**Programming Language - Elixir**

Dhruv Rana

Professor Robila

CSIT313\_01FA23

December 12, 2023

**Table of Contents**

[**1. Introduction 3**](#_9ugg5la1koox)

[**2. Background 3**](#_nnlmozkhmcp3)

[**3. Elixir Compiler 4**](#_d30v2rwg1ke5)

[**4. Elixir Default Library and Environment 4**](#_tgf4tpysxgh4)

[**5. Elixir Syntax 5**](#_omsqqxbs97rq)

[5.1 Identifiers and Reserved Words 5](#_qj943m491hk7)

[5.2 Variable Declarations 6](#_m5h90akqe57z)

[5.3 Data Types 7](#_35e868qf75im)

[5.3.1 Numerical Types 7](#_qbt64f6390jl)

[5.3.2 Atoms 7](#_3zzpiloo266h)

[5.3.3 Booleans 8](#_nks16ynirir)

[5.3.4 Strings 9](#_5tzz90w8wzwf)

[5.3.4.1 Character Lists 9](#_hlriyppykiso)

[5.3.5 Lists, Tuples, and Binaries 10](#_x9056w5oqt5)

[5.4 Operators 12](#_j1wulhnmmhib)

[**6. Printing onto Screen 12**](#_eg24ksvqol5h)

[**7. Modules and Functions 13**](#_m0sw77hb6zj2)

[**8. Control Flow and Loops 15**](#_7n01ywkeb4n9)

[8.1 Control Flow 15](#_9q3ip6xhlxos)

[8.2 Loops 16](#_a2066mm115qq)

[**9. Features 17**](#_wrpp04rlrpmr)

[9.1 Immutability 17](#_k40oh4nc32ju)

[9.2 Pattern Matching 18](#_m22x5wf32yf2)

[9.3 Concurrency and Processes 19](#_4k2pszjjycoy)

[**10. Future Direction and Evaluation 20**](#_xem7aeu7ism0)

[**11. Citations 22**](#_eq3c4g89cyus)

# **Introduction**

There are many programming languages out in the world, some well known and others not so much but learning a programming language is very important to develop skills. Elixir is a dynamic functional, concurrent, and fault-tolerant programming language built on the Erlang Virtual Machine (BEAM). Elixir was developed by José Valim in 2012. “Elixir is widely used in fields such as web development, embedded software systems, telecommunications, and multimedia networking.” (Elixir-lang 1). In fact, it is used by some well known companies like Discord, Pepsico, and even Duffal. This essay delves into the world of Elixir, exploring its uniqueness, the basic syntax and semantics, data types, functions and more while also highlighting some key features such of Elixir such as immutable data, pattern matching, concurrency, and processes, but also some of its limitations and the future direction of the language.

# **Background**

Elixir is a dynamic, functional high-level general purpose programming language created by José Valim that operates on the Erlang VM (BEAM). José Valim actually created Elixir when he was working on the Ruby on Rails core team project. According to the Ruby on Rails application, “Ruby on Rails is a web-application framework that includes everything needed to create database-backed web applications according to the Model-View-Controller (MVC) pattern.” (Ruby on Rails 1). Valim was working on an issue related to improving the performance of Rails when he noticed a lack of concurrency tools for software development. This prompted him to work on a side project Elixir, which later became the programming language Elixir. Elixir is a language that was inspired by a mixture of other languages such as Ruby and Erlang, so it tends to be more user-friendly, while also borrowing much of its syntax from those languages. “Elixir is known for its high concurrency, low latency, and scalability for building distributed systems” (Elixir-lang 1).

# **Elixir Compiler**

Elixir, like most other mainstream programming languages like Python, has a shell environment of its own to run Elixir code. Elixir code can run on the Elixir shell, but to run the Elixir code and to get to the shell you must download Erlang and Elixir. You must download the appropriate versions of both Erlang and Elixir that are compatible with each other to open the Elixir shell. Then on the Erlang shell, type the command iex to open the Elixir shell. Next, after you're done writing the code, you can compile the written code using the command elixirc to get an executable Elixir program file with the extension .exs. For the sake of simplicity, an Ubuntu virtual machine is used to test and demonstrate Elixir code with an Elixir shell version of 1.13.4. Something interesting to note is that code developed in Elixir is compatible with the code developed in Erlang and vice versa since both languages operate on BEAM. Since Elixir is built upon the Erlang, Elixir programmers also have access to most of Erlangs libraries. This covers Elixir’s shortcomings as a newer and less popular language, since it can use the resources of another already well developed language like Erlang. For more detailed information on the Elixir compiler, please visit the Hexdocs Elixir documentation.

# **Elixir Default Library and Environment**

While Elixir runs on the Erlang VM, it possesses its own programming rules, parameters, and environment within the system. According to Elixir’s written documentation in Hexdocs, “Kernel is Elixier’s default environment” (Hexdocs 1). Kernel is Elixir’s most basic and fundamental module. It contains the basic building blocks of the language. Kernel contains the rules for basic mathematical operations, data types, variables and more. You can tell how important Kernel is to the language by the fact that you can implement Kernel functions without using the Kernel keyword. While Kernel is very important to the language, if a programmer wishes to use their own defined functions instead of the Kernel functions, you opt out of them by using the except clause with an import statement. As shown below in Figure 1, you can opt out of the Kernel functions by importing Kernel, the module name, followed by a comma, and then the keyword except, and then the functions or macros enclosed in the square brackets. For more detailed information on Kernel and Elixir’s default libraries, please visit the Hexdocs Elixir documentation.



[**Figure 1:** Kernel import function to show how to opt out of Kernel functions using the keyword except. Code from Hexdocs Elixir Documentation from Kernel module.]

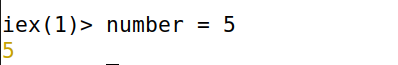
# **Elixir Syntax**

## **5.1 Identifiers and Reserved Words**

The Elixir syntax is quite simple and straightforward. Like many programming languages, Elixir contains reserved words that can not be used as identifiers. Reserved words are words that hold a special meaning within the language. Words such as, true, false, nil, when, and, or, not, in, fn, do, end, catch, rescue, after, and else are all reserved words that cannot be used as identifiers. In Elixir, identifiers must start with lowercase alphabetic characters or underscores, but not uppercase alphabetic characters, numbers, or any other special characters. After the first character, the identifier can contain any sequence of alphabetic characters, numbers, or underscores. The recommended naming convention for identifiers is snake\_case format, where words or phrases are separated by an underscore for better readability. For more detailed information on Elixir’s identifiers and reserved words, please visit the Hexdocs Elixir documentation.

## **5.2 Variable Declarations**

A variable declaration is done to tell the interpreter where and how much data to be stored in a memory location. Elixir is a dynamic programming language, as such it is not required to explicitly declare a variable’s data type at the creation of the variable. In dynamic typing, the data type of a variable is decided upon at the runtime of the program depending on the value of the variable. As shown below in Figure 2, there is a variable named number which is equal to 5, but the variable data type is not explicitly declared like in languages like Java where the variable data type must be declared at the creation of the variable. But, the data type that the variable number holds is an integer data type because of the integer value 5. An important thing to note is that ‘=’ is not an assignment operator but in Elixir it is a match operator, used for pattern matching. According to the Elixir programming language textbook *Elixir in Action* by Sas̄a Jurić “In Elixir terms, assignment is called *binding* and when you initialize a variable with a value, the variable is *bound* to that value…” (Jurić 21). At first, you may think binding and assignment mean the same thing but in actuality they are not. You can think of binding as giving a value a name in some context or scope, while assignment can be thought of as storing a value in a location which is the variable name in a given scope. Variable binding is much more helpful in pattern matching than variable assignment, especially since Elixir’s data is immutable, meaning once the data is created it can not be changed. For more detailed information on Elixir’s variable declaration, please visit the Hexdocs Elixir documentation.



[**Figure 2:** Creates the variable number bound to an integer value 5. Code made by Dhruv Rana using Ubuntu VM Elixir shell 1.13.4]

## **5.3 Data Types**

### **5.3.1 Numerical Types**

As a programming language, Elixir supports many data types. At the core of the language, Elixir uses Erlang’s type system, which allows for easy integration with Erlang libraries. Elixir supports numerical data types that are defined as integers and floats. What is interesting about Elixir’s numerical data types is that they can be represented in many formats. The integer data type can be represented in their binary, octal, and hexadecimal forms, and even using underscores for readability, replacing commas with underscores such as 1\_000. Floats can also be written in scientific notation such as 54.4e10. Elixir is not the only language, in fact these numeric data type formats are also in Python. Having so many representations for integers and floats means that Elixir provides the programmer the ability to manipulate data at the bit level. For more detailed information on the Elixir’s numerical data types, please visit the Hexdocs Elixir documentation.

### 

### **5.3.2 Atoms**

A special data type that Elixir supports is called an atom. Atoms are named constants whose value is their name. As shown in Figure 3, you can create an atom using the colon (:) character, followed by alphanumeric characters and underscores.



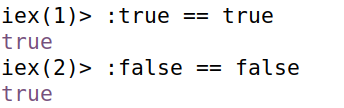
[**Figure 3:** Creates an atom called :this\_is\_an\_atom. Code made by Dhruv Rana using Ubuntu VM Elixir shell 1.13.4]

According to Sas̄a Jurić in his book *Elixir in Action*, “An atom consists of two parts, the text and the value. The text consists of the characters following the colon, which at runtime is kept in the atom table. The value is bound to the variable through a reference to the atom table instead of the text.” (Jurić 34). This means that when an atom is bound to a variable, the variable has access to the location of the atom in the atom table, but not the value itself, so the atom value is not lost when a variable is rebound with a new value. When an atom of the same name is created again, it would reference the last instance of the atom in the atom table, so no duplicate atoms can be created. Since the variable only contains the reference location of the atom, less memory is used to store an atom. For more detailed information on the Elixir’s atom data type, please visit the Hexdocs Elixir documentation.

### 

### **5.3.3 Booleans**

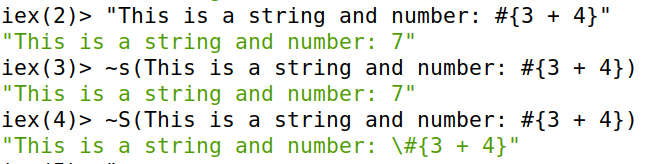
Elixir also supports booleans, however, according to Elixir’s written documentation in Hexdocs, “Elixir does not have a dedicated boolean type, but instead uses atoms: true and :false.” (Hexdocs 1). Elixir supports true and false as booleans without having to reference true and false as atoms with a colon. As shown in Figure 4, a boolean is an atom with the value of true or false as such, the atom :true is equal to the boolean value true, and the atom :false is equal to the boolean value false. For more detailed information on the Elixir’s boolean data type, please visit the Hexdocs Elixir documentation.



[**Figure 4:** Demonstrates atom :true equal to boolean value true, and atom :false equal to boolean value false. Code made by Dhruv Rana using Ubuntu VM Elixir shell 1.13.4]

### **5.3.4 Strings**

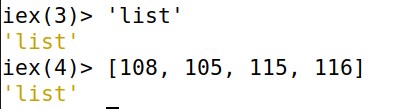
Strings in Elixir work a little differently from strings in other programming languages like Java. In fact, Elixir does not have a dedicated string type, instead “...strings are represented by using either a binary or a list type.” (Jurić 45-46). As shown below in Figure 5, strings in Elixir can be made using double quotes around the desired characters for single lined strings. Binary strings support escape characters and embedded string expressions using #{} where the expression is placed within the curly brackets. Elixir also provides another syntax for declaring strings using something called sigils. Sigils are an alternative syntax for working with literal. Figure 5 demonstrates how the characters enclosed inside ~s( ) are strings, but the upper case ~S() can also be used to create strings but ~S() does not support escape characters and embedded string expressions as indicated by the ~S() sigil printing the embedded string expression #{3 + 4}. For more detailed information on Elixir’s string data type, please visit the Hexdocs Elixir documentation.



[**Figure 5:** Demonstrates various ways to declare a string. Code made by Dhruv Rana using Ubuntu VM Elixir shell 1.13.4]

#### **5.3.4.1 Character Lists**

An alternative way to represent strings is through a character list. A character list is a list of integers, where each integer in the list represents an ASCII value of a character. However, character strings are not compatible with binary strings, and String module operations. As shown in Figure 6, a list is created with the integer values 108, 105, 115, and 116, which then corresponds to the letters ‘l’, ‘i’, ‘s,’ ‘t’, to form the character list ‘list’. You can also create a character list simply by using single quotes around the desired characters. An important thing to note however is that simple string operations like concatenation are not possible with character lists. For more detailed information on the Elixir’s character lists, please visit the Hexdocs Elixir documentation.

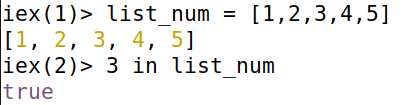


[**Figure 6:** Showcases how to create a string using character lists. Code made by Dhruv Rana using Ubuntu VM Elixir shell 1.13.4]

### 

### **5.3.5 Lists, Tuples, and Binaries**

Elixir also supports lists, tuples, and binaries. Lists are used to store a collection of data. As shown in Figure 7, lists can be defined by binding them to a variable where each element in the list is separated by commas. The variable list\_num is bound to a list containing the values 1, 2, 3, 4, and 5. You can perform operations on a list and manipulate the data with a list. A basic operation is to check if a value is in a list. As shown in Figure 7, after the creation of the list, the next line of the code checks if the value 3 is in the list list\_num using the ‘in’ command, which is true as 3 is within the list. For more detailed information on Elixir lists, please visit the Hexdocs Elixir documentation.



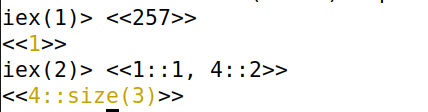
[**Figure 7:** Demonstrates how to create a list. Code made by Dhruv Rana using Ubuntu VM Elixir shell 1.13.4]

According to Sas̄a Jurić in his book *Elixir in Action*, “Tuples are used to group a fixed number of elements together.” (Jurić 36). Figure 8 demonstrates an example where a tuple is used to group a pet’s name and its age together, where a tuple is created by enclosing data elements between curly brackets, where each element in a tuple is separated by commas. Tuples are very useful for organizing large sets of data and for pattern matching with different data types since you can group two different data types together. For more detailed information on Elixir tuples, please visit the Hexdocs Elixir documentation.



[**Figure 8:** Demonstrates how to create a tuple. Code made by Dhruv Rana using Ubuntu VM Elixir shell 1.13.4]

A binary is a special Elixir data type that represents a piece of bytes. Binaries allow you to match byte size to a value. Binaries allow you to specify the size of each value you use and tell the compiler to determine how many bits to use for the value. As shown in Figure 9, you can create binaries by enclosing the byte sequence between << and >> operators. If the byte value provided is bigger than 255, the value is reduced down to 0. As shown in Figure 9, the bytes value is 257 and it is reduced down to 1, which shows that byte values work in a sequence of 0 to 255. Binaries are helpful when you are working with bit level data, making it easier to manipulate the memory. For more detailed information on Elixir binaries, please visit the Hexdocs Elixir documentation.



[**Figure 9:** Demonstrates how to create a binary. Code made by Dhruv Rana using Ubuntu VM Elixir shell 1.13.4]

## 

## **5.4 Operators**

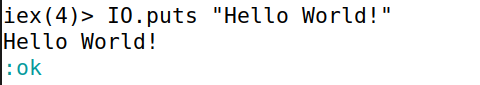
Elixir, like many programming languages, supports operators. Elixir posses standard arithmetic operators +, =, \*, and /, but also supports two special operators div and rem. The operator div gives the whole number of a division, without the remainder, while rem gives the remainder of a division. Elixir also supports logical operators such as and (&&), or (||), and not (!). For comparison operators, look at the Figure 10 below. Important comparison operators to look out for are the strict equality, strict inequality, weak equality, and weak inequality. They may look similar but in fact they are different. The difference between == and === is when it comes to comparing integers to floats. 1 == 1.0 is true because they hold the same value but under 1 === 1.0, the comparison is false because it is a comparison between an integer and float data types, both of which are different data types. For more detailed information on Elixir’s operators, please visit the Hexdocs Elixir documentation.

| ===, !== | Strict equality, inequality |
| --- | --- |
| ==, != | Weak equality, inequality |
| <, >, <=, >= | Less than, greater than, less than or equal, greater than or equal |

[**Figure 10:** Table of comparison operators. Made by Dhruv Rana]

# **Printing onto Screen**

Printing is an important feature to any programming language. To print a statement in Elixir, you must use built-in modules and functions. In Elixir, the Input-Output module or the IO module can be used to perform various I/O functions such as taking user input. The puts function in the IO module can be used to print onto the screen. In Figure 11, a print statement is shown printing “Hello World” with the IO module name followed by the dot (.) operator and the puts function and lasting followed by the characters, Hello World, in double quotes that are printed. For more detailed information on Elixir’s IO module, the puts functions, and printing in general, please visit the Hexdocs Elixir documentation.

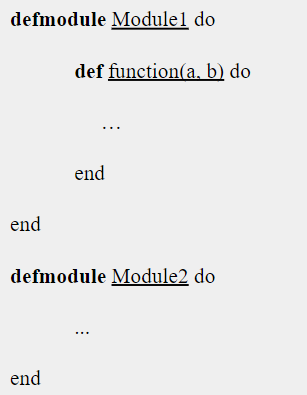


[**Figure 11:** Shows printing “Hello World”. Cody made by Dhruv Rana using Ubuntu VM Elixir shell 1.13.4]

# **Modules and Functions**

As a functional language Elixir has built-in modules and functions. Elixir possesses built-in modules which can be used through Elixir’s standard libraries. An example of a built-in module and function include the IO module and the puts function used to print a message onto screen. As shown below in Figure 12, to define your own module, use the defmodule construct, and then followed by the module name and the command do for action or start. The module name must start with an uppercase letter, consisting of alphanumeric characters, underscores, and the dot character, and generally written in CamelCase style. According to Sas̄a Jurić in his book *Elixir in Action*, “The dot character is a convenience that can be used to organize modules in hierarchical relationships but there is no hierarchical relationship between modules…” (Jurić 23). This means that the dot character is used in the module name for visual aid rather than giving a module any special precedence or rights. It holds little weight on the module. For more detailed information on Elixir’s modules, please visit the Hexdocs Elixir documentation.

Within a module is a function. Every Elixir function must be defined inside a module, and follow the same naming convention as variables and can also end in the characters ? and !. The ? character is conventionally used to indicate a function that returns true or false, and the ! character is conventionally used to indicate a function that returns an runtime error. As shown in Figure 12, to define a function, use the def construct followed by the function parameters in parentheses. If a function has no parameters, parentheses can be ignored. Additionally, when a function is called, it must refer to the module it belongs to in its syntax, for example, IO.puts(“Hello World”), where IO is the module and puts is the function name. Something to note is that both the module and function bodies are closed using the keyword end. Figure 12 also defines another module name Module2, which means that an Elixir file can hold multiple modules and multiple functions within a module. For more detailed information on Elixir’s functions, please visit the Hexdocs Elixir documentation.

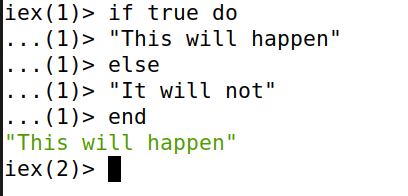


[**Figure 12:** Shows a module and a function shell. Code from *Elixir in Action* by Sas̄a Jurić on pages 22-23]

# **Control Flow and Loops**

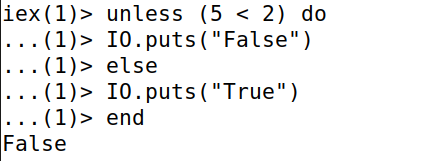
## **8.1 Control Flow**

Elixir supports control flow through the use of if-else, unless, and case statements. If-else statements allow one or the other branch of code to execute depending on if the condition is true or false. In Figure 13, an if-else statement is defined where the if condition is always true due to the true boolean value for the condition. This results in the string “This will happen” to be returned. If the if-expression condition is not satisfied, and no else condition is present, it returns the atom *nil*, similar to null in languages like C/C++. For more detailed information on Elixir’s if-else statement, please visit the Hexdocs Elixir documentation.



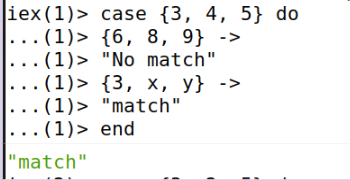
[**Figure 13:** Demonstrates if-else statement with conditions. Code made by Dhruv Rana using Ubuntu VM Elixir shell 1.13.4]

The unless statement works opposite of the if-else statement. Where the if-else statement evaluates the expression when the condition is true, the unless condition evaluates the expression when the condition is false. In Figure 14, the unless statement condition is 5 < 2 which is false so the expression is evaluated printing “False” onto the screen. For more detailed information on Elixir’s unless statement, please visit the Hexdocs Elixir documentation.



[**Figure 14:** Demonstrates unless statement with conditions. Code made by Dhruv Rana using Ubuntu VM Elixir shell 1.13.4]

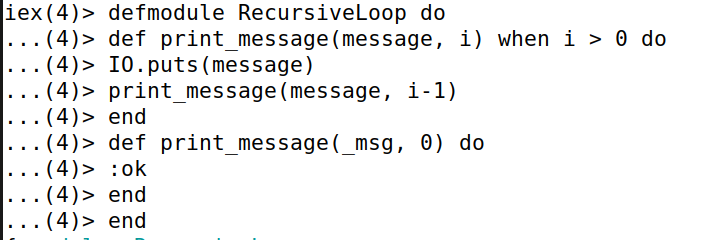
The case statement is used for pattern matching. In a case statement the given expression is compared with the given clauses for a match. If a match is found, the condition is true and the conditions expression is executed. In Figure 15, a case statement has a given expression with the values {3, 4, 5} that do not match the first given clause {6, 8, 9}, but do match with the second given clause {3, x, y} returning the string “match”. This is because the value 3 is in the same location in both expressions, and x and y are variables so the values 4 and 5 can be bound to them, creating a pattern match. If no clauses match with the given expression, an error is raised. For more detailed information on Elixir’s case statement, please visit the Hexdocs Elixir documentation.



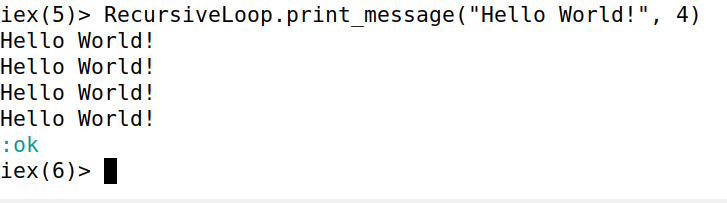
[**Figure 15:** Demonstrates case statement with conditions. Code made by Dhruv Rana using Ubuntu VM Elixir shell 1.13.4]

## **8.2 Loops**

Elixir also supports loops, but in the form of recursive loops. In Elixir, there are not built-in loop constructs like a for-loop, or a while-loop. This is because the data in Elixir is immutable, as such, looping in Elixir is heavily focused on recursive looping and function calls. In Figure 16, a function called RecursiveLoop is created, which has a function called print\_message, with the parameters for a message, and a counter i, which prints a message when the counter i is greater than 0, and after printing the message the counter i is reduced by 1. This part of the code creates the conditions for looping. Following that, the same function print\_message is defined again which checks when the courter i is equal to 0, to stop the loop. In Figure 17, the function print\_message is called with the parameters “Hello World!” for the message and a counter of 4. After the function is called, “Hello World!” is printed 4 times. For more detailed information about Elixir loops, please visit the Hexdocs Elixir documentation.



[**Figure 16:** Creates a recursive loop through a module and a function. Code made by Dhruv Rana using Ubuntu VM Elixir shell 1.13.4]



[**Figure 17:** Calls a recursive loop module and function to print “Hello World!” 4 times. Code made by Dhruv Rana using Ubuntu VM Elixir shell 1.13.4]

# **Features**

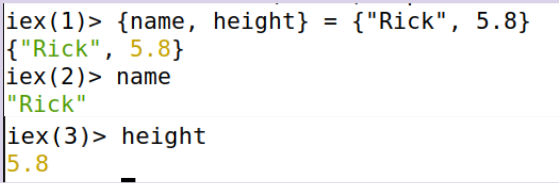
## **9.1 Immutability**

Elixir data is immutable meaning once the data is assigned, the data value in memory will not be modified. Rebinding the value to a new variable will not change the existing memory location as the system will use new memory to create a slot for the new variable and point to the newly created location. The two main benefits of immutability are side-effect-free functions and data consistency. Side-effect-free functions mean that the function will no longer change any value outside of its scope. This makes code more predictable and easier to understand and test. Data consistency means that you receive expected outputs from the functions. This means that you hold all versions of a data structure in the program, making it possible for you to perform detailed operations on individual memory locations. For more detailed information on Elixir’s immutable data, please visit the Hexdocs Elixir documentation.

## 

## **9.2 Pattern Matching**

Pattern matching is a key feature of Elixir. Elixir allows for pattern matching because the = operator is not assignment but a binding of the variable to the value. “The operator = is called the match operator.” (Hexdocs 4). The pattern matching expression consists of two parts, the pattern or variable, on the left side, and the term or the value on the right side. As shown in Figure 18 below, tuples are used for pattern matching, where the left hand side of the match operator is the pattern, the tuples name and height and the right hand side of the match operator is the term, Rick and 5.8. This establishes a pattern where the variable name is bound to Rick, and the variable height is bound to 5.8. When name and height are called it returns “Rick” and 5.8. Pattern matching is useful because it allows you to bind multiple variables with values in a tuple or list, making it easier to manipulate the data within a list or tuple. For more detailed information on pattern matching in Elixir, please visit the Hexdocs Elixir documentation.



[**Figure 18:** Pattern matching with tuples. Code made by Dhruv Rana using Ubuntu VM Elixir shell 1.13.4]

## **9.3 Concurrency and Processes**

Elixir uses processes and supports the creation of processes by programmers. Elixir supports concurrent and parallel processes. Concurrency is when multiple processes run at overlapping times, whereas parallelism is when processes run at the same time. Elixir’s concurrency relies on what is known as Actor models. According to the Elixir School, a teaching environment for Elixir, “Actors are a contained process that communicates with other processes through message passing.” (Elixir School 1). In a sense they are isolated processes that communicate with other processes by using send and receive messages. In fact, all Elixir processes go through BEAM. BEAM processes are lightweight unlike OS processes or threads. BEAM uses schedulers to determine process executions, running tasks in different processes. This improves BEAM’s fault tolerance with Elixir in the form of fault detection and correction of failed processes. Figure 19 shows how to start a process. A process is created using the spawn function. Before creating a process, make the program for the process to do something. In Figure 19, the module Example defines a message passing process scenario, where the function listen uses the built-in receive function to wait to receive the tuple input {:ok, “hello”}, and then print the statement “World”. The function listen() is recursive in the module so that multiple messages can be handled. After defining the message passing, create a process with the spawn function bound to the variable pid, where the parameters for spawn are the module name, and function name. “PID is a process identifier which is created when a process is created.”(Elixir School 1). In Figure 19, the process identifier is <0.108.0>. After creating the process, the send function is used to send the inputs {:ok, “hello”} to the process, printing the statement “World”. For more detailed information on concurrency and processes in Elixir, please visit the Hexdocs Elixir documentation.



[**Figure 19:** Creates a message passing process. Code from Elixir School, concurrency lesson on page 1]

# **Future Direction and Evaluation**

The Elixir’s syntax is pretty simple and easy to understand. Additionally, pattern matching and immutable data support each other, which makes working with complex data like lists and tuples much easier. Not to mention that since Elixir runs on BEAM, it gains much from both BEAM and Erlang, such as since BEAM supports isolated lightweight processes it allows Elixir to be a great choice for running concurrent processes, but Elixir’s speed is also affected since Elixir programs run on BEAM instead of an OS which results in much slower speeds compared to languages like C/C++. I can see Elixir rising in popularity for mainstream web development, telecommunications, and multimedia networking due to features like pattern matching and Elixir’s support for running concurrent processes. Erlang also provides Elixir with many of its own library resources which greatly increases Elixir’s available resources. But Elixir lacks when it comes to its own individual libraries. Elixir is far too reliant on Erlang and BEAM, to become an independent language. In fact, this is one of the reasons I found installing Elixir to be quite complex because it requires you to download packages for both languages just to write code in one of them, and then having to connect the two languages together. This takes up a lot of hard drive storage. This is also a reason why I opted to use an Ubuntu VM as installing Elixir on Ubuntu VM is much easier compared to the conventional way. Another one of Elixir’s weaknesses is that Elixir still has a smaller community, and being a functional programming language, it is hard to compete with object oriented languages as they have become more mainstream. Having a smaller community is not a complete downside, it just means the language is still developing and growing. My evaluation of Elixir is that it still has room to grow.

# **Citations**

“Elixir-Lang.github.com.” *Elixir-Lang.github.com*, elixir-lang.org/. Accessed 10 Dec. 2023.

“Home · Elixir School.” *Elixirschool.com*, elixirschool.com/en. Accessed 10 Dec. 2023.

“Introduction — Elixir V1.16.0-Rc.1.” *Hexdocs.pm*, hexdocs.pm/elixir/1.16/introduction.html. Accessed 13 Dec. 2023.

“Ruby on Rails.” *Ruby on Rails*, rubyonrails.org/. Accessed 12 Dec. 2023.

Sas̄a Jurić. *Elixir in Action*. Shelter Island, Ny, Manning, 2015.