# Linguagens de Programação

## Tipos de Dados e Condicionais

Samuel da Silva Feitosa

Aula 7

# Modelo de Dados

# Comandos Úteis do GHCi

| **comando** | abrev | **significado** |
|---|---|---|
| `:load` *name* | `:l` | carrega o programa fonte *name* |
| `:reload` | `:r` | recarrega o programa fonte atual |
| `:edit` *name* | `:e` | edita o programa fonte *name* |
| `:edit` | `:e` | edita o programa fonte atual |
| `:type` *expr* | `:t` | mostra o tipo de *expr* |
| `:info` *name* | `:i` | dá informações sobre *name* |
| `:browse` *Name* | | dá informações sobre o módulo *Name*, se ele estiver carregado |
| `let` *id* `=` *exp* | | associa a variável *id* ao valor da expressão *exp* |
| `:!` *comando* | | executa *comando* do sistema |
| `:help` | `:h, :?` | lista completa dos comandos do GHCi |
| `:quit` | `:q` | termina o GHCi |

# Principais Conceitos

- Aprenderemos os ingredientes básicos para utilizar valores de determinados tipos, criar nossos próprios tipos e definir funções para manipular nossos dados.
- Você já aprendeu a trabalhar com números, e agora vai adicionar outros tipos de dados ao seu conhecimento, além do uso de tuplas e listas.

# Caracteres, Números e Listas

- Caracteres e números são os tipos de dados mais básicos que uma linguagem pode ter.
- A partir destes dados, podemos criar outros mais complexos, ou utilizar mecanismos para criar *strings* ou listas.

# Caracteres

- Haskell possui um tipo chamado **Char** para representar caracteres.
  - Para criar um valor do tipo **Char**, geralmente utilizamos aspas simples, como por exemplo: ‘a’.
- Usando o GHCi, podemos verificar o tipo de uma expressão usando o comando **:t**.

```
Prelude> :t 'a'
'a' :: Char
```

# Caracteres

- Haskell possui funcionalidades pré-definidas para **Char**.

```
Prelude> import Data.Char
Prelude Data.Char> toUpper 'a'
'A'
```

- Podemos verificar o tipo de uma função:

```
Prelude Data.Char> :t toUpper
toUpper :: Char -> Char
```

- Esta seta -> especifica o tipo de funções.
  - **toUpper** é uma função que recebe um caracter e retorna outro caracter.

# Caracteres

- As funções podem ter tipos de entrada e saída diferentes:
  - A função **chr** recebe um **Int** e retorna um **Char**.

```
Prelude Data.Char> chr 97
'a'
Prelude Data.Char> :t chr
chr :: Int -> Char
```

- Podemos testar o sistema de tipos do Haskell.

```
Prelude Data.Char> chr 'a'
<interactive>:16:5: error:
    • Couldn't match expected type ‘Int’ with actual type ‘Char’
    • In the first argument of ‘chr’, namely ‘'a'’
      In the expression: chr 'a'
      In an equation for ‘it’: it = chr 'a'
```

# Números

- Já utilizamos alguns números na aula passada.
- A linguagem Haskell possui suporte a uma grande variedade de tipos numéricos:
  - **Int** é do tipo inteiro limitado a um certo tamanho.
  - **Integer** é um tipo inteiro sem limite.
  - **Ratio** é o tipo de números racionais.
  - **Float** e **Double** são números com casas decimais.
- Funções para converter entre tipos numéricos:
  - **fromInteger**, **toInteger**, **fromRational**, **toRational**

# Números

- Verificando o tipo de constantes numéricas.

```
Prelude> :t 5
5 :: Num p => p
Prelude> :t 3.4
3.4 :: Fractional p => p
```

- Haskell possui um mecanismo avançado para especificar um tipo numérico.
  - Ele faz o ‘agrupamento’ de tipos similares.
  - Desta forma, fica mais fácil realizar operações entre diferentes constantes.
- Se precisar usar uma constante numérica negativa: (-4)

# Strings

- Já vimos como utilizar caracteres em Haskell.
- Para trabalhar com **String** utilizamos uma forma similar ao da linguagem C.
  - Basta utilizar aspas duplas.

```
Prelude> :t "Hello World"
"Hello World" :: [Char]
```

- Ao invés do tipo **String**, vemos o tipo **[Char]**.
  - Algo entre colchetes indica o uso de uma **lista de caracteres**.
  - Lista é a estrutura de dados mais utilizada em FP.

# Listas

- Listas são escritas com **vírgulas** separando cada valor **entre colchetes**.

```
Prelude> :t [1,2,3]
[1,2,3] :: Num a => [a]
```

- Funções sobre Listas

```
Prelude> :t reverse
reverse :: [a] -> [a]
Prelude> reverse [1,2,3]
[3,2,1]
Prelude> reverse "abc"
"cba"
```

- Concatenação de listas

```
Prelude> :t (++)
(++) :: [a] -> [a] -> [a]
Prelude> [1,2,3] ++ [4,5,6]
[1,2,3,4,5,6]
```

- Note neste exemplo que **reverse** e **(++)** podem operar em qualquer tipo de lista.

# Listas

- Listas em Haskell são homogêneas.
  - Cada lista pode ter elementos de um único tipo.

```
Prelude> [1,2,3,'a','b','c']
<interactive>:20:2: error:
    • No instance for (Num Char) arising from the literal ‘1’
    • In the expression: 1
      In the expression: [1, 2, 3, 'a', ....]
      In an equation for ‘it’: it = [1, 2, 3, ....]
Prelude> "abc" ++ [1,2,3]
<interactive>:21:11: error:
    • No instance for (Num Char) arising from the literal ‘1’
    • In the expression: 1
      In the second argument of ‘(++)’, namely ‘[1, 2, 3]’
      In the expression: "abc" ++ [1, 2, 3]
```

# Listas

- Haskell implementa Listas ligadas (*Linked Lists*).
- Outra forma para criar listas:

```
Prelude> 1 : 2 : 3 : []
[1,2,3]
Prelude> 'a' : 'b' : 'c' : []
"abc"
```

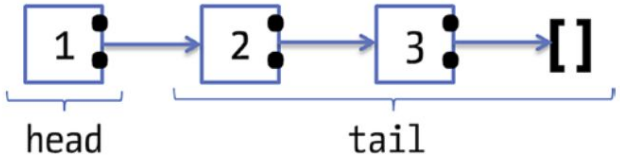
- Três funções principais para obter informações sobre uma lista:
  - **null** para verificar se uma lista está vazia.
  - **head** para buscar o primeiro elemento.
  - **tail** para retornar a lista sem o primeiro elemento.

# Listas

- Vejamos alguns exemplos:

```
Prelude> null [1,2,3]
False
Prelude> null []
True
Prelude> head [1,2,3]
1
Prelude> tail [1,2,3]
[2,3]
Prelude> head []
*** Exception: Prelude.head: empty list
```

- Representação gráfica



# Booleanos

- Booleanos tem apenas dois valores possíveis:
  - **True** e **False**.
- Existem diversas funções padrão para trabalhar com booleanos.
  - **(&&)**, **(||)** e **not** estão disponíveis no **Prelude**.
- Exemplos

```
Prelude> (True && False) || (False && not True)
False
Prelude> or [True, False, and [False, True, True]]
True
Prelude> (2 == 2.1) || (2 < 2.1) || (2 > 2.1)
True
```

# Condicionais

- Expressões no formato: **if** b **then** t **else** f.
  - Ambos **then** e **else** devem estar presentes.
  - Os tipos de dados resultantes da avaliação do **then** e do **else** devem ser os mesmos.
- Exemplo

```
Prelude> if 3 < 4.5 then "3 menor que 4.5" else "3 maior que 4.5"
"3 menor que 4.5"
```

- Verificando se a lista está vazia:

```
Prelude> if not (null []) then (head []) else "empty"
"empty"
```

# Múltiplas linhas no GHCi

- Também é possível escrever expressões multi-linha no GHCi.
  - Porém, é preciso utilizar o padrão **{: :}** para tal.

```
Prelude> :{
Prelude| if not (null ["hello", "ola"])
Prelude| then (head ["hello", "ola"]) else "empty"
Prelude| :}
"hello"
```

# Listas dentro de Listas

- Listas podem conter outras listas como elementos, ou qualquer nível de aninhamento.
- Exemplos:

```
Prelude> :t [['a','b','c'],['d','e']]
[['a','b','c'],['d','e']] :: [[Char]]
Prelude> head [['a','b','c'],['d','e']]
"abc"
Prelude> head (head [['a','b','c'],['d','e']])
'a'
Prelude> head [[]]
[]
```

- Note o detalhe do último exemplo:
  - Uma lista vazia dentro de uma lista.

# Considerações Finais

- Nesta aula vimos diversos tipos de dados e como utilizá-los no GHCi.
  - Utilizando **:t** podemos verificar o tipo de uma constante.
- Listas são muito utilizadas em programação funcional.
- Vimos diversas funções padrão para lidar com os mais diferentes tipos de dados.

# Exercícios

- Reescreva as listas anteriores utilizando apenas **(:)** e o construtor de lista vazia **[]**.
- Escreva uma expressão que verifica se uma lista está vazia, **[]**, ou se o primeiro elemento é vazio, como **[[],[‘a’,‘b’]]**.
- Escreva uma expressão que verifica se uma lista tem somente um elemento. Ela deve retornar **True** para **[‘a’]** e **False** para **[]** ou **[‘a’, ‘b’]**.
- Escreva uma expressão que concatena duas listas dadas dentro de outra lista. Por exemplo, ela deve retornar **“abcde”** para **[“abc”, “de”]**.