Introduction to Functional Programming

Jean-Louis Giordano (@jellismymind)

<2015-04-08 Wed>

Contents

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:
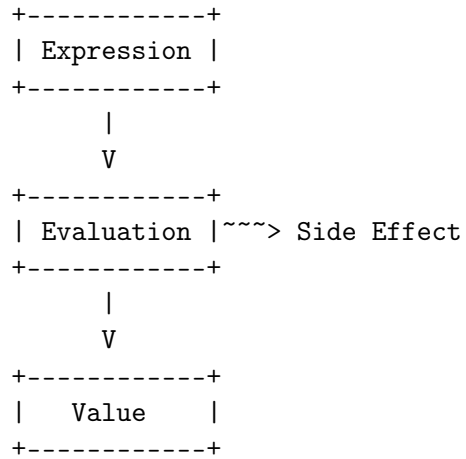
```
a = 2
```

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:

```
print "hello"
```

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

```
+------------+
| Expression |
+------------+
      |
      V
+------------+
| Evaluation |~~~> Side Effect
+------------+
      |
      V
+------------+
|   Value    |
+------------+
```

7 Statement

A statement is an expression that evaluates to nothing.

7.1 In Python:

Expression:

```
a = 1 + 1
a = lambda x: x ** 2
```

Statement:

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

7.2 In Clojure and Haskell

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.

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

8   What is a function?

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

def inc (x):
  return x + 1

inc = lambda 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
def zero ():
  return 0

# arity 1
def inc (x):
  return x + 1

# arity 2
def add (x, y):
  return x + y
```

10   Routine vs Function

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

```
def a (x):
  print x

def 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?

```
def inc (x):
  return x + 1

def one (x):
  print x
  return x

def rand (x):
  # non deterministic!
  return x * random.random()
```

12   Immutable vs Mutable

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

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

```
(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

```
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

18.7   function composition

```
(def twice (partial * 2))
(def thrice (partial * 3))
(def six-times (comp twice thrice))

twice = (*) 2
thrice = (*) 3
sixtimes = twice . thrice
```

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? #user.Cow{:spotted? true})
(flies? #user.Goose{:silly? true})
```

```
(flies? #user.Ostrich{:height 100})
(flies? #user.Duck{:daffy? true})
(flies? #user.Duck{:daffy? false})
```

20.3   parametric polymorphism

```
(defn walk [animal]
  ((get animal :walk) animal))

(walk {:species :Duck,
       :daffy? false,
       :walk (fn [duck]
               (if (:daffy? duck) "run" "wobble"))})
(walk {:species :Horse, :walk (fn [_] "gallop")})

(defn walk-a-lot [animal]
  (take 3 (repeatedly (partial walk animal))))

(walk-a-lot {:species :Duck,
             :daffy? false,
             :walk (fn [duck]
                     (if (:daffy? duck) "run" "wobble"))})
(walk-a-lot {:species :Horse,
             :walk (fn [_] "gallop")})
```