F# - PATTERN MATCHING

Pattern matching allows you to “compare data with a logical structure or structures, decompose data into constituent parts, or extract information from data in various ways”.

In other terms, it provides a more flexible and powerful way of testing data against a series of conditions and performing some computations based on the condition met.

Conceptually, it is like a series of if… then statements.

Syntax

In high level terms, pattern matching follows this syntax in F# −

```
match expr with
| pat1 - result1
| pat2 -> result2
| pat3 when expr2 -> result3
| _ -> defaultResult
```

Where,

- Each | symbol defines a condition.
- The -> symbol means "if the condition is true, return this value...".
- The _ symbol provides the default pattern, meaning that it matches all other things like a wildcard.

Example 1

The following example, calculates the Fibonacci numbers using pattern matching syntax −

```
let rec fib n =
   match n with
   | 0 -> 0
   | 1 -> 1
   | _ -> fib (n - 1) + fib (n - 2)
for i = 1 to 10 do
   printfn "Fibonacci %d: %d" i (fib i)
```

When you compile and execute the program, it yields the following output −

```
Fibonacci 1: 1
Fibonacci 2: 1
Fibonacci 3: 2
Fibonacci 4: 3
Fibonacci 5: 5
Fibonacci 6: 8
Fibonacci 7: 13
Fibonacci 8: 21
Fibonacci 9: 34
Fibonacci 10: 55
```

You can also chain together multiple conditions, which return the same value. For example −

Example 2

```
let printSeason month =
   match month with
   | "December" | "January" | "February" -> printfn "Winter"
   | "March" | "April" -> printfn "Spring"
   | "May" | "June" -> printfn "Summer"
```

```
    | "July" | "August" -> printfn "Rainy"
    | "September" | "October" | "November" -> printfn "Autumn"
    | _ -> printfn "Season depends on month!"

printSeason "February"
printSeason "April"
printSeason "November"
printSeason "July"
```

When you compile and execute the program, it yields the following output −

```
Winter
Spring
Autumn
Rainy
```

Pattern Matching Functions

F# allows you to write pattern matching functions using the **function** keyword −

```
let getRate = function
    | "potato" -> 10.00
    | "brinjal" -> 20.50
    | "cauliflower" -> 21.00
    | "cabbage" -> 8.75
    | "carrot" -> 15.00
    | _ -> nan (* nan is a special value meaning "not a number" *)

printfn "%g"(getRate "potato")
printfn "%g"(getRate "brinjal")
printfn "%g"(getRate "cauliflower")
printfn "%g"(getRate "cabbage")
printfn "%g"(getRate "carrot")
```

When you compile and execute the program, it yields the following output −

```
10
20.5
21
8.75
15
```

Adding Filters or Guards to Patterns

You can add filters, or guards, to patterns using the **when** keyword.

Example 1

```
let sign = function
    | 0 -> 0
    | x when x < 0 -> -1
    | x when x > 0 -> 1

printfn "%d" (sign -20)
printfn "%d" (sign 20)
printfn "%d" (sign 0)
```

When you compile and execute the program, it yields the following output −

```
-1
1
0
```

Example 2

```
let compareInt x =
   match x with
   | (var1, var2) when var1 > var2 -> printfn "%d is greater than %d" var1 var2
   | (var1, var2) when var1 < var2 -> printfn "%d is less than %d" var1 var2
   | (var1, var2) -> printfn "%d equals %d" var1 var2

compareInt (11,25)
compareInt (72, 10)
compareInt (0, 0)
```

When you compile and execute the program, it yields the following output −

```
11 is less than 25
72 is greater than 10
0 equals 0
```

Pattern Matching with Tuples

The following example demonstrates the pattern matching with tuples −

```
let greeting (name, subject) =
   match (name, subject) with
   | ("Zara", _) -> "Hello, Zara"
   | (name, "English") -> "Hello, " + name + " from the department of English"
   | (name, _) when subject.StartsWith("Comp") -> "Hello, " + name + " from the
department of Computer Sc."
   | (_, "Accounts and Finance") -> "Welcome to the department of Accounts and Finance!"
   | _ -> "You are not registered into the system"

printfn "%s" (greeting ("Zara", "English"))
printfn "%s" (greeting ("Raman", "Computer Science"))
printfn "%s" (greeting ("Ravi", "Mathematics"))
```

When you compile and execute the program, it yields the following output −

```
Hello, Zara
Hello, Raman from the department of Computer Sc.
You are not registered into the system
```

Pattern Matching with Records

The following example demonstrates pattern matching with records −

```
type Point = { x: float; y: float }
let evaluatePoint (point: Point) =
   match point with
   | { x = 0.0; y = 0.0 } -> printfn "Point is at the origin."
   | { x = xVal; y = 0.0 } -> printfn "Point is on the x-axis. Value is %f." xVal
   | { x = 0.0; y = yVal } -> printfn "Point is on the y-axis. Value is %f." yVal
   | { x = xVal; y = yVal } -> printfn "Point is at (%f, %f)." xVal yVal

evaluatePoint { x = 0.0; y = 0.0 }
evaluatePoint { x = 10.0; y = 0.0 }
evaluatePoint { x = 0.0; y = 10.0 }
evaluatePoint { x = 10.0; y = 10.0 }
```

When you compile and execute the program, it yields the following output −

```
Point is at the origin.
Point is on the x-axis. Value is 10.000000.
Point is on the y-axis. Value is 10.000000.
Point is at (10.000000, 10.000000).
```