Forms

- Clojure code is composed of nested expressions, or *forms.*

- The simplest of forms evaluate to themselves.

Self-evaluating forms, or literals

=> 42

=> “Hello World!”

=> nil

Function Calls

- A list is denoted by a pair of parentheses.

- To call a function, write the function name at the beginning of a list followed by its arguments.

- The arguments of a function can be any Clojure form.

(<fn-name> <arg1> <arg2>)

Function Calls

```
=> (+ 1 2)
=> (+ 1
      (* 5 7))
=> (str “Hello” “ ” “World!”)
=> (println “Hello World!”)
=> (println (str “Hello” “ ” “World!”))
```

Prefix notation

- Eliminates precedence rules
- Supports an arbitrary number of operands easily
- Makes the syntax very consistent

Naming values with def

- To assign a name to the result of a form, use def.
- def is a *special form* — it is an important language primitive that does not follow the same evaluation rules as function calls.
- Clojure has only a few special forms — see https://clojure.org/reference/special_forms
- Names defined using def can be used in all subsequent expressions.

Naming values with def

```
=> (def pi 3.14159)
=> (def radius 10)
=> (* pi (* radius radius)) ;; similar to (* pi radius radius)
=> (def area (* pi (* radius radius)))
=> (println pi)
```

Function literals

- Functions in clojure are defined using the *fn* special form.
- (fn [<arguments…] <body>)
- fn evaluates to the function that was defined.
- Functions themselves are first-class objects in Clojure, and evaluate to themselves just like numbers and strings.
  - Interesting fact: All Clojure functions are instances of first-class Java classes under the hood. **+** is *clojure.core$_PLUS_*, for example)

Functions

```
=> (fn [a b] (+ a b))
=> ((fn [a b] (+ a b)) 2 3)
=> (def add (fn [a b] (+ a b)))
=> (add 2 3)
```

Defining functions with defn

- defn is syntactic sugar that allows for defining functions.

- defn is implemented as a macro — a special Clojure function that transforms Clojure code.

- (defn <optional docstring> [<arguments>] <body>)

Defining functions with defn

```
=> (defn square
#_=>   "Squares a number."
#_=>   [n]
#_=>   (* n n))
```

```
=> (square 6)
```

Conditionals

- Conditionals are defined using the *if* special form.
- (if <predicate> <consequent> <alternative>)
    - The if block is an expression (like mostly everything else). It evaluates to an appropriate value.
- nil and false represent logical falsehood. All other values are logically true.

Conditionals

```
=> (if (> 3 2)
#_=>   "greater"
#_=>   "lesser")
```

```
=> (if nil
#_=>   "it's true!"
#_=>   “it's false")
```

Side effects with do

- An expression has a side effect if it modifies a state or has some interaction with the outside world besides simply evaluating to a value.

- Ex: println

- *do* is a special form that evaluates all expressions in order and returns the result of the last one.

Side effects with do

```
  => (do
#_=>  (println "Welcome to IN/Clojure!")
#_=>  (* 4 3))
```

Local bindings with let

- To assign names to values locally, use let. (Analogy: scoped variables)
- (let [<name> <expression>…] <body>)
- let wraps its body in an implicit do block.

Local bindings with let

```
  => (let [radius 42]
#_=>   (println radius)
#_=>   (* 3.14 radius radius))

    => radius
```

```
  => (let [circle-area (fn [radius] (* 3.14 radius radius))]
#_=>   (circle-area 42))

    => (circle-area 69)
```

Vectors

- A vector is an ordered, indexed collection of values.
- [1 42 “baz” :quux]
  - Equivalent to (vector 1 42 “baz” :quux)
- Vector literals are denoted by square brackets.
- Vectors can be heterogeneous.

Vectors

=> (def colours [“red” “orange” green”])

=> (nth colours 1)

=> (conj colours “blue”)

Maps

- Maps are associative, unordered data structures.
- They map keys of any type to values of any type.
- {:foo “3” :bar 4}
  - Equivalent to (hash-map :foo “3” :bar 4)
- Although map keys can be of any type, keywords are most commonly used.

Sets

- Sets are collections of unique items, unordered.
- #{:foo “3” :bar 4} ;; A set with 4 elements. Note the #
  - Equivalent to (hash-set :foo “3” :bar 4)

Lists (again)

- A linked-list is created as follows
- (1 2 3 4) ;; Boom! No, doesn’t work. Why?
- ‘(1 2 3 4)
    - Equivalently (list 1 2 3 4)