Haskell

Allek Cezana Rajab
Henrique Bertolo Selga
João Paulo Coelho

Índice

Índice

Introdução - Histórico

Influências:

- 1930: Alonzo Church desenvolveu o cálculo de lambda, um simples, mas poderoso teorema de funções.
- 1950: John McCarthy, influenciado pela teoria do lambda, desenvolveu Lisp, a primeira linguagem funcional.
- 1970: Um grupo de pesquisadores, liderados por Robin Milner, desenvolveram ML, a primeira linguagem funcional moderna introduzindo a inferência de tipos e tipos polimórficos na linguagem.

Introdução - Histórico

Origem:

- 1987: Em uma conferência realizada em Amsterdã, a comunidade de programação funcional decidiu criar uma linguagem puramente funcional padronizada e semântica não rígida, designando um comitê para o desenvolvimento desse projeto.
- A linguagem possui esse nome em homenagem ao lógico Haskell Brooks Curry.
- 1999: Haskell 98 define uma versão estável da linguagem.

Programação Funcional

Linguagens funcionais operam apenas sobre funções, as quais recebem listas de valores e retornam um valor. O objetivo da programação funcional é definir uma função que retorne um valor como a resposta do problema.

Um programa funcional é uma chamada de função que normalmente chama outras funções para gerar um valor de retorno. As principais operações nesse tipo de programação são a composição de funções e a chamada recursiva de funções. Outra característica importante é que funções são valores de primeira classe que podem ser passados para outras funções.

Programação Funcional

De maneira geral, pode-se pensar na programação funcional simplesmente como avaliação de expressões, cujo fluxo é:

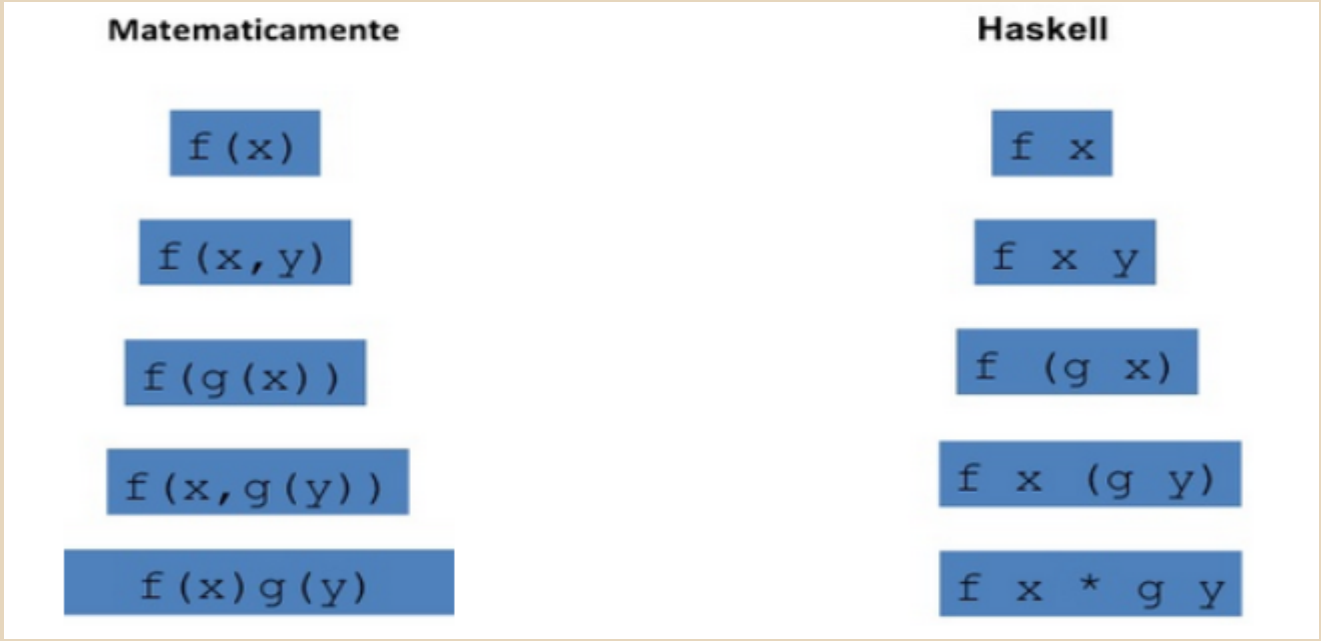
- O programador define uma função para resolver um problema e a passa para o computador;
- Uma função pode envolver várias outras funções em sua definição, inclusive si própria;

Programação Funcional

- O computador funciona então como uma calculadora que avalia expressões escritas pelo programador através de simplificações até chegar a uma forma base.

Programação Funcional

- Equivalência Funções Matemáticas - Haskell



Características Gerais da LP

- Haskell é uma LP puramente funcional;
- Os tipos de dados são bem definidos, sendo fortemente tipado. Toda função, variável, constante tem um tipo de dado, que sempre pode ser determinado;
- Sistema de tipos estático;
- Utiliza avaliação de expressões tardia (*lazy*);
- Ausência de desvios incondicionais;
- Polimorfismo universal paramétrico;
- Suporte a recursão, casamento de padrões, list comprehensions e guard statements;

Características Gerais da LP

- Utiliza Identação;
- É Case-sentive;
- Inferência de tipos;
- Muitos conceitos de alto nível;
- É compilável e/ou interpretada;
- Gerenciamento automático de memória;
- Passagem de parâmetros por cópia;
- Suporte à concorrência através de *Threads*.

Requerimentos/Instalação

- Para começarmos, tudo que necessitamos é um editor de texto e um compilador Haskell. O compilador mais utilizado é o GHC
- O GHC pega um script em Haskell de extensão .hs e o compila, mas é possível que você o utilize de forma interativa.
- Para chamar o modo iterativo basta digitar ghci no seu prompt.

Requerimentos/Instalação

Implementadores:

Hugs

- Interpretador;
- Escrito em C;
- Portável;
- Leve;
- Ideal para iniciantes;

GHC

- Interpretador (GHCi) e Compilador;
- Escrito em Haskell;
- Menos portável;
- Mais lento;
- Exige mais memória;
- Produz programas mais rápidos;

Requerimentos/Instalação

- Isto é o que deve aparecer quando você chama o GHCi

```
GHCi, version 7.8.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude>
```

- Para mudar o nome do prompt, basta digitar “:set prompt NomeDesejado”

```
Prelude> :set prompt ghci>
ghci>
```

Requerimentos/Instalação

- Suponha que seja criado algumas funções dentro de um arquivo chamado teste.hs (hs significa Haskell Script). Para você carregá-lo basta digitar *“: l teste” e suas funções estarão prontas para serem utilizadas (desde que teste.hs esteja na mesma pasta a qual ghci foi invocado)*
- *Se por acaso alguma alteração for feita em teste.hs, você pode simplesmente digitar “: l teste” novamente ou “: r”*

Tutorial Básico

- Em Haskell, diferente das linguagens imperativas ao qual estamos mais familiarizados, você não pode indicar que uma variável é uma coisa, e mais tarde modificá-la sendo uma outra coisa. Por exemplo, se você define “a” sendo 3, você não pode depois dizer que “a” vale 6
- Com isso, evitamos efeitos colaterais.

Tutorial Básico

- Vamos começar mostrando alguns exemplos auto explicativos.

```
ghci> 1+8
9
ghci> 23 * 53
1219
ghci> 18 / 6
3.0
ghci>  2 + 3.75
5.75
```

- Todos os operadores aritméticos comuns podem ser utilizados livremente na mesma forma usual ao qual estamos acostumados, devemos ter apenas cuidado quando formos utilizar os números negativos

```
ghci> 3 * -3

<interactive>:16:1:
    Precedence parsing error
        cannot mix `*' [infixl 7] and prefix `-' [infixl 6] in the same infix expression
ghci> 3 * (-3)
-9
```

Tutorial Básico

- Você pode não ter percebido, mas estávamos utilizando funções o tempo todo. Por exemplo o “/” é uma função que pega dois números e os divide. Como mostrado no slide anterior, chamamos ela colocando-a entre dois números, este tipo de função é conhecida como função inserida.

- Podemos definir então dois tipos de funções:

  ⇒ Funções inseridas(infixas)

  ⇒ Funções prefixas

Tutorial Básico

1. ***Funções inseridas***

- Como vimos anteriormente, as funções inseridas, são funções normalmente pré estabelecidas pela LP e que estão contidas entre dois tipos primitivos. Veja alguns exemplos de funções inseridas abaixo

```
ghci> "Juntos" ++ " somos mais" ++ " " ++ "fortes"
"Juntos somos mais fortes"
ghci>
```

- Repare que o “++” está inserido entre duas strings, daí vem o nome função inserida

Tutorial Básico

- Outros exemplos de funções inseridas são: “+” “*” “-” “ ==”
- Agora é possível entender o por que esta imagem apresentada em alguns slides anteriores mostra que quando multiplicamos 3* -3 ocorre um erro.

```
ghci> 3 * -3

<interactive>:16:1:
    Precedence parsing error
        cannot mix `*' [infixl 7] and prefix `-' [infixl 6] in the same infix expression
ghci> 3 * (-3)
-9
```

- Você está utilizando duas funções inseridas seguidas, é como se você tentasse multiplicar “3” com “-”, gerando o erro.

Tutorial Básico

2. ***Funções prefixas***

- Este é o tipo de função ao qual estamos habituados.
- Sua estrutura é a seguinte: Primeiro vem o nome da função e logo em seguida os seus parâmetros.

```
ghci> succ 13
14
ghci> succ 'a'
'b'
ghci>
```

- A função succ mostrada acima pega qualquer coisa, define o seu sucessor e depois retorna.

Tutorial Básico

- Como é possível reparar nós apenas escrevemos o nome da função e a separamos do parâmetro por um espaço, a mesma estrutura continua quando chamarmos funções com mais parâmetros.
- Um exemplo simples de uma função com dois parâmetros é a “min” e a “max”. Elas pegam coisas que podem ser colocadas em ordem e retornam o maior ou menor valor.

```
ghci> min 9 13
9
ghci> max 5 8
8
ghci>
```

Tutorial Básico

- Caso o número de parâmetros não esteja correto, uma mensagem de erro irá aparecer.
- Quando escrevemos min 12 9 2 a função min olha apenas os dois primeiros números e encontra o seu valor, que no caso é o 9, mas logo em seguida se depara com um 2 solto
- Repare o mesmo erro acontece quando escrevemos 3 4

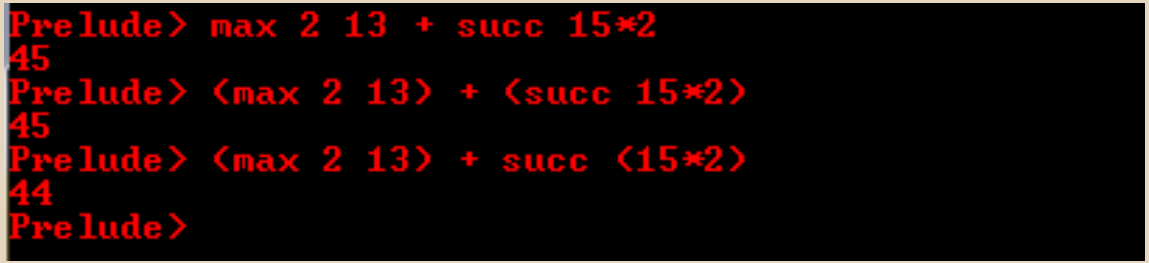
```
ghci> min 12 9 2

<interactive>:90:1:
    Could not deduce (Ord (a0 -> t))
      arising from the ambiguity check for 'it'
    from the context (Ord (a -> t), Num (a -> t), Num a)
      bound by the inferred type for 'it':
                 (Ord (a -> t), Num (a -> t), Num a) => t
      at <interactive>:90:1-10
    The type variable 'a0' is ambiguous
    When checking that 'it'
      has the inferred type 'forall a t.
                             (Ord (a -> t), Num (a -> t), Num a) =>
                             t'
    Probable cause: the inferred type is ambiguous
ghci> 3 4

<interactive>:91:1:
    Could not deduce (Num (a0 -> t))
      arising from the ambiguity check for 'it'
    from the context (Num (a -> t), Num a)
      bound by the inferred type for 'it': (Num (a -> t), Num a) => t
      at <interactive>:91:1-3
    The type variable 'a0' is ambiguous
    When checking that 'it'
      has the inferred type 'forall a t. (Num (a -> t), Num a) => t'
    Probable cause: the inferred type is ambiguous
ghci> min 12 9 +2
11
ghci>
```

Tutorial Básico

- Se quisermos então fazer uma especificação dos parâmetros, basta colocarmos entre parênteses.



- Repare nos diferentes resultados gerados, a diferença está nos parênteses.

No primeiro e no segundo caso o programa faz: 13 + (16*2) = 45. No terceiro caso temos: 13+ 31 = 44

Tutorial Básico

- Caso a função receba dois parâmetros, podemos escreve-la de forma infixa, ou seja como se fosse uma função inserida, para isto basta colocarmos aspas simples:

```
Prelude> elem 4 [2,5,8,13,5,2,8,4]
True
Prelude> 4 `elem` [2,5,8,13,5,2,8,4]
True
Prelude>
```

- Isto se torna logicamente muito mais claro, e acaba contribuindo com a legibilidade do programa.

Tutorial Básico

3. ***Como criar funções***

- O primeiro passo é criar um arquivo .hs.
- Nele você poderá implementar suas funções e depois carregá-lo para o ghci.
- Um exemplo simples de função pode ser visto na imagem abaixo:

```
double x = x + x
```

- Repare que as funções são definidas de forma semelhante a como são chamadas.

Tutorial Básico

- Quando definimos funções, elas terão um "=" e logo em seguida definiremos o que ela faz.
- Depois de criarmos, basta carregar os scripts e verificar seu funcionamento:

```
*Main> double 4
8
*Main> double 3.15
6.3
*Main> double 'a'

<interactive>:6:1:
    No instance for (Num Char) arising from a use of `double'
    In the expression: double 'a'
    In an equation for `it': it = double 'a'
*Main> _
```

Tutorial Básico

- Para criar funções com dois ou mais parâmetros, segue-se a mesma lógica:

```
double2 x y = (x*2) + (y*2)

double3 x y z = (x*2) + (y*2) + (z*2)

double4 x y z k = x + x + x + x
```

- Repare que você não precisa necessariamente utilizar todos os parâmetros passados.

Tutorial Básico

- Eis alguns resultados gerados utilizando esta função:

```
*Main> double2 3 2.5
11.0
*Main> double3 13 5 9
54
*Main> double4 1 4 3 2
4
*Main> double4 1 2

<interactive>:6:1:
    No instance for (Show (t10 -> t20 -> a0))
      arising from a use of `print'
    In a stmt of an interactive GHCi command: print it
*Main> _
```

- Repare que mesmo que o double4 utilize apenas o primeiro parâmetro, ainda assim é necessário que você passe o mesmo número de parâmetros pré-definido.

Tutorial Básico

- Como esperado, você pode chamar suas funções dentro de outras funções.

```
double x = x + x

double2 x y = double x + double y
```

- Modularizar seus problemas é algo eficiente e confiável, sem dizer que pode ajudar muito na modificabilidade e reutilização do seu código.
- Funções em Huskell não necessitam estar escritas em ordem,ou seja não importa se você define primeiro double ou double2

Tutorial Básico

- Agora dê uma olhada em uma função que calcula o dobro apenas se os números forem maiores que 200:

```
doubleBigNumber x = if x > 200
                    then x*2
                    else x
```

```
*Main> doubleBigNumber 3
3
*Main> doubleBigNumber 300
600
*Main> doubleBigNumber 200
200
*Main> doubleBigNumber 200.00000001
400.00000002
*Main>
```

Palavras reservadas

case	class	data	deriving	do
else	if	import	in	infix
infixl	infixr	instance	let	of
module	newtype	then	type	where

Operadores Básicos

>	Maior
>=	Maior ou igual
==	Igual
/=	Diferente
<	Menor
<=	Menor ou igual
&&	e
||	ou
not	Negação
++	Concatenação

+	Soma
-	Subtração
*	Multiplicação
^	Potência
div	Divisão inteira
mod	Resto da divisão
abs	valor absoluto de um inteiro
negate	troca o sinal do valor
:	Composição de lista

Amarrações - Convenções

- Apresenta sistema de tipos estático.
- Todos os tipos são conhecidos em tempo de compilação.
- Existe inferências de tipos. Não é preciso explicitar de qual tipo é um certo identificador.
- Haskell permite que um mesmo identificador seja declarado em diferentes partes do programa, possivelmente representando diferentes entidades.
- As conversões de dados são explicitas.

Amarrações - Convenções

- Os identificadores necessariamente devem começar com uma letra maiúscula ou minúscula - seguida por uma sequência opcional de letras,dígitos, sublinhas ou apóstrofes.
- Definições de funções ou variáveis devem começar com letra minúscula.
- Tipos, construtores, módulos e classes de tipos têm que começar com letras maiúsculas.

Tipos de dados

- Tipos Primitivos:
  - Tipo Unitário: Unit, implementação do conjunto 1;
  - Tipo Vazio: Void;
  - Tipo Lógico: Bool;
  - Tipo Caracter: Char;
  - Tipos Numéricos:
    - Inteiros:
      - Int, Integer;
    - Reais:
      - Float, Double.

Tipos de dados

- Inteiros:

Tipos	`Int, Integer`
Valores	`Int`, inteiros de comprimento fixo. A gama é de, pelo menos, -2^{29} a $2^{29}-1$. `Integer`, inteiros de comprimento arbitrário.
Operadores	`+, *, -, negate` ($\doteq$ `-` unário), `quot, div, rem, mod`

Os operadores ‘quot’ e ‘rem’, fazem arrendondamento em direção ao infinito negativo, enquanto ‘div’ e ‘mod’ em direção ao zero.

Tipos de dados

- Reais:

Tipos	`Float, Double`
Valores	`Float`, reais precisão simples. `Double`, reais precisão dupla.
Operadores	`+, *, /, -, negate, ^, pi, exp, log, sqrt, **, logBase, sin, cos, tan, asin, acos, atan, sinh, cosh, tanh, asinh, acosh, atanh.`

O operador ‘^’ é usado quando o expoente é inteiro, enquanto o operador ‘**’ quando o expoente é real.

Tipos De Dados - Compostos

- Tuplas:
  - São a implementação dos produtos cartesianos em Haskell.

$$T = T_1 \times T_2 \times \cdots \times T_m$$

Tipo	$(\mathtt{T}_1, \ldots, \mathtt{T}_n)$
Valores	$(\mathtt{e}_1, \ldots, \mathtt{e}_k)$, com $k \geq 2$
Construtores	$(,\ldots,)$
Operadores	`fst`, `snd`, só aplicáveis a pares.

Tipos De Dados - Compostos

```
ghci> zip [5,3,2,6,2,7,2,5,4,6,6] ["im","a","turtle"]
[(5,"im"),(3,"a"),(2,"turtle")]
```

```
ghci> zip [1..] ["apple", "orange", "cherry", "mango"]
[(1,"apple"),(2,"orange"),(3,"cherry"),(4,"mango")]
```

Tipos Dados - Compostos

- Listas:
  - As listas são um conjunto de sequências finitas de elementos de um dado tipo A, ou seja, diferentemente das tuplas as listas em Haskell são homogêneas.
  - Um aspecto importante das listas em Haskell é que elas podem ser definidas por compreensão, de maneira muito semelhante ao que estamos habituados com a definição de conjuntos na matemática.
  - Ex:

```
λ [n * n | n <- [1..10]]
[1,4,9,16,25,36,49,64,81,100]
```

Tipos Dados - Compostos

- Listas

Lista A = Vazia+A×Lista A

Tipo	[A], listas de elementos de um tipo A
Valores	[] (lista vazia); [e_0,...,e_m] (lista não vazia)
Construtores	[] : 1 ⟶ Lista * ⟼ [] : : A × Lista A ⟶ Lista A (a,l) ⟼ l′ = a:l [e_0,e_1,...,e_m] ≐ e_0:(e_1:(...:(e_m : [])))
Operadores	++, head, last, tail, init, nul, length, !!, foldl, foldl1, scanl, scanl1, foldr, foldr1, scanr, scanr1, iterate, repeat, replicate, cycle, take, drop, splitAt, takeWhile, dropWhile, span, break, lines, words, mlines, mwords, reverse, and, or, any, all, elem, notElem, lookup, sum, product, maximun, minimun, concatMap, zip, zip3, zipWith, zipWith3, unzip, unzip3

Tipos dados - Compostos - Listas

Como criar:

```
ghci> let lostNumbers = [4,8,15,16,23,48]
ghci> lostNumbers
[4,8,15,16,23,48]
```

Concatenar:

```
ghci> [1,2,3,4] ++ [9,10,11,12]
[1,2,3,4,9,10,11,12]
ghci> "hello" ++ " " ++ "world"
"hello world"
ghci> ['w','o'] ++ ['o','t']
"woot"
```

```
ghci> 'A':" SMALL CAT"
"A SMALL CAT"
ghci> 5:[1,2,3,4,5]
[5,1,2,3,4,5]
```

Tipos de Dados Compostos - Listas

Obter elemento por índice:

```
ghci> "Steve Buscemi" !! 6
'B'
ghci> [9.4,33.2,96.2,11.2,23.25] !! 1
33.2
```

Lista de listas:

```
ghci> let b = [[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
ghci> b
[[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
ghci> b ++ [[1,1,1,1]]
[[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3],[1,1,1,1]]
ghci> [6,6,6]:b
[[6,6,6],[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
ghci> b !! 2
[1,2,2,3,4]
```

Funções Básicas Em Listas

head:

```
ghci> head [5,4,3,2,1]
5
```

tail:

```
ghci> tail [5,4,3,2,1]
[4,3,2,1]
```

last:

```
ghci> last [5,4,3,2,1]
1
```

init:

```
ghci> init [5,4,3,2,1]
[5,4,3,2]
```

```
ghci> head []
*** Exception: Prelude.head: empty list
```

length:

```
ghci> length [5,4,3,2,1]
5
```

null:

```
ghci> null [1,2,3]
False
ghci> null []
True
```

reverse:

```
ghci> reverse [5,4,3,2,1]
[1,2,3,4,5]
```

maximum
minimum

```
ghci> minimum [8,4,2,1,5,6]
1
ghci> maximum [1,9,2,3,4]
9
```

Funções Básicas Em Listas

take:

```
ghci> take 3 [5,4,3,2,1]
[5,4,3]
ghci> take 5 [1,2]
[1,2]
ghci> take 0 [6,6,6]
[]
```

drop:

```
ghci> drop 3 [8,4,2,1,5,6]
[1,5,6]
ghci> drop 0 [1,2,3,4]
[1,2,3,4]
ghci> drop 100 [1,2,3,4]
[]
```

sum:

```
ghci> sum [5,2,1,6,3,2,5,7]
31
```

product:

```
ghci> product [6,2,1,2]
24
```

elem:

```
ghci> 4 `elem` [3,4,5,6]
True
ghci> 10 `elem` [3,4,5,6]
False
```

Rangers

```
ghci> [1..20]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
ghci> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"
ghci> ['K'..'Z']
"KLMNOPQRSTUVWXYZ"
```

```
ghci> [2,4..20]
[2,4,6,8,10,12,14,16,18,20]
ghci> [3,6..20]
[3,6,9,12,15,18]
```

```
ghci> [0.1, 0.3 .. 1]
[0.1,0.3,0.5,0.7,0.8999999999999999,1.0999999999999999]
```

Compreensão de Listas

```
ghci> [x*2 | x <- [1..10]]
[2,4,6,8,10,12,14,16,18,20]
```

```
ghci> [x*2 | x <- [1..10], x*2 >= 12]
[12,14,16,18,20]
```

```
ghci> [ x | x <- [50..100], x `mod` 7 == 3]
[52,59,66,73,80,87,94]
```

```
boomBangs xs = [ if x < 10 then "BOOM!" else "BANG!" | x <- xs, odd x]
ghci> boomBangs [7..13]
["BOOM!","BOOM!","BANG!","BANG!"]
```

Compreensão de Listas

```
ghci> [ x | x <- [10..20], x /= 13, x /= 15, x /= 19]
[10,11,12,14,16,17,18,20]
```

```
ghci> [ x*y | x <- [2,5,10], y <- [8,10,11]]
[16,20,22,40,50,55,80,100,110]
```

```
ghci> [ x*y | x <- [2,5,10], y <- [8,10,11], x*y > 50]
[55,80,100,110]
```

```
ghci> let xxs = [[1,3,5,2,3,1,2,4,5],[1,2,3,4,5,6,7,8,9],[1,2,4,2,1,6,3,1,3,2,3,6]]
ghci> [ [ x | x <- xs, even x ] | xs <- xxs]
[[2,2,4],[2,4,6,8],[2,4,2,6,2,6]]
```

Tipos Compostos

- Strings
  - São simplesmente sequências de caracteres, ou seja, uma lista de Char.

Tipo	String
Valores	sequências de caracteres entre aspas. Por exemplo `"abc"` ($\doteq$ ['a','b','c'])
Operadores	Os operadores das listas.

  - Haskell utiliza a tabela Unicode como padrão para representação de caracteres.

I/O

I/O

- Vamos começar escrevendo um arquivo que imprima o famoso “Hello world” no terminal.

```
escreve = putStrLn "Hello world"
```

- Agora basta chamarmos a função criada.

I/O

```
main = do
    foo <- putStrLn "Hello, what's your name?"
    name <- getLine
    putStrLn ("Hey " ++ name ++ ", you rock!")
```

```
import Data.Char

main = do
    putStrLn "What's your first name?"
    firstName <- getLine
    putStrLn "What's your last name?"
    lastName <- getLine
    let bigFirstName = map toUpper firstName
        bigLastName = map toUpper lastName
    putStrLn $ "hey " ++ bigFirstName ++ " " ++ bigLastName ++ ", how are you?"
```

Avaliação da LP

- Redigibilidade e Legibilidade:
  - Ambas são altas por conta da proximidade das funções (do programa) com as funções matemáticas, facilitando assim também seu aprendizado.
- Desempenho/Eficiência:
  - Alto uso da CPU e da memória principal, sendo assim um aspecto negativo da LP. Esse baixo desempenho é também consequência de seu aspecto puramente funcional, que obriga a utilização intensiva de recursão.

Avaliação da LP

- Reusabilidade:
  - Apresenta um bom nível em função da estrutura modular da linguagem, no entanto, por não ser uma linguagem orientada a objetos essa aspecto fica um pouco limitado
- Confiabilidade:
  - Alto grau de confiabilidade em função da ausência de desvios incondicionais, estados nas variáveis e de efeitos colaterais.

Referências Bibliográficas

- http://haskell.tailorfontela.com.br/introduction
- http://pt.wikipedia.org/wiki/Haskell_(linguagem_de_programa%C3%A7%C3%A3o)
- https://www.facebook.com/l.php?u=https%3A%2F%2Fwww.haskell.org%2Fhaskellwiki%2FHaskell_em_10_minutos&h=jAQE3GatF
- http://www.infoq.com/br/news/2009/02/rwh-book-interview
- http://www.decom.ufop.br/lucilia/func/slides/fun00.pdf
- http://www.inf.ufpr.br/andrey/ci062/ProgramacaoHaskell.pdf
- http://tryhaskell.org/
- https://www.haskell.org/haskellwiki/Haskell