Introduction to Functional Programming

Jean-Louis Giordano (@jellismymind)

<2017-02-21 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 equality?

Two values are equal if they are the same at any point in time.

```
1 === 1;
"hello" === "hello";
a = {x: 1, y: 2};
b = {x: 1, y: 2};
c = a;
a === b; //=> false
a === c; //=> true
[] === []; //=> false
{x: 1, y: 2} === {x: 1, y: 2}

(= {:x 1, :y 2} {:x 1, :y 2})

{x=1, y=2} == {x=1, y=2}
```

4 Immutable vs Mutable values

Values that never change over time are immutable.

In JavaScript, numbers and strings are immutable, arrays and objects are mutable. So it's easy to compare numbers and strings, but hard to compare arrays and objects.

In ClojureScript and Elm, all values are immutable by default.

5 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.

6 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    |
+------------+
```

7 What is a variable?

A variable is a reference to a value.

In:

```
var a = 2;
```

a is a variable, referencing the value 2

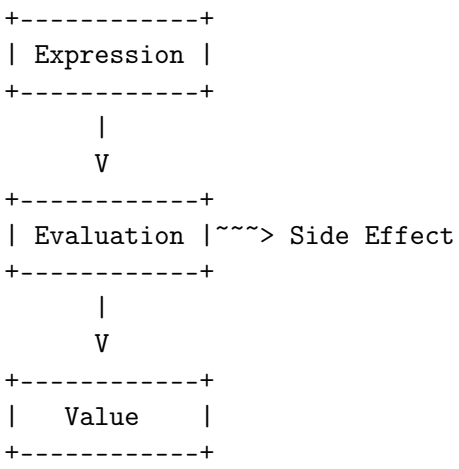
8 What are side-effects?

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;
```



9 Statement

A statement is an expression that evaluates to nothing.

9.1 In JavaScript:

Expression:

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

Statement:

```
var a = 1;
```

9.2 In ClojureScript and Elm:

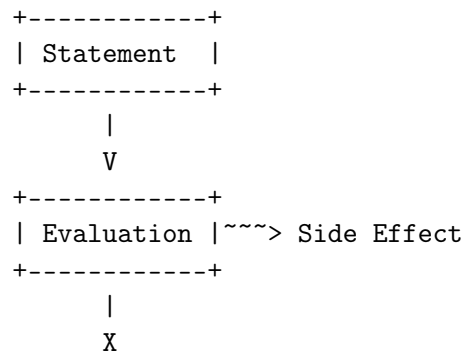
Only expressions

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

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

9.3 Why is that?

Statements require side effects, intrinsically imperative.



10 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?

11 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
```

12 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;
```

13 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.

14 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;
};
```

15 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.

16 Higher order function

Functions can return functions, and take functions as argument.

```
var apply = (f, x, y) => f(x, y);

apply(add, 2, 3);
```

```
var makeIncrementer = (n) => (m) => m + n;

var add6 = makeIncrementer(6);
add6(4);
```

17 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
```

18 Partial Application (bis)

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

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

19 How to do things?

19.1 map

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

19.2 filter

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

19.3 reduce / fold

```
(reduce + 0 [1 2 3])
```

19.4 recursion

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

19.5 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"
```

19.6 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
```

20 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

21 What is polymorphism?

A function that provides a single interface for different types.

22 ad hoc polymorphism

The same function has a different implementation depending on the type of its inputs when applied.

```
(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}))
```

```
function Cow (spotted) { this.spotted = spotted; };
function Duck (daffy) { this.daffy = daffy; };

var talk = (x) => talk[x.constructor.name](x);

talk.Cow = (cow) => "Muuu";
talk.Duck = (duck) => duck.daffy ? "What's up?" : "Quack Quack";

talk(new Duck(true));
talk(new Cow(false));
```

23 subtyping

```
(defrecord Cow [spotted?])
(defrecord Ostrich [height])
(defrecord Duck [daffy?])
```

```
(defrecord Goose [silly?])
(defrecord Dog [grumpy?])

(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)

(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}))
```

24 parametric polymorphism

Also known as "generics"

```
List.map

type alias Point2D a = {a | x : Float, y : Float}

move : Float -> Float -> Point2D a -> Point2D a
move x y point = {point | x = (.x point) + x,
                          y = (.y point) + y}
```

Note: in Elm, map is "polymorphic" because lists have different types:

```
[1,2,3] : List Float
```

```
["1", "2", "3"] : List String
[{x=0,y=0}, {x=1,y=0}] : List Point2D {}
```

But in ClojureScript and JavaScript, all lists have the same type.

```
typeof [1,2,3];
typeof ["1","2","3"];

(type [1 2 3])
(type ["1" "2" "3"])
(type [{:x 0 :y 0}])
```