Amrita Vishwa Vidyapeetham
TIFAC-CORE in Cyber Security

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

Suvetha D P

21st January, 2024

Introduction to Paradigm

- A programming paradigm is a fundamental approach or style of programming that provides a set of principles, concepts, and techniques for designing and implementing computer programs.
- It defines the structure, organization, and flow of the code, as well as the methodologies for problem-solving and expressing computations.
- Different paradigms have their strengths and weaknesses, and choosing the right paradigm for a given task can greatly impact the efficiency, maintainability, and scalability of a program.

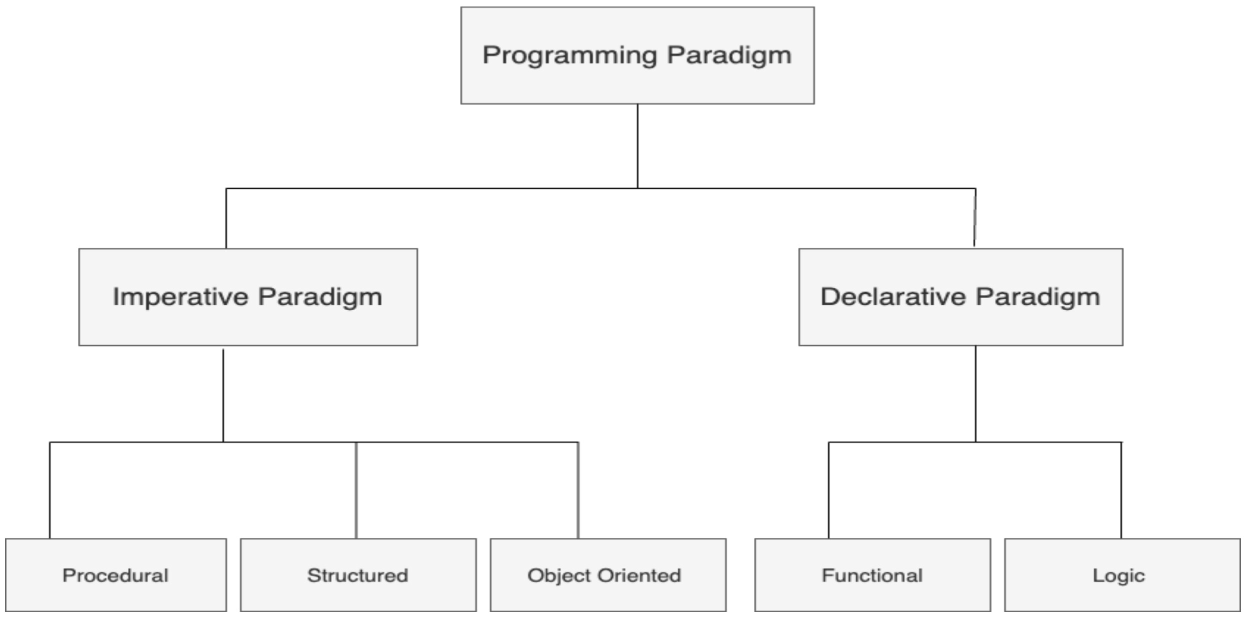


Figure 1: Programming Paradigm

Imperative Programming Paradigm

It works by changing the program state through assignment statements. It performs step by step task by changing state. The main focus is on how to achieve the goal. The paradigm consist of several statements and after execution of all the result is stored.

Declarative Programming Paradigm

It is a style of building programs that expresses logic of computation without talking about its control flow. It often considers programs as theories of some logic.It may simplify writing parallel programs.The focus is on what needs to be done rather how it should be done basically emphasize on what code is actually doing. It just declares the result we want rather how it has be produced.

- The only difference between imperative and declarative programming paradigms is how to do and what to do.

Paradigm 1: Functional Paradigm

Functional Paradigm

Functional programming is a programming paradigm which has its roots in mathematics, primarily evolved from lambda calculus.It is a declarative type of programming style. Its main focus is on "what to solve".

This is the style of writing programs through the compilation of a set of functions. Its basic principle is to wrap almost everything in a function, write many small reusable functions, and then simply call them one after another.

The primary aim of this style of programming is to avoid the problems that come with shared state, mutable data and side effects which are commonplace in object oriented programming. Functional programming tends to be more predictable and easier to test than object oriented programming.

Difference between Functional Paradigm and Object Oriented Programming

Functional Paradigm

- This programming paradigm emphasizes on the use of functions where each function performs a specific task.
- Fundamental elements used are variables and functions.The data in the functions are immutable(cannot be changed after creation).
- Importance is not given to data but to functions.
- It follows declarative programming model.
- It uses recursion for iteration.
- It is parallel programming supported.
- The statements in this programming paradigm does not need to follow a particular order while execution.
- Does not have any access specifier.

- To add new data and functions is not so easy.
- No data hiding is possible. Hence, Security is not possible.

Object Oriented Programming

- This programming paradigm is based on object oriented concept. Classes are used where instance of objects are created.
- Fundamental elements used are objects and methods and the data used here are mutable data.
- Importance is given to data rather than procedures.
- It follows imperative programming model.
- It uses loops for iteration.
- It does not support parallel programming.
- The statements in this programming paradigm need to follow an order i.e., bottom up approach while execution.
- Has three access specifiers namely, Public, Private and Protected.
- Provides an easy way to add new data and functions.
- Provides data hiding. Hence, secured programs are possible.

Principles and Concepts of Functional Paradigm

- Purity
- Immutability
- Disciplined State
- First class functions and high order functions
- Type systems
- Referential transparency
- Recursion

Purity

A pure function is a function which always returns the same output when called with the same argument values. It has no side effects like modifying an argument or global variable or outputting something.

Pure functions are predictable and reliable. Most of all, they only calculate their result. The only result of calling a pure function is the return value.

For example,The computation in square function depends on the inputs. No matter how many times we call a square function with the same input, it will always return the same output.

Immutability

It means that once you assign a value to something, that value won't change.This eliminates side effects (anything outside of the local function scope), for instance, changing other variables outside the function.

Immutability helps to maintain state throughout the runtime of a program. Since your function has a disciplined state and does not change other variables outside of the function, you don't need to look at the code outside the function definition.

Disciplined State

A shared mutable state is hard to keep correct since there are many functions that have direct access to this state.It is also hard to read and maintain.With mutable state, you need to look up for all the functions that use shared variables, in order to understand the logic. It's hard to debug for the very same reason.

When you are coding with Functional Programming principles in mind, you avoid, as much as possible, having a shared mutable state. Of course you can have state, but you should keep it local, which means inside your function. This is the state discipline : you use state,but in a very disciplined way.

First class functions and high order functions

Functional programming treats functions as first-class citizens.This means that functions are able to be passed as arguments to other functions, returned as values from other functions, stored in data structures and assigned to variables.

Higher order function is the function that takes one or more functions as arguments or returns a function as its result.

Type System

Type systems help the compiler in making sure that you only have the right types as arguments, turn statements, function composition, and so on. The compiler will not allow you to make any basic mistakes.

For example, if you have a function that receives a string, it could be dangerous,because you can pass pretty much anything in a string, say abc.

fun assignRole(role : String) { }

Another great benefit of strong typing is that you have better documentation, as your code becomes your documentation, and it's way more clear what you can or can't do.

Referential transparency

An expression is said to be referentially transparent if it can be replaced with its corresponding value without changing the program's behaviour.

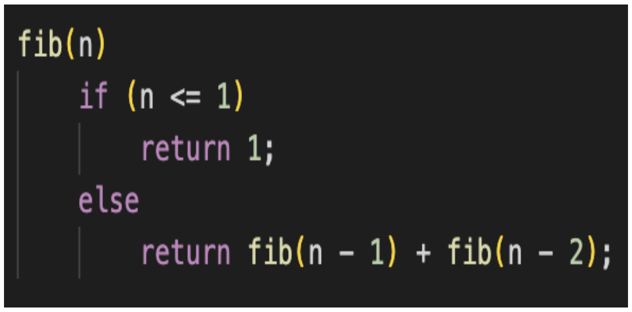
To achieve referential transparency, a function must be pure. This has benefits in terms of readability and speed. Compilers are often able to optimize code that exhibits referential transparency.

```
fun add(a: Int, b: Int) = a + b
fun sub(a: Int, b:Int) = a — b
val x = add(2, sub(7, 3))
```

In this example, the sub method is referentially transparent because any call to it may be replaced with the corresponding return value. This may be observed by replacing sub(7, 3) with 4.

Recursion

There are no "for" or "while" loop in functional languages. Iteration in functional languages is implemented through recursion. Recursive functions repeatedly call themselves, until it reaches the base case.



```
fib(n)
    if (n <= 1)
        return 1;
    else
        return fib(n - 1) + fib(n - 2);
```

Figure 2: Recursion

Principles and Concepts of functional programming makes our code more readable, predictable and testable. This allows us to have code which contains less bugs, easier on boarding and a generally nicer code base

Language for Paradigm 1: F#

- F# is pronounced as F Sharp. It is a functional programming language that supports approaches like object oriented and imperative programming approach. It is a cross-platform and .Net Framework language. The filename extension for F# source file is .fs.
- It was designed and developed by Microsoft. It was first appeared in 2005. Current stable version of F# is 4.0.1.20 which was released on November 13, 2016.
- It is influenced by Python, Haskell and Scala. It influenced to C#, F* etc.
- F# interacts with .Net library so that it can access the entire library and tools.

History

- F# emerged from the fertile ground of Microsoft Research in Cambridge, UK, in the late 1990s. Don Syme, its primary architect, envisioned a language that blended the purity of functional programming with the practicality of the .NET ecosystem. The "F" is often attributed to "Fun," reflecting the team's desire to create an enjoyable and expressive coding experience.
- Initially used within Microsoft for internal projects, F# gained traction due to its ability to tackle complex scientific and financial computations with elegance and efficiency. Its type safety, immutability, and powerful data structures resonated with developers seeking reliable and scalable solutions.
- In 2005, F# officially became part of the .NET platform, opening its doors to a wider audience. Visual F# tools for Visual Studio made development more accessible, and libraries blossomed to support various domains like web development, machine learning, and data analysis.
- Today, F# thrives as a mature and versatile language. Its integration with .NET allows seamless collaboration with other languages and tools within the ecosystem. The thriving community constantly innovates, pushing the boundaries of F# with new libraries, frameworks, and applications.
- From its research lab roots to its real-world applications, F# has come a long way. Its unique blend of functional elegance and practical power continues to attract developers seeking a different, yet familiar, way to build impactful software solutions. The future of F# shines bright, as its community and applications continue to grow, leaving a mark on the ever-evolving landscape of programming languages.

Characteristics

- Lightweight syntax
- Immutable by default
- Type inference and automatic generalization
- First-class functions
- Powerful data types
- Pattern matching
- Async programming

Lightweight syntax

- Functional languages are often praised for their concise and readable syntax. This means less code to write and maintain, allowing developers to focus on the core logic of their programs.
- Boilerplate code, the repetitive parts that can clutter code, is minimized in functional languages. This makes the code cleaner and easier to follow.
- The syntax often directly mirrors the concepts of the problem domain, making it more intuitive to express problem solutions.

Immutable by default

- In most functional languages, data values are immutable by default. This means that once a value is created, it cannot be modified.
- This immutability leads to more predictable and reliable code, as there is no risk of unexpected side effects from modifying data in place.
- It also encourages a functional style of programming, where data is transformed through pure functions rather than mutated directly.
- Immutability also promotes thread safety, as concurrent access to immutable data is inherently safe without the need for complex synchronization mechanisms.

Type Inference and Automatic Generalization

- Functional languages often have powerful type inference systems, which can automatically deduce the types of data and functions without the need for explicit type annotations.
- This reduces boilerplate code and makes code more concise.
- It also enables the creation of generic functions that can work with a variety of data types, further promoting code reusability and flexibility.

First-Class Functions

- Functions are treated as first-class citizens in functional languages. This means that they can be assigned to variables, passed as arguments to other functions, and returned as results from functions.
- This enables powerful abstractions and higher-order functions, such as map, filter, and fold, which can concisely operate on collections of data.
- First-class functions also facilitate techniques like currying and partial application, providing flexible ways to compose and reuse functions.

Powerful Data Types

- Functional languages offer a rich set of built-in data structures, including lists, tuples, records, unions, options, and more.
- Algebraic data types (ADTs) allow for the creation of custom types with multiple variants, providing a structured way to model complex data.
- Pattern matching provides an elegant and concise way to decompose data structures based on their patterns, extracting values and handling different data types effectively.

Pattern Matching

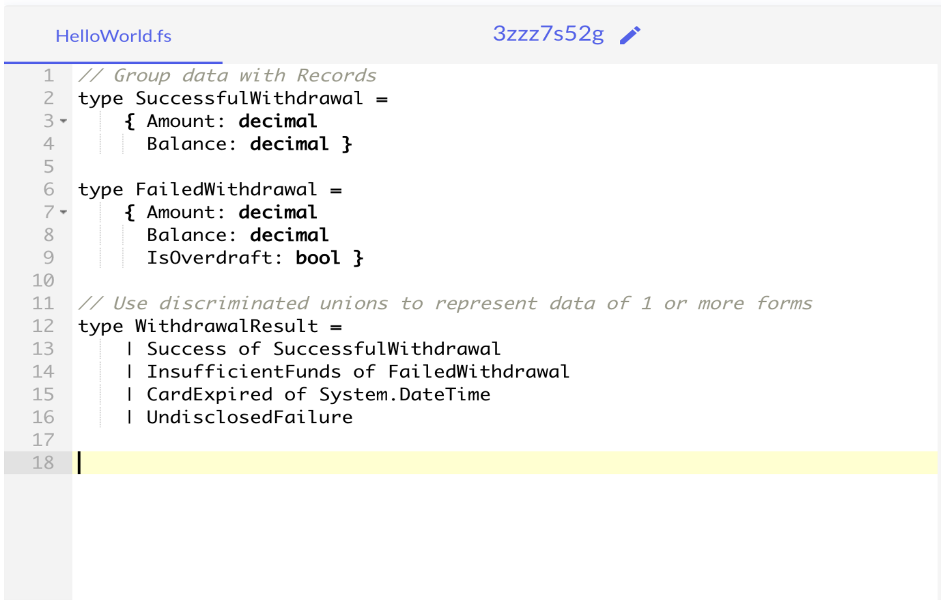
- Pattern matching is a powerful tool for decomposing data structures based on their patterns.
- It can be used in function arguments, conditional branching, and other contexts to extract values and handle different data types in a concise and expressive way.
- Pattern matching often leads to more readable and maintainable code compared to traditional conditional statements.

Async Programming

- Functional languages often have built-in constructs for asynchronous programming, allowing the handling of long-running tasks without blocking the main thread.
- This improves responsiveness and concurrency, making it ideal for tasks involving I/O, network requests, and other asynchronous operations.
- Async programming features enable efficient handling of these tasks while maintaining a responsive user experience.

Rich data types

Types such as Records and Discriminated Unions let you represent your data.



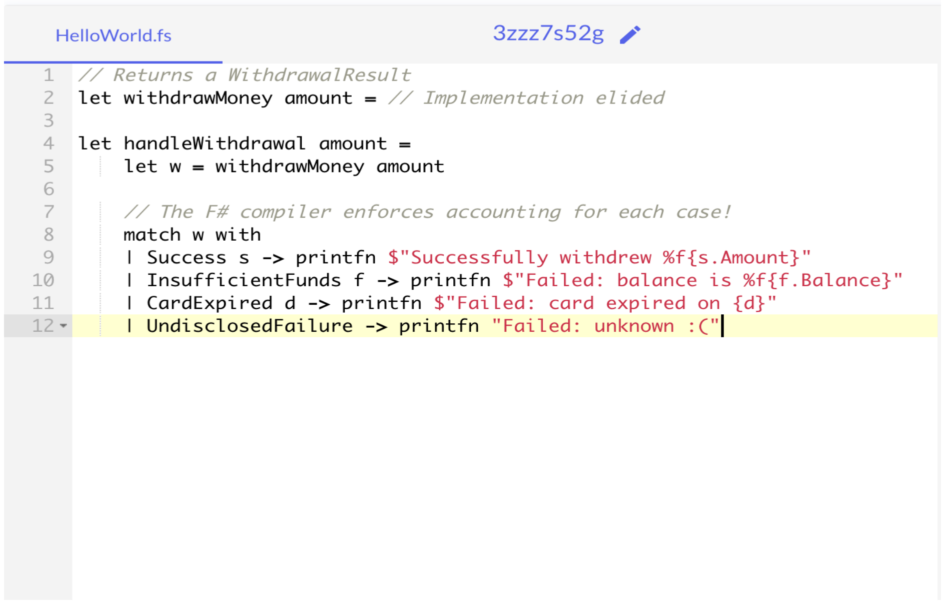
```fsharp
// Group data with Records
type SuccessfulWithdrawal =
    { Amount: decimal
      Balance: decimal }

type FailedWithdrawal =
    { Amount: decimal
      Balance: decimal
      IsOverdraft: bool }

// Use discriminated unions to represent data of 1 or more forms
type WithdrawalResult =
    | Success of SuccessfulWithdrawal
    | InsufficientFunds of FailedWithdrawal
    | CardExpired of System.DateTime
    | UndisclosedFailure

```

Figure 3: Data Type

F# records and discriminated unions are non-null, immutable, and comparable by default, making them very easy to use.

Correctness with functions and Pattern Matching

F# functions are easy to define. When combined with pattern matching, they allow you to define behavior whose correctness is enforced by the compiler.



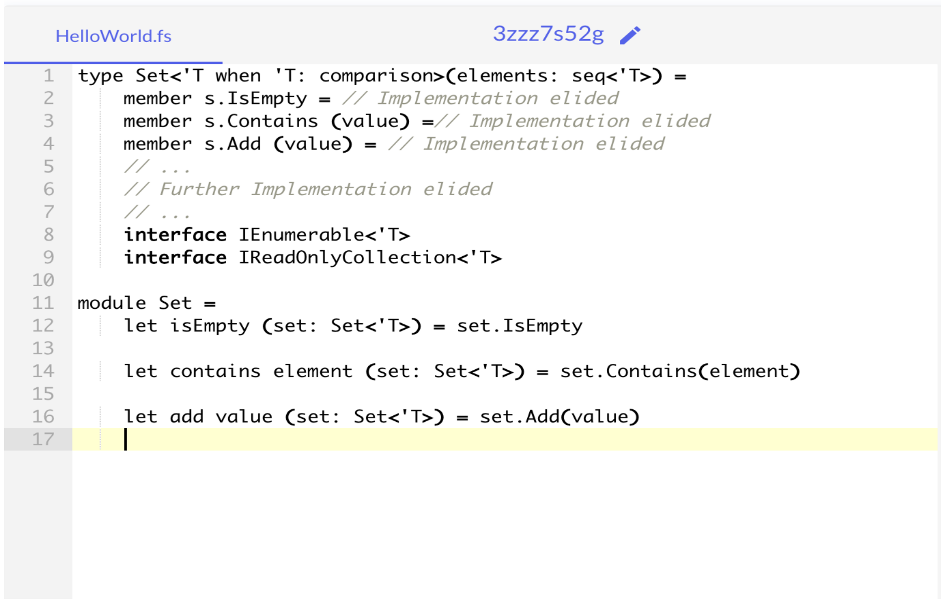
```fsharp
// Returns a WithdrawalResult
let withdrawMoney amount = // Implementation elided

let handleWithdrawal amount =
    let w = withdrawMoney amount

    // The F# compiler enforces accounting for each case!
    match w with
    | Success s -> printfn $"Successfully withdrew %f{s.Amount}"
    | InsufficientFunds f -> printfn $"Failed: balance is %f{f.Balance}"
    | CardExpired d -> printfn $"Failed: card expired on {d}"
    | UndisclosedFailure -> printfn "Failed: unknown :("
```

Figure 4: Pattern Matching

F# functions are also first-class, meaning they can be passed as parameters and returned from other functions.

Functions to define operations on objects

F# has full support for objects, which are useful when you need to blend data and functionality. F# members and functions can be defined to manipulate objects.



```
type Set<'T when 'T: comparison>(elements: seq<'T>) =
    member s.IsEmpty = // Implementation elided
    member s.Contains (value) =// Implementation elided
    member s.Add (value) = // Implementation elided
    // ...
    // Further Implementation elided
    // ...
    interface IEnumerable<'T>
    interface IReadOnlyCollection<'T>

module Set =
    let isEmpty (set: Set<'T>) = set.IsEmpty

    let contains element (set: Set<'T>) = set.Contains(element)

    let add value (set: Set<'T>) = set.Add(value)
```

Figure 5: Functions

In F#, you will often write code that treats objects as a type for functions to manipulate. Features such as generic interfaces, object expressions, and judicious use of members are common in larger F# programs.

Features

- Conciseness
- Convenience
- Correctness
- Concurrency
- Completeness

Conciseness

F# provides clean and nice code to write no curly brackets, no semicolons and so on. Even you don't have to specify type in your code just because of type inference. Moreover, you can finish your code in less line compared to other languages.

Convenience

Common programming tasks are much simpler in F#. You can easily define and process your complex problems. Since functions are first class object so it is very easy to create powerful and reusable code by creating functions that uses other functions as a parameter.

Correctness

F# provides powerful type system which helps to deal with common type errors like null reference exception etc. F# is a strongly typed language which helps to write error free code. It is easily caught at compile time as a type error.

Concurrency

F# provides number of built-in functions and libraries to deal with programming system when multiprocessing is occurred. F# also supports asynchronous programming, message queuing system and support for event handling. Data in F# is immutable by default so sharing of data is safe. It avoids lock during code communication.

Completeness

F# is a functional programming language but it also supports other programming approaches like object oriented, imperative etc. which makes it easier to interact with other domains. Basically, we can say that F# is designed as a hybrid language by which you can do almost everything that you can do with other programming languages like C#, Java etc.

Applications

F# has a rich set of library and supports multi-paradigm which helps to deal with every domain whether it is desktop based application, web based application or mobile based application.

- Data analysis
- Scientific research
- Data statistical
- Design games
- Artificial Application
- Desktop application
- Mobile application

Paradigm 2: Logic Paradigm

Logic Paradigm

A logic programming paradigm is a set of principles and techniques that guide the design and implementation of logic programs.

A logic program consists of a collection of facts and rules that describe the relationships and properties of entities, and a query language that allows asking questions and obtaining answers from the program.

A logic programming paradigm defines the syntax and semantics of the facts, rules, and queries, as well as the inference mechanism that derives new facts and rules from the existing ones.

Major logic programming language families include Prolog, Answer Set Programming (ASP) and Datalog.

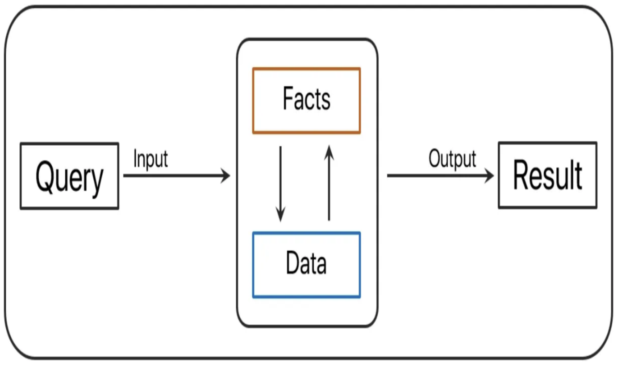


Figure 6: logic Workflow

Principles and Concepts of Logic Paradigm

- Declarative Nature
- Logic Rules and Facts
- Predicate Logic
- Unification
- Backtracking
- Negation as Failure

Declarative Nature

Logic programming emphasizes what needs to be achieved rather than how to achieve it. Programs are expressed as a set of logical statements that declare relationships and conditions, leaving the execution details to the underlying logic programming system.

Logic Rules and Facts

Rules: Logical relationships are defined using rules, typically in the form of Horn clauses. Rules consist of a head and a body, with the body specifying conditions that, when satisfied, lead to the conclusion stated in the head.
Facts: Facts represent basic truths or knowledge in the form of simple statements.
Eg: father(charles, william) -> Charles is William's father.

Predicate Logic

Predicate logic is a formal system that uses predicates to express relationships and conditions. Predicates are statements that involve variables and constants, and they evaluate to true or false.

Unification

Unification is a process used in logic programming to find substitutions for variables that make two logical expressions equivalent. It plays a crucial role in pattern matching and solving queries.
Eg: For the query "parent(mary, X)" and the rule "parent(Y, Z) :- mother(Y, Z)," unification binds Y to "mary" and Z to "X," making the query and rule heads identical.

Backtracking

Backtracking is a search strategy employed by logic programming systems to explore alternative paths when a failure occurs. It involves undoing decisions and exploring different branches of the search space.

Negation as Failure

In logic programming, negation is often implemented as failure. If a query cannot be proven true, it is assumed false. This concept, known as "negation as failure," is a way of expressing negation in a logic programming context.

Eg: If the program tries to prove "friend(john, bob)" using various rules but fails in all attempts, it can then infer "not friend(john, bob)" under negation as failure.

Language for Paradigm 2: Mercury

- Mercury, a general-purpose programming language, originated from the University of Melbourne, Australia. It embodies the paradigm of purely declarative programming, designed for creating robust real-world applications by seamlessly integrating logic and functional programming.
- It enforces a disciplined approach by necessitating type, mode, and determinism declarations for predicates and functions. The compiler rigorously checks these declarations, enhancing reliability and productivity. The correctness of declarations not only prevents certain errors but also serves as valuable documentation for maintenance, while optimizing the efficiency of the generated code.
- Mercury stands out as a language that prioritizes reliability, productivity, and efficiency through its declarative paradigm, stringent type checking, and the integration of logic and functional programming features.

History

- Mercury's story begins in the 1990s, a time when logic programming languages like Prolog held sway. A team at the University of Melbourne, led by Zoltan Somogyi, saw potential for a more practical and efficient logic language. Thus, Mercury was born, inheriting Prolog's declarative spirit but shedding its limitations.
- The key focus was on strong, static typing and a powerful mode system, ensuring predictable and error-free code. This made Mercury stand out, offering the precision of logic alongside the reliability of statically typed languages.
- Initially embraced by academics and AI researchers, Mercury's strengths soon attracted users in diverse fields. Its declarative style found favor in tasks like configuration management, natural language processing, and rule-based systems. Its logic reasoning capabilities proved invaluable in solving constraint satisfaction problems in areas like scheduling and optimization.
- While not as widely adopted as some mainstream languages, Mercury carved its niche as a dedicated tool for logic programming. Its open-source nature and active community foster continuous development, with advancements in areas like performance optimization and language features.
- Today, Mercury stands as a testament to the power of logic programming in addressing specific problems. Its elegant syntax, declarative approach, and robust logic capabilities continue to attract developers seeking innovative solutions in domains where traditional languages fall short. Though not a household name, Mercury's impact on logic programming and its future as a niche but potent language remain strong.

Characteristics and Features of Mercury

Declarative

Mercury is purely declarative. Predicates and Functions in Mercury do not have non-logical side effects.

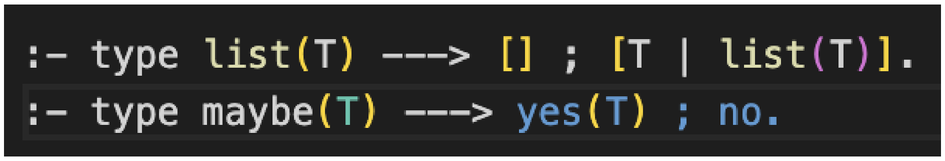
Mercury handles I/O through built-in and library predicates that operate on an old state of the world, returning a new state and potential results. The language enforces a destructive update model by requiring the input reference to the old state to be the last one. I/O is restricted to program sections where backtracking is unnecessary.

Mercury handles dynamic data structures by providing several abstract data types in the standard library that manage collections of items with different operations and tradeoffs. Programmers can also create their own abstract data types.

Strongly Typed Language

Mercury's type system is based on many-sorted logic with parametric polymorphism, very similar to the type systems of modern functional languages such as ML and Haskell.

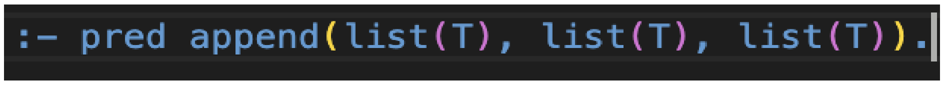
Programmers must declare the types they need using declarations such as



```
:- type list(T) ---> [] ; [T | list(T)].
:- type maybe(T) ---> yes(T) ; no.
```

Figure 7: Declaration

They must also declare the type signatures of the predicates they define, for example



```
:- pred append(list(T), list(T), list(T)).
```

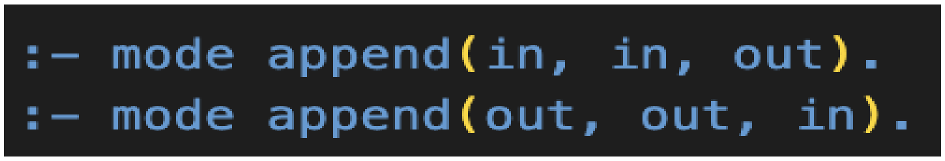
Figure 8: Declaration Type

The compiler infers the types of all variables in the program. Type errors are reported at compile time.

Strongly Moded Language

The programmer must declare the instantiation state of the arguments of predicates at the time of the call to the predicate and at the time of the success of the predicate. Currently only a subset of the intended mode system is implemented. This subset effectively requires arguments to be either fully input (ground at the time of call and at the time of success) or fully output (free at the time of call and ground at the time of success).

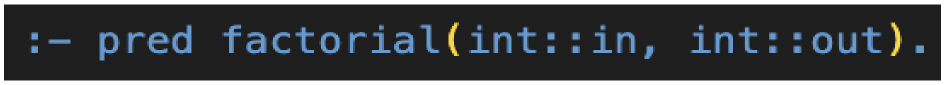
A predicate may be usable in more than one mode. For example, append is usually used in at least these two modes:



```
:- mode append(in, in, out).
:- mode append(out, out, in).
```

Figure 9: Predicates using two mode

If a predicate has only one mode, the mode information can be given in the predicate declaration.



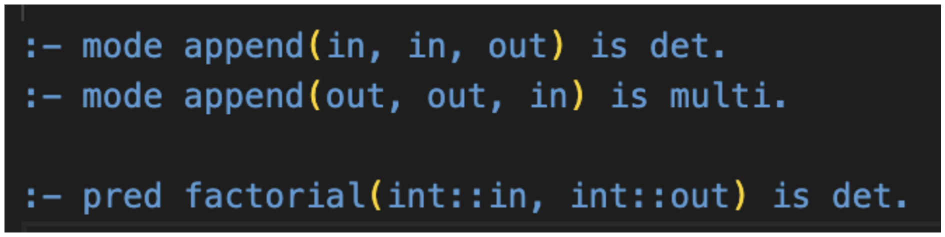
```
:- pred factorial(int::in, int::out).
```

Figure 10: Predicates using one mode

The compiler will infer the mode of each call, unification and other builtin in the program. It will reorder the bodies of clauses as necessary to find a left to right execution order; if it cannot do so, it rejects the program. Like type-checking, this means that a large class of errors are detected at compile time.

Strong Determinism System

For each mode of each predicate, the programmer should declare whether the predicate will succeed exactly once (det), at most once (semidet), at least once (multi) or an arbitrary number of times (nondet). These declarations are attached to mode declarations like this:



```
:- mode append(in, in, out) is det.
:- mode append(out, out, in) is multi.

:- pred factorial(int::in, int::out) is det.
```

Figure 11: Declarations of modes

The compiler will try to prove the programmer's determinism declaration using a simple, predictable set of rules that seems sufficient in practice (the problem in general is undecidable). If it cannot do so, it rejects the program.

As with types and modes, determinism checking catches many program errors at compile time. It is particularly useful if some deterministic (det) predicates each have a clause for each function symbol in the type of one of their input arguments, and this type changes; you will get determinism errors for all of these predicates, telling you to put in code to cover the case when the input argument is bound to the newly added function symbol.

Module System

Programs consist of one or more modules. Each module has an interface section that contains the declarations for the types, functions and predicates exported from the module, and an implementation section that contains the definitions of the exported entities and also definitions for types and predicates that are local to the module.

A type whose name is exported but whose definition is not, can be manipulated only by predicates in the defining module; this is how Mercury implements abstract data types. For predicates and functions that are not exported, Mercury supports automatic type, mode, and determinism inference.

Characteristics

- Mercury supports higher-order programming, with closures, currying, and lambda expressions.
- Strong types, modes, and determinism provide the compiler with the information it needs to generate very efficient code.
- The Mercury compiler is written in Mercury itself. It was bootstrapped using NU-Prolog and SICStus Prolog. This was possible because after stripping away the declarations of a Mercury program, the syntax of the remaining part of the program is mostly compatible with Prolog syntax.
- The Mercury compiler compiles Mercury programs to C, which it uses as a portable assembler. The system can exploit some GNU C extensions to the C language, if they are available: the ability to declare global register variables, the ability to take the addresses of labels, and the ability to use inline assembler. Using these extensions, it generates code that is significantly better than all previous Prolog systems known to us. However, the system does not need these extensions, and will work in their absence.

Applications

- Medical Diagnosis and Decision Support
- Financial Risk Management and Trading
- Robotics and Automation
- Logistics and Supply Chain Management
- Software Development and Testing

Analysis

Functional Paradigm

Advantages

- Simplicity
- Scalability
- Code Reusability
- Testing and Debugging
- Readability
- Lazy Evaluation
- No Side Effects

Simplicity

The programming languages those are based on the functional paradigms are known as pure functional languages. They are only dependent on the input. So the output is definitely the return value. By looking at the function signature, programmer can easily identify important details. And also, pure functions create a clean structure which is the reason for the programmers to write a simple code.

Scalability

A program developed needs to handle multiple threads simultaneously. Otherwise, there can be memory inconsistencies within the threads. If multiple threads can read/write precisely, we can say that the program is scalable.

Code Reusability

The functions used in a Functional Paradigm programming is made to be reusable. Regardless of any scenario it can be reused. In case of necessity, the functions can be passed as parameters or else returned again as functions.

Testing and Debugging

The testing and debugging process is also easier with pure functions. There is only input values in pure functions which makes it simple to identify errors. And while handling errors, you need to deal with one exact function. As a result of this, everyone can enhance the correctness of their application.

Readability

Compared to imperatives, the codes in functional programming is less complicated. The time spent by a programmer in understanding the code structure is very less. Pure functions doesn't often change states which increases the readability of values.

Lazy Evaluation

Functional programming comes with a feature known as lazy evaluation. This ensures that repeated evaluations are prevented. Only the required values are stored. Eventually, the program will perform faster because all the unnecessary details needed for the queries are eradicated.

No Side Effects

To make sure that there is good coding approach, all the side effects needs to be necessarily be removed. If not, it can result in different outcomes for the same input. One popular example for a side effect is the shared variable. To overcome this, functional code makes use of immutability.

Disadvantages

- Performance
- Memory Consumption
- Tools and Framework Support
- Loops Compatibility
- Data Duplication

- Mathematical Concepts
- Community Support

Performance

Immutability in functional programming approach is well known to generate garbage. Therefore, the tasks assigned will require significant amount of variables which will eventually decrease performance.

Memory Consumption

Although immutable variables does not alter the existing variables, a new variable is created instead. This is a problematic situation where the actual data just needs to be copied. One such example is the nested data structure. If the data is frequently copied, it will further increase the memory consumption.

Tools and Framework Support

Most of the cloud service providers and other services impose restrictions with regards to the programming language. As compared to other programming languages like JavaScript or Java, the community in functional programming is basically smaller. Hence, very less number of tools and frameworks are supported.

Loops Compatibility

Functional programming does not support loops. Thus, for iterations there is no proper method other than recursions. Unfortunately, the concept of recursion are bit complicated similar to mathematics. Especially, for the people are only familiar with the OOP approach.

Data Duplication

Functional programming also struggles in cases where is a need to duplicate data. One such instance is when handling data structures. In the above mentioned situation, functional programming does not allow you to update variable. The result of this is again the loss of performance.

Mathematical Concepts

Functional programs are logically connected to mathematics. The mathematical concepts included in functional programming languages are way too complex. This is one of the reasons why new developers often refuse to learn FP. It takes considerable amount of time and effort to grasp these concepts.

Community Support

Since the community is not so great in functional programming languages, we can say that the experts in this field is also lesser. If a developer encounters a problem in functional programming, they need to rely on third party vendors. Now these vendors might charge an amount for their service.

Functional Paradigm - F#

Advantages

Algebraic Data Types

F# promotes ADT entirely and helps organizations be more precisely defined with respect to data structures, reducing the probability of misinterpretation and improving coding consistency. Basically, in F#, unnecessary circumstances can be avoided by rendering them unpresentable, simply by not allowing an incorrect source code to be written.

Transformations and Mutations

F# supports the use of transformations instead of mutations. The entity is not changed in a transformation, and so the condition does not change. It just builds a new object and leaves the existing unchanged. This implies that the entity will still be delivered in a single state at birth. It simplifies the creation process significantly when working with only one condition, which decreases the amount of code so once we see an entity, it is in the only possible state, and it is safe to use it in the absence of additional tests.

Concise Syntax

F# provides a sophisticated inference style scheme that deduces values types from how certain values are used. Thus in most situations, types in the code are to be omitted so that F# can be identified. Less findings must be provided in order to maximize efficiency and boost code maintenance. Some people consider the syntax F# is easier to read and elegant.

Disadvantages

Naming is More Challenging

F# has no overload feature like C# has for methods. So two F# functions that are stored in the same module can't have the same name, which makes it impossible to name them. It is a struggle in F# to come up with a clear naming convention.

More Complex Data Structures

More sophisticated data structures involve effective manipulation using the transformation-over-mutation method. For e.g., a binary tree in F# is to be used instead of a hash table as in C#. The extensive use of zippers instead of iterators would be another example.

Less Advanced Tools

Microsoft spends a great deal in making C# programmers the right tools. Sadly, there are not so many F# resources that make coding less convenient. F# has no basic methods for refactoring.

Notable Features of Functional Paradigm - F#

- Immutability
- Pure Functions
- Higher-Order Functions
- Pattern Matching

- Strong Typing
- Type Inference
- Concise and Expressive Syntax

Logic Paradigm

Advantages

- Logic programming can be used to convey knowledge in a way that is independent of implementation, allowing programmes to be more flexible, compressed, and understandable.
- It allows information to be separated from use, allowing the machine architecture to be altered without affecting programmes or their underlying code.
- It can be naturally altered and extended to accommodate special types of knowledge, such as meta-level or higher-order knowledge.
- It can be applied in non-computational disciplines that depend on reasoning and precise expression.

Disadvantages

- Initially, consumers were underserved due to a lack of investment in complementary technologies.
- In the outset, the lack of facilities for supporting arithmetic, types, and so on discouraged the programming community.
- There is no adequate method of representing computational concepts contained in built-in state variable mechanisms. (as is usually found in conventional languages).
- Some programmers always have, and always will prefer the overtly operational character of machine operated programs, since they prefer the active control over the 'moving parts'.

Logic Paradigm - Mercury

Advantages

- Declarative approach, focusing on what needs to be achieved rather than how.
- Powerful reasoning and inference capabilities through rules and facts.
- Expressive and concise syntax for readable code.
- Suitable for complex domains like natural language processing and constraint satisfaction.
- Static type checking for error prevention and code robustness.
- Tabling for efficiency to avoid redundant computations.

Disadvantages

- Non-deterministic execution due to backtracking, potentially leading to unpredictable behavior and debugging challenges.
- Limited concurrency support compared to some other languages.
- Steeper learning curve for those unfamiliar with logic programming paradigms.
- Error handling can be complex, especially for intricate problems.
- Smaller community and ecosystem, with fewer resources and libraries available.

Notable Features of Logic Paradigm - Mercury

- Declarative Approach
- Facts and Rules
- Unification
- Backtracking and Reasoning
- Pure Logic
- Static Type Checking

Comparison

Similarities

Declarative Style

Both F# and Mercury emphasize "what" needs to be accomplished rather than "how," focusing on describing problems through data and rules.

Immutability

Both languages promote immutable data structures, leading to predictable behavior and thread safety.

Strong Typing

Both F# and Mercury employ static type systems, catching errors early and improving code robustness.

Conciseness and Readability

Both strive for clean and concise syntax, promoting code readability and maintainability.

Expressive Power

Both offer powerful abstractions and data manipulation capabilities, making them suitable for complex problems.

Differences

Core Paradigm

- F#: Primarily functional with influences from imperative languages. Utilizes functions as first-class citizens and emphasizes data transformations.
- Mercury: Pure logic language built on facts and rules. Focuses on reasoning and inference through unification and backtracking.

Execution and Control Flow

- F#: Traditional execution order with explicit constructs like loops and conditionals. Recursion allows for iterative processes.
- Mercury: Non-deterministic due to backtracking, exploring alternative solutions and potentially unpredictable execution paths. Implicit control flow driven by rule matching.

Error Handling

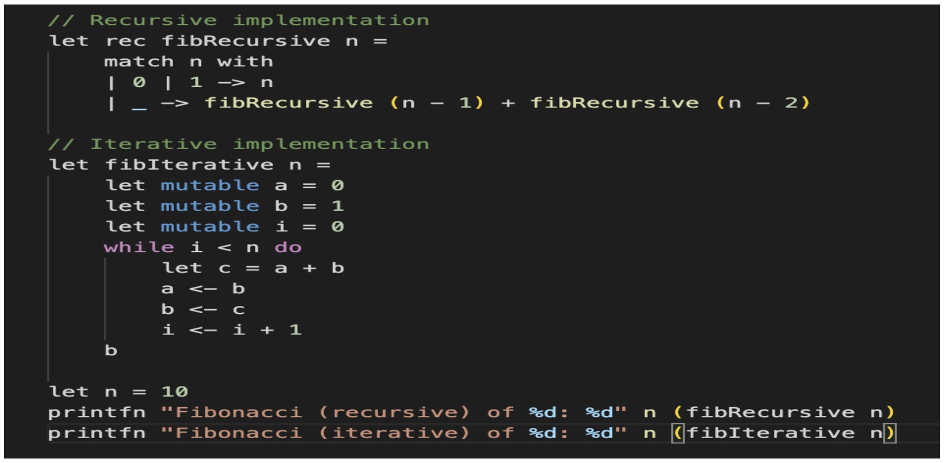
- F#: Requires explicit handling of exceptions and potential error states.
- Mercury: Non-deterministic execution and negation as failure can lead to complex debugging scenarios.

Concurrency

- F#:Built-in features for parallel and asynchronous programming leverage multi-core architectures.
- Mercury: Limited built-in support, requiring specific constructs and libraries for parallel processing.

Program

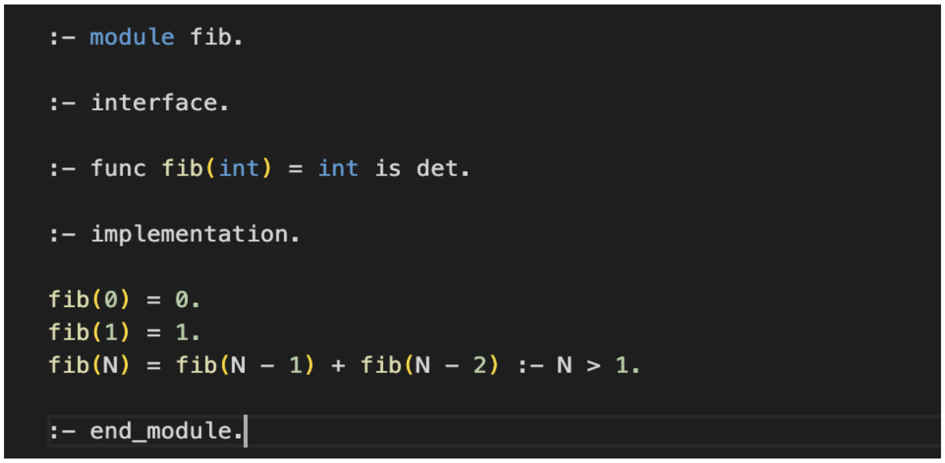
Functional Paradigm - F#



```
// Recursive implementation
let rec fibRecursive n =
    match n with
    | 0 | 1 -> n
    | _ -> fibRecursive (n - 1) + fibRecursive (n - 2)

// Iterative implementation
let fibIterative n =
    let mutable a = 0
    let mutable b = 1
    let mutable i = 0
    while i < n do
        let c = a + b
        a <- b
        b <- c
        i <- i + 1
    b

let n = 10
printfn "Fibonacci (recursive) of %d: %d" n (fibRecursive n)
printfn "Fibonacci (iterative) of %d: %d" n (fibIterative n)
```

Figure 12: Fibonacci in F#

Logic Paradigm - Mercury



```
:- module fib.

:- interface.

:- func fib(int) = int is det.

:- implementation.

fib(0) = 0.
fib(1) = 1.
fib(N) = fib(N - 1) + fib(N - 2) :- N > 1.

:- end_module.
```

Figure 13: Fibonacci in Mercury

Difference between Functional Paradigm and Logic Paradigm

Functional Paradigm

- It is totally based on functions.
- In this programming paradigm, programs are constructed by applying and composing functions.
- These are specially designed to manage and handle symbolic computation and list processing applications.
- Its main aim is to reduce side effects that are accomplished by isolating them from the rest of the software code.
- Some languages used in functional programming include Clojure, Wolfram Language, Erland, OCaml, etc.
- It reduces code redundancy, improves modularity, solves complex problems, increases maintainability, etc.
- It usually supports the functional programming paradigm.
- Testing is much easier as compared to logical programming.
- It simply uses functions.

Logic Paradigm

- It is totally based on formal logic.
- In this programming paradigm, program statements usually express or represent facts and rules related to problems within a system of formal logic.
- These are specially designed for fault diagnosis, natural language processing, planning, and machine learning.

- Its main aim is to allow machines to reason because it is very useful for representing knowledge.
- Some languages used for logic programming include Absys, Cycl, Alice, ALF (Algebraic logic functional programming language), etc.
- It is data-driven, array-oriented, used to express knowledge, etc.
- It usually supports the logic programming paradigm.
- Testing is comparatively more difficult as compared to functional programming.
- It simply uses predicates. Here, the predicate is not a function i.e., it does not have a return value.

Challenges Faced

Challenges faced during the exploration of programming paradigms

- Some Concepts related to the languages are difficult to understand.
- Resources for Mercury language is too less.
- Compiler for both the languages is too less.
- Paradigm and their associated languages does not contain similar features which is difficult to understand.

Conclusion

In conclusion, the exploration of the Functional and Logic programming paradigms, along with their associated languages F# and Mercury, reveals distinct characteristics and applications that cater to different problem-solving approaches.

F# exemplifies the principles of functional programming, emphasizing immutability, pure functions, and expressive syntax. Its focus on concise and declarative coding makes it suitable for a wide range of applications, from general-purpose programming to data science. F# leverages the .NET ecosystem, providing strong interoperability with other languages and frameworks. Its adoption of type inference, pattern matching, and asynchronous programming contributes to writing robust and maintainable code.

Mercury, as a logic programming language, diverges from the operational nature of traditional languages. It excels in expressing logical relationships, rule-based systems, and constraint logic programming. With features like logic variables, unification, and backtracking, Mercury finds applications in areas such as expert systems, symbolic computation, and formal verification. Its deterministic execution, termination analysis, and unique modes contribute to creating reliable and logically sound applications.

While both paradigms share features like immutability and strong typing, they diverge in their core principles. F# prioritizes functional programming concepts, making it versatile for a broad spectrum of applications, including web development and data science. On the other hand, Mercury's logic paradigm shines in scenarios requiring symbolic reasoning, rule-based systems, and automated decision-making.

The choice between the functional and logic paradigms, represented by F# and Mercury, depends on the nature of the problem at hand. Functional programming suits scenarios where concise and expressive code is paramount, while logic programming excels in rule-based systems and complex reasoning tasks.

In the dynamic landscape of programming languages, the choice of paradigm plays a crucial role in shaping the development process. F# and Mercury, each rooted in a distinctive paradigm, offer developers powerful tools for addressing diverse challenges. Whether it is the elegant, functional nature of F# or the logical reasoning capabilities of Mercury, the selection should align with the specific requirements and nature of the problem domain. As technology evolves, the exploration and integration of diverse paradigms contribute to the richness and adaptability of the programming landscape.

References

1. https://www.geeksforgeeks.org/functional-programming-paradigm/
2. https://www.freecodecamp.org/news/an-introduction-to-programming-paradigms/
3. https://hackr.io/blog/functional-programming
4. https://www.tutorialspoint.com/fsharp/fsharp$_{o}verview.htmhttps : //www.javatpoint.com/what - is - f - sharp$
5. earn.microsoft.com/en-us/dotnet/fsharp/what-is-fsharp
6. https://www.linkedin.com/advice/0/what-main-characteristics-examples-logic-programming
7. https://www.allassignmenthelp.com/blog/logic-programming-what-are-its-techniques/
8. https://mercurylang.org/about.html