Amrita Vishwa Vidyapeetham

TIFAC-CORE in Cyber Security

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

Kavali Sai Suvarchala

21st January, 2024

Paradigm 1: Functional

Functional programming is a programming paradigm in which we try to bind everything in pure mathematical functions style. It is a declarative type of programming style.

It uses expressions instead of statements. An expression is evaluated to produce a value whereas a statement is executed to assign variables

- **Immutability**

  A mutation is a modification of the value or structure of an object. Immutability means that something cannot be modified.

- **Deterministic Function**

  A function is deterministic if, given the same input, it returns the same output. The function produces the same output for the same set of inputs every time it is called.

- **Pure functions**

  A pure function is a function that is deterministic and has no side effects. A side effect is a modification of state outside the local environment of a function.

- **Functions are First-Class and can be Higher-Order**

  First-class functions are treated as first-class variable. The first class variables can be passed to functions as parameter, can be returned from functions or stored in data structures. Higher order functions are the functions that take other functions as arguments and they can also return functions.

- **Referential transparency**

  In functional programs variables once defined do not change their value throughout the program. Functional programs do not have assignment statements. If we have to store some value, we define new variables instead. This eliminates any chances of side effects because any variable can be replaced with its actual value at any point of execution. State of any variable is constant at any instant.

Language for Paradigm 1: Ocaml

OCaml, short for Objective Caml, is a programming language that evolved from the Caml language family .Key Features of Ocaml Language:

- **Ocaml Programming Language**

  In 1996, the OCaml project was initiated as an evolution of Caml. The "Objective" in "Objective Caml" reflects its support for object-oriented programming, which was introduced in this version. OCaml aimed to combine functional and imperative programming paradigms while maintaining a strong type system and efficient performance.

  OCaml, short for Objective Caml, is a statically typed, functional, and imperative programming language with a strong emphasis on type safety and expressive language features. It was developed in the late 1990s as an evolution of the Caml (Categorical Abstract Machine Language) programming language. OCaml is known for its expressive type system, efficient performance, and a rich ecosystem of libraries and tools.

- **Data Types in Ocaml Language**

  int — Integers,float — Floating-Point Numbers,char — Characters,string — Character Strings,bool — Booleans,unit — The Unit Value,Tuples,list — Lists,array — Arrays,exn — Exceptions,format — printf Formats,option — Optional Values,Records(like structures in c),ref — References,fun in functions that takes a single parameter.

- **Functional Programming**

  OCaml is rooted in functional programming paradigms, supporting first-class functions, higher-order functions, and immutable data structures. It encourages pure and declarative programming styles.

  Example Code:

```
          (* Define a function that takes two integers and returns their sum *)
let add_numbers (x: int) (y: int) : int =
  x + y

(* Define a function that takes a function and an integer, and applies the function twice *)
let apply_twice (f: int -> int) (x: int) : int =
  f (f x)

(* Example usage *)
let result1 = add_numbers 3 4

(* Use the apply_twice function with add_numbers to calculate (3 + 4) + (3 + 4) *)
let result2 = apply_twice (add_numbers 3) 4

(* Print the results *)
let () =
  Printf.printf "Result 1: %d\n" result1;
  Printf.printf "Result 2: %d\n" result2
```

  Define a First-Class Function (add numbers): The add numbers function takes two integers as arguments and returns their sum. Functions in OCaml are first-class citizens, meaning they can be passed as arguments to other functions

  Define a Higher-Order Function (apply twice): The apply twice function takes a function f and an integer x, and applies the function twice to the integer. This demonstrates higher-order function capabilities.

Example Usage:The example calculates the sum of 3 and 4 using the add numbers function and prints the result. It then uses the apply twice function to apply the add numbers function twice to calculate (3 + 4) + (3 + 4) and prints the result.

The apply twice function showcases the higher-order function concept by taking a function as an argument.

- **Strong Typing**

  OCaml employs a strong and static type system. Type inference allows developers to write expressive code without explicit type annotations while ensuring type safety and early error detection during compilation.

  OCaml has a strong and static type system, which helps catch many errors at compile-time.

  Example Code:

```
let add x y = x + y;;
(* add : int -> int -> int *)
```

  This code defines a function add that takes two parameters (x and y) and returns their sum. The type annotation indicates that the function takes two integers as input and returns an integer.

- **Pattern Matching**

  OCaml's pattern matching feature allows developers to destructure and manipulate complex data structures with ease. It simplifies control flow and enhances code readability.

  Example Code:

```
   let rec factorial n =
match n with
| 0 -> 1
| _ -> n * factorial (n - 1);;
```

  This code defines a recursive function factorial that calculates the factorial of a given number n. The match expression is used for pattern matching.

- **Polymorphic Variants**

  Polymorphic variants in OCaml enable the creation of types with multiple constructors, providing flexibility and expressiveness in data modeling and manipulation.

  Example Code:

```
    (* Define a polymorphic variant type representing different animals and their sounds *)
type animal =
  [ `Dog
  | `Cat
  | `Bird of string   (* bird species *)
  | `Elephant of int   (* trunk length in inches *)
  ]

(* Function to make an animal sound *)
let make_sound (a: animal) : string =
  match a with
  | `Dog -> "Woof!"
  | `Cat -> "Meow!"
  | `Bird species -> "Chirp chirp! I'm a " ^ species
```

```
  | `Elephant trunk_length -> "Trumpet! I have a trunk of length " ^ string_of_int trunk_length

(* Example usage *)
let () =
  let dog = `Dog in
  let cat = `Cat in
  let parrot = `Bird "Parrot" in
  let elephant = `Elephant 60 in

  Printf.printf "Dog says: %s\n" (make_sound dog);
  Printf.printf "Cat says: %s\n" (make_sound cat);
  Printf.printf "Parrot says: %s\n" (make_sound parrot);
  Printf.printf "Elephant says: %s\n" (make_sound elephant)
```

The type animal is defined as a polymorphic variant, which can represent different animals: Dog, Cat, Bird (with a string parameter for the species), and Elephant (with an integer parameter for trunk length).

The "make sound" function takes an animal as an argument and uses pattern matching to return a string representing the sound that particular animal makes.

The example creates instances of different animals (dog, cat, parrot, elephant) and uses the $make_sound function to print the sou

- **Static Memory Management**

  OCaml features automatic memory management through garbage collection, relieving developers from manual memory allocation and deallocation tasks. This reduces the risk of memory-related errors.

- **Concurrency and Parallelism**

  OCaml provides support for concurrency through lightweight threads and asynchronous I/O. Developers can write concurrent and parallel code to leverage modern multicore architectures.

- **Open Source**

  OCaml is open source, and its development benefits from contributions and collaboration from a worldwide community of users and developers.

- **Type Inference**

  OCaml's powerful type inference system can deduce types in many situations, reducing the need for explicit type annotations and promoting concise code.

Paradigm 2: Logic

Logic programming is a programming, database and knowledge-representation and reasoning paradigm which is based on formal logic.

A program, database or knowledge base in a logic programming language is a set of sentences in logical form, expressing facts and rules about some problem domain.

Major logic programming language families include Prolog, Answer Set Programming and Datalog. In all of these languages, rules are written in the form of clauses.

- **Data Types in Oz Language**

  Integers represent whole numbers.

  Atoms are basic constants denoted by names. They represent unique and indivisible entities.

  Floating-point numbers represent real numbers with decimal points.

  Records are compound data types that group multiple values together under named fields.

  Lists are ordered collections of elements.

  Tuples are ordered collections of elements, similar to lists, but are immutable.

  Strings,Boolean.

  Oz supports object-oriented programming with classes and objects.

  Procedures represent imperative code blocks and are used for sequencing and side effects.

  Example: Browse 30 (a procedure that displays the value 30)

  Functions are first-class citizens in Oz and can be assigned to variables, passed as arguments, and returned as results.

  Dictionaries are key-value pairs that provide an associative array data structure.

  Records with Named Arguments are similar to records, but with named arguments in function calls.

  Example: Personname:"Jay" age:22

Key Concepts of Logic Programming:

- **Logic and Rules**

  Programs in logic programming are expressed using formal logic, typically in the form of first-order logic. The core elements include facts and rules. Facts state what is true, and rules specify how to derive new truths from existing ones.

- **Declarative Style**

  Logic programming is declarative, meaning that programs are written as a set of logical statements that describe relationships and constraints rather than a sequence of steps to execute. Programmers specify what needs to be achieved rather than how to achieve it.

- **Predicates and Relationships**

  Predicates represent relationships between objects or conditions. They are used to express facts and rules in logic programming. For example, in Prolog, predicates are defined as clauses with a head and a body.

- **Horn Clauses**

  Logic programming often relies on Horn clauses, a type of logical formula where the clause is a disjunction of literals, all but one of which are positive. This structure simplifies the process of resolution and inference.

- **Backtracking**

  Backtracking is a fundamental mechanism in logic programming. If a branch of the logical tree fails to find a solution, the system backtracks to the most recent choice point and explores alternative branches. This allows for an exploration of multiple possibilities.

- **Goal-Driven Execution**

  In logic programming, execution is goal-driven. The program execution involves proving goals by searching for facts and rules that satisfy the given goals. This is in contrast to imperative programming, where the control flow is explicitly specified.

- **Recursive Rules**

  Logic programming languages like Prolog support recursive rules. This allows for the expression of repetitive patterns or recursive relationships in a concise and elegant manner.

- **Logical Variables**

  Variables in logic programming are logical variables rather than storage locations as in imperative programming. Logical variables are used in rules and queries, and their values are determined through unification..

- **Non-Determinism**

  Logic programming allows for non-deterministic execution. When multiple solutions exist, the system can explore different possibilities, and programmers can express this non-determinism explicitly..

- **Unification**

  Unification is a key concept in logic programming. It is the process of finding substitutions for variables in logical expressions such that the expressions become equal. Unification is used in pattern matching and resolving queries.

Language for Paradigm 2: Oz

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

- **Constraint Logic Programming**

  One of the distinctive features of Oz is its strong support for constraint logic programming. In constraint logic programming, you express relationships between variables using constraints, and the system automatically searches for values that satisfy those constraints.

  Example Code:

```
   declare
X Y Z in

% Constraint: X + Y = Z
{X + Y == Z}

% Constraint: Z is greater than 5
{Z > 5}

% Search for values of X, Y, and Z that satisfy the constraints
{Browse {Search X Y Z}}
```

In te above code :

The declare statement is used to declare logical variables X, Y, and Z. Logical variables in Oz are subject to unification.

Constraints are specified using the == operator. In this example, the constraint X + Y = Z is defined, expressing a relationship between the variables.

Another constraint Z > 5 is added, stating that Z must be greater than 5.

The Search operation is used to explore possible values for X, Y, and Z that satisfy the defined constraints.

The Browse statement is used to print the results of the search, displaying values for X, Y, and Z that satisfy the constraints.

The solution for the above code will be X + Y = Z (0 + 6 = 6) and Z > 5 (6 > 5).

- **Variables and Unification**

  Variables in Oz are logical variables, and they are subject to unification. Unification is a fundamental operation in logic programming, allowing the values of variables to be determined such that two terms become identical.

  Example Code:

```
        declare
X Y Z in

% Unify X and Y
X = Y

% Unify Y and Z
Y = Z

% Display the values of X, Y, and Z after unification
{Browse 'X=' X 'Y=' Y 'Z=' Z}
```

  In the above Code:

  The declare statement is used to declare logical variables X, Y, and Z.

  The variables are unified using the = operator. In the example, X is unified with Y, and Y is unified with Z.

  The Browse statement is used to print the values of X, Y, and Z after unification.

  The output of the will be :

  'X=' _0 'Y=' _0 'Z=' _0

  In this output, _0 is a logical variable that has been unified with X, Y, and Z, making them all identical. The concept of unification allows logical variables to take on values that make different terms equal, which is a key aspect of logic programming in Oz. The use of logical variables and unification is central to expressing relationships and constraints in a declarative manner.

- **Constraints and Solvers**

  Oz provides a rich set of built-in constraints and constraint solvers. Constraints can represent arithmetic relationships, inequalities, and other logical conditions. The system's constraint solver works to find solutions that satisfy these constraints.

  Example Code:

```
        declare
X Y Z in

% Constraint: X + Y = Z
{FD.int X 0 10}  % X is an integer between 0 and 10
{FD.int Y 0 10}  % Y is an integer between 0 and 10
{FD.int Z 0 20}  % Z is an integer between 0 and 20

{FD.plus X Y Z}  % Constraint: X + Y = Z

% Search for values of X, Y, and Z that satisfy the constraints
{FD.distribute [X Y Z]}

% Display the values of X, Y, and Z after constraint propagation
{Browse 'X=' X 'Y=' Y 'Z=' Z}
```

  In the above code:

  The declare statement is used to declare logical variables X, Y, and Z

  FD.int is used to define that X, Y, and Z are integer variables within specified ranges.

  FD.plus X Y Z sets up the constraint that X + Y = Z.

  FD.distribute [X Y Z] triggers constraint propagation, allowing the system's constraint solver to search for values that satisfy the specified constraints.

  The Finite Domain library allows to define constraints on these variables and then use a constraint solver to find values that satisfy those constraints.

  he Browse statement is used to print the values of X, Y, and Z after constraint propagation.

  The Ouput will be :

  'X=' 5 'Y=' 10 'Z=' 15

  The constraint solver has found values for X, Y, and Z that satisfy the constraint X + Y = Z and the specified integer ranges.

- **Goal-Directed Execution**

  Oz follows a goal-directed execution model. Programs are written in terms of goals, and the execution mechanism works towards achieving these goals by applying constraints and searching for solutions.

```
        declare
X Y Z in

% Constraint: X + Y = Z
{FD.int X 0 10}  % X is an integer between 0 and 10
{FD.int Y 0 10}  % Y is an integer between 0 and 10
{FD.int Z 0 20}  % Z is an integer between 0 and 20
```

```
{FD.plus X Y Z}  % Constraint: X + Y = Z

% Goal: Search for values of X, Y, and Z that satisfy the constraints
{Search X Y Z}

% Display the values of X, Y, and Z after goal-directed execution
{Browse 'X=' X 'Y=' Y 'Z=' Z}
```

Inthe above Code:

Declare logical variables X, Y, and Z.Set up constraints to define the characteristics of these variables.

The Search X Y Z statement initiates goal-directed execution. The system's constraint solver works towards finding values for X, Y, and Z that satisfy the specified constraints.

The Browse statement is used to print the values of X, Y, and Z after goal-directed execution.

The output will be:

'X=' 5 'Y=' 10 'Z=' 15

After finidng the Values for X, Y, and Z that satisfy the goal of the search, taking into account the specified constraints. Goal-directed execution, combined with constraint solving, allows for a natural and expressive way of expressing problems in Oz and finding solutions.

- **Dataflow Variables**

  Oz supports dataflow variables, which allow for synchronization and communication between concurrent threads. Dataflow variables are closely related to logic variables and contribute to the concurrent and reactive programming features of Oz.

- **Higher-Order Programming**

  Oz supports higher-order programming, allowing functions to be passed as arguments and returned as results. This feature enhances the expressiveness of the language and facilitates the development of modular and reusable code.

- **Pattern Matching**

  Oz includes pattern matching as a language construct. Pattern matching is a powerful feature that allows concise and expressive handling of complex data structures.

```
    declare
% Define a record type representing shapes
Shape = {record shape Type Size},

% Example shapes
Circle = {shape 'circle' 5},
Rectangle = {shape 'rectangle' 8 6},
Triangle = {shape 'triangle' 4 3 5},

% Function to process different shapes using pattern matching
proc {ProcessShape S}
  case S
  of {shape 'circle' R} then
    {Browse 'Processing a circle with radius:' R}
  [] {shape 'rectangle' W H} then
    {Browse 'Processing a rectangle with width:' W 'and height:' H}
  [] {shape 'triangle' A B C} then
```

```
      {Browse 'Processing a triangle with sides:' A B C}
    else
      {Browse 'Unknown shape'}
    end
  end

in
  % Process the example shapes
  {ProcessShape Circle}
  {ProcessShape Rectangle}
  {ProcessShape Triangle}
```

In the above code:

Shape is a record type with fields Type and Size. This represents different shapes, and the Type field specifies the type of shape, while Size may represent dimensions.

Circle, Rectangle, and Triangle are instances of the Shape record type, representing a circle, rectangle, and triangle, respectively.

The ProcessShape function takes a shape (S) as an argument and uses pattern matching to determine the type of shape and process it accordingly. The case statement checks the structure of the shape using patterns.

Patterns like shape 'circle' R, shape 'rectangle' W H, and shape 'triangle' A B C are used to match the structure of different shapes.

The matched values (radius, width, height, sides) are then used in the corresponding branches.

The example shapes (Circle, Rectangle, Triangle) are passed to the ProcessShape function to demonstrate the pattern matching.

The Browse statements within the function show the processed information for each shape.

The Output will be:

Processing a circle with radius: 5

Processing a rectangle with width: 8 and height: 6

Processing a triangle with sides: 4 3 5

This output demonstrates how pattern matching in Oz allows concise and expressive handling of different shapes by extracting and processing relevant information based on their structures.

Analysis

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

- **Analysis of Functional Paradigm - OCaml Language**

  OCaml supports a concise and expressive syntax, making it easy to write clear and readable code.

  OCaml features strong type inference, reducing the need for explicit type annotations while maintaining type safety.

  Immutability is encouraged in OCaml, promoting safer and more predictable code.

  Powerful pattern matching capabilities allow developers to handle complex data structures elegantly.

  Static typing helps catch errors at compile-time, improving code robustness and maintainability.

  Caml has a strong ecosystem, it may not be as extensive as some mainstream languages, which can limit libraries and tools available.

  The functional paradigm, concepts like currying and pattern matching will be bit difficult for the beginners to learn.

  Notable Features:

  OCaml's module system allows for the creation of abstract data types and supports powerful features like functors.

  Caml promotes concise code through features like type inference, algebraic data types, and pattern matching.

  Real World Projects Associated with Functional paradigm of Ocaml Language are:

  OCaml has been used in various real-world projects, especially in industries like finance, formal verification, and system programming.

  MirageOS is a library operating system that is written in OCaml. It is designed for building secure and efficient unikernels for cloud services. MirageOS leverages OCaml's functional programming features for safety and expressiveness.

  ReasonML, a syntax extension and toolchain for OCaml, is used in React programming. It brings OCaml's functional features to the world of front-end web development.

- **Analysis of Logic Paradigm - Oz Language**

  Oz's support for constraint logic programming enables elegant modeling and solving of problems with constraints.

  Dataflow variables facilitate communication between concurrent threads, contributing to reactive and concurrent programming features.

  Oz is a multiparadigm language, allowing developers to seamlessly integrate logic programming, functional programming, and concurrent programming.

  Similar to OCaml, Oz includes pattern matching as a language construct, providing expressive handling of complex data structures.

  Oz is designed for distributed computing, providing abstractions for distributed programming.

  Oz is not as widely adopted in industry compared to mainstream languages, which may limit its usage in certain contexts.

  The multiparadigm nature of Oz, along with its unique features, might learning challenge for developers unfamiliar with logic programming concepts.

  Notable Features:

  Oz's Finite Domain (FD) library provides powerful facilities for working with finite domain variables and expressing constraints.

Oz's interactive programming environment supports exploratory programming and experimentation.

Real World Projects Associated with logic paradigm of Oz Language are:

Oz is often used in academic and research settings, and its applications may be more prevalent in educational or experimental projects rather than large-scale commercial projects.

Conculusion:

Both OCaml and Oz offer unique strengths and features associated with their respective paradigms. OCaml excels in functional programming with strong type inference and pattern matching, while Oz provides a multiparadigm approach with a focus on logic programming, dataflow variables, and distributed computing.

Comparison

Similarities:

Both OCaml and Oz support pattern matching, allowing developers to concisely express and handle complex data structures.

Both paradigms encourage immutability, providing a more predictable and safer approach to programming.

OCaml and Oz feature expressive type systems, although they approach type systems from different perspectives. OCaml has a strong and statically inferred type system, while Oz provides flexibility through its multiparadigm approach.

Both languages offer features for concurrent programming. OCaml supports lightweight threads and asynchronous programming, while Oz has built-in support for concurrent programming through dataflow variables.

Differences:

OCaml is primarily a functional programming language with a strong focus on immutability, first-class functions, and algebraic data types. In contrast, Oz is a multiparadigm language with a unique emphasis on logic programming, constraint logic programming, and concurrent programming.

OCaml has a statically typed system with strong type inference, while Oz has a more flexible approach with a dynamically typed system. OCaml's type system is more rigid and helps catch errors at compile-time, whereas Oz's dynamic typing allows for more flexibility at runtime.

While both languages support concurrency, their models differ. OCaml's concurrency model involves lightweight threads and asynchronous programming. In contrast, Oz employs dataflow variables for synchronization between concurrent threads.

Oz excels in constraint logic programming with a rich set of built-in constraints and a dedicated Finite Domain (FD) library. OCaml, being primarily a functional language, lacks the specialized support for constraint logic programming that Oz provides.

OCaml has a more mature ecosystem with a strong emphasis on functional programming, and it is used in both academia and industry. Oz, on the other hand, is less widely adopted and is known for its unique features in academic and research settings.

The difficulty in learning for OCaml may be less for developers familiar with functional programming languages, while Oz's multiparadigm approach, especially with logic programming, may present a more significant learning challenge.

Challenges Faced

Learning different ways of programming was tough for me because my knowledge in programming was limited. Understanding new ideas and switching between different ways of writing code felt challenging. It was hard to grasp the rules and styles of each method. Figuring out how to use multiple ways of programming in one project and finding good learning materials were also tricky, but I managed by taking small steps and using helpful resources. I sometimes worried about learning and understanding the concepts and made mistakes. Learning all of this took time, and I needed to practice a lot. It is not only about reading and understanding but i also I had to try things out and improve my skills.It was not easy for me to understand and learn all the things mentioned here but i took help from the resources available.

Conclusion

This took us through two different ways of programming: Functional Programming with OCaml and Logic Programming with Oz. We discovered that OCaml is a go-to language for real-world projects, especially in finance and system programming, because it's great for functional programming. While Oz, which leans more towards logic, might not be as popular in big projects, it's still known for its unique logic and concurrent programming features. This assignment helped us to know how these programming styles are used in real projects.

References

1. Video Tutorial: https://youtu.be/spwvg0DThh4?si=2VMSC61mkOCY$_lfm$
2. Video Tutorial :https://youtu.be/dAPL7MQGjyM?si=LjNuvqjcLksPyky2
3. Video Tutorial : https://youtu.be/U4U7KOBV7OI?si=qbVIMr0oyXNKWkqb

1. GeeksforGeeks - Functional Programming Paradigm: https://www.geeksforgeeks.org/functional-programming-paradigm/
2. OCaml Control Structures: https://www2.lib.uchicago.edu/keith/ocaml-class/control.html: :text=While
3. Hacker Noon - 9 Functional Programming Concepts: https://hackernoon.com/9-functional-programming-concepts-everyone-should-know-uy503u21
4. Introduction to OCaml Programming Language: https://piembsystech.com/introduction-to-ocaml-programming-language/
5. AllAssignmentHelp - Logic Programming Techniques: https://www.allassignmenthelp.com/blog/logic-programming-what-are-its-techniques/
6. GeeksforGeeks - Difference between Functional and Logical Programming: https://www.geeksforgeeks.org/difference-between-functional-and-logical-programming/
7. Wikipedia - Oz Programming Language: https://en.wikipedia.org/wiki/Oz$_{(programming_language)}ACMDigitalLibra$ $LogicProgramminginOz : https : //dl.acm.org/doi/10.1145/3386333$
8. Wikipedia - Comparison of Programming Paradigms: https://en.wikipedia.org/wiki/Comparison$_of_programming_paradi$ $DifferencebetweenLogicandFunctionalProgramming : https : //stackoverflow.com/questions/8297574/differenc$
9. OCaml Documentation - Basic Data Types: https://ocaml.org/docs/basic-data-types
10. OCaml Core Examples: https://v2.ocaml.org/manual/coreexamples.html
11. OCaml Official Website: https://ocaml.org/
12. Wikipedia - OCaml: https://en.wikipedia.org/wiki/OCaml

Chatgpt Prompts:

what is meant by pure functions in functional programming

key concepts of functional programming paradigm

characteristics and features of the ocaml language associated with Functional Programming Paradigm

Characteristics and features of the ocaml language associated with Functional Programming Paradigm

data types in oz language

Functional Programming Paradigm in Ocaml

Logic Programming Paradigm in Oz

ocaml language and how functional programming paradigm is implemented in this language

Oz language and how Logic programming paradigm is implemented in this language

characteristics and features of the oz language associated with Logic Paradigm

Polymorphic variants in OCaml

principles and concepts of the programing paradigm logic

what is FD

Real world applications /projects implemented using logic paradigm using oz language

Real world applicationsprojects implemented using Functional Paradigm using Ocaml Language

Codes explaining logic paradigm feactures of Oz Language

Codes explaining Functional paradigm featues of Ocaml Language

Explanations of some codes i pasted in the report