20CYS312 - Principles of Programming Languages
Exploring Programming Paradigms

Assignment-01

Presented by Aishwarya G
CB.EN.U4CYS21003
TIFAC-CORE in Cyber Security
Amrita Vishwa Vidyapeetham, Coimbatore Campus

Feb 2024

Outline

Introduction

What is Programming Paradigm?

- Paradigm refers to a method to solve some problem or perform some task.
- Programming paradigm is an approach to solve problem using some programming language.
- Programming paradigms are fundamental styles or approaches to programming that dictate how code is structured, organized, and executed. They are frameworks or models that provide a set of principles, concepts, and rules for designing and implementing software.
- Several programming paradigms exist, and each has its own set of principles and characteristics.
- Major programming paradigms include Imperative, Declarative, Object-Oriented, Functional, Procedural, Structural and Concurrent.

Different Programming Paradigms

- **Imperative programming** – focuses on how to execute, defines control flow as statements that change a program state.
  Example: C programming language, where you use statements to explicitly define the sequence of operations.
- **Declarative programming** – focuses on what to execute, defines program logic, but not detailed control flow.
  Example: HTML, SQL (Structured Query Language) for database queries, where you specify the desired data without detailing the steps of retrieval.
- **Structural Programming** - Programming with clean control structures.
  Example: Pascal, where programs are organized into structures such as functions and procedures to enhance readability and maintainability.

Different Programming Paradigms

- **Procedural Programming** - Imperative programming with procedure calls. Example: Fortran, which involves defining procedures (subroutines and functions) to execute a series of steps to achieve a specific task.
- **Functional Programming** - Programming with function calls that avoid any global state.
  Example: Haskell, where functions are treated as first-class citizens, and programs are built by composing and applying functions, avoiding mutable state.
- **Object-oriented programming (OOP)** – organizes programs as objects: data structures consisting of attributes and methods together with their interactions. Example: Java, where you create and manipulate objects with encapsulated data and behavior, fostering modularity and reusability.
- **Concurrent programming** - It is a paradigm in software development that focuses on executing multiple tasks or processes simultaneously to improve program performance and responsiveness.
  Exmaple:Erlang - Originally designed for telecom applications, Erlang is known for its lightweight processes and fault-tolerant design.

Why Programming Paradigm

- Guides programmers in how to think about and structure their code.
- Influences the way programs are written and the techniques used to solve problems.
- Shapes the overall design philosophy of software development.
- Choosing the right paradigm can impact the efficiency, maintainability, and scalability of a program.
- Many languages support multiple paradigms, allowing for a flexible and adaptable approach.("multi-paradigm" languages, meaning you can adapt your code to fit a certain paradigm or another)
- New paradigms may emerge as programming evolves, integrating new techniques and methodologies.

Functional Programming

- The functional programming paradigm is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data.
- It is a **declarative type of programming style** that focuses on what to solve rather than how to solve.
- The functions and function calls are directly used by the functional programming language. Functional programming language does not support the flow of the controls like statements of the loop, and conditional statements such as If-Else and Switch Statements.
- The statements can be executed in any order.
- The first high-level functional programming language, Lisp, was developed in the late 1950s.

Characteristics of Functional Programming

- Functions are also treated like **first-class citizens**—meaning they can pass as arguments, return from other functions, and attach to names.
- Functional programming typically incorporates the utilization of **pure functions**, which are deterministic mathematical functions. These functions consistently generate identical output for a given input and do not cause any side effects. This characteristic enhances the predictability of the code and simplifies the process of testing and maintaining it.
- **Immutable data**, No value assigned to a variable in functional programming can be changed; the only way to change a value is to create a new variable.
- **Referential Transparency** - This property ensures that a function, given the same inputs, will always produce the same outputs, with no hidden states or side effects.
- Functional programming supports functions in **higher-order** and features of **lazy evaluation**.
- The functional programming paradigm aligns well with parallel and concurrent programming. The use of immutability and the absence of shared state make it easier to develop programs that can run in **parallel** without the risk of data corruption.

Advantage of Functional Programming

- **Code without bugs**: Functional programming is stateless, there are no side effects. As a result, we can write error-free code.
- **Easy to debug**: Since pure functions generate the same output for a given input, there are no unexpected changes or hidden outputs. Additionally, the immutability of functional programming functions makes it simpler to identify and rectify errors in the code more efficiently.
- **Efficient programming language**: Functional programming languages do not have mutable state, so there are no issues with state changes. We can use functions to work parallel to instructions, allowing for code that is easily reusable and testable.
- **Efficiency**: Functional programs consist of independent units that can run concurrently, making them more efficient.
- **Support for nested functions**: Functional programming supports nested functions.
- **Lambda Calculus**: Since functional programming is based on the concepts of lambda calculus, it is also ideal for mathematical operations.
- **Lazy evaluation**: Functional programming also supports lazy constructions such as lazy lists and lazy maps.

Limitations of Functional Programming

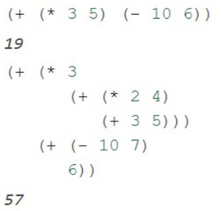
- Pure functions are not optimal for unpredictable arguments like user input.
- Function recursion can be more complex and perplexing compared to traditional loops.
- Due to the no side effects in pure functions, their combination with other functions and I/O often poses difficulties and results in decreased performance.
- The utilization of recursion and variable immutability frequently leads to increased memory usage and reduced performance.
- The absence of loops can present challenges: Transforming programs into a recursive style instead of using loops can be an difficult task.

Scheme

- Scheme is a programming language in the Lisp family.Scheme is a small, yet powerful language in the Lisp family.
- Scheme is formally defined in the Scheme report [Abelson98], which is revised from time to time. Currently, the seventh revision is the most current one.R7RS-'The seventh Revised Report on the Algorithmic Language Scheme'.
- Scheme is a high-level programming language that encourages a functional style.
- Types are checked and handled at run time - Dynamic type checking.

Scheme

- In Scheme, everything is an expression; parenthesized lists in which a prefix operator is followed by its arguments.
- Scheme programs are made up of keywords, variables, structured forms, constant data (numbers, characters, strings, quoted vectors, quoted lists, quoted symbols, etc.), whitespace, and comments.
- Scheme exclusively uses prefix notation. Operators are often symbols, such as + and *. Call expressions can be nested, and they may span more than one line.



```
(+ (* 3 5) (- 10 6))
19
(+ (* 3
      (+ (* 2 4)
         (+ 3 5)))
   (+ (- 10 7)
      6))
57
```

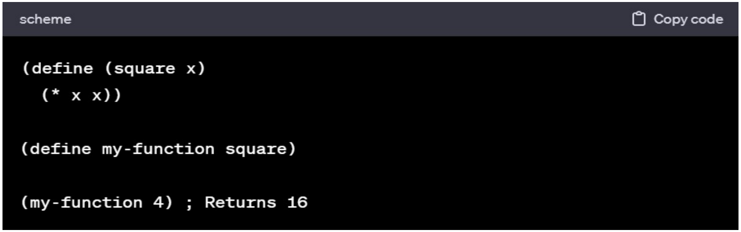
- The if expression in Scheme is an example of a special form. Despite its syntactical resemblance to a function call, it has a different evaluation procedure.The gener form of an if expression is:
  (if <predicate> <consequent> <alternative>)

Scheme

- Numerical values can be compared using familiar comparison operators, but prefix notation is used in this case as well:
  > (>= 2 1)
  #t
- Truth values in Scheme, including the boolean values #t (for true) and #f (for false), can be combined with boolean special forms, and or not.
  (and <e1> ... <en>)
  (or <e1> ... <en>)
  (not <e>)
- Values can be named using the **define** special form:
  (define pi 3.14)
  (* pi 2)
  6.28
- In Scheme, any expression that is not evaluated is said to be quoted.When an expression is quoted, it is treated as data, and its literal form is used rather than evaluating its content.
  (display 'x) ; Prints the symbol 'x', not the value of x

Scheme - Functional Programming

- Functions as First-Class Citizens- where functions can be passed as arguments and returned as values:

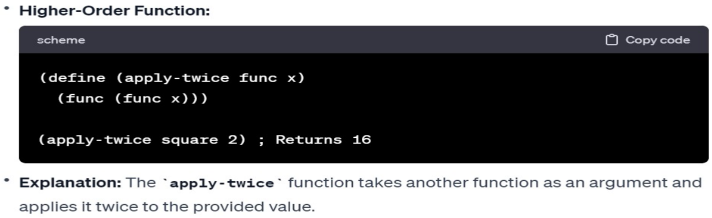


```scheme
(define (square x)
  (* x x))

(define my-function square)

(my-function 4) ; Returns 16
```

- High order functions- It takes one or more functions as arguments and/or returns a function as its result.



- **Higher-Order Function:**

```scheme
(define (apply-twice func x)
  (func (func x)))

(apply-twice square 2) ; Returns 16
```

- **Explanation:** The `apply-twice` function takes another function as an argument and applies it twice to the provided value.

Key features of Scheme

- Minimalism: Scheme has a small, clean, and simple syntax with a minimal set of core features. This makes it easy to learn and use.
- Lexical Scoping: Scheme adopts lexical scoping, wherein the scope of a variable is determined by its placement in the source code. This stands in contrast to dynamic scoping, where scope is determined by the sequence of function calls.
- First-Class Functions: Functions in Scheme are treated as first-class citizens, signifying that they can be assigned to variables, passed as arguments to other functions, and returned as values from functions.
- Recursion: Scheme promotes the utilization of recursion as a primary control structure. Iteration is frequently expressed through recursion rather than traditional loop constructs.
- Dynamic Typing: Scheme is dynamically typed, enabling variables to hold values of any type.
- Macro System: Scheme possesses a robust macro system that empowers users to define their own syntactic extensions. This facilitates the creation of domain-specific languages and enhances expressiveness.
- Immutable Data Structures

Concurrent Programming

- Concurrent programming is a paradigm in software development that focuses on executing multiple tasks or processes simultaneously to improve program performance and responsiveness.
- Tasks are called threads or processes, which can run independently, share resources, and Interact with each other.
- Concurrent programming plays a crucial role in the development of software systems that are high-performing, scalable, and responsive.
- **Concurrency vs Parallelism**: Concurrency is about managing multiple tasks in overlapping time periods, while parallelism is about executing multiple tasks simultaneously.

Concurrent Programming

- Concurrent programs can be represented using threads, processes, or asynchronous tasks, depending on the programming language and platform.
- Shared variables, message passing, and synchronization primitives are widely used techniques to facilitate communication and coordination between concurrent processes.
- **How do concurrent programmes work?**
  Concurrent programs work by executing multiple tasks or threads simultaneously, allowing for increased efficiency and better use of computing resources. They divide complex tasks into smaller subtasks, which can be processed concurrently by separate threads or processes. These threads or processes then work independently while sharing computing resources, such as memory and processing power. To ensure correct results and prevent conflicts, concurrent programs use synchronization techniques and mechanisms to manage the coordination and communication between threads.
- **Basic Principle** -Parallelism (multiple processes or threads running simultaneously), Non-determinism (unpredictable execution order, different result on different runs), and Synchronization (coordination and mutually exclusive access to shared resou to prevent data inconsistency and race conditions).

Key concept in Concurrent Programming

- **Processes**: Independent units of execution with their own memory space. Processes do not share memory directly and communicate through inter-process communication (IPC).
  Threads: Lighter-weight units of execution within a process. Threads share the same memory space, simplifying communication but requiring synchronization mechanisms to avoid data conflicts.
- **Shared Memory** : Concurrent entities communicate by sharing a common area of memory. Synchronization mechanisms are needed to control access and ensure data consistency.
  Message Passing: Entities communicate by sending messages to each other. This can be achieved through inter-process communication (IPC) mechanisms, message queues, or other communication channels.
- **Deadlocks**: Situations where two or more processes are unable to proceed because each is waiting for the other to release a resource.
  Race Conditions: Occur when the behavior of a program depends on the relative timing of events, leading to unpredictable results.
- **Mutex** (Mutual Exclusion): A mutex is a synchronization primitive that allows o one thread or process to access a shared resource at a time. It helps prevent rac conditions.

Why Concurrent Programming

- Performance Improvement: Concurrent programs boost efficiency by efficiently utilizing multiple CPU cores, resulting in significant enhancements for CPU-bound tasks.
- Responsiveness: Concurrency maintains application responsiveness during background operations, crucial for tasks in user interface applications.
- Resource Utilization: Concurrent programs optimize system resources, including CPU time, memory, and I/O devices, leading to efficient resource management.
- Parallelism.
- Platform independent.
- Safe from bugs : Concurrency bugs are some of the hardest bugs to find and fix, and require careful design to avoid.
- Easy to understand: Predicting how concurrent code might interleave with other concurrent code is very hard for programmers to do. It's best to design in such a way that programmers don't have to think about that.

ErLang

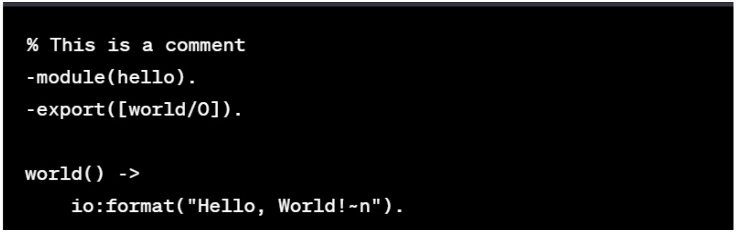
- The Erlang programming language is a general-purpose, simultaneous and garbage-collected programming language, which also serves as a runtime system.
- Open Telecom Platform (OTP) is a large collection of libraries for Erlang to perform in.
- Erlang is dynamically typed, has immutable data and a pattern matching syntax.
- Distributed and fault-tolerant (a piece of failing software or hardware should not bring the system down).
  Concurrent (it can spawn many processes, each executing a small and well-defined piece of work, and isolated from one another but able to communicate via messaging)
  Hot-swappable (code can be swapped into the system while it's running, leading to high availability and minimal system downtime).

Why ErLang

- Erlang is specifically designed to gracefully handle failures by incorporating a built-in error management mechanism known as "supervisors". With this mechanism, Erlang can automatically detect and recover from errors,. This makes it ideal for building systems that need to be highly available and reliable.
- Erlang is designed to be highly concurrent. It can handle a large number of requests and processes simultaneously. This makes it ideal for building systems that need to scale to handle large amounts of traffic.
- Erlang is optimized for low-latency and high-throughput systems. It has a lightweight process model. This model allows it to handle a large number of concurrent processes without incurring a significant performance overhead.
- Erlang offers a built-in message-passing mechanism that facilitates quick and efficient communication between processes. This feature simplifies the development of distributed systems that require seamless data exchange between different nodes.

ErLang

- Erlang has a clear and simple syntax. It uses periods (.) to terminate statements.
- Basic code:



```
% This is a comment
-module(hello).
-export([world/0]).

world() ->
    io:format("Hello, World!~n").
```

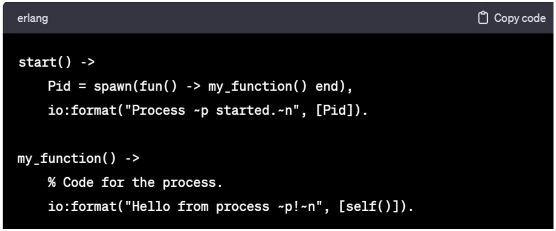
  Module Declaration: A module is defined with the -module(ModuleName). directive.
  Export Declaration: Functions to be accessed from outside the module are declared with -export([FunctionName/Arity])..
  Function Definition: Functions are defined using the FunctionName(Parameters) -> Body. syntax.
  Print to Console: io:format/1 is used for printing.
- Erlang makes heavy use of recursion. Since data is immutable in Erlang, the use while and for loops (where a variable needs to keep changing its value) is not allowed.

ErLang

- Erlang has eight primitive data types:
  Integers,Atoms,Floats,Refrences,Binaries,Pids,Ports,Funs.
  Three Compound data type:Tuples,Lists,Maps.
- Pattern matching is a powerful feature in Erlang used in function clauses and assignments.
  % Pattern Matching in Function Clause
  is_zero(0) -> true;
  is_zero(_) -> false.
  % Pattern Matching in Assignment
  X, Y, Z = 1, 2, 3.
- Erlang functions are defined with the fun keyword. Anonymous functions are commonly used.
  % Named Function
  add(X, Y) ->
  X + Y.
  % Anonymous Function
  Multiply = fun(X, Y) -> X * Y end.

ErLang

- Erlang's concurrency model is based on lightweight processes. Processes are created using the spawn function.



erlang Copy code

```erlang
start() ->
    Pid = spawn(fun() -> my_function() end),
    io:format("Process ~p started.~n", [Pid]).

my_function() ->
    % Code for the process.
    io:format("Hello from process ~p!~n", [self()]).
```

- **What is Beam?** BEAM stands for Bogdan's Erlang Abstract Machine. It is named after its developer Bogumil Bogdan Hausman. It is an Erlang virtual machine, often confused with the Erlang runtime system(ERTS). It is used for executing user code in the Erlang runtime system. There are no preassumptions for processes, tables, ports, etc., in it. Not only Erlang but Elixir also operates on the BEAM. Along with an introduction to Erlang, let's discuss what Elixir is and how it differs from Erl

Comparison of Functional and Concurrent Paradigm

Functional	Concurrent
Focus on Computation: Emphasizes the evaluation of mathematical functions and the avoidance of mutable state. Computation is expressed as the evaluation of expressions.	Focus on concurrent and parallel execution of tasks or processes. It deals with managing multiple computations that may execute simultaneously.
Immutability: Promotes immutable data structures, reducing the need for locks or synchronization mechanisms. Minimizes shared mutable state and side effects.	Shared State: Often involves managing shared mutable state. Synchronization mechanisms, like locks or semaphores, are required to prevent race conditions.
Promotes pure functions that do not have side effects, making it easier to reason about and test code.	Acknowledges the presence of side effects, especially in shared state scenarios. Requires careful handling to ensure correctness.
Immutability contributes to better error handling, as state remains consistent.	Error handling in concurrent programs needs to address challeng such as deadlocks, race conditions.

Comparison of Functional and Concurrent Paradigm

Functional	Concurrent
Declarative Style: Focuses on what the program should accomplish rather than specifying how to achieve it. Expresses computations as the composition of functions.	Imperative and Declarative: Involves both imperative (specifying the sequence of steps) and declarative style depending on the concurrency model used.
Deterministic: Pure functions produce the same output for the same input, promoting determinism and predictability.	Non-Deterministic: Concurrent programs may have non-deterministic outcomes due to the unpredictable interleaving of tasks.
Easier Debugging: Due to immutability and lack of side effects, debugging is often more straightforward.	Complex Debugging: Debugging concurrent programs can be more challenging due to potential race conditions and timing issues.
Languages: Haskell, Scala, Lisp,Scheme and functional features in languages like JavaScript and Python	Languages: Java, Go, Erlang, and languages with built-in support threads or processes.

Comparison of Scheme and ErLang

Paradigm:

- Scheme: Primarily a functional programming language.
  Emphasizes immutability, first-class functions, and simplicity.
- ErLang:Erlang is designed for concurrent and distributed systems. It is built to handle large-scale, fault-tolerant, and real-time applications.

Syntax:

- Scheme: Lisp dialect with a minimalistic and consistent syntax.
  Scheme uses prefix notation
  Scheme code may be standalone without the need for explicit module declarations.
  Functions are defined using the define keyword, and function application is done by enclosing the function and its arguments in parentheses.
- ErLang: Unique syntax influenced by Prolog.
  Erlang uses a more traditional infix notation.
  Erlang requires module declarations.
  functions are defined using the fun keyword, and the end of a function is denoted by a period (.).

Distribution

- Scheme:Portable code but not inherently designed for distributed systems.
- Erlang:Built for distributed systems with easy scalability. Features like hot code swapping support continuous operation during updates.

Comparison of Scheme and ErLang

Concurrency and Fault tolerance

- Scheme: Limited built-in support for concurrency and fault tolerance. Concurrency features may need to be added through external libraries.
- ErLang: Built around the Actor model. Has a built-in concurrency model with lightweight processes and message passing contribute to fault tolerance.

Community and Support

- Scheme: Has a diverse but smaller community compared to more mainstream languages.
- Erlang: Has a specialized community focused on building fault-tolerant and concurrent systems and strong support from Ericsson.

Learning Curve

- Scheme: Considered beginner-friendly due to its simple syntax.
- Erlang: Learning Erlang might be steeper, especially for those new to its syntax and concurrency model.

Comparison of Scheme and ErLang

Use cases

- Scheme: General-purpose programming, scripting, and educational purposes. Used in various applications but not specifically tailored for concurrent or distributed systems.
- Erlang: Telecommunications, distributed and concurrent systems. Particularly suited for applications requiring high concurrency, fault tolerance, and distributed computing.

Execution:

- Scheme: Scheme is an interpreted language, which means that the code is usually executed by an interpreter at runtime.
- ErLang: Erlang code is typically compiled into bytecode (BEAM bytecode) and executed on the Erlang Virtual Machine (BEAM VM).

- Both are dynamically typed.
- Both languages typically use garbage collection to manage memory automatically.

Application

- **Functional Programming**: Functional programming languages are often preferred for academic purposes, rather than commercial software development. Nevertheless, several prominent functional languages like Clojure, Erlang, F, Haskell, and Racket, are used for developing a variety of commercial and industrial applications.
- **Concurrent Programming**:Concurrent programming finds applications in diverse domains, including telecommunications for handling multiple connections, web servers for managing concurrent requests, database systems for simultaneous queries, real-time systems, parallel processing, cloud computing for scalable services, game development, and operating systems for multitasking. Its role is critical in ensuring responsiveness, scalability, and efficient resource utilization in modern computing environments.
- **Scheme**:Scheme, a functional programming language, finds applications in various domains, particularly in educational settings, prototyping, and scripting. Its minimalistic syntax and expressive nature make it suitable for teaching programming concepts. Scheme is often used in academia to introduce fundamental principles due to its simplicity.
- **ErLang**:Erlang is widely used in telecommunications for its concurrency and fault-tolerance, making it suitable for real-time messaging, chat applications(Whatsapp), and distributed systems. Its lightweight processes and f on fault tolerance contribute to the development of highly reliable and scalable applications.

References

- https://www.composingprograms.com/pages/32-functional-programming.html
- https://www.scheme.com/tspl3/intro.html
- https://eecs390.github.io/notes/functional.html
- ChatGPT prompt: give code in scheme for functional programming features like first class citizen, high order function
- https://www.studysmarter.co.uk/explanations/computer-science/computer-programming/concurrent-programming/
- https://insights.sei.cmu.edu/documents/1549/1990_007_001_15815.pdf
- https://tipsontech.medium.com/concurrent-programming-in-java-863d2a3f3c1
- https://web.mit.edu/6.005/www/fa14/classes/17-concurrency/concurrency_is_hard_to_test_and_debug