PATTERN MATCHING

by Carlos Rodrigues

```
Github:  carlosnsr
Twitter: @carlos1nsr
Email:   carlos.all.mail@gmail.com
```

RESOURCES

- Programming Elixir by Dave Thomas
- http://elixir-lang.org/getting-started
- https://elixirschool.com/

WHY?

- Fundamental feature
- Impacts:
  - Deconstructing complex types
  - Function calls
  - Comparisons

OBJECTIVES

- Differentiate between matching and assigning
- Preserve values by using ^ (pin operator)
- Use matching to de-construct complex types
- Use matching to select which function to execute

RE-THINK =

- In Elixir, = is not the assignment operator
- It is the *match operator*
- Used for matching the LHS with the RHS

= IS LIKE AN ASSERTION

```
1 = 1  # true
2 = 2  # true
2 = 1  # not true
```

- Elixir tries to match the LHS to the RHS to make the assertion true

```
a = 1  # is true, only if `a` is `1`
```

- Elixir deduces that a is 1 and binds a to the value 1

PASS THE DUTCHIE ON THE LEFT HAND SIDE

- Binding varibles to values only happens on the LHS

```
a = 1  # matches.  Elixir binds `a` to `1`
1 = a  # matches.  `a` is already bound to `1` so `1 = 1` is a
2 = a  # does not match.  Earlier matches said that `a = 1`.
2 = b  # CompileError because b is not defined/bound
```

- Variables can be re-bound: match it to a new value

```
a = 2  # matches.  Elixir binds `a` to `2`
```

^ PREVENTS BINDING

- The *pin operator* ^ prevents binding
- When rebinding isn't desired, prefix variables with ^

```
a = 1   # matches.  Elixir binds `a` to `1`
b = 2   # matches.  Elixir binds `b` to `2`
^a = b  # does not match.  `a` does not get bound to `2`
```

WHAT'S THE PATTERN?

- The LHS is the *pattern*
- The RHS is the *values*
- Elixir will try to match the *pattern* with the *values*

MATCH CONDITIONS

- Both sides must have the same structure
- Each term in the pattern (LHS) must be matched to the corresponding term in the values (RHS)

```
[a, b] = {4, 5}  # no match.  List vs Tuple
{a, b} = {4, 5, 6}  # no match.  Different length tuples
{a, b} = {4, 5}  # match.  Tuple vs Tuple
a  # 4
b  # 5
{4, 5} = {a, b}  # match.  Each LHS term matches each on RHS
```

LITERALS AND VARIABLES

- A literal must match that *exact value* on the RHS

```
[1, 2] = [1, 2]  # matches
[2, 3] = [1, 3]  # does not match
```

- A variable matches *by taking* the value on the RHS

```
[1, a] = [1, 2]  # matches.  `a` is now `2`
[2, a] = [1, 3]  # does not match.  `a` is unchanged
```

MATCH ANYTHING OPERATOR _

- _ is a special variable (so LHS only)

```
[1, 2] = [1, _]  # CompileError: unbound variable _
```

- _ matches *any value* on the RHS

```
[_, a] = [5, 6]  # matches
a  # 6

[_, a] = [7, 8]  # matches
a  # 8

[_, a] = [[4, 5], 6]  # matches
a  # 6
```

PATTERNS CAN BE RECURSIVE

```
[d = [a, b], c] = [[:first, :second], :third] # match
d  # [:first, :second]
a  # :first
b  # :second
c  # :third
```

DECONSTRUCT COMPLEX TYPES

- Pattern match to get values out of complex types
  - Lists
  - Tuples
  - Maps

TUPLES

- Basic type
- Store elements contiguously in memory
- Can be accessed by index
- Have a size

```
tuple = {:ok, "hello"}
elem(tuple, 1)  # "hello"
tuple.size(tuple)  # 2
```

PATTERN MATCHING TUPLES

```
{a, b} = {:left, :right} # match
a # left
b # right
{:left, :right} = {a, b} # match

{c, :right} = {:left, :right} # match
c # left

{{d, e}, f} = {{:first, :second}, :third} # match
d  # :first
e  # :second
f  # :third
```

LISTS

```
[a, b, c] = [:first, :second, :third] # match
a  # :first
b  # :second
c  # :third

[d, _, e] = [4, 5, 6]  # matches
d  # 4
e  # 6
```

LISTS ARE A RECURSIVE DATA-STRUCTURE

- A new element is prepended to the list using `|`

```
a = [2, 3]
a = [1 | a]
a  # [1, 2, 3]
```

USE | IN LIST MATCHES

- to match the head and the tail

```
[h | t] = [1, 2, 3]
h  # 1
t  # [2, 3]

[1, 2, 3] = [h | t] # match
[1 | [2, 3]] = [h | t] # match
[1, 2, 4] = [h | t] # NO match.  t != [2
```

- to match more than one head

```
[a, b, c | t] = [1, 2, 3]
a  # 1
b  # 2
c  # 3
t  # []
```

MAPS

- Maps are key-value stores

```
map = %{ lang: "Elixir", talk: "Pattern Matching" }
map[:lang]  # "Elixir"
map.talk  # "Pattern Matching"
```

PATTERN MATCHING MAPS

- Match an entry

```
map = %{ lang: "Elixir", talk: "Pattern Matching" }
%{ lang: _ } = map  # matches
%{ address: _ } = map  # no match.  MatchError
```

- Match an entry that has an exact value

```
%{ lang: "Elixir" } = map  # matches
%{ lang: "Java" } = map  # no match.  MatchError
```

- Extract a value from an entry

FUNCTION PARAMETER LISTS ARE ACTUALLY PATTERNS

- Parameters are pattern matched to the passed in values
- So one can deconstruct complex types easily

```
sum_3_list = fn [a, b, c] = a + b + c end
sum_3_list.( [1, 2, 3] )  # 6

invert_pair = fn {a, b} -> {b, a} end
invert_pair.( {1, 2} )  # {2, 1}
```

FUNCTIONS ARE CHOSEN BY WHICH PATTERN MATCHES THE VALUES

- When a function is called, the argument list is matched to the function's parameter list
- The function with the matching pattern, is the function that gets called

GREAT FOR...

- executing code depending on the passed-in values

```
def divide(_, 0) do
  raise ArithmeticError, message: "division by zero"
end
def divide(a, b), do: a / b
```

GREAT FOR...

- handling errors

```
def read_first_line( {:ok, file} ), do: IO.read(file, :line)
def read_first_line( {:error, reason} ) do
  "Error: #{:file.format_error(reason)}"
end
```

GREAT FOR...

- recursion

```
def factorial(0), do: 1
def factorial(n), do: n * factorial(n - 1)
```

CONCLUSION

THANK YOU

QUESTIONS?


```
Github:  carlosnsr
Twitter: @carlos1nsr
Email:   carlos.all.mail@gmail.com
```