

Projeto e escrita de Programas

Programação Funcional
DCOMP-UFS

- Implementar
 - Escrever um programa
- Projetar
 - Planejamento prévio à implementação
 - Arquitetar uma solução
 - Tudo o que fazemos antes de implementar

Por onde começo?

- Eu entendo o problema?
 - Descrições incompletas, ambíguas e/ou inconsistentes

“Devolva o valor do meio dentre três números”

- Para 2, 4 e 3 o resultado é 3
- Para 2, 4 e 2?

- Ainda para problemas simples podem existir detalhes a serem pensados
- Não há resposta correta – programador e usuário devem definir
- Exemplos são facilitadores
- Quais são os casos limites?
- Importante descobrir falhas na especificação o mais cedo possível

Outro exemplo

- Dados três números, calcule a soma daqueles que são pares
- Se somente um for par?
- Se nenhum for par?

Defina os tipos das funções cedo

Use nomes significativos

```
valorDoMeio :: Int -> Int -> Int -> Int
```

Defina seus próprios tipos se necessário (veremos depois)

Pensamento *botom-up*

- Que recursos são disponíveis
 - pré-definidos, bibliotecas, outros problemas resolvidos, ...
- Já resolvi algum problema similar?
 - Posso usá-lo como modelo?
 - Posso usá-lo “dentro da minha solução”?

Calcular o maior de três números

```
maior :: Int -> Int -> Int
maior x y
  | x >= y      = x
  | otherwise   = y
```

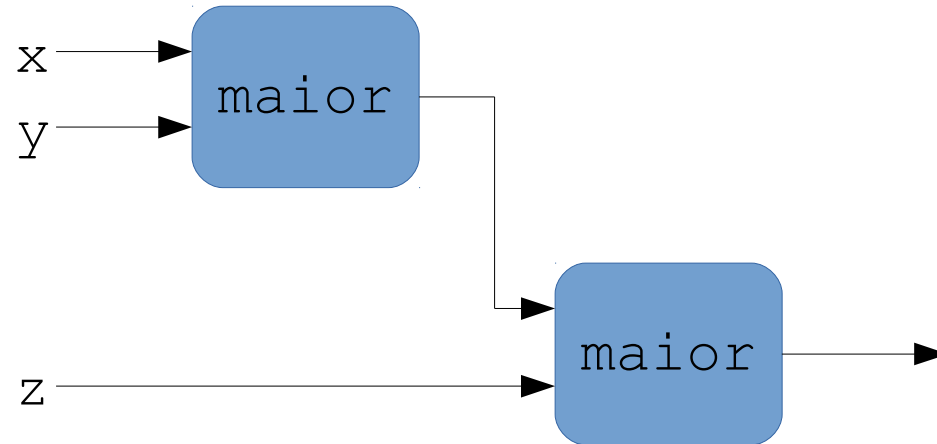
Podemos usar esta solução como modelo

```
maiorDe3 :: Int -> Int -> Int -> Int
maiorDe3 x y z
  | x é o maior dos três    = x
  | y é o maior dos três    = y
  | otherwise               = z
```



```
maiorDe3 :: Int -> Int -> Int -> Int
maiorDe3 x y z
  | x >= y && x >= z = x
  | y >= x && y >= z = y
  | otherwise       = z
```

Solução prévia como parte da solução

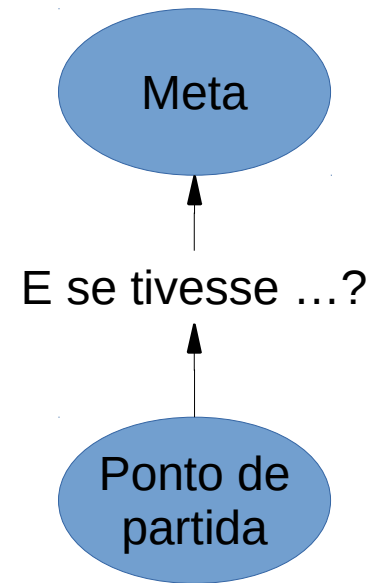
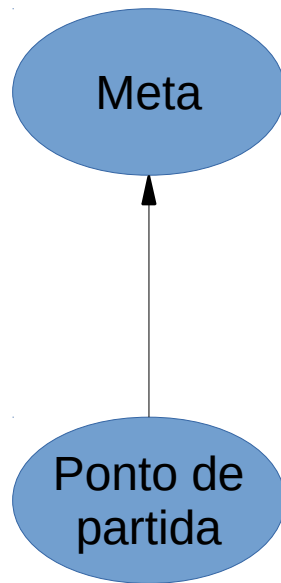


```
maiorDe3 :: Int -> Int -> Int -> Int  
maiorDe3 x y z = maior (maior x y) z
```

```
maiorDe3 :: Int -> Int -> Int -> Int  
maiorDe3 x y z = (x `maior` y) `maior` z
```

Pensamento top-down

- Decomponha o problema em partes (dividir para conquistar)
 - Resolva aspectos do problema separadamente
 - Componha a solução
- Como decompor
 - Use abstração
E se tivesse uma função que preciso?



```
valorDoMeio :: Int -> Int -> Int -> Int
valorDoMeio x y z
  | condição para x ser a solução = x
  | condição para y ser a solução = y
  | otherwise                     = z
```

```
valorDoMeio :: Int -> Int -> Int -> Int
valorDoMeio x y z
  | entre y x z = x
  | entre x y z = y
  | otherwise    = z
```

```
entre :: Int -> Int -> Int -> Bool
entre a b c = a <= b && b <= c ||
              a >= b && b >= c
```

A solução é correta?

- Testes
- Provas

A solução pode ser melhorada?

```
maiorDe3 :: Int -> Int -> Int -> Int
maiorDe3 x y z
  | x >= y && x >= z = x
  | y >= x && y >= z = y
  | otherwise       = z
```

Para a segunda condição podemos usar o fato que x não pode ser o maior dos três

```
maiorDe3 :: Int -> Int -> Int -> Int
maiorDe3 x y z
  | x >= y && x >= z = x
  | y >= z           = y
  | otherwise       = z
```

Resumo

- Entenda o problema
- Defina os tipos das funções cedo
- Use estratégias
 - Que recursos tenho
 - Já resolvi algo similar
 - Decomponha em subproblemas
- Valide a solução
- Analise a solução para solidificar o aprendizado

Definições locais

```
valorDoMeio :: Int -> Int -> Int -> Int
valorDoMeio x y z
  | entre y x z = x
  | entre x y z = y
  | otherwise   = z
where
    entre a b c = a <= b && b <= c ||
                  a >= b && b >= c
```

Útil para escrever top-down e passo a passo

A definição de `entre` dentro do **where** é local

Só é visível dentro da definição de `valorDoMeio`

```
bmiTell :: Float -> Float -> String
bmiTell weight height
  | weight / height ^ 2 <= 18.5 = "You're underweight"
  | weight / height ^ 2 <= 25.0 = "You're normal"
  | weight / height ^ 2 <= 30.0 = "You're fat"
  | otherwise                  = "You're a whale"
```

Código repetido

- suscetível a erros
- difícil manter
- ineficiente
- difícil de ler

```
bmiTell :: Float -> Float -> String
bmiTell weight height
  | bmi <= 18.5 = "You're underweight"
  | bmi <= 25.0 = "You're normal"
  | bmi <= 30.0 = "You're fat"
  | otherwise   = "You're a whale"
where bmi = weight / height ^ 2
```

```
bmiTell :: Float -> Float -> String
bmiTell weight height
  | bmi <= skinny = "You're underweight"
  | bmi <= normal = "You're normal"
  | bmi <= fat    = "You're fat"
  | otherwise     = "You're a whale"
where bmi = weight / height ^ 2
      skinny = 18.5
      normal = 25.0
      fat    = 30.0
```

Nomear constantes
melhora legibilidade
permite que modificações sejam localizadas

Escopos

Escopo = trecho de alcance de uma definição

- Nomes definidos numa cláusula **where** são visíveis somente na definição da função
- Parâmetros formais são também locais
- Definições locais podem “esconder” definições não locais
- Dentro de um escopo, valores podem ser usados antes ou depois da definição

```
bmiTell :: Float -> Float -> String
bmiTell weight height
  | bmi <= skinny = "You're underweight"
  | bmi <= normal = "You're normal"
  | bmi <= fat    = "You're fat"
  | otherwise     = "You're a whale"
where bmi = weight / height ^ 2
      skinny = 18.5
      normal = 25.0
      fat = 30.0
```

```
heightTell :: Float -> String
heightTell height
  | height <= dwarf = "You're dwarf"
  | height >= normal = "You're giant"
  | otherwise      = "You're normal"
where dwarf = 1.5
      normal = 1.9
```

```
normal :: Int -> Bool
normal a = 5 <= a && a <= 10
```

```
normalTell :: Int -> String
normalTell n
  | normal n = "Normal"
  | otherwise = "Anormal"
```

Expressões **let**

```
> 4 * (if 10 > 5 then 10 else 0) + 2  
42  
>  
> 4 * (let a = 9 in a + 1) + 2  
42
```

- Similar com **where**, produz bindings locais
- Diferentemente do **where**, expressões **let** são expressões

```
cylinder :: Float -> Float -> Float
cylinder r h =
    let sideArea = 2 * pi * r * h
        topArea = pi * r^2
    in sideArea + 2 * topArea
```

```
cylinder :: Float -> Float -> Float
cylinder r h = sideArea + 2 * topArea
    where sideArea = 2 * pi * r * h
          topArea = pi * r^2
```

O mais comum é usar cláusulas `where`