Introduction to Functional Programming

Jean-Louis Giordano (@jellismymind)

<2016-11-22 Tue>

Contents


cljs
JS

1 What is FP?

2 What is a value?

A value is the final result of a computation.

The value of 1 + 1 is 2.

3 What is an expression?

A symbol or combination of symbols that represents a value or a relationship between values

1 + 1 is an expression, it reduces to the value 2.

2 is also an expression as well as a value.

4 What is evaluation?

Evaluation is the reduction of an expression to its value.

Evaluate: e- (ex-, out) + value "To extract the value"

Example of evaluation:

```
1 + 1 + 1
1 + 2
3
```

```
+------------+
| Expression |
+------------+
      |
      V
+------------+
| Evaluation |
+------------+
      |
      V
+------------+
|   Value    |
+------------+
```

5 What is a variable?

A variable is a reference to a value.

In:

```
var a = 2;
```

a is a variable, referencing the value 2

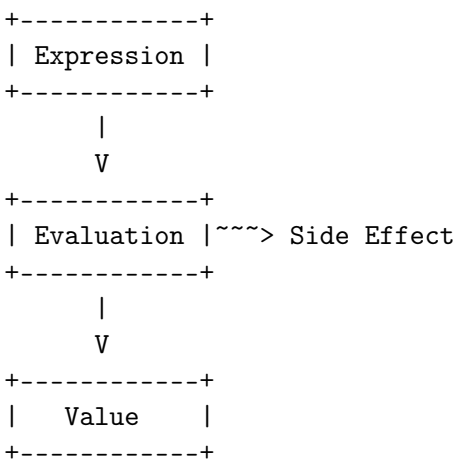
6 What are side-effects?

A side effect is a step in the evaluation of an expression that has effects outside of the expression itself.

Examples:

```
console.log("hello");

a = 1; a += 1;
```



7 Statement

A statement is an expression that evaluates to nothing.

7.1 In JavaScript:

Expression:

```
1 + 1;
(x) => x ** 2;
```

Statement:

```
var a = 1;
```

7.2 In ClojureScript and Elm:

Only expressions

```
(def a
  (if true "hello" "goodbye"))
```

```
a = if True
    then "hello"
    else "goodbye"
```

7.3 Why is that?

Statements require side effects, intrinsically imperative.

```
+------------+
| Statement  |
+------------+
      |
      V
+------------+
| Evaluation |~~~> Side Effect
+------------+
      |
      X
```

8 What is a function?

A function is an abstraction for an expression, where one or several values in the expression are replaced by variables.

Let's abstract the following expression:

```
1 + 1
```

```
inc = function (x) { return x + 1; };
```

```
inc = (x) => {return x + 1;};
inc = (x) => x + 1;
```

```
(+ 1 1)
(def inc (fn [x] (+ x 1)))
(defn inc [x] (+ x 1))
```

```
1 + 1
inc = \x -> x + 1
inc x = 1 + x
```

Question: Is a function a value?

9 What is the arity of a function?

The number of arguments a function takes is its arity.

```
// arity 0
zero = () => 0;

// arity 1
inc = (x) => x + 1;

// arity 2
add = (x, y) => x + y;

// infinite arity
countArgs = (...args) => args.length;
```

10 Routine vs Function

A routine is an abstraction that do not return a value.

```
a = (x) => {
  console.log(x);
}

b = (x) => {
  return x;
};
```

a is a routine, b is a function.

A procedure can either be a routine or a function.

11 What is a pure function?

A pure function is a side-effect free function that always maps a given input to the same output.

Which of the following is a pure function?

```
a = (x) => x + 1;

b = (x) => {
  console.log(x);
  return x;
};

c = (x) => x * Math.random();

d = (x) => x.push("hello");

e = (x) => {
  var result = [];
  while (x > 0) {
    result.unshift(x);
    x--;
  }
  return result;
};
```

12 Immutable vs Mutable

Immutable means that cannot change. Think "read only", "constants".

Persistent Datastructures are immutable, and can't be updated in-place.

13 Referencial Transparency

An expression that is deterministic and without side-effects is referencially transparent.

It means it can be replaced by its value without changing the behaviour of the program.

14 What is Application?

Calling a function with some arguments is applying that function to the value of those arguments.

Abstraction and Application are the core concepts of functional programming.

```
f(arg1, arg2);
1 + 2;

(f arg1 arg2)
(+ 1 2)

f arg1 arg2
1 + 2
(+) 1 2
```

15 Higher order function

Functions can return functions, and take functions as argument.

```
def apply (f, x, y):
  return f(x, y)

apply(add, 1, 2)

def incrementer (n):
  return lambda m: m + n

add2 = incrementer(2)
add2(4)
```

16 Partial Application

Take a function of arity n, and m $<$ n arguments, and return a function of arity n - m.

Example:

```
(+ 1 2 3)
((partial +) 1 2 3)
```

```
((partial + 1) 2 3)
((partial + 1 2) 3)
((partial + 1 2 3))

1 + 1
(+) 1 1
((+) 1) 1
-- Currying
```

17 Partial Application (bis)

```
(defn part [f & args]
  (fn [& rest]
    (apply f (concat args rest))))

((part + 1 2) 3 4)
```

18 How to do things?

18.1 map

```
(map inc [1 2 3])
```

18.2 filter

```
(filter even? [1 2 3 4])
```

18.3 reduce / fold

```
(reduce + [1 2 3])
```

18.4 list comprehension / generators

```
(for [x (range 1 10) :when (even? x)
      y (range 1 10) :when (odd? y)]
  (* x y))
```

18.5 recursion

```
(defn factorial [n]
  (if (zero? n)
    1
    (* n factorial)))
```

18.6 pattern matching

```
-- Lists in Elm
[1,2,3] == (1 :: 2 :: 3 :: [])
[1,2,3] == 1 :: [2,3]

eval things = case things of
               [] -> ""
               ["surprise", x] -> String.concat [x, "!!!"]
               "concat" :: rest -> String.concat rest
               _ -> "no match"
```

18.7 function composition

```
(def inc (partial + 1))
(def twice (partial * 2))
(def inc-and-double (comp twice inc))
(def double-and-inc (comp inc twice))

inc = (+) 1
twice = (*) 2
incAndDouble = inc >> twice
doubleAndInc = inc << twice
```

19 What are Types?

Types are sets of values.

1 belongs to several types: it's an Integer, a Number, a Value, the value 1.

One of the elements of the set of all Values.

One of the elements of the set of all Integers.

The only element in the set of all values that are 1.

1 has the type Value, Integer, Being 1

20 What is polymorphism?

20.1 ad hoc polymorphism

```
(defrecord Cow [spotted?])
(defrecord Duck [daffy?])
(defmulti talk type)
(defmethod talk Cow [_] "Muuu")
(defmethod talk Duck [_] "Quack Quack")

(talk (map->Cow {:spotted? true}))
(talk (map->Duck {:daffy? true}))
```

20.2 subtyping

```
(defrecord Cow [spotted?])
(defrecord Ostrich [height])
(defrecord Duck [daffy?])
(defrecord Goose [silly?])
(defrecord Dog [grumpy?])

(def h
  (-> (make-hierarchy)
      (derive ::bird ::animal)
      (derive Dog ::animal)
      (derive Cow ::animal)
      (derive Duck ::bird)
      (derive Goose ::bird)
      (derive Ostrich ::bird)))

(defn dispatch [v] (type v))

(defmulti flies? #'dispatch :hierarchy #'h)

(defmethod flies? ::animal  [_] false)
(defmethod flies? ::bird    [_] true)
(defmethod flies? Ostrich [_] false)
(defmethod flies? Duck [duck] (not (:daffy? duck)))

(flies? (map->Cow {:spotted? true}))
(flies? (map->Goose {:silly? true}))
```

```
(flies? (map->Ostrich {:height 100}))
(flies? (map->Duck {:daffy? true}))
(flies? (map->Duck {:daffy? false}))
```

20.3 parametric polymorphism

```
data KindOfDuck = Duck | DaffyDuck

instance Show KindOfDuck where
  show Duck = "Duck"
  show DaffyDuck = "DaffyDuck"

data KindOfHorse = Horse deriving (Show)

class Walker a where
  walk :: a -> String

instance Walker KindOfDuck where
  walk Duck = "wobble"
  walk DaffyDuck = "run"

instance Walker KindOfHorse where
  walk horse = "gallop"

walkTheSame :: Walker a, Walker b => a -> b -> Bool
walkTheSame a b = (walk a) == (walk b)

bunchWalk :: Walker a => [a] -> [String]
bunchWalk = map walk
```