MTAT.03.295
Agile Software Development

Lecture 2: Elixir fundamentals

Luciano García-Bañuelos

Elixir's numeric values

› Integer arbitrary-precision arithmetic

› Float 64 bits, double precision

```
1250
0xB21
0o732
0b10111
```

```
+ - *

div rem  /

is_number
is_integer
is_float
```

```
1.25
1.0e-5
```

```
1_500_000
```

Atoms & Booleans

› Atoms are constant strings that are used to refer to a unique value. Start with ":" or capital letter. (Well suited for pattern matching)

› Booleans values: `true` and `false`

is_atom
is_boolean

Boolean operators:

and or not

Relaxed boolean operators:

&& || !

Strings & Charlists

- Double-quoted sequence of characters are strings, e.g.
  "This is a string"
- Single-quoted sequence of characters are not strings, e.g.
  'This is not a string'

is_binary
is_list

```
> iex.bat
iex(1)> IO.inspect 'string', charlists: :as_lists

iex(2)> IO.inspect "string", binaries: :as_binaries
```

More on strings

› String interpolation

```
> iex.bat
iex(1)> a = 'string'; b = [1,2,3]
iex(2)> "Value of 'a': #{a}"
iex(3)> "Value of 'b': #{b}"
iex(4)> "Value of 'b': #{inspect(b)}"
```

› Multi-line strings

```
mlstr = """
        This is a multi-line
        string!!!
        """
```

Collection types

› Tuples

{:ok, 123, false}

› Lists

[1, 2, 3, 4]

```
[1, 2]++[3]      # Concatenation
[1, 2, 3]--[2]   # Difference
hd [1, 2, 3]     # List head
tl [1, 2, 3]     # ... Tail
[1 | [2, 3]]     # “Cons” operator
```

› Maps

%{:name => "Alfonso Cuarón", :age => 55}

Functions in Elixir

› Two types: anonymous and named

```
sum = fn (a,b) -> a + b end
sum.(3, 5)
```

```
sum = &(&1 + &2)
sum.(3, 5)
```

```
defmodule Example do
  def factorial(0), do: 1
  def factorial(n) when n > 0, do: n * factorial(n - 1)
end

Example.factorial(6)
```

Pattern matching

- In Elixir `a = 1` does not mean we are assigning 1 to variable a
- The symbol "`=`" asserts that the left-hand side (LHS) matches the right-hand side (RHS)

```
> iex.bat
iex(1)> a = 1
1
iex(2)> 1 = a
1
iex(3)> a = 2
2
```

variables can be rebound

unless variables are pinned

```
> iex.bat
iex(1)> a = 1
1
iex(2)> 1 = a
1
iex(3)> ^a = 2
** (MatchError) no match ...
```

Pattern matching on collections

› Let us play a little bit

```
> iex.bat
iex(1)> [1, a, 3] = [1, 2, 3]
[1, 2, 3]
iex(2)> a
2
iex(3)> [1, a, a] = [1, 2, 2]
[1, 2, 2]
iex(3)> a
2
iex(4)> [1, a, a] = [1, 2, 3]
** (MatchError) no match ...
```

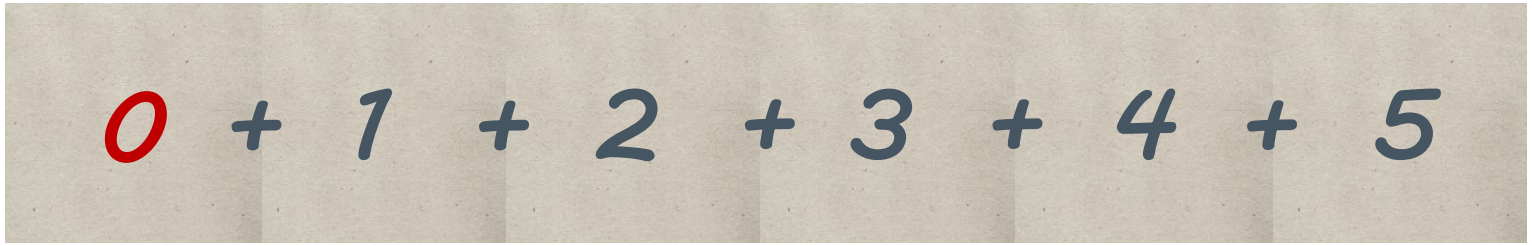
```
> iex.bat
iex(1)> [a | b] = [1, 2, 3]
[1, 2, 3]
iex(2)> a
1
iex(3)> b
[2, 3]
```

Fold operation

foldl
(a.k.a. reduce)

foldr

Adding the elements in a list with a foldl




```
defmodule Example do
  def foldl([], acc, _f), do: acc
  def foldl([h|t], acc, f), do: foldl(t, f.(acc, h), f)
end
```

```
> iex.bat -S mix
iex(1)> Example.foldl([1,2,3,4,5], 0, fn(a,n) -> a + n end)
15
```

Higher order functions

- Functions that take another function as input and/or produces another one as output
- Three widely used:

- Fold (left or right)
- Map
- Filter

Higher order functions (map)

› MAP applies a given function to each element of the input list and produces a new list thereof

```
defmodule Example do
  def map([], _f), do: []
  def map([h|t], f), do: [f.(h) | map(t, f)]
end
```

Higher order functions (filter)

› FILTER copies elements from an input to an output list, keeping only those for which a given function returns true

```
defmodule Example do
  def filter([], _f), do: []
  def filter([h|t], f) do
    case f.(h) do
      true -> [h | filter(t, f)]
      _ -> filter(t, f)
    end
  end
end
```

In class exercise

- Write a function that computes the average of an input list of numbers
- You can use any of the three functions presented here (i.e. foldl, map, filter)

Binary trees

```
defmodule BinaryTree do
  def insert(nil, value), do: {value, nil, nil}

  def insert({value, left, right}, new_value)
      when new_value < value do
    {value, insert(left, new_value), right}
  end

  def insert({value, left, right}, new_value) do
    {value, left, insert(right, new_value)}
  end
  ...
end
```

In class exercise

- Write a function that implements an “in-order” traversal on a given binary tree
- Generalize the code above to a sort of “fold” function that takes as input an “accumulator” and an anonymous function and produces an ordered list (i.e. similar to the one produced by the in-order traversal)