MTAT.03.295
Agile Software Development

Lecture 3: Elixir standard libraries and control flow

Luciano García-Bañuelos
Map

```
fn n -> n * 2 end
```

<table>
<thead>
<tr>
<th>2</th>
<th>7</th>
<th>12</th>
<th>15</th>
<th>17</th>
<th>22</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>14</td>
<td>24</td>
<td>30</td>
<td>34</td>
<td>44</td>
</tr>
</tbody>
</table>
Filter

\[
\text{&\texttt{Integer.is_even/1}}
\]

\[
\begin{array}{cccccc}
2 & 7 & 12 & 15 & 17 & 22 \\
\end{array}
\]

\[
\begin{array}{ccc}
2 & 12 & 22 \\
\end{array}
\]
Foldl/Reduce

\[
\text{fn } n, \text{ acc } \rightarrow n + \text{ acc } \text{ end}
\]

```
foldl(fn n, acc -> n + acc, 0)
```

```
<table>
<thead>
<tr>
<th>0</th>
<th>7</th>
<th>12</th>
<th>15</th>
<th>17</th>
<th>22</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>12</td>
<td>15</td>
<td>17</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>12</td>
<td>15</td>
<td>17</td>
<td>22</td>
<td>19</td>
</tr>
<tr>
<td>19</td>
<td>15</td>
<td>17</td>
<td>22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>17</td>
<td>22</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>51</td>
<td>22</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>73</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Structured types

› Tuples

{:ok, 123, false}

[1, 2, 3, 4]

› 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}
Structured types

› Tuples
\{\:ok, 123, false\}

= t = \{\:ok, 123, false\}

elem(t, 0) # :ok

put_elem(t, 0, :nok) # \{\:nok, 123, false\}

elem(t, 3) # FAILS!!!!

› Lists
[1, 2, 3, 4]

› Maps
\%\{\:name => "Alfonso Cuarón", \:age => 55\}
Structured types

 › Tuples
   `{:ok, 123, false}`

 › Lists
   `[1, 2, 3, 4]`

 › Maps
   `%{:name => "Alfonso Cuarón", :age => 55}`
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)
Keyword lists

› Collection of key-value pairs (key is always an atom)

```
[fg: "blue", bg: "white", font: "Helvetica"]
```

```
[{:fg, "blue"}, {:bg, "white"}, {:font, "Helvetica"}]
```

› Extensively used!

```
if shoe_size <= 35, do: "child", else: "young adult"
```

```
if(shoe_size <= 35, [do: "child", else: "young adult"])
```

```
query = from c in Customer,
     where: c.type == "Loyal customer",
     select: c
```
Structs

Derived from Maps, they come with some constraints that are checked at compilation time

```elixir
defmodule User do
defstruct first_name: nil, last_name: nil, active: false
end
```

```
iex> user = %User{first_name: "Philip", last_name: "Brown"}
%User{active: false, first_name: "Philip", last_name: "Brown"}

iex> user = %User{first_name: "Philip", last_name: "Brown", location: "UK"}
** (KeyError) key :location not found in: %User{active: false, first_name: "Philip", last_name: "Brown"}

iex> user[:first_name]  # This is wrong
```

Agile Software Development - Elixir std libs & ctrl flow
Standard libraries for structured types

- Tuple
- List
- Keyword
- Tuples
- Lists
- Keyword lists
- Enum
- Stream
- Map
- Struct
- Sets
- MapSet
Standard libraries for simple types

› Worth having a look at the following libraries:
  – Regex
  – String
  – Time
  – Range
  – Integer
## Sigils

<table>
<thead>
<tr>
<th>~r</th>
<th>Regular expression with escaping and interpolation</th>
</tr>
</thead>
<tbody>
<tr>
<td>~R</td>
<td>Regular expression, no escaping nor interpolation</td>
</tr>
<tr>
<td>~w</td>
<td>Word list with escaping and interpolation</td>
</tr>
<tr>
<td>~W</td>
<td>Word list, no escaping nor interpolation</td>
</tr>
<tr>
<td>~c</td>
<td>Charlist with escaping and interpolation</td>
</tr>
<tr>
<td>~C</td>
<td>Charlist, no escaping nor interpolation</td>
</tr>
<tr>
<td>~s</td>
<td>String with escaping and interpolation</td>
</tr>
<tr>
<td>~S</td>
<td>String with no escaping nor interpolation</td>
</tr>
<tr>
<td>~N</td>
<td>Naïve date time struct</td>
</tr>
</tbody>
</table>
**if/else vs. cond**

```elixir
def children_shoe(size) when size in 20..42 do
  if size <= 23 do
    "toddler"
  else
    if size <= 35 do
      "child"
    else
      "young adult"
    end
  end
end
```

```elixir
def children_shoe(size) when size in 20..42 do
  cond do
    size <= 23 -> "toddler"
    size <= 35 -> "child"
    true -> "young adult"
  end
end
```
case

› Similar to switch/case found in other languages

› In Elixir, however, we can use pattern matching in this construction

```elixir
case Repo.insert(%Post{title: "Ecto is great"}) do
  {:ok, struct} -> # Inserted with success
  {:error, changeset} -> # Something went wrong
end
```

Errors is usually handled in this way and not by raising exceptions!
Comprehensions

› Not as conventional for loops … but rather a combination of map+filter

```elixir
for x <- [1, 2, 3, 4, 5], do: x*x
for x <- [1, 2, 3, 4, 5], y <- ["a", "b"], do: {x, y}
for {_key, val} <- [one: 1, two: 2, three: 3], do: val
for x <- 1..10, is_even(x), do: x
```