Article

Ryan and Dhaval

Code Jugalbandi

*In 2013, **Dhaval Dalal** was inspired by **Jugalbandi**, an Indian classical music form in which musicians have an improvised musical conversation, each using a different instrument, while exploring a shared musical theme together akin to Jazz in some ways. Dhaval thought, "What if we could have a conversation about some programming concepts in this way? Instead of musical instruments, what if we use different programming languages?"*

This led to the first Code Jugalbandi between us. Code Jugalbandi inspired and fuelled our learning by focusing on dialogue and exploration. More languages in the room meant more perspectives too. Exploring a paradigm through many languages gave us a better map of the territory than any single language could. Above all, Code Jugalbandi was a playful way to learn together!

Creator and Listener

We met regularly with over a period of several months, exploring some key issues related to Functional Programming (FP). This culminated in a live Code Jugalbandi at the Functional Conference in Bengaluru, 2014. During the FP Code Jugalbandi, we explored nine **themes**, which became **melodies**. Similar to the musical Jugalbandi, there are two roles: **creator** and **listener**. We called the creator of the melody **Bramha** and the one who listens and responds with a different instrument, **Krishna**. + In the last Code Jugalbandi Article, we looked at the melody, the Currying and Partial Function problem. This time we will look at **De-Structuring and Pattern Matching**.

De-structuring

BRAMHA: In FP, we favour simple data structures over classes, e.g. instead of a collection of *people* objects, we might have a collection of *person* tuples, and containing only the fields we're interested in:

```
//Tuple with 2 elements - name and age
type Person = (String, Int)
val p = ("alan", 30)
```

BRAMHA: … and then we have some functions acting on this *person* data *type*. Whatever we'll do with *person*, we'll need code to get at the innards of the *person* tuple.

```
def showPerson(p: Person) = p match {
  case (name, age) =>     println(s"$name - $age")
}

showPerson(p) //alan - 30
```

BRAMHA: How does this look in Clojure?

KRISHNA: Most functional languages provide some level of *de-structuring*:

```
(def a-person ["alan" 30])
(defn print-person [p]
  (println (first p) " " (second p)))
(defn print-person2 [[name age]]
  (println name " - " age))
```

KRISHNA: De-structuring gives you a shorthand way of naming parts of a structure. If we were using a hash-map to represent our person:

```
(def h-person {:name "alan"
               :age 30 })
;; de-structuring hash-maps looks like this:
(defn print-person3 [{:keys [name age]}]
 (println name " - " age))
```

KRISHNA: There are libraries that allow for even more compact and sophisticated de-structuring syntax, but let's leave it at that for de-structuring in Clojure.

BRAMHA: This is interesting, I am not aware of de-structuring Maps in Scala, out-of-the-box. However, you can de-structure *Lists*, *Vectors* and *Streams* in Scala. If Person was a Vector like you showed in Clojure, I could do the following

```
val p = List("alan", 30)
  def showPerson(p: List[Any]) = p match {
   case List(name, age) =>  println(s"$name - $age")
}
showPerson(p) //alan - 30
```

Pattern Matching

BRAMHA: So far we've just talked about picking apart the structure of things, but we can generalize de-structuring into something even more powerful: Pattern Matching **(not to be confused with string pattern matching)**

BRAMHA: Let's explore this by example. Say we have a journal with some fields.

```
case class Journal(op: String, amt: Double)
```

BRAMHA: Let's say, we want to save the journal entries like *debit* and *credit*.

```
def save(j: Journal) = j match {
   case Journal("debit" , _) => println("Debit")
   case Journal("credit", _) => println("Credit")
   case Journal(_       , _) => println("Unknown")
}
save(Journal("debit",    23))  //Debit
save(Journal("credit",   10))  //Credit
save(Journal("transfer", 50))  //Unknown
```

KRISHNA: Does the underscore "_" here mean wildcard?

BRAMHA: Yes, it means that we don't care about it and is used as a placeholder during pattern matching.

BRAMHA: We get pattern matching in Scala for free by using *case* classes. Scala compiler implicitly adds a companion object for the *case* class and provides an implementation for the *apply()* method **(basically factory method to create the classes)** and an *unapply()* method to get the constituents back.

KRISHNA: Yes, I see, we want to pick only those journals where the first value is *debit* or *credit*. So, you can't do this kind of thing in Clojure.

Pattern Matching - Clojure

KRISHNA: Idiomatic Clojure doesn't have "Pattern Matching" in that way. But, there is a way to achieve this (there are libraries that implement sophisticated Pattern Matching)

```
(def debit-journal  [:debit  25.4])
(def credit-journal [:credit 26.5])
```

Then one can use multi-methods in Clojure like this.

```
(defmulti save-j (fn [j] (first j)))
(defmethod save-j :debit [j]
    (println "DEBIT: " j))
(defmethod save-j :credit [j]
     (println "CREDIT: " j))
(save-j debit-journal)
(save-j credit-journal)
```

KRISHNA: Multi-methods allow dispatching on the arguments in a very flexible way.

Pattern Matching - Haskell

KRISHNA: In Haskell, this would have been expressed much more directly.

```
data Journal = Journal String Float
save_j :: Journal -> IO()
save_j (Journal "debit"  _ _)
    = print ("Debit")
save_j (Journal "credit" _ _)
    = print ("Credit")
save_j (Journal _        _ _)
    = print ("Unknown")

save_j (Journal "debit"    20.0)  –– Debit
save_j (Journal "credit"   100.0) –– Credit
save_j (Journal "transfer" 50.0)  –– Unknown
```

KRISHNA: So, we use pattern matching when we define a function, that is, embed pattern matching in function definitions. However, you can also do *case* distinctions explicitly, similar to Scala.

```
save_j :: Journal -> IO()
save_j jrnl = case jrnl of
    Journal "debit"  _
-> print ("Debit")
    Journal "credit" _
-> print ("Credit")
    Journal _        _
       -> print ("Unknown")
```

KRISHNA: In Haskell, we can pattern match on tuples as well.

```
showPerson :: (String, Int) -> IO()
showPerson (name, age)
    = print (name ++ " - " ++ (show age))
showPerson ("alan", 30)
```

KRISHNA: In Haskell, pattern matching is a deep part of the language and we have just scratched the surface here.

Reflections

KRISHNA: There is a difference between De-structuring and Pattern Matching. In pattern matching, we're not only matching on relative position of elements in the data structure, we're trying to dispatch to different functions based on a value inside the data structure! This dispatch is a kind of conditional construct, while with de-structuring there is nothing conditional about it.

BRAMHA: In Clojure, we saw basic de-structuring, and then more sophisticated pattern matching made possible by **Multi-method** dispatch. In Scala and Haskell, de-structuring and pattern matching are unified using the *case* expressions.

KRISHNA: Pattern matching with all the case distinctions is like generalization of *switch* statements in Java/C#, where we used Ints, Enums etc., but we can now use class hierarchies or data types. It so happens that the syntax is different.

Ryan is a software developer, coach and advisor, based in Cape Town. He has been working with code for more than 15 years. Ryan assists individuals and teams to manage the evolution and complexity of their software systems. Ryan is a passionate learner and enjoys facilitating learning in others.

Dhaval a hands-on developer and mentor, believes that software development is first an art and then a craft. For him writing software brings his creativity to the fore. His interests are in architecting applications ground up, estabilishing environments, transitioning and orienting teams to Agile way of working by embedding himself within teams.

You can follow them on Twitter @ CodeJugalbandi

Pragmatic
Programmer Quote

"In the old days, people robbed stagecoaches and knocked off armored trucks. Now they're knocking off servers."

- Richard Power

"The inside of a computer is as dumb as hell but it goes like mad!"

- Richard Feynman

"The question of whether computers can think is just like the question of whether submarines can swim."

- Edsger W. Dijkstra