Amrita Vishwa Vidyapeetham

TIFAC-CORE in Cyber Security

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

Gokul

21st January, 2024

Paradigm 1: Imperative

Discuss the principles and concepts of Paradigm 1.

1 Introduction:

- Imperative programming paradigm focuses on describing sequence of steps to be executed by a computer to achieve a specific goal. Program consists of a series of statements that change a program state by manipulating state variables.

- It assumes that the computer keeps track of the state variables that are modified in a computation process. This computation is performed through guided sequence of steps in which these variables are referred to or changed.

- The order of the steps is crucial because a given step will have different consequences depending on the current values of variables when the step is executed.

- It is also one of the oldest programming paradigms. It has close relation to machine architecture. It is based on Von Neumann architecture, that works by changing program state through assignment statements. It performs step by step task by changing state.

- The main focus of this paradigm is how to achieve the goal. Step by step instruction in specified and there is very little abstraction.

2 Principles and Concepts of Imperative Paradigm:

2.1 Variables

Imperative program mainly works by manipulating state. State is the current snapshot of the program at any given time. State is mostly represented by variables. Changes to these states are modified by assignment statements.

2.2 Sequence of Code

The order of statements is of high importance in this paradigm, as upcoming statements depend on the state variable that was supposed to be modified by a previous statement. The order of execution determines the flow of the program. Various control flow statements are used to manage the flow of the code.

2.3 Control Flow

Control flow statements include if-else conditions, loops, etc. These form a major part of imperative programming. These control the flow of the program based on certain conditions. They execute different blocks of code based on the conditions and loops help run the code until certain conditions are met.

2.4 Mutation and side effects

The process of changing state variable is referred to as mutation. This mutation during the execution of code results in side effects. Managing side effects and monitoring variables is an important part of imperative programming.

3 Advantages and Disadvantages:

3.1 Advantages:

1. Easy to read

2. Easy to learn

3. Provide explicit control over the execution of the program

4. Allows programmers to optimise the code for efficiency

5. Allows for direct manipulation of memory, making it suitable for low-level and system programming.

3.2 Disadvantages:

1. Code becomes confusing and very large soon.

2. More prone to errors.

3. Requires time for maintenance which reduces time for actual development.

4. Extension and optimisation is difficult.

5. Limited abstraction.

Language for Paradigm 1: Rust

Discuss the characteristics and features of the language associated with Paradigm 1.

4 Introduction to Rust:

- Rust is a programming language that's designed for systems and can be used for wide range of applications. Well suited for system programming where performance and safety are critical, but it is also used in web development, game development and embedded systems.

- Rust is a multi paradigm, general purpose programming language that emphasizes on performance, type safety and concurrency. It enforces memory safety and doesn't require automated memory management techniques.

- Rust is named after the rust fungus because it is robust, distributed and parallel.

- Rust is not bound to any one paradigm. An interesting facet of rust is that you can program imperatively with some of the same benefits as functional programming, because of immutability by default and mutability aliasing checking.

- In a way this is more work than equivalent functional code but safer than the equivalent imperative code.

Hello World:

```
    // This is the main function.
fn main() {
    // Statements here are executed when the compiled binary is called.

    // Print text to the console.
    println!("Hello World!");
}
```

4.1 Association with Imperative Paradigm

- Mutable variables
  - Rust allows the creation of mutable variables using the keyword ‘mut‘.

```
fn main() {
    let _immutable_binding = 1;
    let mut mutable_binding = 1;

    println!("Before mutation: {}", mutable_binding);

    // Ok
    mutable_binding += 1;

    println!("After mutation: {}", mutable_binding);

    // Error! Cannot assign a new value to an immutable variable
    _immutable_binding += 1;
}
```

- Control Flow
  - Rust has the traditional conditional statements and loops like if-else, for, while, etc.

```
fn main() {
let n = 5;

if n < 0 {
    print!("{} is negative", n);
} else if n > 0 {
    print!("{} is positive", n);
} else {
    print!("{} is zero", n);
}

let big_n =
    if n < 10 && n > -10 {
        println!(", and is a small number, increase ten-fold");

        // This expression returns an ‘i32‘.
        10 * n
    } else {
        println!(", and is a big number, halve the number");

        // This expression must return an ‘i32‘ as well.
        n / 2
        // TODO ^ Try suppressing this expression with a semicolon.
    };
//   ^ Don’t forget to put a semicolon here! All ‘let‘ bindings need it.

println!("{} -> {}", n, big_n);
}
```

- Ownership and borrowing
  - Because variables are in charge of freeing their own resources, resources can only have one owner. This prevents resources from being freed more than once. Note that not all variables own resources (e.g. references).
  - When doing assignments (let x = y) or passing function arguments by value (foo(x)), the ownership of the resources is transferred. In Rust-speak, this is known as a move.
  - After moving resources, the previous owner can no longer be used. This avoids creating dangling pointers.

```
    // This function takes ownership of the heap allocated memory
fn destroy_box(c: Box<i32>) {
    println!("Destroying a box that contains {}", c);

    // ‘c‘ is destroyed and the memory freed
}

fn main() {
    // _Stack_ allocated integer
    let x = 5u32;

    // *Copy* ‘x‘ into ‘y‘ - no resources are moved
    let y = x;

    // Both values can be independently used
    println!("x is {}, and y is {}", x, y);

    // ‘a‘ is a pointer to a _heap_ allocated integer
    let a = Box::new(5i32);

    println!("a contains: {}", a);

    // *Move* ‘a‘ into ‘b‘
    let b = a;
    // The pointer address of ‘a‘ is copied (not the data) into ‘b‘.
    // Both are now pointers to the same heap allocated data, but
    // ‘b‘ now owns it.

    // Error! ‘a‘ can no longer access the data, because it no longer owns the
    // heap memory
    //println!("a contains: {}", a);
    // TODO ^ Try uncommenting this line

    // This function takes ownership of the heap allocated memory from ‘b‘
    destroy_box(b);

    // Since the heap memory has been freed at this point, this action would
    // result in dereferencing freed memory, but it’s forbidden by the compiler
    // Error! Same reason as the previous Error
    //println!("b contains: {}", b);
    // TODO ^ Try uncommenting this line
}
```

- Most of the time, we'd like to access data without taking ownership over it. To accomplish this, Rust uses a borrowing mechanism. Instead of passing objects by value (T), objects can be passed by reference (T).
- The compiler statically guarantees (via its borrow checker) that references always point to valid objects. That is, while references to an object exist, the object cannot be destroyed.

```
    // This function takes ownership of a box and destroys it
fn eat_box_i32(boxed_i32: Box<i32>) {
    println!("Destroying box that contains {}", boxed_i32);
}

// This function borrows an i32
fn borrow_i32(borrowed_i32: &i32) {
    println!("This int is: {}", borrowed_i32);
}

fn main() {
    // Create a boxed i32 in the heap, and a i32 on the stack
    // Remember: numbers can have arbitrary underscores added for readability
    // 5_i32 is the same as 5i32
    let boxed_i32 = Box::new(5_i32);
    let stacked_i32 = 6_i32;

    // Borrow the contents of the box. Ownership is not taken,
    // so the contents can be borrowed again.
    borrow_i32(&boxed_i32);
    borrow_i32(&stacked_i32);

    {
        // Take a reference to the data contained inside the box
        let _ref_to_i32: &i32 = &boxed_i32;

        // Error!
        // Can't destroy ‘boxed_i32‘ while the inner value is borrowed later in scope.
        eat_box_i32(boxed_i32);
        // FIXME ^ Comment out this line

        // Attempt to borrow ‘_ref_to_i32‘ after inner value is destroyed
        borrow_i32(_ref_to_i32);
        // ‘_ref_to_i32‘ goes out of scope and is no longer borrowed.
    }

    // ‘boxed_i32‘ can now give up ownership to ‘eat_box‘ and be destroyed
    eat_box_i32(boxed_i32);
}
```

5 Real World Projects:

- Alacritty
  Alacritty is a modern terminal emulator that comes with sensible defaults, but allows for extensive configuration. By integrating with other applications, rather than reimplementing their functionality, it manages to provide a flexible set of features with high performance. The supported platforms currently consist of BSD, Linux, macOS and Windows.

  Github: https://github.com/alacritty/alacritty

- Lapce Editor

  Lapce (IPA: /læps/) is written in pure Rust with a UI in Floem. It is designed with Rope Science from the Xi-Editor which makes for lightning-fast computation, and leverages Wgpu for rendering. More information about the features of Lapce can be found on the main website and user documentation can be found on GitBook.

  Github: https://github.com/lapce/lapce

Paradigm 2: Functional

Discuss the principles and concepts of Paradigm 2.

6 Introduction:

- The functional programming paradigm has its roots in mathematics. The central model for abstraction is the function which are meant for some specific computation and not the data structure.

- Data is not coupled to the functions. No global variables are changed in this paradigm. The values are passed in as arguments and a result is returned back. No variable outside the scope of the function is changed.

- These functions provide abstraction by hiding the implementation and only showing the result. The arguments can be changed without changing the meaning of the program.

- The paradigm views all subprograms as functions in the mathematical sense-informally, i.e., it takes in arguments and returns a solution. The solution returned is based entirely on the input and the time at which it is called has no relevance.

- It does not manipulate state variables, relies on the principles of immutability and absence of side effects.

- The main focus here is what should the program achieve rather than how it should do.

7 Principles and Concepts of Functional Paradigm:

7.1 Pure Functions:

In functional programming, functions are pure, meaning they have no side effects and produce the same output for the same input every time they are called. Pure functions do not modify global state or variables; they only depend on their input parameters.

7.2 Immutability:

Functional programming encourages the use of immutable data structures. Once a data structure is created, it cannot be modified. Instead, new data structures are created to represent changes, promoting a more predictable and safer programming style.

7.3 Modularity:

Functional programming encourages composing small, focused functions to build more complex functions. Function composition involves combining functions to create new functions, providing a concise and expressive way to model complex behavior.

7.4 Recursion:

Recursion is often favored over traditional iterative constructs like loops. Recursive functions call themselves with modified arguments, emphasizing the self-contained nature of functional programming.

7.5 Pattern Matching:

Pattern matching is a powerful technique in functional programming for checking a value against a pattern. It simplifies code by allowing developers to express complex conditional logic more concisely.

7.6 First-Class Functions:

First-class functions in functional programming are treated as data type variables and can be used like any other variables. These first-class variables can be passed to functions as parameters, or stored in data structures.

7.7 High Order Functions:

A function that accepts other functions as parameters or returns functions as outputs is called a high order function. This process applies a function to its parameters at each iteration while returning a new function that accepts the next parameter.

7.8 Lazy Evaluation:

Lazy evaluation is a strategy where the evaluation of an expression is delayed until its value is actually needed. This can lead to more efficient use of resources and enables the creation of infinite data structures.

8 Advantages and Disadvantages:

8.1 Advantages:

- Pure functions are easier to understand because they don't change any states and depend only on the input given to them. Whatever output they produce is the return value they give. Their function signature gives all the information about them i.e. their return type and their arguments.

- The ability of functional programming languages to treat functions as values and pass them to functions as parameters make the code more readable and easily understandable.

- Testing and debugging is easier. Since pure functions take only arguments and produce output, they don't produce any changes don't take input or produce some hidden output. They use immutable values, so it becomes easier to check some problems in programs written uses pure functions.

- It is used to implement concurrency/parallelism because pure functions don't change variables or any other data outside of it.

- It adopts lazy evaluation which avoids repeated evaluation because the value is evaluated and stored only when it is needed.

- Functional programming languages often provide concise and expressive syntax for manipulating data and defining transformations. This can lead to more readable and maintainable code.

- Functional programming promotes predictability by avoiding mutable state and side effects. This makes it easier to understand and reason about the behavior of functions.

8.2 Disadvantages:

- Sometimes writing pure functions can reduce the readability of code.

- Writing programs in recursive style instead of using loops can be difficult.

- Writing pure functions are easy but combining them with the rest of the application and I/O operations is a difficult task.

- Immutable values and recursion can lead to decrease in performance.

- Functional programming introduces a different mindset compared to imperative programming, which can result in a steep learning curve for developers unfamiliar with the paradigm.

- While immutability has advantages, there are cases where mutable state is more efficient or practical. Functional programming may limit certain programming patterns that rely heavily on mutability.

Language for Paradigm 2: Scala

Discuss the characteristics and features of the language associated with Paradigm 2.

9 Introduction to Scala:

- Scala is a modern multi-paradigm programming language designed to express common programming patterns in a concise, elegant, and type-safe way. It seamlessly integrates features of object-oriented and functional languages.

- Scala is also a functional language in the sense that every function is a value.

- Scala provides a lightweight syntax for defining anonymous functions, it supports higher-order functions, it allows functions to be nested, and it supports currying.

- Scala's case classes and its built-in support for pattern matching provide the functionality of algebraic types, which are used in many functional languages.

- Singleton objects provide a convenient way to group functions that aren't members of a class.

- Scala lets you write less to do more. As a high-level language, its modern features increase productivity and lead to more readable code.

- With Scala, you can combine both functional and object-oriented programming styles to help structure programs.

- Scala is well suited to building fast, concurrent, and distributed systems with its JVM, JavaScript and Native runtimes.

- Scala prioritises interoperability, giving easy access to many ecosystems of industry-proven libraries.

- Scala's static types help you to build safe systems by default. Smart built-in checks and actionable error messages, combined with thread-safe data structures and collections, prevent many tricky bugs before the program first runs.

Hello World:

```
object Hello {
    def main(args: Array[String]) = {
    println("Hello, world")
    }
}
```

9.1 Association with Functional Paradigm:

- Immutable data structures
  - In pure functional programming, only immutable values are used. In Scala this means:
    - All variables are created as val fields
    - Only immutable collections classes are used, such as List, Vector, and the immutable Map and Set classes
  - When it comes to using collections, one answer is that you don't mutate an existing collection; instead, you apply a function to an existing collection to create a new collection. This is where higher-order functions like map and filter come in.

```
val a = List("jane", "jon", "mary", "joe")
val b = a.filter(_.startsWith("j"))
     .map(_.capitalize)
```

- First-Class Functions:
  - Functions are first-class citizens in Scala, meaning they can be assigned to variables, passed as arguments to other functions, and returned as values. This allows for the creation of higher-order functions and functional composition.
  - While every programming language ever created probably lets you write pure functions, a second important Scala FP feature is that you can create functions as values, just like you create String and Int values.
  - This feature has many benefits, the most common of which are (a) you can define methods to accept function parameters, and (b) you can pass functions as parameters into methods. You've seen this in multiple places in this book, whenever methods like map and filter are demonstrated:

```
val nums = (1 to 10).toList

val doubles = nums.map(_ * 2)           // double each value
val lessThanFive = nums.filter(_ < 5)   // List(1,2,3,4)
```

- Pure Functions:
  - Another feature that Scala offers to help you write functional code is the ability to write pure functions. A pure function can be defined like this:
    - A function f is pure if, given the same input x, it always returns the same output f(x)
    - The function's output depends only on its input variables and its implementation
    - It only computes the output and does not modify the world around it

  - This implies:
    * It doesn't modify its input parameters
    * It doesn't mutate any hidden state
    * It doesn't have any "back doors": It doesn't read data from the outside world (including the console, web services, databases, files, etc.), or write data to the outside world
  - As a result of this definition, any time you call a pure function with the same input value(s), you'll always get the same result. For example, you can call a double function an infinite number of times with the input value 2, and you'll always get the result 4.

```
def sum(xs: List[Int]): Int = xs match
case Nil => 0
case head :: tail => head + sum(tail)
```

- Functional Error Handling:
  - Functional programming is like writing a series of algebraic equations, and because algebra doesn't have null values or throw exceptions, you don't use these features in FP.
  - Scala's solution is to use constructs like the Option/Some/None classes.

```
def makeInt(s: String): Option[Int] =
try
  Some(Integer.parseInt(s.trim))
catch
  case e: Exception => None
```

- Immutability by Default:
  - Scala emphasizes immutability by default, encouraging developers to use the val keyword for immutable variables. While mutable variables can be defined using var, the preference is for immutability when possible.
- Lambda Expressions:
  - Scala supports concise syntax for anonymous functions through lambda expressions, making it easy to define functions inline.

```
val f = { (x: Int) => x * x }
```

- Lazy Evaluation
  - Scala provides lazy evaluation, allowing the delay of computation until the result is actually needed. This can improve performance and supports the creation of infinite data structures.

- Lazy evaluation or call-by-need is a evaluation strategy where an expression isn't evaluated until its first use i.e to postpone the evaluation till its demanded. Functional programming languages like Haskell use this strategy extensively.

- C, C++ are called strict languages who evaluate the expression as soon as it's declared. Then there are languages like Scala who are strict by default but can be lazy if specified explicitly i.e. of mixed type.

```
val geeks = List(1, 2, 3, 4, 5)

lazy val output2 = geeks.map(l=> l*5)

println(output2)
```

10 Real World Projects:

- Scalaz
  - Scalaz is a Scala library for functional programming.
  - It provides purely functional data structures to complement those from the Scala standard library. It defines a set of foundational type classes (e.g. Functor, Monad) and corresponding instances for a large number of data structures.

  Github: https://github.com/scalaz/scalaz

- Doobie
  - doobie is a pure functional JDBC layer for Scala.

  Github: https://github.com/tpolecat/doobie

Analysis

Provide an analysis of the strengths, weaknesses, and notable features of both paradigms and their associated languages.

11 Strengths of Imperative Paradigm in Rust:

1. Easy to Read:

- Rust uses a syntax similar to C like languages, this makes it relatively straightforward.
- This contributes to easy code readability of seasoned developers.

2. Easy to Learn:
   - Rust is considered easy to learn for those who are already familiar with C.
   - It has a wide community support.
   - It has lower learning curve that C++ and is faster than C++.
3. Explicit Control over Execution:
   - Imperative languages like rust provide explicit control over the flow of code.
   - This explicit control allows developers to micro-manage the program execution and optimize performance.
4. Optimization for efficiency:
   - Rust's zero-cost abstraction are meant to mean that you can use higher-level abstractions, and still get the same performance as if you wrote lower-level code optimized by hand.
   - The language allows developers to control memory allocation and deallocation explicitly, making it possible to write highly efficient code without sacrificing safety.
   - This level of control over performance is especially valuable in systems programming where efficiency is critical.
5. Direct Memory Manipulation:
   - Rust's ownership system allows for direct manipulation of memory.
   - The ownership system prevents common memory related errors like dangling pointers.

12 Weakness of Imperative Paradigm in Rust:

1. Code becomes very large soon:
   - Imperative languages can have high code complexity and verbosity.
2. More Prone to Errors:

- Imperative Paradigm allows direct manipulation of mutable state.

3. Requires Time for Maintenance, Reducing Time for Actual Development:
   - Maintenance overhead is a concern in any large codebase, and imperative languages may require more attention to detail in terms of managing mutable state and ensuring correct resource management.

4. Extension and Optimization is Difficult:
   - While the imperative paradigm in Rust allows for explicit control over execution and performance optimizations, it might be challenging for developers to optimize code effectively.

5. Limited Abstractions:
   - Imperative languages, including Rust, may be perceived as having limited abstraction compared to some functional languages. This can impact code expressiveness and conciseness.

13 Strengths of Functional Paradigm in Scala:

1. Pure Functions and Immutability:
   - Scala, as a functional programming language, encourages the use of pure functions and immutability. The ability to define functions that don't change states and rely only on their input parameters contributes to code clarity, ease of understanding, and predictability.

2. Higher-Order Functions:
   - Scala treats functions as first-class citizens, allowing them to be passed as arguments to other functions. This feature facilitates the creation of higher-order functions, making the code more readable and allowing for expressive and concise solutions to complex problems.

3. Testing and Debugging:
   - In Scala, functional programming practices, including the use of pure functions and immutability, make testing and debugging easier.
   - The absence of side effects simplifies the process of identifying and fixing issues, contributing to code reliability.

4. Concurrency/Parallelism:
   - Immutability and the absence of shared mutable state make it easier to reason about and manage concurrent operations.

- Scala provides constructs like Futures and Actors for concurrent programming.

5. Predictability:

    - Functional programming in Scala promotes predictability by avoiding mutable state and side effects.
    - This makes it easier to reason about the behavior of functions, understand their effects, and predict the program's overall behavior.

14 Weaknesses of Functional Paradigm in Scala:

1. Reduced Readability with Pure Functions:

    - While pure functions enhance code predictability and maintainability, they introduce verbosity, especially when dealing with complex operations.

2. Difficulty in Recursive Programming:

    - Recursive programming, often emphasized in functional languages, can be challenging for some developers who are more accustomed to using loops.

3. Integration with I/O Operations:

    - Integrating pure functions with the rest of the application, especially when dealing with I/O operations, can be challenging.

4. Performance Impact of Immutability and Recursion:

    - Immutability and recursion, while promoting code safety and simplicity, can lead to performance considerations.

5. Learning Curve for Functional Mindset:

    - Functional programming introduces a different mindset compared to imperative programming, and developers may face a learning curve when transitioning.

6. Limitations on Mutable State:

    - While immutability has advantages, certain scenarios may benefit from mutable state in terms of efficiency or practicality.
    - Functional programming in Scala is not strictly enforced, allowing developers to choose mutable data structures when needed.

Comparison

Compare and contrast the two paradigms and languages, highlighting similarities and differences.

15 Similarities between the two paradigms and languages:

1. Memory Safety:

   - Rust provides features for low-level concurrency and parallelism with its ownership system and thread support.
   - Scala supports concurrent and parallel programming through features like Futures and Actors, making it suitable for scalable and concurrent applications.

2. Concurrency Support:

   - Rust provides features for low-level concurrency and parallelism with its ownership system and thread support.
   - Scala supports concurrent and parallel programming through features like Futures and Actors, making it suitable for scalable and concurrent applications.

3. Versatility:

   - Rust, while primarily imperative, incorporates functional programming concepts.
   - Scala, designed as a hybrid language, seamlessly blends object-oriented and functional programming.

4. Pattern Matching:

   - Rust's match expression and Scala's pattern matching provide expressive ways to handle different cases in code.

5. Immutable Collections:

   - Rust has immutable variables.
   - Scala's collections are immutable by default, promoting functional programming practices.

16 Dissimilarities between the two paradigms and languages:

1. Mutability:

   - Rust allows mutable state but enforces strict rules through its ownership system to prevent data races and memory issues.

   - Scala encourages immutability, making data structures and variables immutable by default. Functional programming in Scala avoids mutable state, promoting safer code.

2. Side Effects:

   - Imperative programming allows side effects, but Rust minimizes them through its ownership system and borrowing rules.

   - Functional programming emphasizes the avoidance of side effects, promoting pure functions that rely solely on input parameters.

3. Lazy Evaluation:

   - Rust does not natively support lazy evaluation.

   - Scala supports lazy evaluation, allowing expressions to be evaluated only when their values are needed, contributing to performance optimizations.

4. Code Order:

   - In Rust the command execution order is fixed.

   - In Functional Paradigm, the command execution order is not fixed.

5. Expressiveness:

   - Imperative is more expressive but less safer

   - Functional safer than imperative paradigm.

Challenges Faced

Discuss any challenges you encountered during the exploration of programming paradigms and how you addressed them.

Navigating Documentation:

- Large documentation of Scala and Rust become overwhelming when learning new concepts from scratch.
- Overcame by reading only the relevant parts and skimming through others.

Referencing Multiple Materials:

- Referencing multiple sources online is a great way to learn, but quickly leads to information overload because different sources may contradict each other.
- Overcame this by priortizing sources based on legibility and content quality.

Understanding Language-Specific Concepts:

- Both the language introduce unique concepts that are not present in the other requiring additional effort to comprehend. Learning about two language at a time also introduces an additional overhead.
- Overcome by learning only the relevant bits and leaving others things to learn as you go.

Version Compatibility Issues:

- Overcame this by using online playground provided the official websites, since I only need to run basic examples.

Conclusion

Summarize your findings and conclude the assignment.

We talked about two different ways of writing computer programs: one is like giving clear instructions step by step (like Rust does), and the other is like using building blocks to create a puzzle (like Scala does). Each way has its own strengths and challenges.

When we looked at Rust, which follows the step-by-step way, it's good for controlling how things happen and making sure the computer doesn't make mistakes with memory. It's like having a detailed plan for everything.

On the other hand, Scala, with its puzzle-building approach, makes code shorter and more like plain English. It's good for describing what you want without worrying too much about the exact steps.

Learning about these ways of programming, we found that each has its difficulties. Understanding and getting used to the new ideas can be tricky. It's like learning a new language or a new game.

The choice between these ways of programming depends on what you're building and what you feel comfortable with.

References

Include any references or sources you consulted for your assignment.

https://www.geeksforgeeks.org/introduction-of-programming-paradigms/

https://www.cs.ucf.edu/ leavens/ComS541Fall97/hw-pages/paradigms/major.htmlimperative

https://youtu.be/dAPL7MQGjyM?si=VkWwRZTroHLhZM2y

https://youtu.be/E7Fbf7R3x6I?si=2zsX0aMOF-hKKCnv

https://youtu.be/sqV3pL5x8PI?si=ENUdLypdk8U8u0cV

https://www.techtarget.com/whatis/definition/imperative-programming: :text=Imperative

https://react.dev/learn

https://en.wikipedia.org/wiki/Rust$_{(}$*programming*$_{l}$*anguage*) : : *text* = *Rust*

https : //*users.rust* − *lang.org*/*t*/*idiomatic* − *rust* − *favors* − *functional* − *or* − *imperative* − *style*/31720/3

https : //*doc.rust* − *lang.org*/

https : //*www.geeksforgeeks.org*/*functional* − *programming* − *paradigm*/

https : //*www.turing.com*/*kb*/*introduction* − *to* − *functional* − *programming*

https : //*www.scala* − *lang.org*/

https : //*docs.scala* − *lang.org*/*scala*3/*book*/*introduction.html*

https : //*www.geeksforgeeks.org*/*scala* − *lazy* − *evaluation*/

https : //*users.rust* − *lang.org*/*t*/*rust* − *has* − *zero* − *cost* − *abstraction* − *what* − *does* − *this* − *mean* − *in* − *a* − *practical* − *sense*/100556

https : //*www.linkedin.com*/*advice*/0/*what*−*differences*−*between*−*functional*−*imperative*−*languages*−1*wire*

https : //*www.geeksforgeeks.org*/*difference* − *between* − *functional* − *and* − *imperative* − *programming*/