Haskell: Introdução

Programação Funcional

Prof. Maycon Amaro

Visão Geral

- Criada em 1990 por um comitê.
- Novas especificações lançadas em 1998 e 2010.
- É uma linguagem funcional pura, de propósito geral, com tipos estáticos.

Instalação

A implementação padrão de Haskell é o Glasgow Haskell Compiler (GHC).

Ele está disponível gratuitamente no site oficial https://haskell.org. Para ajudar a gerenciar a compilação, dependências e testes, utilizem uma opção que também instala o Stack.

Se preferirem, podem instalar *apenas* o Stack. Ele baixará um ghc quando necessário.
https://docs.haskellstack.org/en/stable/install_and_upgrade/.

Editor de texto: qualquer um. *Syntax highlighting* é útil, mas evitem *linters* por enquanto.

Compilando um arquivo simples

Exemplo mínimo

```haskell
main :: IO ()
main = putStrLn "Hello, World!"
```

Com Stack, compile-o com `stack ghc <nome>.hs`.
Sem o Stack, compile-o com `ghc <nome>.hs -o <nome>`.

Use o comando `ghci <nome>.hs` ou `stack ghci <nome>.hs` para abrir o arquivo com o interpretador interativo.

Gerenciando Projetos com Stack

Gerenciando Projetos com Stack

Em projetos grandes, é indispensável um gerenciador de build. Vamos tentar nos preocupar com isso, mesmo criando projetos pequenos.

- Com Stack instalado, o comando `stack new <nome>` cria uma pasta de projeto Haskell.
- O comando `stack build` compila o projeto e o otimiza, baixando as depêndencias se necessário.
  - Use `stack build --fast` para compilar sem otimizações, o que poupa tempo durante o desenvolvimento.
- O comando `stack run` executa um projeto compilado, mas também o compila se for necessário.

Gerenciando Projetos com Stack

- O comando `stack ghci` inicia um interpretador interativo, que pode ser usado para executar funções específicas do seu projeto e fazer alguns testes simples.
- O comando `stack test` roda a bateria de testes especificada.
- O comando `stack clean` limpa alguns arquivos temporários.
- O comando `stack purge` limpa todos os arquivos temporários.

Criação do projeto

Ao criar um projeto com `stack new`, serão criados alguns arquivos e pastas.

- O arquivo `package.yaml` contém as configurações do projeto. Futuramente, incluiremos dependências aqui para serem baixadas automaticamente pelo Stack.
- A pasta `app` onde ficam os arquivos de código que podem se tornar executáveis, ou seja, arquivos com `main`.
- A pasta `src` onde ficam os arquivos da aplicação que podem se tornar bibliotecas.
- A pasta `test` onde ficam os códigos de teste.
- Um repositório git que pode ser associado a um local remoto.

Criação do projeto

A pasta `src` virá com um arquivo `Lib.hs`

```
module Lib
    ( someFunc
    ) where

someFunc :: IO ()
someFunc = putStrLn "someFunc"
```

Criação do projeto

A pasta app virá com um arquivo `Main.hs` que dependerá de `Lib.hs`

```haskell
module Main (main) where

import Lib

main :: IO ()
main = someFunc
```

Exemplo mínimo de projeto

arquivo `src/Lib.hs`

```haskell
module Lib (hello) where

hello :: String
hello = "Hello, world!"
```

arquivo `app/Main.hs`

```haskell
module Main (main) where

import Lib

main :: IO ()
main = putStrLn hello
```

Execute `stack run` na pasta do projeto para ver `Hello, world!` na tela.

Sintaxe

Comentários

Use `--` para comentários em uma linha e `{- -}` para comentários em bloco.

```
-- Comentário de linha

{- Comentário
   de bloco
-}
```

Definição de funções

Numa linha, escreva o nome da função e utilize `::` para começar a escrever o tipo. O formato dos tipos segue o estilo matemático usando $\rightarrow$ para marcar o domínio e contradomínio.

```
sucessor :: Int -> Int
sucessor x = x + 1
```

Esta função existe no prelúdio como `succ`.

Vários argumentos

Um termo em λ-cálculo tem "vários argumentos" quando há abstrações aninhadas. Exemplo: $\lambda x.\lambda y.\ldots.$

Ou seja, vários parâmetros são apenas funções retornando funções. Assim, o tipo da função mais interna é $A \to B$, e a de uma função mais externa sendo $C \to (A \to B)$.

No entanto, não é necessário colocar parênteses quando a associação for à direita.

```
soma :: Int -> Int -> Int
soma x y = x + y
```

Currying e Uncurrying

Uma outra forma equivalente, porém menos utilizada, é usar tuplas para agrupar parâmetros. Uma forma com tuplas é chamada de *uncurried* e uma forma com apenas -> é chamada de *curried*.

```
soma :: (Int, Int) -> Int
soma (x, y) = x + y
```

Entrada e Saída

Como Haskell controla muito bem os efeitos colaterais e a mudança de estado, só vamos entender direito entrada e saída no final da disciplina.

Mas uma forma de imprimir mensagens e ler um número inteiro é:

```haskell
-- com stack, isso vai no arquivo Lib.hs
suc :: Int -> Int
suc x = x + 1

-- com stack, isso vai no arquivo Main.hs
main :: IO ()
main = do
  putStrLn "Digite um número:"
  x' <- getLine
  let x = read x' :: Int
  putStrLn "A resposta é:"
  print (suc x)
```

Entrada e Saída tem Efeitos Colaterais

Lembre-se que o compilador rejeitará programas com entrada e saída fora da função `main`. Entenderemos o porquê disso em aulas posteriores.

Alternativa

Se quiser apenas observar o resultado de uma função, não é preciso escrever uma função `main`. Utilize `stack ghci` e execute a sua função manualmente. Use `:q` para sair do ambiente interativo.

Arquivo app/Main.hs

```haskell
module Main (main) where

main :: IO ()
main = putStrLn "Not yet"
```

Arquivo `src/Lib.hs`

```haskell
module Lib (suc) where

suc :: Int -> Int
suc x = x + 1
```

Linha de comando

```
stack ghci
ghci> suc 1
2
ghci> :q
Leaving GHCi.
```

Expressões lógicas

Haskell suporta booleanos nativamente. Note que `if-then-else` é uma expressão, não um comando.

```
exemplo :: Bool
exemplo =
  if True
     then False || True
     else True && (not False)
```

Regra de Layout

Assim como Python, Haskell usa a distância da coluna para delimitar escopos. Usá-la torna o código mais organizado, mas não é sempre *obrigatório*.

O seguinte código é equivalente:

```
exemplo :: Bool
exemplo = if True then False || True else True &&
  (not False)
```

Expressões Aritméticas

Haskell suporta diversas operações aritméticas

```
exemplo :: Int
exemplo = 2 + (div 6 3) * 5 - (mod 3 2)
```

As operações / (divisão) e ** (potenciação) retornam números fracionários mesmo quando os operandos são inteiros. Isso é reforçado pelo sistema de tipos.

```
exemplo :: Double
exemplo = 2 + 6 / 3 - 1 ** 2
```

Consultando o tipo

Dentro do ambiente interativo, digite `:t <funcao>` ou `:t (operador)` para verificar o tipo dele.

```
stack ghci
ghci> :t (+)
(+) :: Num a => a -> a -> a
ghci> :t (/)
(/) :: Fractional a => a -> a -> a
ghci> :t div
div :: Integral a => a -> a -> a
```

Estudaremos classes de tipos futuramente, mas por hora é suficiente saber apenas alguns aspectos.

O operador + funciona com qualquer tipo numérico, mas ambos os operandos precisam ser do mesmo tipo.

```
(+) :: Num a => a -> a -> a
```

O seguinte código causará um erro de tipo.

```
x :: Int
x = 4

y :: Double
y = 5

soma :: Double
soma = x + y
```

O operador / funciona com qualquer tipo que possa ser entendido como fracionário. Novamente, os operandos precisam ser do mesmo tipo.

A função `div` funciona com qualquer tipo que possa ser entendido como inteiro. Os operandos precisam ser do mesmo tipo também.

As seguintes funções de conversão de números podem ser úteis.

```
fromIntegral :: (Integral a, Num b) => a -> b
truncate :: (RealFrac a, Integral b) => a -> b
```

`round`, `ceil` e `floor` tem o mesmo tipo de `truncate`.

Casamento de Padrão

Ao definir funções podemos realizar casamento de padrão nos parâmetros para realizar o mapeamento (como na matemática). O símbolo _ é um *wildcard*.

```
conjuncao :: Bool -> Bool -> Bool
conjuncao False _ = False
conjuncao True  x = x

temAlgo :: Int -> Bool
temAlgo 0 = False
temAlgo _ = True
```

Casamento de Padrão

Também é possível casar o padrão no corpo da função, mas prefira o casamento nos parâmetros, quando possível.

```
conjuncao :: Bool -> Bool -> Bool
conjuncao x y =
  case x of
    False -> False
    True -> y
```

Guardas

Separar a definição em casos que envolvem testes booleanos.

```
ehPar :: Int -> String
ehPar x
  | x % 2 == 0 = "Sim"
  | otherwise  = "Não"
```

O que é equivalente a

```
ehPar :: Int -> String
ehPar x =
  if x % 2 == 0
    then "Sim"
    else "Não"
```

`ehPar` está no prelúdio como `even`

Guardas

Mais organizado que `ifs` aninhados.

```haskell
sinal :: Int -> String
sinal x
  | x > 0 = "Positivo"
  | x < 0 = "Negativo"
  | otherwise = "Neutro"

sinal :: Int -> String
sinal x =
  if x > 0
    then "Positivo"
    else
      if x < 0
        then "Negativo"
        else "Neutro"
```

Função anônima

Use \ para representar os λ de λ-cálculo e -> para representar o . que marca o início do corpo da função.

```
soma1 :: Int -> Int
soma1 = (\ x y -> x + y) 1
```

Currying

A seguinte forma é equivalente.

```
soma1 :: Int -> Int
soma1 = (\ x -> \ y -> x + y) 1
```

Recursão

Em Haskell não há `for`, `while` ou estrutura semelhante. Essas construções são do paradigma imperativo/estruturado. Para repetições, temos que usar recursividade. Não é preciso um operador de ponto-fixo, Haskell tem suporte nativo à recursão.

```
fatorial :: Int -> Int
fatorial 0 = 0
fatorial x = x * fatorial (x-1)
```

Teremos uma aula sobre recursão e outras formas de repetir computações em programação funcional.

Imutabilidade

Toda variável ou estrutura em Haskell é **imutável**. Esse é uma das grandes características da programação funcional.

Simplesmente não há como expressar uma mudança de valor. O máximo que conseguimos é **shadowing**.

```
let x = 4 in x + let x = 2 in x + 1
```

Algum grau de mutabilidade é necessário, veremos como fazer isso quando estudarmos mônadas.

Considere o exemplo em C da primeira aula:

```c
#include <stdio.h>
int z = 2;

int f(int x) {
  z++; return x + 1;
}

int main() {
  printf("%d\n", f(1));
  printf("%d\n", z);
}
```

Em Haskell não dá pra escrever uma função `f` como essa. A transparência referencial é **garantida**.

Não importa como `f` está implementado, temos a garantia de que ela não mudará o estado do programa.

```haskell
z :: Int
z = 2

f :: Int -> Int
f x = -- implementacao

main :: IO ()
main = do
  print (f 1)
  print z
```

É esse tipo de garantia que torna a programação funcional poderosa.

Juntando tudo

Calculadora de MMC e MDC

O máximo divisor comum (MDC) entre dois números pode ser calculado através do Algoritmo de Euclides.

Já o mínimo múltiplo comum (MMC) pode ser calculado pela seguinte fórmula:

$$mmc(x, y) = \frac{|x| * |y|}{mdc(x, y)}$$

Arquivo src/Lib.hs

```haskell
module Lib (mdc, mmc) where

euclides :: Int -> Int -> Int
euclides x 0 = x
euclides x y = euclides y (mod x y)

mdc :: Int -> Int -> Int
mdc x y = euclides (abs x) (abs y)

mmc :: Int -> Int -> Int
mmc x y = div ((abs x) * (abs y)) (mdc x y)
```

Arquivo app/Main.hs

```haskell
module Main (main) where

import Lib

main :: IO ()
main = do
  putStrLn "Digite o primeiro número:"
  x' <- getLine
  putStrLn "Digite o segundo número:"
  y' <- getLine
  let x = read x' :: Int
  let y = read y' :: Int
  putStrLn "O MMC é: "
  print (mmc x y)
  putStrLn "O MDC é: "
  print (mdc x y)
```

```
stack run
```

```
Digite o primeiro número:
12
Digite o segundo número:
30
O MMC é:
60
O MDC é:
6
```

stack ghci

```
ghci> mdc 12 30
6
ghci> mmc 12 30
60
ghci> :q
Leaving GHCi.
```

Efeitos Colaterais

Considere a assinatura das funções em C

```
int mdc(int x, int y) { ... }
int mmc(int x, int y) { ... }
```

Você tem garantia de que essas funções irão apenas retornar o inteiro, sem afetar qualquer outra parte do programa?

Efeitos Colaterais

E com Haskell?

```
mdc :: Int -> Int -> Int
...

mmc :: Int -> Int -> Int
...
```

Na programação funcional, você não precisa confiar no desenvolvedor. As regras da linguagem e o compilador garantem isso pra você.

Exercícios

No moodle.

Guia de boas práticas

- Complicado de definir
- Deixarei as minhas recomendações no Moodle.