rest parameter. For any $k \geq 0$,

(apply f x_0 x_1 ... x_{k-1} '(x_k x_{k+1} ... x_n)) = (f x_0 x_1 ... x_n)

- If `f` is a function of `n` arguments, then (partial f x_0 x_1 ... x_{k-1}) is the function `f` with it's first `k` arguments already given and the remaining `n - k` elements not yet given.

  (partial f x_0 x_1 ... x_{k-1}) = #(apply f x_0 x_1 ... x_{k-1} %&)

- (complement p) = #(not (apply p %&))
- (identity x) = x
- (comp f_1 f_2 ... f_n) = #(f_1 (f_2 (... (apply f_n %&)))); if $n = 0$, this is just `identity`.

Let and Loop

Clojure's data structures are immutable —one cannot change them in-place— with structural sharing and it has no assignment operator —one cannot associate a new value with a name without creating a new scope, say via `let`.

Let forms allow local names and allow destructuring as functions do.

```
(let [x 2
      [_ y] '(0 1 2 3)] ;; y is 1
  (+ x y)) ;; ⇒ 3
```

```
;; Let's add the first 10 numbers
(loop [i   0
       sum 0]
    (if (< i 10)
      ;; "update variables and continue"
      (recur (inc i) (+ sum i))
      ;; "break, returning this value"
       sum))
;; ⇒ 45
```

Loops are an in-place form of recursion: (loop [i start] ...(recur i')...) = (f start) where f is (fn [i] ...(f i')...). The loop, like the underlying anonymous function, can take any number of arguments/initial-bindings.

Quotes, Quasi-Quotes, and Unquotes

Quotes: 'x refers to the *name* rather than the *value* of x.

- This is superficially similar to pointers: Given int *x = ..., x is the name (address) whereas *x is the value.
- The quote simply forbids evaluation; it means *take it literally as you see it* rather than looking up the definition and evaluating.
- Note: 'x = (quote x).

Akin to English, quoting a word refers to the word and not what it denotes.

This lets us treat *code* as *data*! E.g., '(+ 1 2) evaluates to (+ 1 2), a function call, not the value 3! In general, (eval (quote e)) = e for any expressions e; e.g., '(+ 1 2) is *data* that when evaluated/executed/run yields 3: (eval '(+ 1 2)) ⇒ 3.

An English sentence is a list of words; if we want to make a sentence where some of the words are parameters, then we use a quasi-quote –it's like a quote, but allows us to evaluate data if we prefix it with a tilde " ~ ". It's usually the case that the quasi-quoted sentence happens to be a function call! In which case, we use `eval` which executes code that is in data form; i.e., is quoted.

(*Macros are essentially functions that return sentences, lists, which may happen to contain code.*)

```
;; Quotes / sentences / data
'(I am a sentence)
'(+ 1 (+ 1 1))

;; Executing data as code ;; ⇒ 3
(eval '(+ 1 (+ 1 1)))
(def name "Jasim")
```

```
;; Quasi-quotes: Sentences with a
;; computation, code, in them.
`(Hello ~name and welcome)
     ;; ⇒ '(Hello "Jasim" and welcome)
`(+ 1 ~(+ 1 1))  ;; ⇒ '(+ 1 2)
```

As the final example shows, Lisp treats data and code interchangeably. A language that uses the same structure to store data and code is called 'homoiconic'.

Macros

. . . when I have time. . . !