GHC.IO About Help

```
Welcome to GHC.IO!
Prelude> let fatorial a = product [1..a]
Prelude> fatorial 3
6
Prelude>
```

sumList [2,3,4,5]
= 2 + sumList [3,4,5]
= 2 + (3 + sumList [4,5])
= 2 + (3 + (4 + sumList [5]))
= 2 + (3 + (4 + (5 + sumList [])))
= 2 + (3 + (4 + (5 + 0)))
= 14

Programação Funcional

Modularização

Matheus Carvalho Viana
matheuscviana@ufsj.edu.br

Dcomp
Departamento de Ciência da Computação
UFSJ
Universidade Federal de São João del-Rei

Introdução

Os programas feitos até agora têm todo o código em um único arquivo.

Para programas pequenos, isso não é um problema, mas para programas grandes, dividir o código em mais de um arquivo tem vantagens:

- Cada arquivo pode ser implementado, compilado e testado separadamente;
- Se o código é modificado, só é necessário recompilar o arquivo que foi modificado, e não todo o programa;
- O código fica mais organizado, com cada arquivo sendo responsável por uma funcionalidade do sistema;
- Os arquivos podem ser reutilizados em outros programas que possuem a mesma funcionalidade.

Modularização

É a capacidade de organizar um programa em vários arquivos.

Em cada **módulo** (arquivo) é implementada uma funcionalidade do sistema.

As bibliotecas fornecidas pelas linguagens de programação são implementadas em módulos.

Todo programa possui um **módulo principal** e zero ou mais **módulos auxiliares**.

Em Haskell, o módulo principal é aquele que contém a função **main**.

Modularização

A modularização deve seguir algumas regras básicas, caso contrário, em vez de facilitar, ela pode atrapalhar:

- Cada módulo deve ter um papel claramente definido;
- Cada módulo deve realizar exatamente uma única tarefa;
- Cada módulo deve ser autocontido;
- Deve haver nenhuma ou pouca dependência entre os módulos;
- Os módulos devem ser pequenos.

Modularização

Um módulo deve começar com a seguinte declaração:

```
module <NomeDoModulo> where
```

Em Haskell, o nome do módulo deve ser o mesmo do arquivo, ambos iniciados com letra maiúscula.

Após a palavra reservada **`where`** inicia-se o corpo do módulo, onde são definidos os tipos e funções do módulo.

Exportação

Um módulo pode exportar tipos e funções para outros módulos.

Na notação anterior, todo o conteúdo do módulo pode ser exportado.

Contudo, o ideal é que o módulo permita exportar somente o que for necessário. Nesse caso a sintaxe de declaração do módulo fica:

```
module <NomeDoModulo> (
 TipoExportado1(..),
 ...
 TipoExportadoN(..),
 funçãoExportada1,
 ...
 funçãoExportadaZ
) where
```

Os tipos e funções indicadas na declaração do módulo são os que podem ser exportados.

A notação especial `(..)` que segue a declaração dos tipos indica que o tanto o tipo quanto o construtor do tipo podem ser exportados.

Exportação

Por exemplo, o módulo Pontos permite que sejam exportados somente o tipo **`Ponto`** e a função **`distância`**.

```
module Pontos ( Ponto(..), distancia) where

data Ponto = Pt Float Float deriving (Show,Read,Eq)

distancia :: Ponto -> Ponto -> Float
distancia a b = sqrt ( (subX2 a b) + (subY2 a b) )

subX2 :: Ponto -> Ponto -> Float
subX2 (Pt xa _) (Pt xb _) = (xb-xa)^2

subY2 :: Ponto -> Ponto -> Float
subY2 (Pt _ ya) (Pt _ yb) = (yb-ya)^2
```

Importação

Para importar todo o conteúdo exportável de um módulo, usa-se o comando **`import`** seguido do nome do módulo.

```
import Data.Char
```

Para importar somente uma função ou tipo, basta indicar o nome dela entre parênteses após o comando `import`.

Por exemplo, para importar somente a função `ord` de `Data.Char`, escreve-se:

```
import Data.Char (ord)
```

Importação

Também é possível importar toda a biblioteca, exceto algumas funções ou tipos. Para isso usa-se o comando **hiding** junto ao import..

```
Prelude> import Data.Char hiding (chr)
Prelude Data.Char> ord 'a'
97
Prelude Data.Char> chr 97

<interactive>:12:1: Not in scope: ‘chr’
Prelude Data.Char>
```

Todo o conteúdo exportável de Data.Char foi importado, exceto a função chr.

A função chr não foi importada, assim o GHCi não a reconheceu.

Importação e Exportação

Em um módulo, a palavra **`import`** é usada após a palavra **`where`**.

Por padrão, um módulo exporta somente o seu próprio conteúdo sem exportar o conteúdo de módulos importados por ele.

Mas é possível indicar que um módulo exporta o conteúdo de outro módulo indicando o módulo a ser exportado na lista de exportações.

```
module A ( module A, module B) where
import B
...<conteúdo do módulo A>...
```

No exemplo acima, o módulo A exporta todo o seu conteúdo e o conteúdo exportável do módulo B.

Hierarquia de Módulos

Os módulos podem seguir uma hierarquia que permite agrupá-los.

Por exemplo, os módulos **`Sphere`** e **`Cube`** podem ser considerados submódulos do módulo **`Geometry`**.

Para isso, basta colocar os submódulos em uma pasta e usar o nome dessa pasta antes do nome do submódulo separado por ponto (.).

```
module Geometry.Sphere (area) where
```

```
module Geometry.Cube (area) where
```

Os arquivos Sphere.hs e Cube.hs devem ficar dentro da pasta Geometry.

Namespace

Quando um modulo necessita importar mais de um módulo, podem ocorrer conflitos de nomes.

Por exemplo, considere os módulos **`Sphere`** e **`Cube`**. Ambos possuem uma função chamada **`area`**. Se ambos os módulos forem importados, haverá um conflito de nome.

```
module Geometry.Sphere (area) where
area :: Float -> Float
area radius = 4 * pi * (radius ^ 2)

module Geometry.Cube (area) where
area :: Float -> Float
area side = Cuboid.area side side side
```

Namespace

Nesses casos, é necessário definir um **namespace** para que o compilador saiba diferenciar de onde vem cada função.

Para isso, usam-se as palavras reservadas **`qualified`** e **`as`**.

```
import qualified Geometry.Sphere as Sphere
import qualified Geometry.Cube as Cube

main = do
  putStr "\nDigite um float: "
  ld <- getLine
  putStrLn $ "Área do cubo: " ++ show (Cube.area (read ld))
  putStrLn $ "Área da esfera: " ++ show (Sphere.area (read ld))
```

Compilação de Módulo

Para compilar um módulo, usa-se o atributo -c para indicar que não deve ser gerado um arquivo executável.

```
ghc -c NomeDoArq.hs
```

Esse comando irá gerar dois arquivos:

`NomeDoArq.o` - contém o código de máquina;

`NomeDoArq.hi` - arquivo de interface onde o GHC armazena informações sobre os nomes exportados pelo módulo.

Módulo Main

Um programa completo deve possuir um módulo chamado **`Main`**, implementado no arquivo **`Main.hs`** e que contém a função **`main`**.

Por exemplo, um módulo Main que importa o módulo Pontos:

```
module Main() where
import Pontos
main = do
  putStrLn “Digite dois pontos:”
  a <- getLine
  b <- getLine
  print (distancia (read a) (read b))
```

Módulo Main

Após ter compilado todos os módulos, inclusive o módulo Main, com a diretiva **-c**, pode-se criar o arquivo executável.

Para isso, use a diretiva **-o** e indique o nome do executável seguido pelo nome dos módulos com extensão **.o**.

Por exemplo:

```
ghc -o ponto Main.o Pontos.o
```

Exercícios

1. Crie o módulo Calculadora com as funções polimórficas soma, subtração, multiplicação e divisão. Crie o módulo Main e crie uma função que solicita os números do usuário e usa as funções do módulo Calculadora.

2. Crie o módulo Formas, que importa o módulo Pontos e define o tipo Forma, que pode ser um Círculo com um ponto e um raio, ou um Retângulo com dois pontos. Crie também a função contido que verifica se um ponto está contido em uma forma.