Paradigmas de Programação

Paradigma Funcional com Haskell (Parte 1)

Introdução

Este material contém notas de aula sobre o *Paradigma Funcional*. O objetivo é apresentar os conceitos deste paradigma utilizando a linguagem de programação Haskell.

Paradigma Funcional

O paradigma funcional é também um paradigma declarativo. É um paradigma baseado em funções matemáticas e originário de um sistema formal matemático chamado *cálculo lambda*[1] (*cálculo-λ*).

FUNÇÃO

Relação matemática que mapeia elementos de um conjunto (domínio) em elementos de outro conjunto (contradomínio).

Definição	Aplicação	Explicação
$f(x) = x + x$	$f(3) = 3 + 3 = 6$	3 pertence ao domínio, 6 pertence ao contradomínio.
$g(x) = x + 1$	$g(5) = 5 + 1 = 6$	5 pertence ao domínio, 6 pertence ao contradomínio.

[1]Cálculo lambda (cálculo-λ) é uma coleção de sistemas formais baseados em uma notação criada por Alonzo Church (HINDLEY & SELDIN, 2008). Desenvolvida na década de 1930, corresponde a uma formalização matemática que expressa toda computação possível.

Neste paradigma, questões arquiteturais são abstraídas. Não há alocação explícita de memória nem declaração explícita de variáveis. Essas operações ocorrem automaticamente quando a função é invocada.

Na programação funcional, não há efeitos colaterais no cálculo da função. Diz-se que os programas são legíveis e confiáveis.

Uma função matemática sempre define o mesmo valor para os mesmos argumentos. Se $f(x, y) = 19$ na primeira execução da função, então o resultado será 19 sempre. Isso é chamado de *transparência referencial*.

Em linguagens funcionais não há variáveis com as mesmas prerrogativas das variáveis de linguagens de outros paradigmas, que permitem sua modificação após a atribuição. Em vez disso, as linguagens funcionais dispõem de uma abstração por vezes chamada de *variável imutável* (*vínculo imutável* ou *binding imutável*), que, como o próprio nome sugere, não permite a alteração de valores após sua definição. A maioria das linguagens funcionais enfatiza a imutabilidade, mas há exceções assim, tais como ML, que tem matrizes mutáveis (SEBESTA, 2012).

A imutabilidade e a transparência referencial favorecem o processo de teste, já que um resultado obtido no ambiente de desenvolvimento será o mesmo obtido em produção. Os arquivos `transparencia-referencial.hs` e `transparencia-referencial.c` mostram como funções com os mesmos parâmetros podem retornar resultados diferentes em C, ao passo que em Haskell ela se comporta de maneira igual.

No paradigma funcional, *loops* são possíveis através da recursividade.

Cada vez mais linguagens de programação estão incorporando suporte a recursos da programação funcional (SEBESTA, 2012), demonstrando o crescente reconhecimento da importância desse paradigma na construção de software moderno.

Exemplos de linguagens funcionais:

- Clojure

- Elixir
- Erlang
- F#
- Haskell
- LISP (1ª linguagem funcional)
- ML
- Miranda
- Scheme

Exemplos de linguagens com suporte à programação funcional:

- C++
- C#
- Java
- JavaScript
- Python
- Ruby

Aplicações

O paradigma funcional é visto cada vez mais como uma alternativa em aplicações reais. Entre essas aplicações, podem ser citados:

- Sistemas financeiros e bancários (CARBONE, 2023; MAGALHÃES, 2023)
- Ensino (de matemática, por exemplo)
- Sistemas embarcados
- Matemáticos
- CTFs (RUEF *et al*., 2016)

A Linguagem Haskell

Haskell foi criada em 1990. É uma linguagem parecida com ML, com o diferencial de permitir sobrecarga de funções[2]. É usada em softwares de grandes empresas como Standard Chartered (MAGALHÃES, 2023), ABN AMRO Bank (SCHMIDT; QUI, 2007), AT&T, Bank of America, NVIDIA entre outras que podem ser consultadas na página "*Haskell in industry*".
A ideia principal da linguagem é baseada na avaliação de expressões. A implementação da linguagem avalia (simplifica) a expressão passada pelo programador até sua forma normal (DU BOIS, s.d.).

Tipos

Haskell é fortemente e estaticamente tipada. Os tipos em Haskell são inferidos quando não informados;

Em Haskell, os tipos são separados em *classes de tipo*. Classes de tipo são como interfaces em linguagens de programação orientadas a objetos, fornecendo um conjunto de operações ou comportamentos que um tipo deve suportar. Um tipo pode ser uma instância de uma ou mais classes de tipo se ele implementar todas as operações especificadas pela classe. Por exemplo, a classe de tipo `Eq` em Haskell define operações para testar igualdade entre valores. Para um tipo ser uma instância da classe `Eq`, ele deve implementar a função `==` (igualdade) e `/=` (diferença).

Assim, pode-se dizer que tipos como `Int`, `Float`, `Char`, entre outros, são instâncias da classe `Eq`. Da mesma forma, a classe de tipo `Ord` define operações para comparação de ordem entre valores. Tipos que são instâncias da classe `Ord` devem implementar funções como `<`, `<=`, `>`, `>=`, além de serem instâncias da classe `Eq`. A hierarquia das classes do módulo `Prelude` está disponível na Figura 1 (HASKELL, 2020).

Alguns dos tipos suportados por Haskell e presentes na Figura 1 são:

[2]Conceito de programação que permite que duas ou mais funções tenham o mesmo nome, mas diferentes assinaturas (tipos e/ou quantidade de parâmetros).

- **`Bool`**: tipo lógico que avalia expressões para dois valores possíveis, `True` e `False`.
- **`Char`**: tipo que representa caracteres Unicode.
- **`Int`**: tipo numérico finito que representa inteiros com tamanho limitado pela arquitetura do sistema.
- **`Integer`**: tipo numérico arbitrariamente grande que representa inteiros sem limite de tamanho.
- **`Double`**: tipo numérico que representa números de ponto flutuante de precisão dupla de acordo com o padrão IEEE 754.

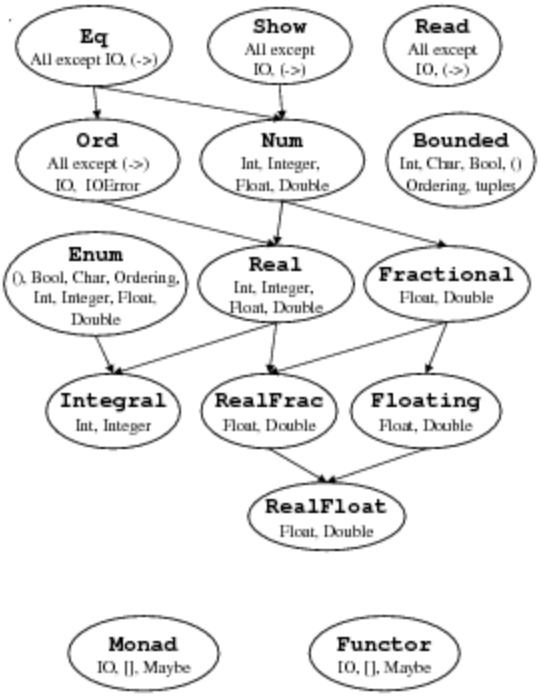

Figura 1. Hierarquia de classes de tipo em Haskell definidas no módulo `Prelude`. Fonte: Haskell, 2024.

Em Haskell, funções também são tratadas como valores comuns. Dessa forma, podem ser

- passadas como argumentos para outras funções;
- retornadas como resultados de funções; e
- armazenadas em estruturas de dados.

Quando uma função permite uma dessas ações, diz-se que ela é uma função de alta ordem ou de ordem superior. Quando uma função possui parâmetros e resultados que não são funções, ela é dita ser de primeira ordem.

Ferramentas

As principais ferramentas para uso da linguagem Haskell são o *Glasgow Haskell Compiler* – GHC e o *Glasgow Haskell Compiler Interactive* – GHCi. Essas ferramentas estão disponíveis para alguns dos principais sistemas operacionais, tais como Linux, Windows, Mac OS X e FreeBSD. Em ambientes Linux, as ferramentas GHC e GHCi podem ser acessadas pelo terminal, respectivamente, digitando `ghc` e `ghci`.

Outras ferramentas que podem ser utilizadas:

- Hugs, não recomendado por não ser atualizada há décadas.
- Replit, alternativa *online* para programação com Haskell.
- Try Haskell, tutorial *online* para programação com Haskell.

Ao executar o GHCi, a expressão "`Prelude>`" será exibida. `Prelude` é o módulo padrão do Haskell. Ele é carregado automaticamente ao iniciar o programa.

Funcionamento

Existem algumas formas para utilizar a linguagem Haskell. Algumas delas são:

1. Utilizar diretamente o terminal interativo do GHCI para fazer os testes.
    - Neste caso, basta digitar `ghci` no terminal e iniciar os testes.
2. Criar um arquivo de extensão `.hs` e carregá-lo na ferramenta GHCi.
    - Neste caso, basta digitar os comandos abaixo e iniciar os testes:
        - `ghci`
        - `:l seu-arquivo.hs`
    - Todas as funções definidas no arquivo `seu-arquivo.hs` são carregadas e podem ser utilizadas.
3. Criar um arquivo de extensão `.hs`, compilá-lo com o GHC e executá-lo.

Os exemplos presentes neste material consideram, em especial, as formas 1 e 2 mencionadas.

Sintaxe

A sintaxe em Haskell é projetada para ser limpa, concisa e expressiva, facilitando a escrita e compreensão do código. Assim como na maioria das linguagens, Haskell é *case sensitive*, ou seja, diferencia letras maiúsculas e minúsculas. Dessa forma, `teste` (começando com "t" minúsculo) e Teste (começando com "T" maiúsculo) seriam identificadores diferentes.

Em Haskell, funções são definidas usando o seguinte padrão:

```
nomeDaFuncao argumentos = expressão
```

Um exemplo válido seria, portanto:

```
f x = x * x
```

Para chamar a função, bastaria definir um valor de *x* (ex: 8) e digitar a função com o valor escolhido (exemplo: `f 8`). Cabe citar também que a função pode ter de zero a muitos argumentos. Caso ela tenha mais de um argumento, eles são separados apenas por espaço em branco.

Para funções que se deseja definir um tipo, usa-se `::` (*dois pontos duplo*) antes de sua definição. Assim, os exemplos do quadro abaixo são válidos:

```
valor :: Int
valor = 32
num :: Integer
num = 9
f :: Int -> Int
f x = x * x
```

No último exemplo, o tipo `Int -> Int` indica que é avaliado um valor do tipo `Int` e, por fim, retorna-se um valor `Int`.

Outro ponto da sintaxe de Haskell é que funções podem ser chamadas nas formas prefixadas e infixadas. Os exemplos do quadro a seguir são todos válidos em Haskell:

```
(+) 9 8
9 + 8
mod 7 3
7 `mod` 3
```

A linguagem também permite a inserção de comentários em seus arquivos. Um comentário de uma linha é precedido por `--` (*dois traços consecutivos*), e comentários de múltiplas linhas são delimitados no início por `{-` (*abre chaves e traço*) e por `-}` (*traço e fecha chaves*) no fim.

Listas em Haskell são delimitadas por colchetes, e seus elementos são separados por vírgula. Assim, `[1, 2, 3]` é uma lista válida. De forma semelhante, elementos de uma tupla também são separados por vírgula, mas a tupla é delimitada por parênteses. Assim, `(1, 2, 3)` é uma tupla válida.

Assim como na maioria das linguagens, o tipo de caractere em Haskell, `Char`, delimita seu valor colocando-o entre aspas simples. Assim, `'A'` é do tipo `Char`. Uma cadeia de caracteres em Haskell pode ser escrita de duas formas:

- como uma lista de `Char`, como no exemplo a seguir:
    - `['I','F','C']`
- como uma String, delimitando o texto com aspas duplas, como no exemplo a seguir:
    - `"IFC"`

Observação: lembre-se de que os caracteres de aspas simples e duplas podem ser representados de formas diferentes e, assim, pode ser necessário inseri-los manualmente para conseguir reproduzir os exemplos.

Mais detalhes da sintaxe serão abordados nos materiais seguintes.

Operadores

Os principais operadores em Haskell são:

- Aritméticos
    - Adição (`+`)

    - Subtração (`-`)
    - Multiplicação (`*`)
    - Divisão (`/`)
    - Divisão inteira (`div`)
    - Resto da divisão inteira (`mod`)
    - Exponenciação inteira (`^`)
    - Exponenciação de ponto flutuante (`**`)
    - Raíz quadrada (`sqrt`)
- Relacionais
    - Menor que (`<`)
    - Maior que (`>`)
    - Menor ou igual a (`<=`)
    - Maior ou igual a (`>=`)
    - Igual a (`==`)
    - Diferente de (`/=`)
- Lógicos
    - E lógico (`&&`)
    - OU lógico (`||`)
    - Negação lógica (`not`)
- De função
    - Composição (`.`)
    - Aplicação (`$`)

Existem ainda os operadores de listas:

- Operador "`:`" (*dois pontos*), serve para construção de lista; e
- Operador "`++`" (*mais mais*), serve para concatenação de listas

Ambos serão explicados na segunda parte do material.

Exemplos

Esta seção mostra exemplos para reprodução na ferramenta GHCi. Há comentários entre os códigos contendo explicações e perguntas sobre os comandos.

```
2 > 1 && 1 < 2
1 > 30 || 30 > 29
not(mod 4 3 == 0)
not $ mod 4 3 == 0
"Aula"
"Oi" + True -- Erro!
sqrt(9^2)
sqrt(9)
2 /= 4-2
2 == 4-2
['A', 'u', 'l', 'a'] -- Os colchetes representam listas.
['A', 'u', 'l', 'a'] == "Aula"
:t "Oi" -- O comando :t retorna o tipo da expressão!
x = 9
sqrt(x)
:t sqrt(x)
y = 9 :: Float
sqrt y
:t sqrt y
:t 2 == 2
:t 2 + 4
:t 2.0 + 4
reverse "Seu nome"
:t reverse "Seu nome"
:t [3, 3.9]
sort [5, 2, 3, 19, 1] -- Não vai funcionar por padrão, pois sort faz
parte do módulo Data.List!
import Data.List
sort [5, 2, 3, 19, 1] -- Agora sim!
Data.List.sort [5, 2, 3, 19, 1] -- Outra forma válida!
sort "Seu nome" -- O que aconteceu aqui?
"Ricardo" ++ " de la Rocha"
```

```
:q -- Comando para encerrar o GHCi
:t (+)
:t (==)
-- O sinal de =>, presente no retorno dos dois comandos acima, indica uma restrição de classe.
-- O retorno do último comando pode ser lido da seguinte maneira: a função igual (==) recebe dois valores do tipo "a" e retorna um "Bool". O tipo desses dois valores deve ser um membro da classe Eq (essa era a restrição da classe).
-- No penúltimo comando, a função soma (+) recebe dois valores do tipo "a" e retorna um valor do tipo "a". O tipo desses valores deve ser um membro da classe Num (restrição da classe).
:t (9, 'a')
(+) 9 8 -- Usando um operador como função!
10 `mod` 4 -- Usando uma função como operador! A isto damos o nome de notação infixa.
(<) 'Z' 'z'
```

Criando Funções

A criação de funções é uma parte essencial na Programação Funcional. Para criar uma função, é possível fazê-lo de diferentes maneiras:

1. Diretamente do terminal interativo (GHCi).
2. Em um arquivo de texto simples, de extensão `.hs`, a ser carregado.

Em relação à sintaxe, a criação de funções pode ser feita:

- Utilizando a palavra reservada `let` (será explicada em material futuro)
- Criando uma função lambda (conteúdo não coberto nesta disciplina, mas sim em Programação Lógica Funcional)
- Utilizando a sintaxe de função citada na Seção Sintaxe
- Utilizando casamento de padrões (será explicada em material futuro)
- Utilizando a palavra-chave `where` (conteúdo não coberto nesta disciplina)

Para exemplificar as formas de criação de função, será criada uma função que soma três valores inteiros, do tipo `Int`. Considerando a criação de funções diretamente do terminal interativo, essa função poderia ser criada e testada das seguintes maneiras:

- Com `let`:

```
let somaTres x y z = x + y + z
somaTres 3 6 9
```

- Criando uma função lambda (as duas formas abaixo são equivalentes):

```
(\x y z -> x + y + z) 3 6 9
(\x -> \y -> \z -> x + y + z) 3 6 9
```

Porém, ao digitar `:t somaTres`, nota-se que o tipo retornado é:
`Num a => a -> a -> a -> a`.

Haskell inferiu a classe de tipo, pois um tipo não foi explicitamente informado. Da mesma forma, ao digitar `:t (\x y z -> x + y + z) 3 6 9`, o retorno obtido é:

```
(\x y z -> x + y + z) 3 6 9 :: Num a => a
```

Os dois casos podem ser resolvidos definindo explicitamente o tipo.

- Com `let`:

```
let somaTres :: Int -> Int -> Int -> Int; somaTres x y z = x + y + z
```

- Criando uma função lambda:

```
(\x y z -> x + y + z) 3 6 9 :: Int
```

Assim, testando o tipo com o comando `:t`, o tipo retornado passa a ser `Int`.

Considerando a criação de função em um arquivo de texto simples, basta criar o arquivo e salvá-lo com a extensão `.hs` (exemplo: `exemplo1.hs`), como mostra o quadro a seguir:

```
somaTres :: Int -> Int -> Int -> Int -- Esta linha não é obrigatória.
-- somaTres :: Integer -> Integer -> Integer -> Integer -- Esta linha
pode substituir a primeira caso o tipo Integer seja desejado.
somaTres x y z = x + y + z
```

A partir do diretório em que `exemplo1.hs` foi salvo, basta abrir o terminal e executar o GHCi digitando `ghci`. Quando o módulo `Prelude` estiver carregado, basta digitar:

```
:l exemplo1.hs
```

O comando deverá retornar algo semelhante a isto:

```
[1 of 1] Compiling Main             ( exemplo1.hs, interpreted )
Ok, modules loaded: Main.
```

Se algum erro for identificado o retorno será diferente, contendo a descrição do erro e, ao final, a frase "`Failed, modules loaded: none.`".

Após o carregamento do arquivo, todas as funções nele definidas poderão ser usadas. No exemplo, pode-se testar a função com o comando abaixo:

```
somaTres 1 3 4
```

A função `somaTres` também poderia ser gerada diretamente a partir do terminal do GHCi. Isso pode ser feito em uma linha ou em múltiplas linhas, como mostra o quadro a seguir:

```
-- Em uma linha
somaTres :: Int -> Int -> Int -> Int; somaTres x y z = x + y + z
-- Em múltiplas linhas
:{
somaTres :: Int -> Int -> Int -> Int
somaTres x y z = x + y + z
:}
```

Uma vantagem de definir funções em arquivos `.hs` é a reusabilidade. Desta forma, basta carregar o arquivo sempre que o GHCi for iniciado. Funções definidas diretamente no terminal interativo possuem menos reusabilidade. Toda função que não for anônima fica disponível enquanto a seção estiver aberta, ou seja, toda vez que a ferramenta for reiniciada é necessário redefinir a função. Como função anônima (lambda), a função não é reusável.

O que fazem as duas linhas de código abaixo?

```
somaDois :: Double -> Double -> Double
```

```
somaDois x y = x + y
```

Sugere-se acrescentá-las ao arquivo `exemplo1.hs` e executá-las. Após atualizar o arquivo, basta digitar:

```
:l exemplo1.hs
somaDois 3 4
```

A função a seguir deve ser acrescentada ao mesmo arquivo. Ela não tem um tipo explicitamente definido. De que tipo ela é? Como é possível saber o tipo dessa função?

```
novoSomaDois x y = x + y
```

Qual é a diferença entre os comandos abaixo?

```
:t novoSomaDois
:t novoSomaDois 1 5
```

A função a seguir também deve ser acrescentada no arquivo exemplo1.hs:

```
f = (+) 1
```

O que faz esta função? Como ela pode ser utilizada? Qual será o retorno dela?

O operador de soma, representado pelo símbolo "`+`", foi utilizado de forma prefixada. A função `f` acrescentará o valor um ao parâmetro informado. Assim, a função pode ser utilizada da seguinte maneira:

```
f 9
```

O retorno será sempre o parâmetro acrescido de um.

Outra forma de criar uma função de soma é através de uma tupla. No exemplo a seguir, a ser inserido no arquivo `exemplo1.hs`, foi criada uma tupla com dois valores (também chamada de *par ordenado*):

```
soma (a,b) = a + b
```

Após salvar o arquivo e recarregá-lo no GHCi, basta chamar a função. Exemplo:

```
soma (8, 9)
```

O arquivo `exemplo1.hs` contém as funções definidas nesta seção.

Listas

Os exemplos a seguir são válidos para listas em Haskell e podem ser digitados diretamente no `ghci`. O tema será abordado com maior profundidade em materiais futuros.

```
[1 + 23]
[1 + 23, 8]
"IFC"
```

Tuplas

Uma tupla é uma agregação de vários componentes (DU BOIS, s.d.), ou uma sequência de elementos que podem ser de tipos variados. Os exemplos a seguir são válidos para tuplas em Haskell e podem ser digitados diretamente no `ghci`.

```
(3, 4)
(3, 4 * 3)
(1077888, "Ricardo")
(3 > 3, 4 + 4, True, 90.0)
```

Avaliação Preguiçosa (*Lazy Evaluation*)

Segundo Du Bois (s.d.), a avaliação preguiçosa é um recurso presente em linguagens funcionais puras como Haskell. Esse recurso faz com que os argumentos das funções sejam avaliados somente quando necessário.

Exemplo:

$f(x) = 20$

$f(-2 + 21\text{^}2/20 - 5/100) = 20$

Em Haskell isso poderia ser feito da seguinte forma:

```
f x = 20
```

Neste caso, independentemente do valor de *x*, o resultado será sempre 20. Portanto, avaliar toda expressão não é necessário. Em linguagens imperativas, tais como C e Pascal, a expressão será completamente avaliada antes da chamada da função.

Desvantagens

Entre as desvantagens da linguagem Haskell, podem ser citadas:

- O custo da recursão. Embora a recursão seja uma técnica poderosa em Haskell, ela pode ter um custo significativo em termos de desempenho e uso de memória. Em Haskell, as chamadas recursivas não são otimizadas automaticamente pelo compilador, o que significa que podem consumir mais tempo e memória em comparação com linguagens imperativas. Além disso, o uso excessivo de recursão em Haskell pode levar a estouro de pilha (*stack overflow*) em alguns casos.
- Ainda há poucos programadores. Embora a comunidade Haskell seja ativa e crescente, ela ainda é consideravelmente menor do que a de linguagens como C, C++, JavaScript e Python. É mais difícil encontrar desenvolvedores com experiência em Haskell para trabalhar em projetos específicos. Além disso, a escassez de programadores pode resultar em menos recursos disponíveis (vídeos, apostilas, *slides* etc.) e menos suporte para resolução de problemas.
- Dificuldade em prever os custos de tempo e memória em programas com avaliação preguiçosa. Como as expressões são avaliadas apenas quando necessário, é difícil determinar quando exatamente uma expressão será avaliada e quanto tempo ou memória ela consumirá. Isso pode levar a situações em que o desempenho do programa não corresponde às expectativas do desenvolvedor.

Observações

Caso alguma ferramenta diferente do GHCi seja utilizada, um aviso sobre a falta da função principal (`main`) pode ser exibido, embora os exemplos presentes no material estejam corretos. Para eliminar o aviso, basta definir, no início do arquivo principal (normalmente intitulado `main.hs` por padrão), uma função `main` sem retorno, da seguinte forma:

```
main = return ()
```

Ao reproduzir os exemplos em ferramentas não interativas, não será possível reproduzir testes interativos. Por exemplo, `:t 2+2` não funcionará em ferramentas não interativas. Uma forma válida e próxima disso poderia ser obtida com os comandos do quadro a seguir:

```
import Data.Typeable
fType :: Int
fType = 2 + 2
main = print (typeOf (fType))
```

Conclusão

O Paradigma Funcional, exemplificado através da linguagem Haskell neste material, oferece uma abordagem poderosa e elegante para a construção de software. Ao basear-se em conceitos de funções matemáticas e no cálculo lambda, o Paradigma Funcional promove a Programação Declarativa com foco na definição de funções e na avaliação de expressões.

Uma das características marcantes do Paradigma Funcional é a imutabilidade, garantindo que uma função sempre produza o mesmo resultado para os mesmos argumentos, o que contribui para a confiabilidade e previsibilidade do código.

A linguagem Haskell, uma das principais representantes do Paradigma Funcional, destaca-se por sua tipagem forte e estática, bem como pelo sistema de tipos baseado em classes. A hierarquia de classes de tipo em Haskell permite uma abordagem flexível e elegante para lidar com diferentes tipos de dados, facilitando a definição e a manipulação de estruturas complexas. Além disso, Haskell incentiva a programação de alto nível, abstraindo questões de gerenciamento de memória, tornando o código mais legível e menos propenso a erros.

Embora o Paradigma Funcional ofereça muitas vantagens, como a ausência de efeitos colaterais e a citada ênfase na imutabilidade, apresenta como desvantagens o custo potencialmente elevado da recursão e a dificuldade em prever os custos de tempo e memória em programas com avaliação preguiçosa. Além disso, a escassez de programadores experientes em Haskell e a falta de recursos disponíveis podem representar obstáculos para a adoção mais ampla desse paradigma. No entanto, o crescente reconhecimento da importância

da programação funcional na construção de software moderno tem levado cada vez mais linguagens de programação a incorporar recursos da Programação Funcional.

Outros elementos da linguagem Haskell e do Paradigma Funcional serão abordados na sequência da disciplina.

Exercícios

1) Aproveitando as funções definidas em `exemplo1.hs`, crie a função `novoSomaTres`, em termos de `novoSomaDois`, ou seja, utilizando `novoSomaDois`. **DESAFIO**: faça o exercício utilizando o operador de aplicação (`$`).

2) Qual é a diferença entre os operadores `^` e `**`?

3) (POSCOMP 2015) A linguagem de programação LISP usa o paradigma de:
   a) programação procedural.
   b) programação de tipos abstratos de dados.
   c) programação orientada a objetos.
   d) programação funcional.
   e) programação declarativa

4) (adaptada de POSCOMP 2011) Relacione corretamente um Paradigma de Linguagem de Programação, citado na primeira coluna da tabela abaixo, com uma Linguagem de Programação, citada na segunda coluna da tabela abaixo:

Paradigmas	**Linguagens**
Imperativo	Scheme
Orientado a Objetos	Smalltalk
Funcional	Pascal
Lógico	Prolog

5) (POSCOMP 2009) Considere as afirmativas abaixo:

I. Fortran, Pascal e Java são linguagens de terceira geração.
II. II. C++ e Java permitem a criação de classes e o uso de herança múltipla.
III. III. Prolog é uma linguagem funcional pura.
IV. IV. PHP, Perl e Ruby são linguagens de sexta geração.

Assinale a alternativa CORRETA:

a) Apenas a afirmativa I é verdadeira.
b) Apenas a afirmativa II é verdadeira.
c) Apenas a afirmativa III é verdadeira.
d) Apenas as afirmativas I e IV são verdadeiras.
e) Apenas as afirmativas II e III são verdadeiras.

6) Resolva os exercícios da Lista #5.

Referências

CARBONE, Heloisa. Programação funcional com Clojure: por que e como o Nubank usa e escala tão bem? 28 mar. 2023. Disponível em: https://building.nubank.com.br/pt-br/programacao-funcional-com-clojure-por-que-e-como-o-nubank-usa-e-escala-tao-bem/. Acesso em: 12 mar. 2024.

DU BOIS, André Rauber. Programação Funcional com a Linguagem Haskell. Disponível em: http://www.inf.ufpr.br/andrey/ci062/ProgramacaoHaskell.pdf. Acesso em: 18 mar. 2024.

HASKELL. Predefined Types and Classes. Disponível em: https://www.haskell.org/onlinereport/haskell2010/haskellch6.html. Acesso em 17 mar. 2024.

HINDLEY, J. Roger; SELDIN, Jonathan P. Lambda-calculus and Combinators, an Introduction. Cambridge: Cambridge University Press, 2008.

MAGALHÃES, José Pedro. Haskell jobs at Standard Chartered - various locations and seniority. 1. Apr. 2023. Disponível em: https://discourse.haskell.org/t/haskell-jobs-at-standard-chartered-various-locations-and-seniority/6157. Acesso em: 12 mar. 2024.

RUEF, Andrew et al. Build it, break it, fix it: Contesting secure development. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. 2016. p. 690-703. Disponível em: https://arxiv.org/pdf/1606.01881.pdf. Acesso em: 17 mar. 2024.

SEBESTA, Robert W. Concepts of Programming Languages. Pearson. 10th ed. 2012.

SCHMIDT, Cyril; QUI, Anne-Elisabeth Tran. The default case in Haskell: counterparty credit risk calculation at ABN AMRO. In: Proceedings of the 4th ACM SIGPLAN workshop on Commercial users of functional programming. 2007. p. 1-2.