# (18/04) Muitas funções em Haskell!

## Função fatorial em Python

```
def fat(n):
    if n == 0:
        return 1

    else:
        return n*(fat(n-1))
```

```
fat(n) = if n == 0

    then 1
    else n*(fat(n-1))
```

## Função sinal

> f(x) = -1, se x < 0 ; 0 se x = 0;  1  se x > 0;

Em haskell:

```
main = do
    l <- getLine
    let x :: Integer
            x = read l
    let s :: sinal(x)
    let str = show s
    putStrLn str
```

```
sinal :: Integer -> Integer
sinal x -- definicao da funcao sinal
    | x < 0 = -1 -- testa primeira guarda
    | x == 0 = 0 -- testa segunda guarda
    | otherwise = 1 -- essa guarda eh sempre verdade (otherwi
```

Em Haskell, a maneira mais matemática de fazer casos não é usando `if else`, e sim da maneira apresentada acima.

Operador `|` se chama guarda.

- Haskell tenta todas as guardas na ordem que estão escritas, até alguma ser verdadeira

# Fatorial em Haskell (nao definitivo ainda)

```
fat :: Integer -> Integer
fat x
    | n == 0 = 1
    | otherwise = n * ( fat(n-1) )
```

## Outras funções

```
tempo :: Integer -> String
tempo t
    | t < 0 = "Congelante"
    | t < 15 = "Fresco"
    | t < 25 = "Agradável"
    | t < 35 = "Quente"
    | t < 45 = "Escaldante"
    | otherwise = "Morto"
```

```
imc :: Float -> Float -> String
img h w
```

```
    | w / (h*h) < 20 = "Abaixo"
    | w / (h*h) < 30 = "Normal"
    | w / (h*h) < 35 = "Sobrepeso"
    | otherwise = "Obeso"
```

Ruim!! definido `w / (h*h)` todas as vezes. Podemos definir de forma matemática:

```
imc :: Float -> Float -> String
img h w
    | b < 20 = "Abaixo"
    | b < 30 = "Normal"
    | b < 35 = "Sobrepeso"
    | otherwise = "Obeso"
    where
        b = w / (h*h)
```

## Baskara

Definindo Baskara da forma matemática.

```
baskara :: Float -> Float -> Float -> [Float]
baskara a b c
    | delta > 0 = [(-b + sqdelta)/ (2*a), (-b - sqdelta)/ (2*
    | delta == 0 = [-b/(2*a)]
    | otherwise []
    where
        delta = b*b - 4*a*c
        sqdelta = delta ** 0.5

        --OBS : lazy programming -> ele testa o guarda primei
        -- se ele so passar no segundo guarda (delta==0) ele
        -- de sqdelta, pois nao sera utilizado!
```

Programação em Haskell é muito próximo de definir matematicamente operações e funções.

## E quando houver muitas denifições?

A melhor maneira é definir a função várias vezes, assim, será escolhido o que melhor encaixa.

```
f :: Integer -> Integer -> Integer
f 5 5 = 15
f 3 6 = 12
f 4 y = y + 3
f x y = x + 9
f x y = x + y
```

## Codigo definitivo fatorial

```
fatorial :: Integer -> Integer
fatorial 0 = 1
fatorial n = n * (fatorial (n-1))
```

```
soma l = if l == []
            then 0
            else (head l) + (soma (tail l)) -- CODIGO PESSIM

-- um pouco melhor
soma l
        | l == [] = 0
        | otherwise = (head l ) + (soma (tail l))

-- melhor mas ainda nao definitivo
soma [] = 0
soma l = (head l ) + (soma(tail l))

-- solucao final
soma :: [Integer] -> Integer
```

```
soma [] = 0 -- uma lista vazia so encaixa aqui
soma (x:xs) = x + (soma xs) -- o resto, aqui, mas encaixa dec
                                                        -- em
```

```
-- definindo outra funcao que soma [4,3,9,1,5,4] -> [7,10,9]
somapares :: [Integer] -> [Integer]
somapares [] = [] -- [] -> []
somapares [x] = [x] -- [3] -> [3]
somapares (x1:x2:xs) = (x1 + x2):(somapares xs)-- [4,3,9,1,5,
-- x1 eh cabeca, (x2:x3 eh a cauda), x2 eh a cabeca da cauda

-- funcao que soma os numeros consecutivamente
somaconsec :: [Integer] -> [Integer]
somaconsec [] => []
somaconsec [x] = [x]
somaconsec (x1:x2:xs) = (x1+x2):(somaconsec( x2:xs ))
--            ^desconstruindo         -- ^reconstruindo

-- dado uma lista, inverte o primeiro segundo, etc
invertepares :: [Integer] -> [Integer]
invertepares [] => []
invertepares [x] = [x]
invertepares (x1:x2:xs) = x2:x1:(invertepares xs)

--
posimpar :: [Integer] -> [Integer]
posimpar [] = []
posimpar [x] = [x]
posimpar (x1:_:xs) = x1:(posimpar xs) -- nao usamos x2, logo,
-- eh diferente de
posimpar (x1:xs) = x1:(posimpar xs)


f [] = 0
f[[x]: _]
```