Funções

PARADIGMAS DE PROGRAMAÇÃO

Entrada de dados

- No F# utilizamos para entrada de dados a função padrão do .NET System.Console.ReadLine(), ela sempre retorna uma string, portanto caso precisar de um valor numérico, o operador |> e um tipo de conversão deve ser passado por parâmetro. É bom lembrar que independente de não passarmos um tipo, a linguagem identifica.

```
let string = System.Console.ReadLine()
let inteiro = System.Console.ReadLine() |> System.Int32.Parse
let real = System.Console.ReadLine() |> System.Double.Parse
```

Funções aninhadas

- Como todas as funções também são tratadas como valores, podemos facilmente trabalhar com funções aninhadas, assim como trabalhamos até agora com desvios condicionais e laços de repetição aninhados.
- Vamos ver um exemplo, faremos uma função que recebe um número por parâmetro e imprime se ele é par ou ímpar. Faremos uma função de validação e internamente construiremos as funções

Funções aninhadas

- Temos funções base, a que realmente verifica o número e as que escrevem na tela:

```
let verificaPar numero =
    numero % 2 = 0
let escrevePar numero =
    printfn "O número %i é par" numero
let escreveImpar numero =
    printfn "O número %i é ímpar" numero
```

Funções aninhadas

- Essas funções não precisam estar visíveis para todo o código, elas podem estar contidas dentro de uma única função:

```
let escreveParImpar numero =

    if verificaPar numero then
        escrevePar numero
    else
        escreveImpar numero
```

Funções aninhadas

```
let escreveParImpar numero =
    let verificaPar numero =
        numero % 2 = 0
    let escrevePar numero =
        printfn "O número %i é par" numero
    let escreveImpar numero =
        printfn "O número %i é ímpar" numero

    if verificaPar numero then
        escrevePar numero
    else
        escreveImpar numero
```

Funções aninhadas

- Outro detalhe é o escopo, já que as funções estão aninhadas, não precisamos redeclarar parâmetros, eles já são acessíveis da função externa.

- Podemos tirar os parâmetros das funções, se tirarmos da primeira, tudo continua normal, se tirarmos das seguintes o programa para de funcionar. Com a falta de parâmetros ele passa a considerar as funções um valor simples, que é atribuído no momento da declaração, nesse caso devemos colocar o operador conhecido de ()

Funções aninhadas

```
let escreveParImpar numero =
    let verificaPar =
        numero % 2 = 0
    let escrevePar() =
        printfn "O número %i é par" numero
    let escreveImpar() =
        printfn "O número %i é ímpar" numero

    if verificaPar then
        escrevePar()
    else
        escreveImpar()
```

Funções de Alta Ordem

- Funções de Alta Ordem recebem como parâmetro ou retornam outras funções. O F# facilita a criação desse tipo de função e possui algumas já na própria linguagem. Vimos na aula passada o List.map que aplica um cálculo para cada número e o List.sum que transforma uma lista em um valor que é a soma de todos os número, outro exemplo é o List.filter.

```
let verificaPar numero =
        numero % 2 = 0

let numeroParAte10 =
   [0..10]
   |> List.filter verificaPar

[<EntryPoint>]
let main argv =
    for num in numeroParAte10 do
        printfn "%i" num
    0
```

Operadores como funções

- Podemos reduzir essa operação, utilizado a aplicação parcial do operador de +

```
let soma3 = (+) 3
```

Operadores como funções

- Como funciona isso?

```
let (+) numero1 numero2 = numero1 + numero2
```

Operadores como funções

- Podemos fazer até mais genérico que isso:

```
let soma = (+)

let soma3 = (+) 3
let soma4 = (+) 4
```

Mas vamos falar de outro operador

- Vamos falar de um operador que vimos algumas vezes, o pipe |>, a definição formal seria algo mais ou menos assim, ele basicamente faz com que o parâmetro anterior a ele seja passado para a função a frente:

```
let (|>) parametro funcao = funcao parametro
```

Operador Pipe

- Vamos fazer como na aula passada, funções aninhadas, dobrarValores, ela vai criar uma lista de 0 até 10 (let lista = [0..10]) e retorna o dobro dos valores menores que, dentro dela devem existir duas funções:
  - Uma que retorna se um número é menor do que cinco
  - Uma multiplica os valores por 2
- Depois vamos utilizar a função List.filter para filtrar os elementos da lista menores de que cinco e depois o List.map para aplicar a multiplicação por 2 pelo filtro definido

Operador Pipe

```
let dobraMenores5 =
    let multiplica2 numero = numero * 2
    let menorQue5 numero = numero < 5

    let lista = [0..10]
    let listaFiltrada = List.filter menorQue5 lista
    let result = List.map multiplica2 listaFiltrada
    result
```

Operador Pipe

- Agora com o operador pipe:

```
let dobraMenores5 =
    let multiplica2 numero = numero * 2
    let menorQue5 numero = numero < 5

    [0..10] |> List.filter menorQue5 |> List.map multiplica2
```

Operador Pipe

- Mesmo que o pipe normalmente seja utilizado com listas, ele pode ser utilizado independente delas, exemplo:

```
let soma2 = (+) 2
let multiplica5 = (*) 5

let operacoes numero1 numero2 =
    numero1 + numero2
    |> soma2
    |> multiplica5
```

Exercícios

- Escreva uma função que receba uma lista com valores entre 0 e 10, depois retorne os número pares e que forem menores do que 5, utilizando o operador pipe

- Transforme a função em pipeline:

```
let multiplicaPor = (*)
let multiplicaPor10 = multiplicaPor 10
let soma = (+)
let soma5 = soma 5
let soma2 = soma 2

let calculos numero =
    let r1 = soma2 numero
    let r2 = multiplicaPor10 r1
    let r3 = soma5 r2
    r3
```