Elixir

Levi Moore
(levimoore@email.arizona.edu)

Benhur Tadiparti
(tadiparti@email.arizona.edu)

April 20, 2020

1 Abstract

2 Introduction

3 History

The Elixir programming language was created by José Valim, the co-founder of Plataformatec, and member of the Ruby on Rails Core Team. He aimed to create a programming language for large-scale sites and apps by combining the best features of Ruby, Erlang, and Clojure.

Elixir was designed to handle large data volumes. The speed and capabilities spread Elixir into telecommunication, eCommerce, and finance industries. (Wiki)

Elixir is definitely not dead nor barely alive due to the avid number of Elixir users, however, compared to big programming languages like Python and Java - Elixir is in-between alive and barely moving.

4 Control Structures

In a program, a control-structure determines the order in which statements are executed. Elixir supports three control-structures. These include

'if and unless', 'case' and 'cond'. In this section these three control-structures will be explored individually, code snippets of each will be included for reference.

In programming, the most well-known control-structure would have to be the if-statement. In simple terms, if a statement is proven true the code below is executed, otherwise a different block of code is executed for when its proven false. Secondly, Elixir also supports unless, this could be thought of as an if statement that works negative. Below is a snippet of how the if-statement is used in Elixir.

SNIPPET HERE:

```
if x == true do
    IO.puts("Hello User!")
unless x == null do
    IO.puts("User might be on the wrong computer")
else
    IO.puts("Unauthorized Access!")
end
```

What determines what we do as humans every day is what day of the week it is. On Sundays we get ready for the work or school week on Monday, whereas on Fridays we get ready to relax and take It easy for a couple days. This is a real-life example of the case control-structure that Elixir supports. This control-structure allows us to compare a value against various patterns until we find the case that suits it and branch into that direction. Below is an example of how the case control-structure is used in Elixir.

SNIPPET HERE:

```
case {"Hi", "Hello"} do
    {"Hi", x} ->IO.puts("The computer response woud be to say 'Hello'")
    {"What", "Excuse Me"} ->IO.puts("That was not a greeting")
end
```

When we need to check many different conditions and find one that evaluates to true, its best to use the cond control-structure. Cond allows us to enter as many conditions as we want to check, similar to an else-if in other languages, in order to evaluate to true. In some cases, all conditions will evaluate to false, so we must add a true condition at the end, similar to an

else statement, to end the conditional. This is useful when we have various conditions needing to be tested. Below is an example of how the cond control-structure is used in Elixir.

SNIPPET HERE:

```
cond do
    1 + 0 == 2 ->IO.puts "This is also false"
    5 * 10 == 0 ->IO.puts "This is false"
    true ->IO.puts "Hello world"
end
```

5 DataTypes

A data type in programming is essentially a data structure that holds data in which the computer can process. Each programming language have similar and unique data types. In Elixir there are seven data types: Strings, Integers, Floats, Lists, Tuples, Booleans and Atoms. Many of these are similar to other coding languages with minor differences and exceptions that will be discussed in this section.

The most well-known data type in programming is a string. A string is encapsulated between two double quotations and everything inside it is the string literal. This is how Elixir handles strings and applies to all other languages as well. Elixir can do many things with strings such as: concatenation, lengths, string equivalence and new line characters. The code snippet below demonstrates how to do all these things.

SNIPPET HERE:

```
def newLine() do
    IO.puts "Insert newline here \nNow I'm on a newline"
end

def concat() do
    IO.puts "This string is its own and" <>" this string was added with '<>'
added between them"
end
```

```
def length() do
    IO.puts "Length of hello is " <>"#String.length("hello")"
end

def equalabc(str) do
    String.equivalent?(str, "abc")
end
```

Another well-known data type is Integers, these are just whole numbers. Elixir supports all the following arithmetic for integers: addition, subtraction, multiplication and division. The integer also supports the following: floor_div, gcd, mod, to_charlist and to_string. floor_div takes in two integers divides the first by the second and floors it. Gcd takes two integers and returns their greatest common denominator, mod takes two integers and returns the remainder of the two divided. To_charlist returns the integer as a character and to_string returns the integer as a string. The following code snippet demonstrates how to use all these in Elixir.

SNIPPET HERE:

```
def arithmetic(x, y) do
    IO.puts x + y
    IO.puts x * y
    IO.puts x - y
    IO.puts x / y
end

def otherFunct(x, y) do
    IO.puts Integer.floor_div(x, y)
    IO.puts Integer.gcd(x, y)
    IO.puts Integer.mod(x, y)
    IO.puts Integer.to_charlist(x)
    IO.puts Integer.to_string(x)
end
```

Floats are very similar to integers with the difference of it being a decimal number instead of a while number. Elixir supports the same arithmetic as integers and has the following methods in the float class: ceil, floor, parse,

ratio, round, to_charlist and to_string. Ceil takes one float and one integer and rounds a float to the smallest integer greater than or equal to the first parameter. Floor rounds a float to the largest number less than or equal to the first parameter. Parse takes a string and returns its float. Ratio returns a pair of integers whose ratio is exactly equal to the original float and with a positive denominator. Round rounds a floating-point value to an arbitrary number of fractional digits (between 0 and 15). The last two do the same as the integer methods. The following code snippet demonstrates how to use all these in Elixir.

SNIPPET HERE:

```
def arithmetic(x, y) do
    IO.puts x + y
    IO.puts x * y
    IO.puts x - y
    IO.puts x / y
end

def otherFunct(x, y) do
    Float.ceil(x, 2)
    Float.floor(x, 3)
    Float.parse(y)
    Float.ratio(x)
    Float.round(x, 4)
    Float.to_charlist(x)
    Float.to_string(x)
end
```

Lists are a collection of values that are stored next to each other. In Elixir a list could hold any type and is done by surrounding the types in brackets. Elixir supports several handy methods and operations that can be done to a list such as: concatenation, subtraction, length, hd and tl. Length simply returns the length of the list, hd returns the first element of the list and tl returns the list minus the first element. The following code snippet demonstrates how to use and do all these in Elixir.

SNIPPET HERE:

```
def listHead() do
    hd([0, "hello", 4])
end

def listTail() do
    tl([0, "hello", 4])
end

def concatLists() do
    [1, 2, 3] ++ [4, 5, 1]
end

def subLists() do
    [1, 2, 3] -- [4, 5, 1]
end

def lenList() do
    length([0, "hello", 4])
end
```

Tuples are very similar to lists where they can hold any types and are mutable, with the difference of surrounding them in curly brackets. Tuples in Elixir have the following methods: elem, put_elem and tuple_size. Elem takes two parameters the first being the tuple and second is the index at which you want to access and returns it. Put_elem takes the same parameters but inserts them into the tuple instead. Tuple_size simply returns the size of the tuple. The following code snippet demonstrates how to instantiate and use these methods in Elixir.

SNIPPET HERE:

```
def newTuple() do
    tuple = "hello", 10, :two
end

def getElemVal() do
    tuple = "hello", 10, :two
```

```
    elem(tuple, 1)
end

def replaceTuple() do
    tuple = "hello", 10, :two
    put_elem(tuple, 1, "world")
end

def sizeTuple() do
    tuple = "hello", 10, :two
    IO.puts tuple_size(tuple)
end
```

Booleans are very self-explanatory, true equal true and false equals false. Elixir also supports an is_boolean method which returns if the parameter passed in is of data type Boolean. The following code snippet shows how to use Booleans in Elixir.

SNIPPET HERE:

```
def trueEqualsFalse() do
    true == false
end

def trueEqualsTrue() do
    true == true
end

def predicateBool(arg) do
    is_boolean(arg)
end
```

Atoms are symbolic constants and are created by putting a colon in from of whatever you'd like to be an atom. Elixir supports the following methods for atoms: is_atom, to_charlist and to_string, you can also compare atoms with double equals signs. Is_atom returns true if the parameter is an atom, to_charlist and to_string do as they say, returns atom as char or string. The following code snippet shows how atoms are used in Elixir.

SNIPPET HERE:

```
def newAtom() do
    atom = :apple
    IO.puts atom
end

def isEqual() do
    IO.puts :apple == :apple
    :orange == :orange
end

def isBoolAtom() do
    IO.puts :true == true
    IO.puts is_atom(false)
    is_boolean(:false)
end

def otherFunct(atom) do
    IO.puts Atom.to_charlist(atom)
    Atom.to_string(atom)
end
```

6 Subprograms

In programming a subprogram can be defined as a set of instructions executed at a remote location in the program, after this subprogram finished it resumes execution of wherever this subprogram was invoked. In object-oriented programming these are called methods or constructors. In Elixir there are modules, which are the equivalent of classes, and within the modules are methods for said module. In the code snippet below there is a method multiply which simply multiplies the parameters a and b and returns them. In the method addMultiply, firstly a and b are multiplied by calling the previous method. This is an example of a subprogram because it executes code not in its method and resumes execution after the method called is complete This demonstrates how to use subprograms in Elixir.

SNIPPET HERE:

```
def multiply(a, b) do
    mul = a * b
end

def addMultiply(a, b) do
    mul = multiply(a, b)
    add = a + b
    addMul = mul + add
end

def sub(a, b) do
    mul1 = multiply(a, a)
    mul2 = multiply(b, b)
    sub = mul1 - mul2
end
```

7 Summary

8 References

References

[1] Jan Dudulski. How we learned elixir—our story and tutorial for beginners, January 2019. Last accessed 6 April 2020.

[2] elixir. elixir, 2011. Last accessed 6 April 2020.

[3] Reddit. The elixir programming language, November 2012. Last accessed 6 April 2020.

[4] Elixir School. Basics. Last accessed 6 April 2020.

[5] tutorialspoint. Learn elixir. Last accessed 6 April 2020.

[6] Wikipedia. Elixir (programming language), 2011. Last accessed 6 April 2020.

[2] [5] [1] [4] [3] [6]