CS 596 Functional Programming and Design
Fall Semester, 2014
Doc 14 Some Review
Oct 21, 2014

Elements of Functional Programming

Pure Functions

First Class Functions

Higher-Order Functions

Immutability

Lazy Evaluation

Recursion

Currying

Memoization

Destructuring

Collection Pipelines

List Compressions

Basic Data Elements

symbols
keywords
literals
lists
vectors
maps
sets

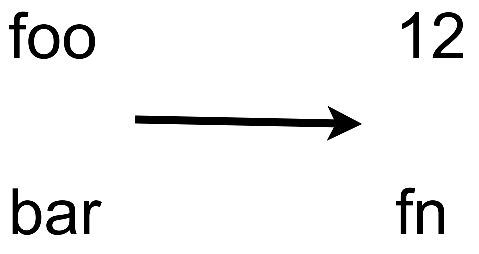
Symbols

Can reference another value

```
(def foo 12)

(defn bar [n] (inc n))
```

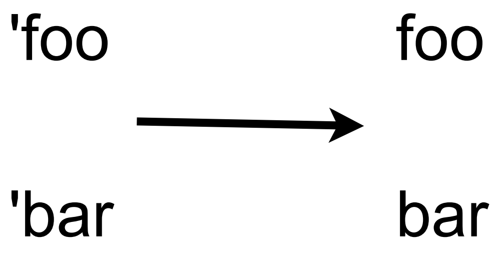
When evaluated returns the value



foo 12

bar fn

When quoted & evaluated
returns it self



'foo foo

'bar bar

Keywords

Like symbols but evaluates to itself

Literal syntax starts with a colon

```
:foobar
:2
:?
:ThisIsALongKeyWordWhichShowsThatTheCanBeLong
```

Colon is part of literal syntax, but not the name of the keyword

```
(= :cat (keyword "cat"))          true
(= :cat (keyword ":cat"))         false
```

Collections

Immutable

Heterogeneous

Persistent

Vectors

Sets

Maps

Lists

Queues

Vectors

```
(vector 8 4 2)                      [8 4 2]

(nth [:a :b :c] 2)                  :c

(get ["a" "b" "c"] 2)               "c"

(["a" "b" "c"] 2)                   "c"

(nth [:a :b :c] 2 "rat")            :c

(nth [:a :b :c] 4 "rat")            "rat"

(.indexOf ["a" "b" "c"] "b")        1

(peek ["a" "b" "c"])                "c"

(pop ["a" "b" "c"])                 ["a" "b"]

(conj [1 2 3]  4)                   [1 2 3 4]
(assoc [1 2 3] 0  9)                [9 2 3]
```

Immutability & Persistence

```
(def a  [1 2 3])

(def b  (conj a 4))

(def c  (assoc b 0 8))
```

a ⟷ [1 2 3]

b ⟷ [1 2 3 4]

c ⟷ [8 2 3 4]

Java

```
int[] d = {1, 2, 3};

d[0] = 8;
```

d ⟷ {8, 2, 3}

Sets

No duplicates

Fast insert & contains

Sets

(contains? #{1 2} 1)	true
(#{2 4} 2)	2
(#{2 4} 3)	nil
(get #{1 2} 1)	1
(get #{1 2} 3)	nil
(get #{1 2} 3 :not-found)	:not-found
(nth #{4 2} 2)	2
(conj #{ 1 2 } 3 4 5)	#{1 2 3 4 5}
(disj #{1 2 3} 2)	#{1 3}
(clojure.set/intersection #{1 2 3} #{2 4 8})	#{2}

Maps (Hash Table)

Key-value map

Keys - any value

Values - any value

Fast insert & find

Very common

```
{:first-name "Roger"
 :last-name "Whitney" }


{:first-name "Roger",
 :last-name "Whitney" }


{:name {:first "Roger" :last "Whitney" }
 :phone-numbers
     ["111-2222" "222-3333"]}


{ "a" 1, 2 "b", [4 3] :me}


{  }
```

Maps (Hash Table)

(get {:a 1} :a)	1
({:a 1} :a)	1
(:a {:a 1})	1
({2 "b"} 2)	"b"
(2 {2 "b"})	Error
(conj {:a 1 :b 2} {:a 3} {:c 4})	{:c 4, :a 3, :b 2}
(merge {:a 1 :b 2} {:a 3 :c 4})	{:c 4, :a 3, :b 2}
(assoc {:a 1 :b 2} :a 3 :c 4)	{:c 4, :a 3, :b 2}

Naming Conventions

Clojure	Java
all-lower-case words-separated-by-hyphen	camelCase

Lists

Linked List

Fast insert & remove at front

```
'( 1 2 3)

'( "cat" {:a 1})

'(+ 1 2)
```

Explain This

```
(defn foo
  [n]
  "How does this work? Not a compile error."
  (if (> 5 n)
    (println "in if")
    (println "else"))
  "This is not a doc comment"
  (+ 10 n))
```

Short Syntax for Lambda

```
(fn [a b] (< (first a) (first b)))
        ↓
#(< (first %1) (first %2))
```

%n -> n'th argument

```
#(+ 2 %)
```

if only one argument can use %

Closure

function + reference to its environment

```
(defn adder
  [n]
  #(+ n %))

(def add-5 (adder 5))

(add-5 10)                    Returns 15
```

Rules for Lazy

Use lazy-seq at outermost level of lazy squence-producing expression

Use **rest** instead of **next** if consuming another sequece

Use higher-order functions when processing sequences

Don't hold on to the **head**

let

threading macros

Symbols, Values & Binding

Recursive Function verses Recursive Process

Private functions, Multiple arities

Tail Recursion

Variable Number of arguments

Truthiness

Lazy Evaluation

if, when, cond, assoc-in

map, reduce, Filter, apply, cons

Namespaces

Destructuring

pre & post conditions

comp, memoize, partial

future, delay

multifunctions

tests

immutability & persistence