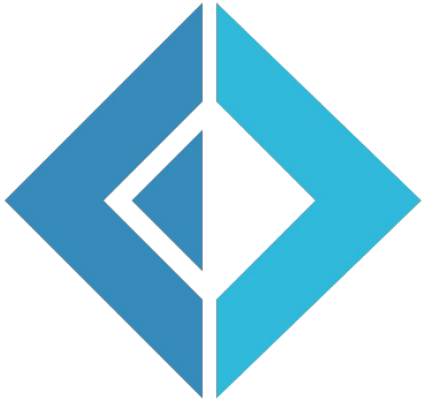


F# overview (fsharp)

Quem sou eu?

Leandro Fernandes Vieira - @leandromoh (github/linkedin)

Senior Software Engineer na Stone

TCC sobre paradigmas de programação 2015

Contribuidor do projeto Morelinq 2018

Autor do projeto RecordParser 2021

Revisor técnico de livro sobre PF 2023

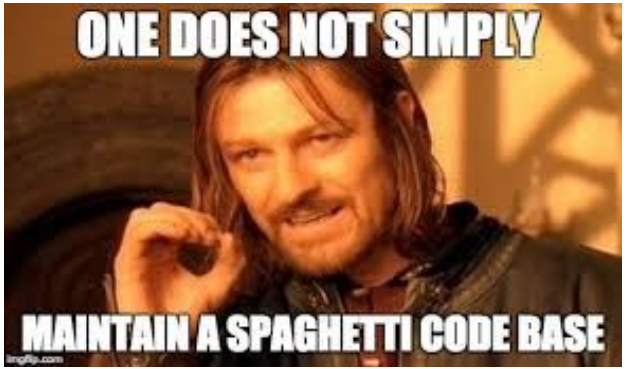
Escrever software de qualidade é difícil

Legibilidade
Manutenabilidade

Performance
Paralelismo

Reusabilidade
Adaptabilidade

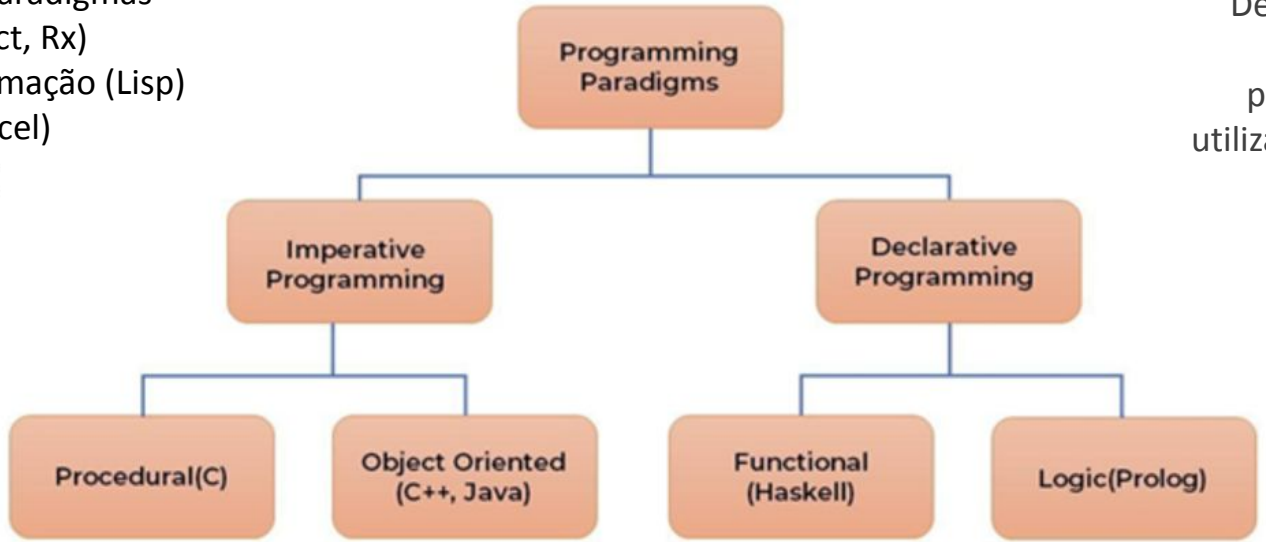
Confiabilidade
Tratamento de erros
Testabilidade



Baixo Acoplamento
Alta Coesão

Paradigmas de Programação

Entre outros paradigmas
- Reativa (React, Rx)
- Meta programação (Lisp)
- Dataflow (Excel)
- etc



Determina a estruturação e execução do programa, permitindo ou proibindo a utilização de algumas técnicas de programação

Programação funcional já está entre nós

C# LINQ e pattern matching

Javascript anonymous functions e callbacks

Java Streams API

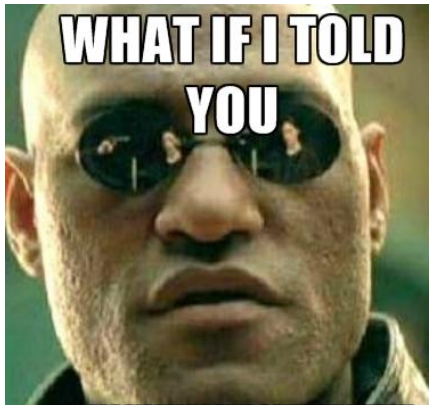
Rust Option Enum e Pattern Matching

Scala Pattern Matching

Elixir Pipe Operator

Clojure funções como cidadãos de primeira classe

etc



Linguagem F#

Surgiu no ecossistema .NET em 2005

Linguagem de propósito geral

Multiplataforma (Windows, Linux, Mac)

Open Source

Multi-paradigma porém Functional-First

Membro da família ML

Jupyter notebooks

Tipagem estática com inferência de tipos

Imutabilidade por default

Pattern Matching

Tipos Base

```fsharp
open System

let x = 42
let price = 1.99M
let username = "leandromoh"
let today = DateTime.Today
let even = true
let tuple = (1, "two", 3.0)
let list = [1; 2; 3; 4; 5]
let array = [|1; 2; 3; 4; 5|]
let unit = ()

printfn "Hello World"
printfn "Hello %s today is %A" username today.DayOfWeek
printfn $"Hello {username} today is {today.DayOfWeek}"
```

Funções

```
// int -> int -> int
let add x y = x + y          // add 2 3 => 5
let square x = x * x         // square 3 => 9
let three = add 1 2

let squareSum x y =          // squareSum 2 3 => 25
    let sum = add x y
    square sum

let checkAge age =
    if age >= 18 then "Adult" else "Young"

let getUserLevel (score: int) : string =
    if score > 90 then
        "Platinum"
    elif score > 70 then
        "Gold"
    elif score > 50 then
        "Silver"
    else
        "Bronze"
```

Funções como cidadãos de primeira classe

```
let bothEven a b =              // bothEven 2 4 => true
    let isEven x = x % 2 = 0    // local function
    isEven a && isEven b

let sum numbers =               // sum [1;2;3;4;5] => 15
    let mutable total = 0       // explict mutable variable
    for n in numbers do         // iterate over a sequence of numbers
        total <- total + n      // increment the variable
    total                       // return the accumulated value

let reduce fn init numbers =    // function that takes another function as parameter
    let mutable acc = init      // fn is the parametrized operation
    for n in numbers do
        acc <- fn acc n
    acc

let sum2 numbers = reduce (+) 0 numbers
let product2 numbers = reduce (*) 1 numbers
```

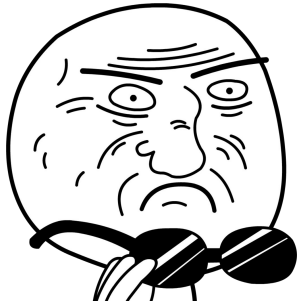
```fsharp
let sum3 : int seq -> int = reduce (+) 0
let product3 : int seq -> int = reduce (*) 1
```

int -> (int -> int)

```fsharp
let add1 x y = x + y

let add2 x =
    let local y = x + y
    local

let add3 = fun x y -> x + y

let add4 = fun x -> fun y -> x + y

let log total =
    printfn "total = %d" total

log (((add1) 2) 3)    // => 15
log (((add2) 2) 3)    // => 15
log (((add3) 2) 3)    // => 15
log (((add4) 2) 3)    // => 15

let sumFiveTo = add 5
let six = sumFiveTo 1  // => 6
```

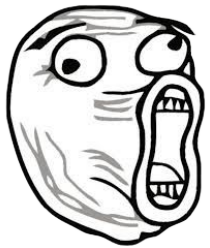
Currying quebra uma função de vários parâmetros em uma sequência de funções aninhadas que recebem um único parâmetro, permitindo a aplicação parcial de argumentos.

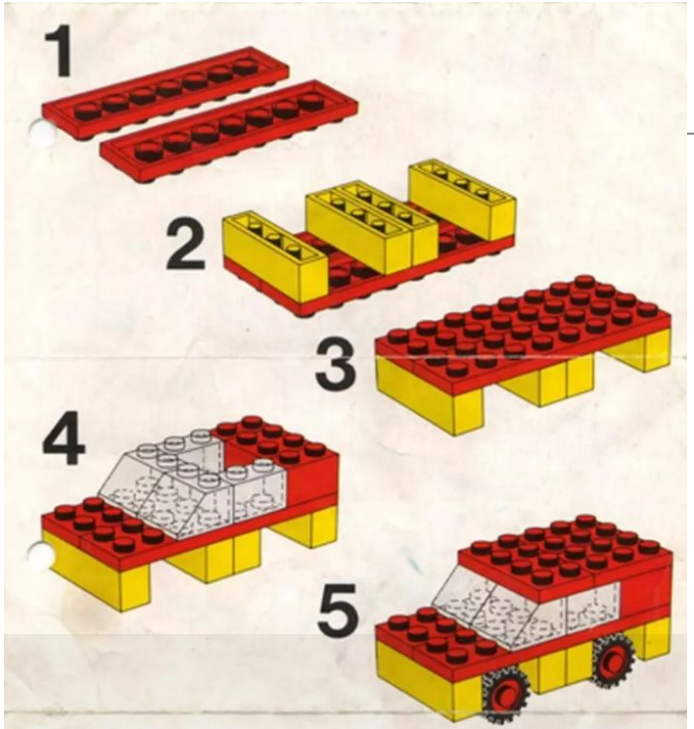
Em vez de fn(a, b, c), obtemos uma estrutura fn(a)(b)(c), que em F# é fn a b c



```fsharp
let numeros = [ 1; 2; 3; 4; 5; 6; 7; 8; 9; 10 ]

// tradicional
let numerosPares = List.filter (fun n -> n % 2 = 0) numeros
let numerosParesQuadrados = List.map (fun n -> n * n) numerosPares
let resultadoSemPipe = List.sum numerosParesQuadrados

// aninhado
let resultadoAninhado =
    List.sum
        (List.map (fun n -> n * n)
            (List.filter (fun n -> n % 2 = 0) numeros))

// com pipe operator
let resultado =
    numeros
    |> List.filter (fun n -> n % 2 = 0)
    |> List.map (fun n -> n * n)
    |> List.sum
```

operações são desenhadas para serem combináveis, com responsabilidade única e ao mesmo tempo adaptáveis atrás da callback



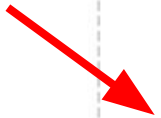


Funções puras como componentes

- responsabilidade única
- resultados determinísticos
- combináveis
- abertas para extensão
- adaptáveis graças a HOF
- manutenção mais simples
- fácil de testar
- fácil de paralelizar

```fsharp
type Customer = {
    Id: Guid
    FirstName: string
    LastName: string
    Email: string
    BirthDate: DateTime
}

let customer = {
    Id = Guid.NewGuid()
    FirstName = "Peter"
    LastName = "Parker"
    Email = "parker.old@example.com"
    BirthDate = DateTime(1990, 5, 20)
}

// Records são imutaveis, mas podemos criar uma nova instância com valores atualizados
let updatedCustomer = { customer with Email = "parker.new@example.com" }

printfn "Email atualizado: %s" updatedCustomer.Email
printfn "Email original inalterado: %s" customer.Email
```

records são imutáveis por padrão
e possuem igualdade por valor



```fsharp
type MutableCustomer = {
    Id: Guid
    FirstName: string
    LastName: string
    mutable Email: string
    BirthDate: DateTime
}

let cust = {
    Id = Guid.NewGuid()
    FirstName = "Peter"
    LastName = "Parker"
    Email = "parker.old@example.com"
    BirthDate = DateTime(1990, 5, 20)
}

cust.Email <- "parker.new@example.com"

printfn "Email alterado: %s" cust.Email
```

mutabilidade é permitida
porém evitada

```fsharp
type Contact =
    | Phone of int
    | Email of string
    | None

type Customer = {
    Id: Guid
    FirstName: string
    LastName: string
    Contact: Contact
    BirthDate: DateTime
}

let customer = {
    Id = Guid.NewGuid()
    FirstName = "Peter"
    LastName = "Parker"
    Contact = Email "parker.old@example.com"
    BirthDate = DateTime(1990, 5, 20)
}
```

Discriminated Union (DU) representam enumeradores; cada item pode armazenar valores

```fsharp
let printNumber x =
    match x with
    | 1 | 2 | 3 -> printfn "Found 1, 2, or 3!"
    | x when x < 0 -> printfn "Negative number: %d" x
    | x -> printfn "Other number: %d" x

let x : obj = DateTime.Now.ToString()
match x with
| :? Customer as cust ->  printfn "matched customer %s %s" cust.FirstName cust.LastName
| :? int as i when i = 23 -> printfn "matched 23"
| :? string as text -> printfn "matched text %s" text
| :? DateTime -> printfn "matched a datetime"
| _ -> printfn "another value"
```

Pattern matching ajuda testar valores, padrões e tipos

default, último case

```fsharp
type Option<'T> =
    | Some of 'T
    | None

let maybeNumber = Some 42 // ou None

let printOption opt =
    match opt with
    | Some value -> printfn "Value: %d" value
    | None -> printfn "No value"

let printContact contact =
    match contact with
    | Phone number -> printfn "Phone: %d" number
    | Email address -> printfn "Email: %s" address
    | Contact.None -> printfn "No contact info"
```

Tipo nativo que representa presença ou ausência de valor; previne null reference exceptions

Pattern Matching para identificar o valor

Pattern matching exaustivo para DU! menos bugs!

```fsharp
type Result<'T,'TError> =
  | Ok of ResultValue: 'T
  | Error of ErrorValue: 'TError

let readFileContent (filePath: string) : Result<string, Exception> =
    try
        use reader = new StreamReader(filePath)
        try
            let content = reader.ReadToEnd()
            Ok content
        with
        | :? OutOfMemoryException as ex ->
            printfn "Close torrent and try again!"
            Error ex
        | ex ->
            Error ex
    finally
        printfn "From the finally block."

let desktopPath = Environment.GetFolderPath(Environment.SpecialFolder.Desktop)
let fileName = "teste.txt"
let filePath = Path.Combine(desktopPath, fileName)
match readFileContent filePath with
| Ok content -> printfn "File content:\n%s" content
| Error ex -> printfn "Error reading file: %s" ex.Message
```

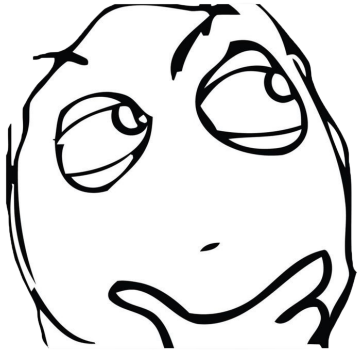
Tipo nativo que representa sucesso ou falha, armazena um valor para cada caso

raise ex para lançar exception
reraise() para relançar dentro do try..with

bloco finally sempre executa e precisa retornar valor do tipo unit

```fsharp
// Complete Active Pattern
let (|PositiveInt|NegativeInt|ZeroInt|) x =
    if x > 0 then PositiveInt
    elif x < 0 then NegativeInt
    else ZeroInt

let testNumber i =
    match i with
    | PositiveInt -> printfn "Positive"
    | NegativeInt -> printfn "Negative"
    | ZeroInt -> printfn "Zero"
```

Active Patterns permitem testar, decompor ou transformar valores e objetos como se fossem DU



```fsharp
// Partial Active Patterns
let (|Integer|_|) (str:string) =
    match System.Int32.TryParse(str) with
    | (true, i) -> Some i
    | _ -> None

let (|Float|_|) (str: string) =
    let mutable f = 0.0
    if System.Double.TryParse(str, &f) then
        Some(f)
    else
        None

let (|Parenthesized|_|) (s: string) =
    if s.StartsWith("(") && s.EndsWith(")") then
        Some (s.Substring(1, s.Length - 2))
    else
        None

let parseExpression (input: string) =
    match input with
    | Integer n -> printfn "Found a integer: %d" n
    | Float n -> printfn "Found a float: %f" n
    | Parenthesized (Integer n) -> printfn "Found a parenthesized integer: %d" n
    | _ -> printfn "Could not parse: %s" input
```

Active Patterns podem ser usados de forma nested, entre outros jeitos

```fsharp
// ('a -> bool) -> seq<'a> -> seq<'a>
let where predicate sequence =
    seq {
        for x in sequence do
            if predicate x then
                yield x
    }
```

```fsharp
// string -> CancellationToken -> Task<Option<string>>
let getMD5HashAsync filePath cancellationToken =
    task {
        if File.Exists(filePath) = false then
            return None
        else
            use stream = File.OpenRead(filePath)
            let! md5 = MD5.HashDataAsync(stream, cancellationToken)
            let hash = BitConverter
                        .ToString(md5)
                        .ToLowerInvariant()
                        .Replace("-", String.Empty)
            return Some hash

    }
```

Computation Expressions abstraem comportamentos atrelados a “fluxos de controle” de cada tipo de CE. Permitem estender a linguagem com novos “fluxos”

```fsharp
let getFilesMD5HashAsync searchPattern directoryPath : Task<list<string * string>> =
    task {
        if not (Directory.Exists directoryPath) then
            return []
        else
            let! (results:(string * Option<string>)[])  =
                    Directory.GetFiles(directoryPath, searchPattern)
                    |> Seq.map (fun file ->
                        task {
                            let! hash = getMD5HashAsync file CancellationToken.None
                            return (file, hash)
                        })
                    |> Task.WhenAll

            return results
                |> Seq.choose (function
                    | (file, Some hash) -> Some (file, hash)
                    | _ -> None)
                |> Seq.toList
    }
```

mapeia cada arquivo para uma operação assíncrona

processamento em paralelo do hash de cada arquivo

```fsharp
type Customer(name: string, height: decimal, age: int) =
    let mutable _age = age

    // optional empty constructor
    new() = Customer(String.Empty, 0, 0)

    // Propriedade com backing field, getter e setter manual
    member x.Age
        with get() = _age
        and set(value) = _age <- value

    // Propriedade automatica: backing field gerado pelo compilador
    member val Height = height with get, set

    // Propriedades somente leitura sem backing field (evaluadas toda vez)
    member x.Name = name
    member x.IsAdult = x.Age >= 18

    member x.Hello() = printfn $"Ola! Meu nome é {x.Name} e tenho {x.Age} anos"

    override x.ToString() =
        $"Customer(Name={x.Name}, Height={x.Height}, Age={x.Age}, IsAdult={x.IsAdult})"

let c = new Customer ("frank", 1.70m, 17)
c.Age <- 18
```

F# também suporta POO, geralmente usada em integrações com APIs do framework ou projetos feitos em C#

```fsharp
type IMultiply =
    abstract member Multiply : int -> int -> int    // F#-style

type ISum =
    abstract member Sum : a: int * b: int -> int    // .NET-style

type SomeClass(x: int, y: float) =

    interface IMultiply with
        member this.Multiply a b =
            let c = a * b
            printfn "%d" c
            c

    interface ISum with
        member this.Sum (a: int, b: int) =
            let c = a + b
            printfn "%d" c
            c

let x = new SomeClass(1, 2.0)
let mult = (x :> IMultiply).Multiply 2 3 // 6
let sum = (x :> ISum).Sum(2, 3)          // 5
```

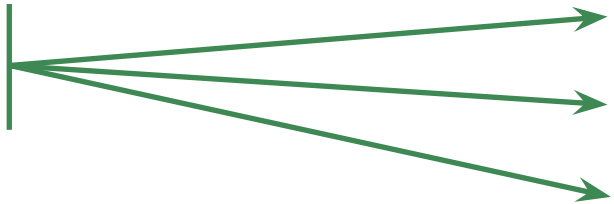
Definição e Implementação de interfaces

Outros features diferenciadas

- Unit of Measure
  - permite associar números a unidades; o compilador verifica se as operações aritméticas têm as mesmas unidades, o que ajuda a evitar bugs.
- Code Quotations:
  - permite representar e manipular código como uma estrutura de dados em forma de árvore (AST)
- Type Providers:
  - plugins que estendem o compilador para fora da linguagem
    - gerar tipos a partir de um csv ou retorno de um endpoint json.
    - verificar se a tabela do banco de dados existe em temp de compilação
    - validar Regular Expressions
- Single-Pass Compiler:
  - só é acessível ao código o que foi declarado antes, minimiza referência circular

Técnicas de programação funcional

- Funções como objeto de primeira classe (HOF)
- Assinatura como interface
- Currying

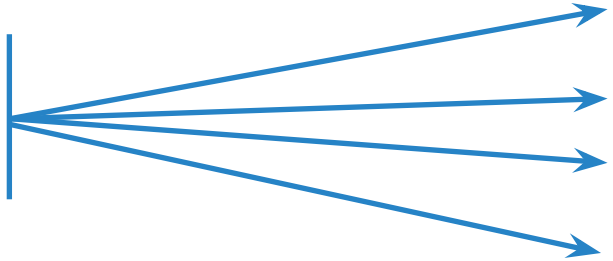


Reutilização

Adaptabilidade

Baixo Acoplamento e Alta Coesão

- Funções sem side effects
- Imutabilidade
- Igualdade por valor
- Discriminated Unions
- Pattern Matching e Active Patterns
- Pipeline Operator



Paralelizável

Testabilidade

Legibilidade

Manutenibilidade

FIM