

Amrita Vishwa Vidyapeetham  
TIFAC-CORE in Cyber Security

**20CYS312 - Principles of Programming Languages**  
**Assignment-01: Exploring Programming Paradigms**

S.Dharmik

21st January, 2024

## Paradigm 1: Meta-Programming

"Metaprogramming is a programming technique in which computer programs have the ability to treat other programs as their data. It means that a program can be designed to read, generate, analyze or transform other programs, and even modify itself while running."

Lisp, a family of programming languages known for its flexible and powerful metaprogramming capabilities, has heavily influenced Julia in this regard.

Representation of Code as Data Structures:

- In Julia, code is not just a sequence of characters; it is represented as data structures within the language itself. This means that the code can be treated as an object that can be created, modified, and manipulated programmatically.
- The ability to represent code as data structures allows for more dynamic and powerful manipulation of the code during runtime.

Since code in Julia is represented by objects that can be created and manipulated within the language, programs can transform and generate their own code dynamically. This is a powerful feature that enables sophisticated code generation without the need for extra build steps.

Lisp-Style Macros:

- Julia supports true Lisp-style macros, which operate at the level of abstract syntax trees (ASTs). Macros in Julia can generate and manipulate code at the AST level, providing a high level of flexibility and expressiveness.
- Lisp macros are known for their ability to transform and generate code at compile time, and Julia inherits this capability.

Contrast with Preprocessor Macros (C and C++): The statement contrasts Julia's metaprogramming with preprocessor "macro" systems in languages like C and C++. In C and C++, macros are essentially textual substitutions performed before the actual parsing or interpretation of the code. Julia's metaprogramming, on the other hand, operates at a higher level, manipulating the abstract syntax tree rather than performing simple text substitutions.

---

## Metaprogramming in Programming

Metaprogramming can be a powerful tool in programming, allowing a program to operate on other programs as well as on itself. Two key concepts in the metaprogramming world are *introspection* and *reflection*. Introspection involves a program using metaprogramming to analyze and report on itself, while reflection involves using metaprogramming to apply modifications.

When a program utilizes metaprogramming to operate on other programs, it can write various functions, including:

- Code Analysis
- Program Maintenance
- Configuration
- Assemblers
- Compilers
- Debuggers
- Interpreters

## Operations in Metaprogramming

Metaprogramming can perform a variety of operations, such as:

- Finding out what file the program is in
- Determining which function is currently running
- Defining the thread that is currently running
- Determining the class, properties, and constructed function of given objects
- Viewing loaded classes and their fields and methods
- Defining anonymous functions and function arguments
- Figuring out class definitions
- Finding out the method body

## Examples of metaprogramming

include template metaprogramming, C++ metaprogramming, and Elixir metaprogramming. These are different types of metaprogramming. This can be supported by different types of programming languages such as

- Python
- Ruby
- Javascript ,
- Java
- Clojure , Go , Julia .. each with its functions and features.

---

## Pros of Metaprogramming

There are many advantages when it comes to metaprogramming. The biggest one is that metaprogramming mainly does:

- **Self-Writing Programs:** Metaprogramming allows programs to generate their own code. This is particularly useful for tasks that involve repetitive or boilerplate code, as the program can dynamically create the required code based on certain conditions or parameters.
- **Error Reduction:** Since metaprogramming generates code automatically, there's less room for human error in the coding process. This can lead to more robust and reliable programs.

Other pros follows as this :

1. **Time Savings:** Automated code generation can save programmers significant time, especially for tasks that involve complex schemas.
2. **Faster Market Turnaround:** The efficiency gained from metaprogramming contributes to a faster turnaround for application releases.
3. **Architecture Stability:** Metaprogramming can contribute to the stability of the overall software architecture.
4. **Reduced Code Repetition:** Metaprogramming helps in eliminating redundant code by dynamically creating it when needed. This reduces the amount of code developers need to write and maintain.
5. **Code Generation in Workflow:** Metaprogramming seamlessly integrates with various workflows, adding flexibility and adaptability to the development process.
6. **Boilerplate Code Assistance:** Metaprogramming is effective in handling boilerplate code, which is the repetitive and standardized code that needs to be included in multiple places.

## Cons of Metaprogramming

Users should, nevertheless, be aware of some drawbacks involved with metaprogramming. A significant disadvantage to many who want to step into the world of metaprogramming is syntax.

- **Steep Learning Curve:** Since metaprogramming primarily creates programs that write other programs, the syntax can be quite complicated, and the learning curve for metaprogramming can be steep.

Other Cons follows as this:

1. **Incorrect Use:** Due to its complexity, metaprogramming can be prone to incorrect use. Developers may make mistakes in generating code, leading to unexpected errors that are difficult to handle.
2. **Limited Applicability:** Metaprogramming may not be beneficial for all types of applications. Certain projects or programming paradigms may not align well with metaprogramming concepts, limiting its usefulness.
3. **Inflexibility in Code Generation:** Generating code through metaprogramming can sometimes be inflexible, especially when dealing with complex logic or dynamic requirements.
4. **Longer Compilation Times:** The process of metaprogramming can result in longer compilation times. This can impact development efficiency, especially in larger codebases.
5. **Potential for Vulnerabilities:** Incorrectly implemented metaprogramming can introduce vulnerabilities and risks within the system.

---

## Language for Paradigm 1: Julia

### Why Julia ???

Julia is a high-performance programming language aimed at scientific computation and data processing. Julia Originally developed by a group of computer scientists and mathematicians at MIT led by Alan Edelman.

Among its competitors, C/C++ is extremely fast and the open-source compilers available for it are excellent, but it is hard to learn, in particular for those with little programming experience, and cumbersome to use. Julia delivers its swift numerical speed thanks to the reliance on a LLVM (Low Level Virtual Machine)-based JIT (just-in-time) compiler.

Not need to “compile” Julia in the way you compile other languages to achieve lightning-fast speed. Julia incorporates in its design important advances in programming languages –such as a superb support for parallelization or practical functional programming orientation– that were not fully fleshed out when other languages for scientific computation were developed a few decades ago.

Among other advances that we will discuss in the following pages, we can highlight multiple dispatching (i.e., functions are evaluated with different methods depending on the type of the operand). **We will mainly discuss about metaprogramming through Lisp-like macros (i.e., a program that transforms itself into new code or functions that can generate other functions), and the easy interoperability with other programming languages (i.e., the ability to call functions and libraries from other languages such as C++ and Python).**

These advances make Julia a general-purpose language capable of handling tasks that extend beyond scientific computation and data manipulation.’

- While Julia’s ecosystem is not as mature as C++, Python or R’s, the growth rate of the penetration of the language is increasing.
- It is true that Julia’s basic syntax (definition of vectors and matrices, conditionals, and loops) is, by design, extremely close to Matlab’s.
- Julia (quite sensibly) passes arguments by reference and not by value as Matlab.

you should try to make an effort to understand how Julia allows you to do many new things and to re-code old things in more elegant and powerful ways than in Matlab.

## Fundamentals of Julia

### Types , Variables , Values ...

Julia has variables, values, and types. A variable is a name bound to a value. Julia is case sensitive: a is a different variable than A .A value is a content (1, 3.2, “economics”, etc.).

Technically, Julia considers that all values are objects (an object is an entity with some attributes). This makes Julia closer to pure object-oriented languages such as Ruby than to languages such as C++, where some values such as floating points are not objects. Finally, values have types (i.e., integer, float, boolean, string, etc.). A variable does not have a type, its value has. Types specify the attributes of the content.

```
Julia 1.10.0
Documentation: https://docs.julialang.org
Type "?" for help, "]" for Pkg help.
Version 1.10.0 (2023-12-25)
Official https://julialang.org/ release

julia> methods(+)
# 189 methods for generic function "+" from Base:
[1] +{x::Dates.CompoundPeriod, y::Dates.CompoundPeriod}
    @ Dates D:\Julia\Julia-1.10.0\share\julia\stdlib\v1.10\Dates\src\periods.jl:334
[2] +{x::Dates.CompoundPeriod, y::Dates.TimeType}
    @ Dates D:\Julia\Julia-1.10.0\share\julia\stdlib\v1.10\Dates\src\periods.jl:362
[3] +{x::Dates.CompoundPeriod, y::Dates.Period}
    @ Dates D:\Julia\Julia-1.10.0\share\julia\stdlib\v1.10\Dates\src\periods.jl:332
```

Figure 1: methods for addition symbol

```
julia> a = 10
10

julia> typeof(a)
Int64

julia> sizeof(a)
8

julia> b = 4 + 3im
4 + 3im

julia> typeof(b)
Complex{Int64}
```

Figure 2: Commands in julia

Functions in Julia will look at the type of the values passed as operands and decide, according to them, how we can operate on the values (i.e., which of the methods available to the function to apply). Adding  $1+2$  (two integers) will be different than summing  $1.0+2.0$  (two floats) because the method for summing two integers is different from the method to sum two floats. In the base implementation of Julia, there are 230 different methods for the function `sum`!

This application of different methods to a common function is known as polymorphic multiple dispatch. This is very important feature .

Being a typed language means that the type of each value must be known by the compiler at run time to decide which method to apply to that value. Being a dynamically typed language means that such knowledge can be either explicit (i.e., declared by the user) or implicit (i.e., deduced by Julia with an intelligent type inference engine from the context it is used). Dynamic typing makes developing code with Julia flexible and fast. Julia is similar to Ruby , Python ..

Julia follows a promotion system where values of different types being operated jointly are “promoted” to a common system: in the sum between an integer and a float, the integer is “promoted” to float.

Julia has a flexible specification for functions (including abstract ones), MapReduce (a particular set of functions), loops, and conditionals. Julia also offers other data structures such as sets , tuples , dictionaries etc ...

When similar logic should be applied to different kinds of objects (i.e., different types), you can write functions that share the same name but have different types or different numbers of parameters (and different implementation).

One of features of julia is multiple-dispatch polymorphism is a generalization of objectoriented runtime polymorphism, where the same function name performs different tasks, depending on which is the object’s class.

---

## Metaprogramming in Julia

Metaprogramming is a concept where by a language can express its own code as a data structure of itself. For example, Lisp expresses code in the form of Lisp arrays, which are data structures in Lisp itself. Similarly, even Julia can express its code as data structures.

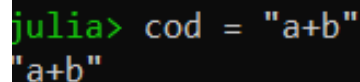
Everything in Julia is an expression that returns a value when executed. Every piece of the program code is internally represented as an ordinary Julia data structure, also called an expression. In this chapter, we will see that by working on expressions, how a Julia program can transform and even generate the new code, which is a very powerful characteristic, also called **homoiconicity**. It inherits this property from Lisp, where code and data are just lists, and where it is commonly referred to with the phrase: “code is data and data is code”.

You will study the life of a Julia program and how it is actually represented and interpreted by Julia. You will also learn what is meant by “a language expressing its own code as a data structure of itself.”

### Procedure

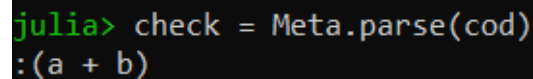
Firstly, it is very important to know that every Julia program starts out as a string. Let’s consider a short program for adding two variables as our Julia code and use it to learn how Julia interprets programs:

```
cod = "a + b"
```



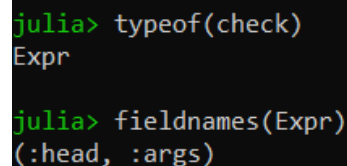
```
julia> cod = "a+b"  
"a+b"
```

if you parse the preceding string code, it would return an object of type Expression. Let’s check it by actually parsing an example Julia program and checking for its type: `check = parse(cod)`



```
julia> check = Meta.parse(cod)  
:(a + b)
```

The parsed string `check` is a Julia expression. You can verify that by checking the type of `check` using the `typeof()` function in Julia and It will return `Expr` as output (Expression).let’s take a close look at the expression object. To look at the different parts of an expression, you can use the `fieldnames()` function.



```
julia> typeof(check)  
Expr  
  
julia> fieldnames(Expr)  
(:head, :args)
```

---

The expression object consists of three parts: the head and the arguments. Each argument can be checked in the following ways: The head of the expression can be checked using this "check.head" and The arguments of the expression can be checked using this "check.args" This returns an array of the arguments used in the code.

```
julia> check.head
:call

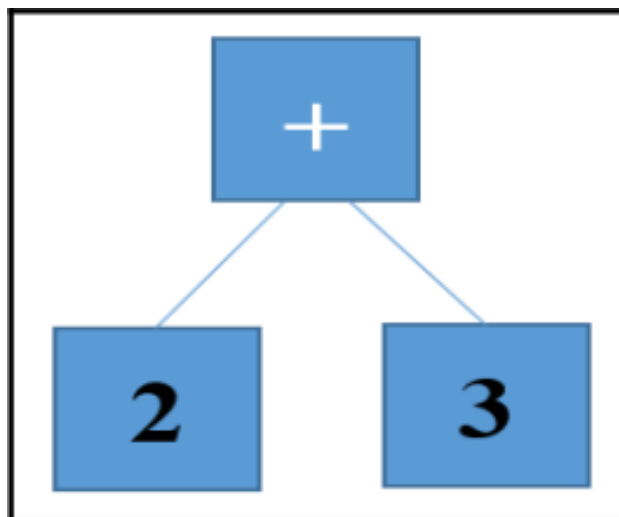
julia> check.args
3-element Vector{Any}:
 :+
 :a
 :b
```

You can clearly infer that the code in Julia is represented as a data structure of the language itself internally. Now, let's take a look at the dump() function, which gives a nice, annotated display of the check expression . It can be done using this "dump(check)".

```
julia> dump(check)
Expr
 head: Symbol call
 args: Array{Any}((3,))
  1: Symbol +
  2: Symbol a
  3: Symbol b
```

An abstract syntax tree (AST) is a tree representation of the abstract syntactic structure of source code written in a programming language. When Julia code is parsed by its LLVM JIT compiler, it is internally represented as an abstract syntax tree. The nodes of this tree are simple data structures of the type expression Expr. We can visualize the hierarchical nature of an expression with the help of dump() function.

An expression is simply an object that represents Julia code. For example,  $2 + 3$  is a piece of code, which is an expression of type Int64. Its syntax tree can be visualized as follows:



We will later explore how and where the details of the program are stored in the Expression objects and how to make use of them.

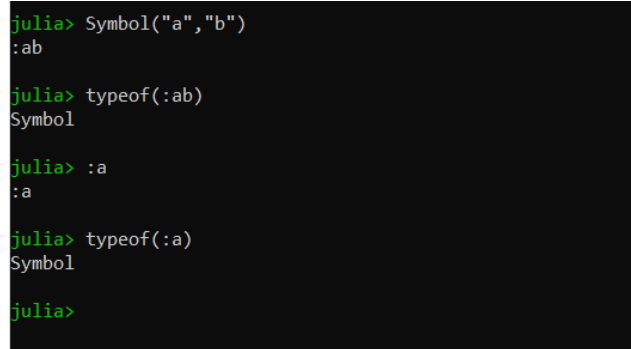
---

## Symbols and Expressions

A Symbol in Julia is denoted by a leading colon (:) followed by a valid identifier, such as :x, :myvariable, etc. Symbols are immutable and unique, meaning that any two symbols with the same identifier represent the same object in memory. Symbols are not evaluated; they represent names or labels in the code.

Symbols create interned strings that are used for building expressions. An interned string is an immutable string that is used during string processing for optimizing time and space. The character : is used to create symbols. So, a symbol always starts with a : symbol.

- Symbols are interned string identifiers. These are the heads of expression objects. Interning a string is a method where one copy of a string is stored. This makes the tasks that need to process the string both time- and space-efficient. Interned strings are also immutable. So, symbols are especially used for performing string processing tasks.
- Symbols can be constructed in two ways. The first way is through the : character, whose significance would be discussed in detail in the next section. Another way to do this is by calling the symbol() function



```
julia> Symbol("a","b")
:ab

julia> typeof(:ab)
Symbol

julia> :a
:a

julia> typeof(:a)
Symbol

julia>
```

Expressions are typically instances of the Expr type in Julia and have a hierarchical tree structure. The Expr type has two main components: the head (a symbol representing the type of expression, e.g., :call, :function, :if) and arguments (an array of sub-expressions or values). Expressions are used to represent the structure of code, and they can be evaluated by the Julia interpreter.

## Ways of Creating Expressions

**Quoting:** The usage of a semicolon to represent expressions is known as quoting. The characters inside the parentheses after the semicolon constitute an Expression object.

**Colon Prefix Operator:** Expressions can be created using the same colon : prefix operator you saw for individual symbols, this time applied to a whole expression (given in brackets)

```
expr = :(a+1)
```

Using this following method, multiple expressions can be represented as a block by quoting them.

**Quote Block:** An alternative to the :( [...]) operator is the quote [...] end block:

```
expr = quote b = a+1 end
```

---

**Expression Interpolation :** Any Julia code which has string or expression is usually unevaluated but with the help of dollar (\$) sign (string interpolation operator), we can evaluate some of the code. The Julia code will be evaluated and inserts the resulting value into the string when the string interpolation operator is used inside a string.



```
julia> a = 1
1
julia> ex = :($a + b)
:(1 + b)
julia>
```

One important feature of such an interpolation is that the expression evaluation evaluates at parse time, whereas other interpolations evaluate only when the `eval()` function is called after parse time.

Once you parsed the expression, there is a way to evaluate the expression also. We can use the function `eval()` for this purpose.

The evaluation happens at the global scope, even if the `eval()` call is done within a function. That is, the expression being evaluated will have access to the global variables but not to the local ones.

```
julia> p = 2
2
julia> q = 3
3
julia> exp = :(p+q)
:(p + q)
julia> eval(exp)
5
```

## Macros

We are now aware of creating and handling unevaluated expressions. In this section, we will understand how we can modify them. Macros are like functions, but instead of values, they take expressions (which can also be symbols or literals) as input arguments. When a macro is evaluated, the input expression is expanded, that is, the macro returns a modified expression. This expansion occurs at parse time when the syntax tree is being built, not when the code is actually executed.

```
julia> macro expFeatures(expression)
    if typeof(expression) == Expr
        println(expression.args)
        println(expression.head)
    end
    answer = eval(expression)
    return answer
end
@expFeatures (macro with 1 method)

julia> @expFeatures 3+4-5
Any{::Integer, ::Integer, Integer}, 5
call
2
```

A lot of macros are predefined in the Julia compiler. One of the most preferred is to time the execution of a code. For Example `@time` macro.

The `macroexpand()` function returns the expanded format (used by the Julia compiler before it is finally executed) of the specified macro. We can view the quoted return expression using the function `macroexpand`.

```
julia> ex = macroexpand(Main, :(@expFeatures(3+4-5)))
Any{::, : (3 + 4), 5}
call
2
```

Macros are necessary because they execute when code is parsed, therefore, macros allow the programmer to generate and include fragments of customized code before the full program is run.

```
julia> macro twostep(arg)
    println("I execute at parse time. The argument is: ", arg)
    return :(println("I execute at runtime. The argument is: ", $arg))
end
@twostep (macro with 1 method)

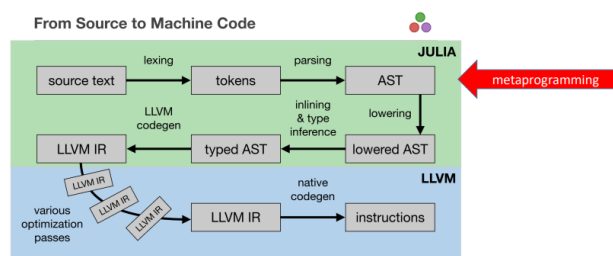
julia> ex = macroexpand(Main, :(@twostep (:1, 2, 3)))
I execute at parse time. The argument is: (:1, 2, 3)
:(Main.println("I execute at runtime. The argument is: ", $(Expr(:copyast, :($(QuoteNode(:1, 2, 3)))))))

julia> ex
:(Main.println("I execute at runtime. The argument is: ", $(Expr(:copyast, :($(QuoteNode(:1, 2, 3)))))))

julia> eval(ex)
I execute at runtime. The argument is: (1, 2, 3)
```

This macro takes a single argument `arg`. It prints a message at parse time (during macro expansion) and then returns a new expression that includes a print statement to be executed at runtime. The `$arg` syntax is used for interpolation, allowing the runtime value of `arg` to be inserted into the returned expression.

The macro expansion triggers the parse-time code, printing: **I execute at parse time. The argument is: ((1, 2, 3))**. This shows that the parse-time code inside the macro is executed, and it prints the macro argument. The macro returns a new expression with runtime code: **:(println("I execute at runtime. The argument is: ", (1, 2, 3)))**. This is the expression that will be executed at runtime. It includes the runtime code to print a message along with the runtime value of the original macro argument. This demonstrates the separation between parse-time and runtime code execution in Julia macros. The parse-time code is executed during macro expansion, while the runtime code becomes part of the generated code to be executed when the program runs.



Macros, as they are generally used in other programs such as Excel, are used for ready-made and quick execution of computations. In this case, a macro is also used similarly. They are used for using generated code for the body of a program. This takes care of the ready-made feature of macros. They also mask the returned expressions to a tuple of arguments so that they can be compiled directly and to avoid the `eval()` function. This would help in automating repetitive tasks easily.

---

## Paradigm 2: Declarative

There are mainly two paradigms, Imperative and Declarative. Logic Programming, Functional Programming and Database Programming comes under declarative paradigm. But we mainly discuss about Declarative Paradigm in this. Declarative programming is a powerful paradigm in computer science that allows developers to express the desired outcome of a program rather than explicitly describing the steps to achieve it. This method of programming focuses on the higher-level objectives of software and promotes greater code readability, modularity, and scalability.

Declarative programming is a programming paradigm that focuses on expressing what the desired result should be, rather than providing a step-by-step solution (as in imperative programming). This style of programming offers many advantages, including increased readability, easier maintainability, and reduced complexity.

Here I will give a short note on the key topics in this :

- **Higher Order Abstractions:** Higher-order abstractions refer to constructs that allow you to express complex logic with simpler and more concise code. This promotes code reuse and modularity.
- **Immutable data:** Immutable data means that data structures cannot be changed after they are created. This prevents many potential bugs caused by unintended side effects and provides better memory management.
- **Pure Functions:** Pure functions are functions that always produce the same output when given the same input and have no side effects. These functions facilitate easier testing and debugging due to their predictability.
- **Declarative DSLs:** Declarative DSLs are programming languages designed specially for a specific domain, allowing you to express the domain requirements more concisely and naturally.



First of all What is declaration ?? In programming, a declaration refers to the process of specifying the properties of a variable, function, or other programming constructs, but not providing its implementation or initial value. This is an important concept in declarative programming as declarations help structure the program and reveal the expected behaviour without describing it in a detailed step-by-step manner. This creates a separation between what the program should achieve and how it should achieve it.

For example, in SQL (Structured Query Language), a common declarative language, you can declare a query to fetch data from a database without specifying the underlying algorithm or data traversal method used to retrieve the data.

Declarative languages come in to flavors: logic languages (expressions are relations) and functional languages (expressions are functions).

---

This paradigm focusses more on this :

- Separating domain logic from control flow
- Emphasizing code readability and expressiveness
- Minimizing state and side effects
- Utilizing recursion and pattern matching

By separating domain logic from control flow, declarative programming allows developers to focus on the problem at hand and express it in a more direct, natural manner. Readability and expressiveness are vital aspects of the declarative programming model, which aims to make it easier for developers to understand and maintain code

This is achieved by using clear, higher-order abstractions and reducing the need for complex logic or nested control structures. Minimizing state and side effects is essential for maintaining predictability and ensuring that code behaves as expected. This is accomplished by using pure functions, immutable data structures, and imposing constraints on data manipulation. Finally, to solve problems recursively and perform pattern matching, declarative programming often leverages functional programming techniques or specific control structures found in declarative languages. You will come to know about this in the coming pages.

## Examples of declarative programming languages

- HTML
- Erlang
- Prolog
- SQL
- CSS
- Xquery
- Lisp and many more ....

## Pros of Declarative Programming

Declarative programming offers several advantages:

- **Readability and Usability:** Declarative programming languages are often closer to natural language, making them more readable and easier to learn, even for non-programmers.
- **Commutativity:** Declarative programming allows expressing an end state without specifying the order of operations, which can simplify parallel programming.
- **Referential Transparency:** This concept, often associated with functional programming, minimizes manual handling of state and relies on side-effect-free functions, making code more predictable.
- **Succinctness:** Declarative languages tend to abstract away boilerplate code, allowing developers to write less code to achieve the same functionality.

- 
- **Minimized Data Mutability:** Immutability in declarative programming improves security and reduces errors.
  - **Reusability :** Declarative functions are typically reusable and can be shared across different parts of an application.

Still there are more advantages other than this ...

## Cons of Declarative Programming

**Complex Search and Backtracking:** Expressing problems involving complex search or backtracking can be cumbersome in a declarative context.

**Efficiency Overhead:** The translation of desired outcomes into executable processes may introduce efficiency overhead, potentially impacting performance.

**Steep Learning Curve:** The declarative paradigm demands a steeper learning curve, necessitating a mindset shift from the more common imperative programming approach.

**Abstraction Obscurity:** Abstraction, a core feature, can obscure understanding for those not well-versed in the language, leading to potential errors.

**Limited Applicability:** The paradigm's applicability is not universal, excelling most in domains that align closely with its strengths.

**Maintenance Challenges:** While abstractions simplify updates, they can pose comprehension barriers for new developers during maintenance, especially in the absence of the original authors.

With this , we can deduce that to maintain more concentration when dealing with declarative paradigm or approach.

## Applications of Declarative Programming

- **Database Querying :** SQL is a prime example where the desired data is specified without detailing the retrieval process.
- **Configuration Management :** Tools like Chef and Puppet use declarative code to manage software configurations.
- **User Interface Development :** Frameworks like React use a declarative approach to define UI components.
- **Functional Programming :** Languages like Haskell are used for tasks that benefit from high-level abstractions and mathematical functions.

In essence, declarative programming is leveraged in areas where the focus is on the outcome rather than the process, which can lead to more efficient and maintainable codebases.

Now we will going to understand Haskell programming language in a declarative approach .. Let's jump to that topic.

---

## Language for Paradigm 2: Haskell

Functional programming is a declarative programming paradigm used to create programs with a sequence of simple functions rather than statements.

All functions in the functional paradigm must be:

Pure: They do not create side effects or alter the input data

Independent from program state: The value of the same input is always the same, regardless of other variable values.

Each function completes a single operation and can be composed in sequence to complete complex operations. For example, we might have one function that doubles an input number, `doubleInput`, another that divides the number by pi, `divPi`.

Either of these functions can be used individually or they can be strung together such that the output of `doubleInput` is the input of `divPi`. This quality makes pieces of a functional program highly modular because functions can be reused across the program and can be called, passed as parameters, or returned.

## Why Haskell???

Haskell is a compiled, statically typed, functional programming language. It was created in the early 1990s as one of the first open-source purely functional programming languages and is named after the American logician Haskell Brooks Curry. Haskell joins Lisp as an older but useful functional language based in mathematics.

The Haskell language is built from the ground up for functional programming, with mandatory purity enforcement and immutable data throughout. It's mainly known for its lazy computation and memory safety that avoids memory management problems common in languages like C or C++.

Haskell belongs to the family of functional languages. It embodies in its core the concept of purity, separating the code with side-effects from the rest of the application. The evaluation model is based on laziness. Types are statically checked by the compiler. Also, Haskell features a type system which is much stronger and expressive than usual. Its approach to polymorphism is based on parametricity (similar to generics in Java and C) and type classes.

## Features of Haskell

### Memory Safe

Includes automatic memory management to avoid memory leaks and overflows. Haskell's memory management is similar to that of Golang, Rust, or Python

### Compiled

Uses the GHC Haskell compiler to compile directly to machine source code ahead of time. GHC is highly optimized and generates efficient executables to increase performance. It also has an interactive environment called GHCi that allows for expressions to be evaluated interactively. This feature is the key to Haskell's popularity for high input data analytics.

### Statically Typed

Has a static type system similar to Java that validates Haskell code within the environment. This lets you catch bugs during development earlier on. Haskell's great selection of types means you always have the perfect type for a given variable.

### Enforced Functional Best Practices

---

Enforces functional programming rules such as pure functions and immutable variables with error messages. This feature minimizes complexity in your program and ensures you're making the best use of its functional capabilities.

### Lazy Evaluation

Defers computation until the results are needed. Haskell is well known for its optimized lazy evaluation capabilities that make refactoring and function composition easy.

### Concurrency

Haskell makes concurrency easy with green threads (virtual threads) and `async` and `stm` libraries that give you all the tools you need to create concurrent programs without a hassle. Enforced pure functions add to the simplicity and sidestep many of the usual problems of concurrent programming.

### Libraries

Haskell has been open source for several decades at this point, meaning there are thousands of libraries available for every possible application. You can be certain that almost all the problems you encounter will have a library already made to solve them. Some popular additions are `Stack`, which builds and handles dependencies, and `Cabal`, which adds packaging and distribution functionality.

## Fundamentals of Haskell

The two most central concepts of Haskell are types and functions.

Types are collections of values that behave similarly, e.g., numbers or strings.

Functions can be used to map values of one type to another. We will use the Haskell notation

`f :: X -> Y`

to assert that `f` is a function taking arguments of type `X` and returning results of type `Y`. For example,

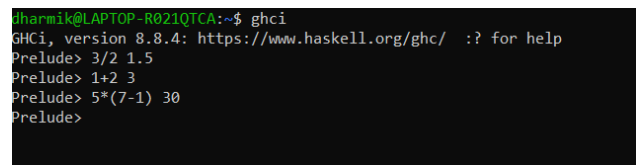
`sin :: Float -> Float`

`age :: Person -> Int`

`add :: (Integer,Integer) -> Integer`

`logBase :: Float -> (Float -> Float)`

Haskell has 3 common numeric types: `Int` for 64 bit (>20 digit) integers `Integer` list of `Int` types that can represent any number (similar to `BigInt` in other languages) `Double` for 64-bit decimal numbers. Each numeric type works with standard operators like `+`, `-`, and `*`. Only `Double` supports division operations and all `Integer` divisions will return as a `Double`.



```
gharmik@LAPTOP-R021QTCA:~$ ghci
GHCi, version 8.8.4: https://www.haskell.org/ghc/  :? for help
Prelude> 3/2 1.5
Prelude> 1+2 3
Prelude> 5*(7-1) 30
Prelude>
```

Haskell also includes predefined functions for common numerical operations like exponents, integer division, and type conversion.

String types represent a sequence of characters that can form a word or short phrase. They're written in double quotes to distinguish them from other data types, like "string string". Some essential string functions are:

Concatenation: Join two strings using the `++` operator

Reverse: Reverses the order of characters in a `String` such that the first character becomes the last.

```

dharmin@LAPTOP-R021QTC:~$ ghci
GHCi, version 8.8.4: https://www.haskell.org/ghc/  ?? for help
Prelude> import Data.Char Prelude Data.Char> "Hello"++"World" "HelloWorld"
Prelude Data.Char> reverse "Hello" "olleH"
Prelude Data.Char>

```

Tuples are simple. They are a group of elements with different types. Tuples are immutable, which means they have a fixed number of elements. They are useful when you know in advance how many values you need to store.

For example, (5, True) is a tuple containing the integer 5 and the boolean True. It has the tuple type (Int, Bool), representing values that contain first an Int value and second a Bool value.

```

dharmin@LAPTOP-R021QTC:~$ ghci
GHCi, version 8.8.4: https://www.haskell.org/ghc/  ?? for help
Prelude> ("First", "second", "third")("First", "second", "third")
Prelude> :t ("First", "second", "third")("First", "second", "third") :: ([Char], [Char], [Char])
Prelude> (1, "apple", pi, 7.2) (1, "apple", 3.141592653589793, 7.2)
Prelude> :t (1, "apple", pi, 7.2) (1, "apple", 3.141592653589793, 7.2)
:: (Floating c, Fractional d, Num a) => (a, [Char], c, d)
Prelude>

```

A list is a similar data structure to a tuple, but lists can be used in more scenarios than tuples. Lists are pretty self-explanatory, but you need to know that they are homogenous data structures, which means the elements are of the same type. You represent lists using square brackets, []. You use ++ when you want to concatenate two lists, so on both sides of the ++ operator you write lists.

```

dharmin@LAPTOP-R021QTC:~$ ghci
GHCi, version 8.8.4: https://www.haskell.org/ghc/  ?? for help
Prelude> [1,2,3][1,2,3]
Prelude> ['a','x'] "ax"
Prelude> [True,False,False] [True,False,False]
Prelude> let numlist = [3,1,0.5] Prelude> :t numlistnumlist :: Fractional a => [a]
Prelude> head [0,1,2,3,4,5] 0
Prelude> tail [0,1,2,3,4,5,6] [1,2,3,4,5,6]
Prelude> [1,2,3] ++ [4,5] [1,2,3,4,5]
Prelude>

```

## Custom Datatypes

Let us create our own datatype !!!Haskell also allows you to create your own data types similar to how we create functions. Each data type has a name and a set of expectations for what values are acceptable for that type. Suppose you want to create a structure that simulates a date. You need three integer values corresponding to the day, the month, and the year.

DateInfo is the name of your new type, and it is called a type constructor, which is used to refer to the type. The Date after the equal sign is the value constructor (or data constructor), which is used to create values of DateInfo type. The three Ints after Date are components of the type. Note that the name of the type constructor and the name of the value constructor begin with capital letters.

For information see Figure 2

To better understand this, take a look at the standard library's Bool type definition: Custom data types are marked with the data keyword and are named Bool by the following item. The = marks the boundary between name and accepted values. Then False | True defines that any value of type Bool must be either true or false.

For information see Figure 1



---

```
data bool = False | True
```

```
gharmik@LAPTOP-R021QTCA:~$ ghci
GHCi, version 8.8.4: https://www.haskell.org/ghc/  :? for help
Prelude> data DateInfo = Date Int Int Int Int Prelude> myDate = Date 1 10 2018
Prelude> :t myDate myDate :: DateInfo
Prelude> :info DateInfo
<interactive>:1:1: error: Not in scope: 'DateInfo'
Prelude> :info DateInfo DateInfo = Date Int Int Int Int -- Defined at <inte
ractive>:1:1
Prelude>
```

In Haskell, the system has the following types:

- **Strong type:** A strong type system ensures that the program will contain errors resulting from wrong expressions. An expression that meets all the conditions of a language is called well-typed; otherwise, it is ill-typed and will lead to a type error. In Haskell, strong typing does not allow automatic conversions. So, if a function has a Double argument but the user provides an Int parameter, then an error will occur. Of course, the user can explicitly convert the Int value to a Double value using the predefined conversion functions and everything will be fine.
- **Static type:** In a static type system, the types of all values and expressions are known by the compiler at compile time, before executing the program. If something is wrong with the types of an expression, then the compiler will tell you, as in the example of lists. Combining strong and static types will avoid runtime errors.
- **Inference type:** In an inference type system, the system recognizes the type of almost all expressions in a program. Of course, the user can define explicitly any variable, providing its type, but this is optional.

## Functions

To create your own functions, using the following definition:

`functionname :: argumenttype -> returntype`

The function name is what you use to call the function, the argument type defines the allowed data type for input parameters, and return type defines the data type the return value will appear in. After the definition, you enter an equation that defines the behavior of the function:

`functionname pattern = expression`

The function name echoes the name of the greater function, pattern acts as a placeholder that will be replaced by the input parameter, and expression outlines how that pattern is used.

```
GNU nano 6.2
add :: Integer -> Integer -> Integer
add x y = x + y
main = do
  putStrLn "Adding two numbers:"
  print(add 3 7)
```

Above one is just a simple example of using functions. The output of the program is given below.

```

sharmik@LAPTOP-R021QTCA:/mnt/e$ nano example.hs
sharmik@LAPTOP-R021QTCA:/mnt/e$ ghc -o example example.hs
[1 of 1] Compiling Main
      ( example.hs, example.o )
Linking example ...
sharmik@LAPTOP-R021QTCA:/mnt/e$ ./example
Adding two numbers:
10

```

On the first line is the function declaration, which tells you the type of inputs and outputs, and on the second line is the function definition. As in the other programming languages, Haskell begins to compile the code from the main function.

## Pattern Matching

Pattern matching means the program checks whether some data matches a certain pattern and then acts accordingly. A function can have different bodies for different patterns. Pattern matching can be applied on any data type. Check out this Code :

```

day :: (Integral a) => a -> String
day 1 = "Monday"
day 2 = "Tuesday"
day 3 = "Wednesday"
day 4 = "Thursday"
day 5 = "Friday"
day 6 = "Saturday"
day 7 = "Sunday"
day x = "The week has only 7 days!"

```

```

sharmik@LAPTOP-R021QTCA:/mnt/e$ ghci
GHCi, version 8.8.4: https://www.haskell.org/ghc/  :? for help
Prelude> :l example.hs[1 of 1] Compiling Main
      ( example.hs, interpreted )
Ok, one module loaded.
*Main> day 7"Sunday"
*Main> day 0"The week has only 7 days!"
*Main> day 3"Wednesday"
*Main>

```

When day is called, the matching begins from the bottom. When there is a match, the corresponding body is chosen. Note that the function contains a default pattern. If you do not put a default pattern and if the function's parameter does not fall into any defined pattern, then you will get an exception.

## Case Expressions

Case expressions are simple. The general syntax is as follows: case expression of pattern -> result .... Let's write the day function using case, as shown here:

```

day :: (Integral a) => a -> String
day x = case x of 1 -> "Monday"
                 2 -> "Tuesday"
                 3 -> "Wednesday"
                 4 -> "Thursday"
                 5 -> "Friday"
                 6 -> "Saturday"
                 7 -> "Sunday"
                 _ -> "The week has only 7 days!"

```

Then write the following:

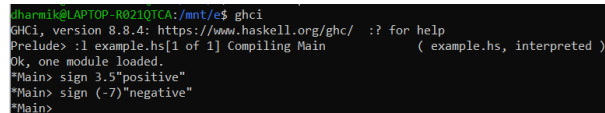
```
Prelude> :l Day.hs
```

```
[1 of 1] Compiling Main ( Day.hs, interpreted )
Ok, one module loaded.
*Main> day 7
"Sunday"
*Main> day 10
"The week has only 7 days!"
```

## Guards

You can use guard to test whether a value has a certain property. Guards are alternative to else..if statements, making the code easier to write and follow. Let's continue with a sign example, shown here:

```
sign :: (RealFloat a) => a -> String
sign x
| x < 0 = "negative"
| x == 0 = "zero"
| otherwise = "positive"
```



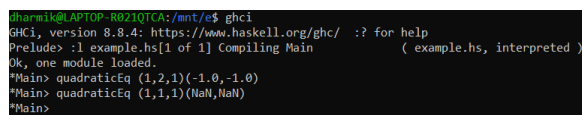
```
sharmik@LAPTOP-R021QTCA:/mnt/e$ ghci
GHCi, version 8.8.4: https://www.haskell.org/ghc/ :? for help
Prelude> :l example.hs[1 of 1] Compiling Main ( example.hs, interpreted )
Ok, one module loaded.
*Main> sign 3.5"positive"
*Main> sign (-7)"negative"
*Main>
```

As you can see in this example, to use guards, you mark them with pipes. The evaluation begins with the bottom expression and continues until a match is found. Note that we have the default case marked by the otherwise keyword; any value that does not meet any of the previous conditions will get the default. You can think of guards as Boolean expressions, where otherwise is always evaluated as True. Pay attention to what's after the parameters in the function definition.

## Clauses

Let's begin with the where clause. As example, think about the quadratic equation defined as  $ax^2 + bx + c = 0$ . The solutions of the equation depend on a discriminant computed as  $\det = b^2 - 4ac$ . If  $\det > 0$ , you will obtain two real solutions:  $x_1, x_2$ . If  $\det = 0$ , you will obtain two real identical solutions:  $x_1 = x_2$ . Otherwise, the equation does not have real solutions. You can think of the three parameters  $a, b, c$  as a triple and the solutions as a pair. Observe that you need  $\det$  in more parts of the algorithm. You can resolve it as follows:

```
quadraticEq :: (Float, Float, Float) -> (Float, Float)
quadraticEq (a, b, c) = (x1, x2)
  where
    x1 = (-b - sqrt delta) / (2 * a)
    x2 = (-b + sqrt delta) / (2 * a)
    delta = b * b - 4 * a * c
```



```
sharmik@LAPTOP-R021QTCA:/mnt/e$ ghci
GHCi, version 8.8.4: https://www.haskell.org/ghc/ :? for help
Prelude> :l example.hs[1 of 1] Compiling Main ( example.hs, interpreted )
Ok, one module loaded.
*Main> quadraticEq (1,2,1)(-1.0,-1.0)
*Main> quadraticEq (1,1,1)(NaN,NaN)
*Main>
```

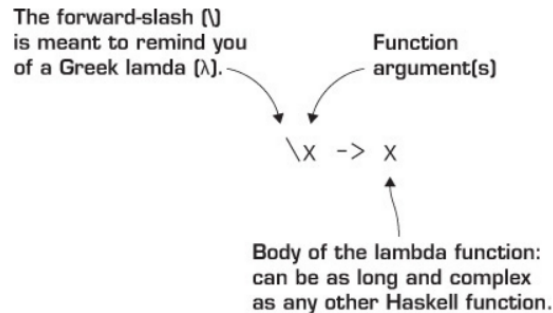
---

The names defined in the where clause are used only in the function, so they will not affect other functions or modules. Be careful about the indentation; all the names should be aligned properly. Do not use the Tab key to add large spaces.

## Lambda Expressions

There are times when you need to use a function just once in your entire application. To not complete with names, you can use anonymous blocks called lambda expressions. A function without definition is called a lambda function, and it is marked by the `\` character.

Example:



```
main = do
  putStrLn "The square of 2 is:"
  print ((\x -> x^2) 2)
```

```
gharmik@LAPTOP-R021QTCA:~$ ghci
GHCi, version 8.8.4: https://www.haskell.org/ghc/  :? for help
Prelude> (\x -> x^2) 2 4
Prelude>
```

## Higher order functions

"A higher-order function is a function that takes other functions as arguments or returns a function as result." The function `applyTwice` takes a function of integers as its first argument and applies it twice to its second argument.

`applyTwice :: (Int -> Int) -> Int -> Int`

`applyTwice f x = f (f x)` Now we'll create some sample functions `double` and `next` to pass to our higher-order function `applyTwice`.

```
applyTwice :: (Int -> Int) -> Int -> Int
```

```
applyTwice f x = f (f x)
```

```
double :: Int -> Int
```

```
double x = 2 * x
```

```
next :: Int -> Int
```

```
next x = x + 1
```

```
main = do
```

```
  print (applyTwice double 2) — quadruples
```

```
  print (applyTwice next 1) —adds 2
```

Using these principles , topics we can make haskell more declarative programming . With higher order functions also can be used to factor out many basic control structures and algorithms from the user code.

---

## Case Studies

The BNDES project team, led by Diogo Barboza Gobira and Felipe Vilhena Antunes Amaral, opted for Julia in their multistage stochastic optimization problem at one of the world's largest development banks. They chose Julia for its speed, elegance, and the Mathematical Optimization Package (JuMP). Unlike traditional approaches that involve a business user prototyping in R and an IT programmer optimizing in C++ or Java, Julia allowed the business side user to create prototypes without the need for subsequent code optimization. This resulted in considerable performance improvements, with the BNDES reporting a speed increase of over 10x, particularly in simulating stochastic models for financial calculations.

The core computations involved simulating stochastic models for stocks, interest rate curves, and inflation indices, with Julia's simplicity, speed, and native parallelism offering a cost-effective and efficient solution. The language's syntax proved simpler than R, eliminating the need for cryptic vectorized code and glue code for datatype translations. The BNDES team anticipates even greater improvements in speed, scalability, and productivity by exploring Julia's native parallelism further.

In developing a JVM prototype using the purely-functional language Haskell, our approach capitalizes on the language's robust features. Strong static typing is employed to construct an intermediate representation, while the expressive abstraction mechanism facilitates the implementation of machine code generation akin to a domain-specific language. The compiler is structured into three key phases: firstly, a transformation pass converts Java bytecode into a register-based intermediate representation; secondly, an existing data-flow analysis framework is applied to this representation; and finally, machine code generation is executed, specifically targeting the x86 architecture. Notably, our implementation adopts a compile-only strategy. To optimize the handling of certain Java features, we leverage code patching. Through diverse code samples, the elegance of our prototype is showcased. The achieved results indicate reasonable performance when compared to real-world implementations. This endeavor highlights the efficiency and expressiveness that Haskell brings to the realm of JVM development, affirming its viability for such intricate tasks.

## Analysis

Metaprogramming excels in dynamic code generation, allowing applications to adapt and generate code at runtime. It promotes code reuse and abstraction, reducing redundancy and enhancing maintainability. Generative programming and the creation of domain-specific languages (DSLs) are facilitated, promoting productivity and a higher level of abstraction.

However, metaprogramming introduces complexity, making code harder to understand and maintain. The performance overhead associated with dynamically generating code at runtime can be a concern. Additionally, security risks may arise if metaprogramming mechanisms are not carefully controlled.

Notable features of metaprogramming include reflection, enabling programs to inspect and modify their own structure at runtime, and template metaprogramming, as seen in languages like C++ and D.

Declarative languages prioritize expressiveness, emphasizing what needs to be achieved over how it should be done. This results in more readable and concise code, making it easier for developers to understand and maintain. The paradigm also supports parallelism and reduces boilerplate code.

While powerful, the declarative paradigm may present a learning curve for developers accustomed to imperative or procedural approaches. Some declarative languages may offer less control over execution flow, which can be limiting in performance-critical scenarios.

Key features of the declarative paradigm include immutability, which reduces side effects and aids in reasoning about program behavior, as well as specialized query languages like SQL for databases or XPath for XML.

---

Julia excels in performance, offering JIT compilation and a type system comparable to low-level languages. Its multiple dispatch enhances flexibility, and it boasts excellent interoperability with other languages. Built-in support for parallelism makes Julia suitable for high-performance computing.

Julia's syntax might present a learning curve, and its ecosystem, while growing, may not match more established languages. However, its metaprogramming capabilities and JIT compilation contribute to its notable features.

Haskell's functional nature promotes robust, reliable code through immutability. Its expressive type system catches errors at compile-time, and lazy evaluation enhances efficiency. Advanced features for concurrency and parallelism make Haskell suitable for scalable applications.

Haskell's functional paradigm and learning curve may challenge developers unfamiliar with these concepts. The language's limitation on mutable state can be a constraint in certain scenarios.

## Comparison

**Expressiveness:** Both Julia and Haskell prioritize expressiveness in their own ways. Julia achieves expressiveness through its multiple dispatch system, while Haskell's functional nature and advanced type system contribute to code clarity and expressiveness.

**Performance:** Both languages are designed with an emphasis on performance, albeit with different approaches. Julia achieves high performance through JIT compilation and dynamic typing, while Haskell focuses on optimization through functional purity and lazy evaluation.

**Parallelism:** Both languages provide features for handling parallelism. Julia has built-in support for parallel computing, while Haskell offers advanced tools for managing concurrency and parallelism.

**Metaprogramming:** Both Julia and Haskell support metaprogramming to some extent. Julia excels in dynamic code generation, while Haskell's template Haskell allows for metaprogramming during compilation.

These are the main similarities I found and there are other too...

when coming to differences, you can come across more differentiation b/w them.

**Programming Paradigm:** The primary difference lies in their programming paradigms. Julia is a multi-paradigm language, incorporating features from procedural, object-oriented, and functional programming. In contrast, Haskell is a purely functional language, emphasizing immutability, referential transparency, and higher-order functions.

**Typing System:** Julia employs dynamic typing, allowing for more flexibility and ease of use. Haskell, on the other hand, utilizes static typing with an advanced type system, catching many errors at compile-time and promoting safer code.

**Learning Curve:** Julia is designed to be accessible to users of other technical computing environments, minimizing the learning curve. Haskell, with its functional paradigm and advanced concepts, may have a steeper learning curve for those unfamiliar with functional programming.

**Immutability:** Immutability is a key principle in Haskell, contributing to the creation of robust and reliable code. While Julia supports immutability, it is not as strictly enforced as in Haskell.

**Use Cases:** Julia is often chosen for its performance in scientific computing, data analysis, and machine learning. Haskell, with its strong emphasis on functional purity, is well-suited for applications where correctness, reliability, and parallelism are crucial, such as finance and concurrent systems.

---

## Challenges Faced

First of all I don't have any idea about these paradigms and languages. This itself a big challenge and interesting. While studying metaprogramming paradigm, it took lot of time to search for the resources as it is a recent paradigm .There are lot of methods for creating expressions , some use different way others use other way. To learn about all ways , it took some time. Initially I couldn't find any difference b/w Symbol and Expression. Later it was manageable. To learn about AST was interesting in this paradigm.

When it comes to Haskell , Initially I have intuition that declarative and functional doesn't have any difference. Later I addressed that functional comes under declarative paradigm. I couldn't find any declarative elements in Haskell as it is functional programming language. But I addressed there is some relation with declarative paradigm in Haskell such as Higher-order functions.It took me time to learn the syntax and later it was manageable.

## Conclusion

Therefore we conclude that, Julia exhibits a robust capability for metaprogramming, leveraging features such as just-in-time (JIT) compilation and dynamic code generation. Its multiple dispatch system and metaprogramming facilities contribute to a highly expressive and flexible programming environment. On the other hand, Haskell's strength lies in its declarative paradigm, emphasizing immutability, referential transparency, and a sophisticated type system. Haskell's declarative nature promotes code clarity and reliability by focusing on specifying what needs to be achieved rather than detailing how it should be done. Both languages showcase unique strengths in their respective programming paradigms, with Julia excelling in dynamic code manipulation and expressiveness, while Haskell prioritizes immutability and declarative clarity for building robust and reliable software.

## References

- <https://docs.julialang.org/en/v1/manual/metaprogramming/>
- <https://github.com/FugroRoames/RoamesNotebooks/tree/master>
- [https://en.wikibooks.org/wiki/Introducing\\_Julia/Metaprogramming](https://en.wikibooks.org/wiki/Introducing_Julia/Metaprogramming)
- [https://www.tutorialspoint.com/julia/julia\\_metaprogramming.htm](https://www.tutorialspoint.com/julia/julia_metaprogramming.htm)
- <https://www.youtube.com/watch?v=2QLhw6LVaq0>
- [https://github.com/dpsanders/Metaprogramming\\_JuliaCon\\_2021/blob/master](https://github.com/dpsanders/Metaprogramming_JuliaCon_2021/blob/master)
- <https://www.oreilly.com/library/view/getting-started-with/9781783284795/ch07.html>
- Getting started with Julia Programming book by Ivo Balbaert
- Julia - Bit by Bit Programming for beginners by Noel Kalicharan
- <https://www.toptal.com/julia/code-writing-code-modern-metaprogramming>
- <https://en.wikipedia.org/wiki/Metaprogramming>
- <https://juliahub.com/case-studies/>
- <https://www.geeksforgeeks.org/difference-between-imperative-and-declarative-programming/>
- <https://www.techopedia.com/definition/18763/declarative-programming>
- <https://www.ionos.com/digitalguide/websites/web-development/declarative-programming/>
- <https://knowledgezone.co.in/trends/explorer?topic=Functional-Programming>
- <https://hyperskill.org/learn/step/15860>

- 
- <https://github.com/thma/WhyHaskellMatters>
  - Learn You a Haskell for Great Good A Beginners Guide book (Miran Lipovaca)
  - Programming in Haskell book by Graham Hutton
  - <https://en.wikibooks.org/wiki/Haskell>
  - <https://www.haskell.org/>
  - <https://blog.stackademic.com/functional-programming-paradigms-from-lisp-to-haskell-af05b375d1c5>
  - <https://www.cmi.ac.in/~madhavan/presentations/mcc-haskell/lecture1.pdf>
  - <https://www.cmi.ac.in/~madhavan/presentations/mcc-haskell/lecture2.pdf>