Clojure

Clojure

- General purpose
- Lisp (List Processing)
- Functional
- Compiled
- Dynamically typed

Literals

```
(class "String")   ;=> java.lang.String
(class 123)        ;=> java.lang.Integer
(class true)       ;=> java.lang.Boolean
(class false)      ;=> java.lang.Boolean
(class :bar)       ;=> clojure.lang.Keyword
(class nil)        ;=> nil
(class 2147483648) ;=> java.lang.Long
(class #"regex")   ;=> java.util.regex.Pattern
(class 123M)       ;=> java.math.BigDecimal
(class 123N)       ;=> clojure.lang.BigInt
(class 42/43)      ;=> clojure.lang.Ratio
(class \c)         ;=> java.lang.Character
(class 'foo)       ;=> clojure.lang.Symbol
```

Collection literals

```
; List
'(3 2 1)

; Vector
[1 2 3]

; Set
#{1 2 3}

; Map
{1 "one", 2 "two", 3 "three"}
```

Collection literals

```
; List
'(3 2 1) -> (list 3 2 1)

; Vector
[1 2 3] -> (vector 1 2 3)

; Set
#{1 2 3} -> (hash-set 1 2 3)

; Map
{1 "one", 2 "two", 3 "three"} ->
      (hash-map 1 "one", 2 "two", 3 "three")
```

This is a list

```
'(+ 1 2)
```

This is a list

```
'(+ 1 2)
  ;=> (+ 1 2)
```

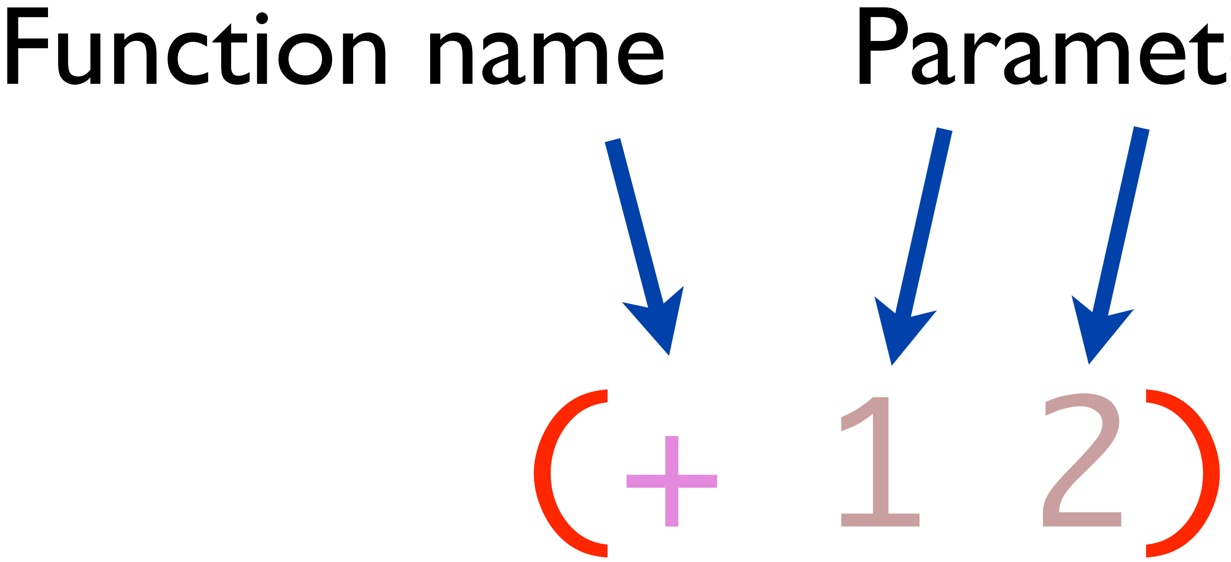
This is a list (and a form)

```
'(+ 1 2)
 ;=> (+ 1 2)
```

This is a form

```
(+ 1 2)
```

Prefix notation



Function name

Parameters

```
(+ 1 2)
```

This is a java.lang.String

```
"(+ 1 2)"
```

```
;=> "(+ 1 2)"
```

This is a java.lang.String
which can be converted to a form

```
(read-string "(+ 1 2)")
     ;=> (+ 1 2)
```

This is a java.lang.String
which can be converted to a form
which can be evaluated

```
(eval (read-string "(+ 1 2)"))
;=> 3
```

Read-compile-evaluate

1. Text is converted to forms by the reader
2. Forms are converted to byte code by compiler
3. Byte code is evaluated by the runtime

Functions

```
(fn [n] (* 2 n))
;=> #<core$eval376$fn__377 user$eval376$fn__377@5c76458f>
```

Functions

```
((fn [n] (* 2 n)) 4)
          ;=> 8
```

Functions

```
(fn [n] (* 2 n))
#(* 2 %)
#(* 2 %1)
```

Functions & Vars

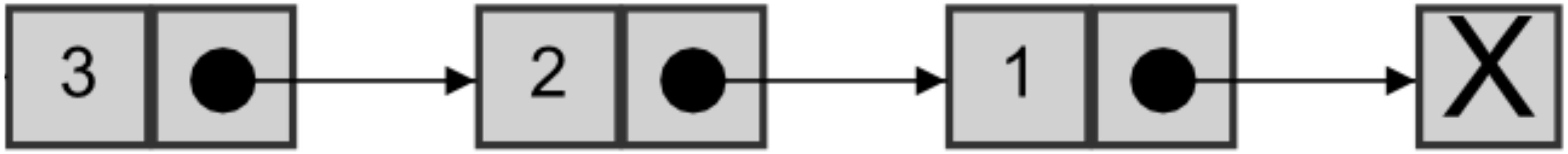
```
(def times-two
  (fn [n] (* 2 n)))
;=> #'user/times-two

(times-two 4)

;=> 8
```

Functions & Vars

```
(def times-two
  (fn [n] (* 2 n)))

(defn times-two
  [n] (* 2 n))
```

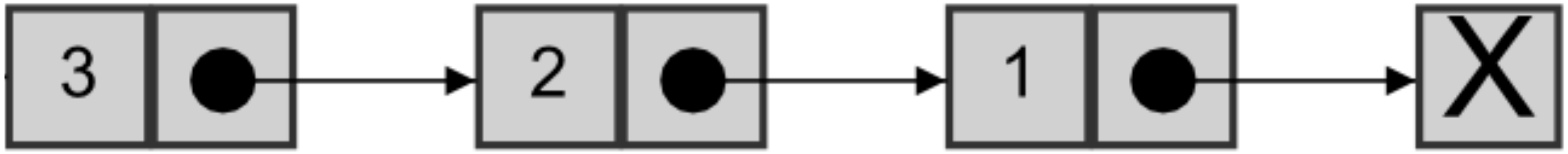
Immutable and persistent data structures

```
(def my-list '(3 2 1))
; => (3 2 1)
```



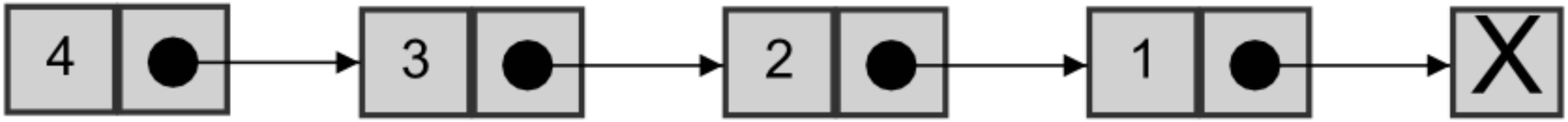
Immutable and persistent data structures

```
(def my-list '(3 2 1))
; => (3 2 1)

(def my-other-list (cons 4 my-list))
; => (4 3 2 1)
```



Immutable and persistent data structures

```
(def my-list '(3 2 1))
; => (3 2 1)

(def my-other-list (cons 4 my-list))
; => (4 3 2 1)
```



Immutable collections

```
; List
'(3 2 1)

; Vector
[1 2 3]

; Set
#{1 2 3}

; Map
{1 "one", 2 "two", 3 "three"}
```

"It is better to have 100 functions operate on one data structure than 10 functions on 10 data structures."

Alan Perlis

first

```
(first '(1 2 3))
;=> 1

(first [1 2 3])

;=> 1

(first #{1 2 3})
;=> 1

(first {:one "one" :two  "two"})
;=> [:one "one"]
```

rest

```
(rest '(1 2 3))
;=> (2 3)

(rest [1 2 3])
;=> (2 3)

(rest #{1 2 3})
;=> (2 3)

(rest {:one "one" :two  "two"})
;=> ([:two "two"])
```

rest

```
(rest '())
;=> ()

(rest [])
;=> ()

(rest #{})
;=> ()

(rest {})
;=> ()
```

collections are functions

```
('(1 2 3) 0)


([1 2 3] 0)
;=> 1

(#{3 2 1} 1)
;=> 1

({:one 1 :two 2 :three 3} :one)
;=> 1
```

some collections are functions

```
('(1 2 3) 0)
;=> java.lang.ClassCastException: clojure.lang.PersistentList cannot be cast to clojure.lang.IFn


([1 2 3] 0)
;=> 1

(#{3 2 1} 1)
;=> 1

({:one 1 :two 2 :three 3} :one)
;=> 1
```

contains?

```
(contains? '(10 11 12) 10)
;=> false

(contains? '[10 11 12] 1)
;=> true

(contains? #{3 2 1} 1)
;=> true

(contains? {:one 1 :two 2 :three 3} :one)
;=> true
```

.contains

```
(.contains '(10 11 12) 10)
;=> true

(.contains '[10 11 12] 1)
;=> false

(.contains #{3 2 1} 1)
;=> true

(.contains {:one 1 :two 2 :three 3} :one)
;=> java.lang.IllegalArgumentException: No matching method
found: contains for class clojure.lang.PersistentArrayMap
```

.contains

```
(.contains '(10 11 12) 10)
;=> true

(.contains '[10 11 12] 1)
;=> false

(.contains #{3 2 1} 1)
;=> true

(.containsKey {:one 1 :two 2 :three 3} :one)
;=> ClassCastException clojure.lang.PersistentArrayMap cannot be
cast to clojure.lang.PersistentHashMap
```

.contains

```
(.contains '(10 11 12) 10)
;=> true

(.contains '[10 11 12] 1)
;=> false

(.contains #{3 2 1} 1)
;=> true

(.containsKey large-map-with-one-as-key :one)
;=> true
```

Java interop

```
(new java.util.ArrayList)
;=> #<ArrayList []>

(java.util.ArrayList.)
;=> #<ArrayList []>
```

Java interop

```
(System/currentTimeMillis)
;=> 1318164613423

(.size (new java.util.ArrayList))
;=> 0

(. (new java.util.ArrayList) size)
;=> 0
```

Macros

```
(infix (1 + 2))
;=> 3


(defmacro infix [form]
  (list (second form) (first form) (first (nnext form))))
;=> #'user/infix
```

Read-compile-evaluate

1. Text is converted to forms by the reader
2. Forms are converted to byte code by compiler
3. If compiler finds macro, expand the macro and start over at 1.
4. Byte code is evaluated by the runtime

Macros

```
(defmacro infix [form]
  (list (second form) (first form) (first (nnext form))))
;=> #'user/infix


(defmacro infix [form]
  `(~(second form) ~(first form) ~@(nnext form)))
;=> #'user/infix

(macroexpand '(infix (1 + 2)))
; => (+ 1 2)
```

Macros

```
(defmacro infix [form]
  (cond (not (seq? form))
        form
        (= 1 (count form))
        `(r-infix ~(first form))
        :else
        (let [operator (second form)
              first-arg (first form)
              others (nnext form)]
          `(~operator
            (r-infix ~first-arg)
            (r-infix ~others))))))
```

Macros

- The two rules of the macro club
    1. *Don't write macros.*
    2. Only write macros *if that is the only way to encapsulate a pattern.*
    3. You can write any macro that makes life easier for your callers when compared with an equivalent function.

Programming Clojure - Stuart Halloway 2009

Tasks

- clojure-functional/src/test/clojure/no/knowit/clojure-functional/exercise_1_intro_tests.clj
- Solve tests one by one
- Remove '#_' to execute test