Primeiros passos com Haskell

Programação Funcional

Prof. Rodrigo Ribeiro

Objetivos

- Configurar ambiente para desenvolvimento em Haskell.
- Usar o interpretador.
- Entender o modelo de execução de programas funcionais.

Setup

- Escolha um editor de texto: Atom, Sublime, Visual Studio Code, Emacs, Vi(m), etc. . .
- Instale o Haskell-stack: https://www.haskellstack.org.

Setup

- Os seguintes itens são opcionais:
  - HLint: ferramenta para sugestões para código Haskell.
  - Pandoc: ferramenta para produção dos slides usados na disciplina.

Haskell Stack

- Ferramenta para gerenciar projetos e bibliotecas.
- Instala todas as dependendências automaticamente (incluindo o compilador GHC).

Intepretador

- No terminal, digite `stack ghci` e você irá obter o prompt do interpretador:

```
Prelude*>
```

Interpretador

- Alguns comandos úteis do interpretador.
  - `:l` carrega um arquivo no interpretador.

```
Prelude*> :l Aula.hs
```

Interpretador

- Alguns comandos úteis do interpretador.
  - `:r` recarrega um arquivo no interpretador (útil após alterações).

```
Prelude*> :r
```

Interpretador

- Alguns comandos úteis do interpretador.
  - `:t` calcula o tipo de uma expressão fornecida como argumento.

```
Prelude*> :t True
True :: Bool
```

Interpretador

- Alguns comandos úteis do interpretador.
  - `:q` encerra a execução do interpretador.

```
Prelude*> :q
$>
```

Interpretador

- O GHCi é um REPL para Haskell.
- Read-Eval-Print-Loop
- Tente usar alguns operadores aritméticos:

```
Prelude*> 2 * (3 + 4)
14
Prelude*> (2 + 3) * 4
20
Prelude*> sqrt (3^2 + 4^2)
5.0
```

Prelude

- Biblioteca automaticamente importada por todo programa Haskell.
- Possui um grande número de tipos e funções pré-definidas.
- Exemplos:
  - Tipos numéricos: `Float`, `Double`, `Int` e `Integer`.
  - Funções usuais: $+$, *, -, $/$, div, mod, etc...
  - Importante: Haskell não possui subtyping.

Aplicação de função

- Em matemática, a aplicação de uma função f aos parâmetros x e y é representada como $f(x, y)$.
- Em Haskell, a aplicação é representada por espaços entre as expressões.

```
Prelude*> f x y
```

Meu primeiro módulo Haskell

- Módulos Haskell devem ter nomes iniciados com letras maiúsculas.
- Chame seu arquivo como `Aula01.hs`.

```haskell
module Aula01 where

main :: IO ()
main = return ()
```

Definindo uma função simples

```
double :: Int -> Int
double x = x + x
```

Componentes da definição

- Assinatura

```
double :: Int -> Int
```

- Equação

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

Requisitos de nomes

- Tipos, módulos e classes de tipos devem possui nomes que iniciam com uma letra maiúscula.
- Funções e variáveis devem iniciar com uma letra minúscula.

Executando o seu código

- No terminal, execute:

```
stack ghci Aula01.hs
```

ou

```
stack ghci
Prelude*> :l Aula01.hs
```

Executando sua função

```
Prelude*> double 5
10
Prelude*> double (double 2)
8
```

Execução do código

- Código Original

```
double :: Int -> Int
double x = x + x
```

- Execução

```
double 5 ==
5 + 5 ==
10
```

Execução do código

- Código Original

```
double :: Int -> Int
double x = x + x
```

- Execução

```
double (double 2) ==
```

Execução do código

- Código Original

```
double :: Int -> Int
double x = x + x
```

- Execução

```
double (double 2) ==
(double 2) + (double 2) ==
```

Execução do código

- Código Original

```
double :: Int -> Int
double x = x + x
```

- Execução

```
double (double 2) ==
(double 2) + (double 2) ==
(2 + 2) + (double 2) ==
```

Execução do código

- Código Original

```
double :: Int -> Int
double x = x + x
```

- Execução

```
double (double 2) ==
(double 2) + (double 2) ==
(2 + 2) + (double 2) ==
(2 + 2) + (2 + 2) ==
```

Execução do código

- Código Original

```
double :: Int -> Int
double x = x + x
```

- Execução

```
double (double 2) ==
(double 2) + (double 2) ==
(2 + 2) + (double 2) ==
(2 + 2) + (2 + 2) ==
4 + 4 ==
8
```

Execução do código

- Código Original

```
double :: Int -> Int
double x = x + x
```

- Execução

```
double (double 2) ==
```

Execução do código

- Código Original

```
double :: Int -> Int
double x = x + x
```

- Execução

```
double (double 2) ==
double (2 + 2) ==
```

Execução do código

- Código Original

```
double :: Int -> Int
double x = x + x
```

- Execução

```
double (double 2) ==
double (2 + 2) ==
(2 + 2) + (2 + 2) ==
```

Execução do código

- Código Original

```
double :: Int -> Int
double x = x + x
```

- Execução

```
double (double 2) ==
double (2 + 2) ==
(2 + 2) + (2 + 2) ==
4 + 4 ==
8
```

Funções puras

- Não realizam nenhum tipo de efeito colateral: I/O, atualização de memória, etc.
- Sempre o mesmo resultado para as mesmas entradas.
- Ordem de avaliação não altera o resultado final.

Exemplo

- Somando os primeiros números inteiros.
- Java style, boring...

```
int sumUpToN ( int n) {
   int total = 0;
   for (int i = 1 ; i <= n ; i ++)
     total += i ;
   return total ;
}
```

Em Haskell...

```haskell
sumUpToN :: Int -> Int
sumUpToN 0 = 0
sumUpToN n = n + sumUpToN (n - 1)
```

Executando sua função

```
Prelude*> sumUpToN 4
10
```

Executando sua função

```
sumUpToN 4 ==
4 + sumUpToN 3 ==
4 + (3 + sumUpToN 2) ==
4 + (3 + (2 + sumUpToN 1)) ==
4 + (3 + (2 + (1 + sumUpToN 0))) ==
4 + (3 + (2 + (1 + 0))) ==
10
```

Mais sobre a sintaxe de Haskell

- Palavras reservadas de Haskell.

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

Sintaxe de Haskell

- Regra de layout: blocos representados por identação de código. Configure seu editor para usar espaços ao invés de TABs.

```
a = b + c
    where
      b = 1
      c = 2
d = 3 * a
```

Sintaxe de Haskell

- Uso de chaves: Praticamente nenhum programador Haskell usa esse estilo.

```
a = b + c
    where {
      b = 1 ;
      c = 2
    }
d = 3 * a
```

Comentários

```haskell
-- comentário de uma linha

triple x = double x + x

{-
múltiplas linhas...
-}
```

Tipos básicos de Haskell

- Tipos numéricos: Int, Integer, Float, Double.
- Caracteres: Char - ‘a’, ‘0’.
- String: “abc”.
- Booleanos: Bool - True, False.

Listas em Haskell

- Sequências de valores de um mesmo tipo.
- Tipo de listas é representado por [a], em que a é um tipo.
- Exemplos

```
[1,3] :: [Int]
[True, False, False] :: [Bool]
['a', 'b', 'c'] :: [Char] == String
```

Tuplas em Haskell

- Sequências de valores de tipos possivelmente distintos.

```
(1, True, "ab") :: (Int, Bool, String)
(True, (1, 'a')) :: (Bool, (Int, Char))
```

Tipos Funcionais em Haskell

- Tipos de funções possuem o formato `T1 -> T2`, em que `T1` é o tipo de um parâmetro e `T2` do resultado.
- Exemplos

```
not :: Bool -> Bool
even :: Int -> Bool
```

Verificando aplicações.

- Se `f :: A -> B` e `x :: A` então `f x :: B`
- Formalmente:

$$\frac{f :: A \to B \quad x : A}{f\ x :: B}$$

Exemplo

- Verificando tipo de `not False`

$$\frac{not :: Bool \to Bool \qquad False : Bool}{not\ False :: Bool}$$

Exercícios

- Apresente a derivação de tipo da expressão `not (not True)`

Exercícios

1. Implemente uma função que, a partir de 3 números inteiros, retorne a soma dos quadrados dos dois maiores números.

Exercícios

1. A função de Fibonacci pode ser definida como:

$$F(n) = \begin{cases} 0 & \textit{se } n = 0 \\ 1 & \textit{se } n = 1 \\ F(n-1) + F(n-2) & \textit{caso contrário.} \end{cases}$$

Implemente uma função em Haskell para calcular `F(n)`.

Exercícios

2. A função de Fibonacci, implementada de acordo com a definição anterior, realiza um número exponencial de chamadas recursivas. Implemente uma função que calcule `F(n)` fazendo um número linear de chamadas recursivas.