Be-a-bá do Haskell

Felipe A. Lessa

Departamento de Ciência da Computação
Universidade de Brasília

Assuntos de hoje

Atenção!

- Pessoal, tudo o que veremos aqui é *muito simples*.
- Todavia é necessário ter muita atenção: os programas serão feitos compondo elementos simples, é extremamente importante tê-los sempre bem entendidos!

Assuntos de hoje

Começando
- O interpretador
- Variáveis
- Funções

Conteiners
- Listas
- Tuplas

Finalizando
- É isso aí...

Abrindo o GHCi

- Nós vamos utilizar o GHCi como interpretador.
- Para abri-lo:
  1. Abra um terminal (e.g. Alt+F2 e `gnome-terminal`).
  2. Digite `ghci` e aperte enter.
- Você deverá ver algo como

```
   ___         ___ _
  / _ \ /\  /\/ __(_)
 / /_\// /_/ / /  | |      GHC Interactive, version 6.6.1, for Haskell
/ /_\\/ __  / /___| |      http://www.haskell.org/ghc/
\____/\/ /_/\____/|_|      Type :? for help.

Loading package base ... linking ... done.
Prelude>
```

- O `Prelude>` indica o prompt atual (você está dentro do módulo `Prelude`).

Expressões matemáticas simples

- Vamos tentar algumas expressões simples:

```
1 Prelude> 2 + 2
2 4

4 Prelude> 5 * 4 + 3
5 23

7 Prelude> 2 ^ 5
8 32
```

Assuntos de hoje

Um problema simples

- Suponha que nós queiramos saber a área de um círculo de 5 cm de raio.
- Simples: sabemos que a área é πr^2 onde $r = 5$ é o raio e $\pi = 3.14\ldots$ é a famosa constante transcendente.
- Podemos utilizar o interpretador para isso! (claro =)

Calculando a área

- Vamos lá então:

```
Prelude> 3.14 * 5^2
78.5
```

- Muito bom! Mas podemos fazer melhor.

Calculando a área II

- Os computadores são bons em fazer contas. Que tal utilizarmos uma precisão maior?

```
Prelude> 3.14159265358979323846264338 3 * 5^2
78.53981633974483
```

- Agora, que tal saber o comprimento da circunferência do círculo? (dica: $2\pi r$)

```
Prelude> 2 * 3.14159265358979323846264338 3 * 5
31.41592653589793
```

Calculando a área III

- Agora, e a área de um círculo de raio 25?

```
Prelude> 3.14159265358979323846264338 3 * 25^2
1963.4954084936207
```

- Isso está ficando chato =). Entram as variáveis!

Sua primeira variável

- Para evitar repetidamente escrever o valor de π com 27 casas decimais, vamos guardá-lo numa variável:

```
Prelude> let pi = 3.14159265358979323846264338 3
```

- Pronto! Agora, vamos usar o valor de π:

```
Prelude> pi
3.141592653589793

Prelude> pi * 5^2
78.53981633974483

Prelude> pi * 25^2
1963.4954084936207
```

Um pouco sobre tipos

- Poderíamos tentar guardar o valor do raio também:

```
Prelude> let r = 25
Prelude> 2 * pi * r

<interactive>:1:9:
    Couldn't match expected type 'Double'
          against inferred type 'Integer'
    In the second argument of '(*)', namely 'r'
    In the expression: (2 * pi) * r
    In the definition of 'it': it = (2 * pi) * r
```

- Oops! Houve um problema com os tipos.

Um pouco sobre tipos II

- O problema é que `25` pode ser interpretado como um `Double` ou um `Integer`, dentre outros tipos.
- Aqui o Haskell "adivinhou" que era um `Integer`. Podemos forçar um tipo diferente:

```
Prelude> let r = 25 :: Double
Prelude> 2 * pi * r
157.07963267948966
```

- Ele não pôde adivinhar o tipo certo porque não havia um contexto de onde o tipo poderia ser inferido.

Um pouco sobre tipos III

- Na verdade existe uma razão muito mais profunda para ele ter "adivinhado" errado. Em primeiro lugar, por que ele tentou adivinhar?
- A variável `r` poderia ter tipo o tipo polimórfico `Num a =>a` que serve para qualquer tipo numérico. Isso não ocorreu devido à *restrição de monomorfismo*.
- Vocês não precisam saber detalhes agora =). Apenas saibam que caso não houvesse essa restrição (e.g. com a opção `-fno-monomorphism-restriction` do `ghci`), então Coisas Ruins Aconteceriam™.

Mais variáveis

- As variáveis não precisam conter apenas constantes, podem ter como valor qualquer expressão válida.
- Por exemplo, para guardar a área do círculo de raio 5:

```
Prelude> let area = pi * 5^2
```

- ...e então recuperá-la mais tarde

```
Prelude> area
78.53981633974483
```

Variáveis mesmo?

- Já que as variáveis guardam qualquer coisa e permitem que não digitemos tudo repetidamente, vamos fazer:

```
1 Prelude> let r = 25.0
2 Prelude> let area2 = pi * r ^ 2
3 Prelude> area2
4 1963.4954084936207
```

- Muito bom. Agora, quando quisermos um $r = 2.0$:

```
1 Prelude> let r = 2.0
2 Prelude> area2
3 1963.4954084936207
```

- Ops! Por que não funcionou?

Variáveis mesmo? II

- O código do slide anterior não funcionou porque *em Haskell as variáveis não variam*.
- O código `let r = 2.0` definiu um outro *r*, diferente daquele primeiro *r* de quando `area2` foi definido.
- Tudo isso tem a ver com o *escopo* das variáveis. Não veremos isso agora, mas para resolver nosso problema basta definir

```
1 Prelude> let area3 = pi * r ^ 2
2 Prelude> area3
3 12.566370614359172
```

- ... mas é claro que existe uma maneira melhor =).

Assuntos de hoje

Entram em cena as funções!

- O que nós queremos é uma *função* que calcule a área.
- Definir funções é quase igual definir variáveis:

```
1 Prelude> let pi = 3.14159265358979323846264338327
2 Prelude> let area r = pi * r^2
```

- Pronto! Para calcular as áreas basta passar para `area`:

```
1 Prelude> area 5
2 78.53981633974483
3 Prelude> area 25
4 1963.4954084936207
```

Funções trazem parametrização

- Na definição da função, escrevemos `area r = ...`.
- Nós dizemos que `r` é um *parâmetro* da função.
- Quando a função é chamada, o parâmetro é passado para ela e seu valor é substituído na definição.
- Tudo isso é muito simples, mas vejamos se está claro...

Exercício

- Não digitem este código no interpretador!

```
Prelude> let r = 0
Prelude> let area r = pi * r ^ 2
Prelude> area 5
```

1. O que você acha que deve acontecer?
2. Qual será o valor de `area 5`?
3. Quais os escopos envolvidos?

Múltiplos parâmetros

- Podemos então brincar com vários parâmetros.
- Digamos que queiramos a área de um retângulo:

```
Prelude> let areaRet comp larg = comp * larg
Prelude> areaRet 5 10
50
```

- Ou a área de um triângulo:

```
Prelude> let areaTri b h = (b * h) / 2
Prelude> areaTri 3 9
13.5
```

Exercício

- Como definiria-se uma função para calcular o volume de uma esfera? E de um paralelepípedo?

Assuntos de hoje

Conhecendo as listas

- Listas são bem úteis em vários lugares, apenas tome cuidado para usá-la onde não deve!
- Elas são análogas às listas ligadas vistas em ED.
- Algumas listas podem ser criadas no interpretador da seguinte maneira:

```
1 Prelude> let numeros = [1,2,3,4]
2 Prelude> let verdades = [True, False, False]
3 Prelude> let strings = ["algumas", "strings",
4                         "numa", "lista"]
```

Conhecendo as listas II

- As listas são sempre homogêneas. Por exemplo, `[True, "pessoas"]` não é uma lista:

```
1 Prelude> let lista = [True, "pessoas"]

3 <interactive>:1:19:
4     Couldn't match expected type `Bool'
5                 against inferred type `[Char]'
6       Expected type: Bool
7       Inferred type: [Char]
8     In the expression: "pessoas"
9     In the expression: [True, "pessoas"]
```

Construindo listas

- Na verdade, a notação de escrever os elementos separados por vírgula entre colchetes é apenas "açúcar sintático" (syntatic sugar).
- As listas na verdade são construídas através do operador `(:)` (lido "cons").
- O cons basicamente coloca algum elemento na cabeça (início) da lista.

Construindo listas II

- Por exemplo:

```
Prelude> let numeros = [1,2,3,4]
Prelude> numeros
[1,2,3,4]
Prelude> 0:numeros
[0,1,2,3,4]
```

- Aqui você adicionou o `0` à cabeça de `numeros`.

Construindo listas III

- Você pode fazer isso indefinidamente:

```
Prelude> [1,2,3,4]
[1,2,3,4]
Prelude> 1:[2,3,4]
[1,2,3,4]
Prelude> 1:2:[3,4]
[1,2,3,4]
Prelude> 1:2:3:[4]
[1,2,3,4]
Prelude> 1:2:3:4:[]
[1,2,3,4]
```

- Hehehe, não dá para fazer indefinidamente =).

Construindo listas IV

- Basicamente, temos uma base (a lista vazia `[]`, lida "nil"), e uma forma de adicionar elementos (o cons).
- Em verdade, quando você escreve `[1,2,3,4]`, o Haskell infere `1:2:3:4:[]` por você. Para isso serve o syntatic sugar, para auxiliar o programador.
- Mas nós temos que novamente dar uma olhada nos tipos.

Problemas com o tipo do cons

- O cons tem dois argumentos, um elemento e uma lista.
- É um erro o segundo não ser uma lista de elementos:

```
Prelude> True : False

<interactive>:1:7:
    Couldn't match expected type `[Bool]'
                against inferred type `Bool'
    In the second argument of `(:)', namely `False'
    In the expression: True : False
    In the definition of `it': it = True : False
```

- Há uma outra sutileza: a lista deve conter elementos *do mesmo tipo* do que será adicionado.

Exercícios

1. O código `3:[True,False]` funcionaria? Por quê?
2. Escreva uma função `cons8` que adiciona `8` a uma lista. Qual o resultado de:
   2.1 `cons8 []`
   2.2 `cons8 [1,2,3]`
   2.3 `cons8 [True,False]`
   2.4 `let foo = cons8 [1,2,3]`
   2.5 `cons8 foo`
3. Escreva uma função que funcione como o cons, pegue um elemento e uma lista e adicione-os.

O que as listas podem ter

- Até agora estudamos poucos tipos, então os exemplos têm sido extremamente simples até agora.
- Apesar disso, as listas podem conter *qualquer* valor.
- Em particular, podemos ter listas de listas: =)

```
1 Prelude> let dentro = [[1,2],[3,4],[5,6]]
2 Prelude> dentro
3 [[1,2],[3,4],[5,6]]
```

Mais exercícios (phew!)

1. Quais abaixo são válidos? Reescreva usando cons.
   1.1 `[1,2,3,[]]`
   1.2 `[1,[2,3],4]`
   1.3 `[[1,2,3],[]]`
2. E desses, quais são? Reescreva usando vírgulas.
   2.1 `[]:[[1,2],[3,4]]`
   2.2 `[]:[]`
   2.3 `[]:[]:[]`
   2.4 `[1]:[]:[]`
   2.5 `([1]:[]):[]`

Assuntos de hoje

Conhecendo as tuplas

- As tuplas possuem uma função semelhante à das listas: guardar vários elementos em um só.
- Apesar disso, há duas diferenças cruciais:
  - As tuplas contêm um número *fixo* de elementos.
  - Em compensação, tuplas são heterogêneas.
- Exemplos comuns de uso: coordenadas (x e y), retorno de vários valores, etc.
- Nós *não* usamos tuplas sempre: em Haskell, definir novos tipos é moleza (em breve veremos).
- No mundo matemático elas são chamadas de “ênuplas”. Quando possuem dois ou três valores, então são chamadas de duplas (ou pares ordenados) e triplas.

Conhecendo as tuplas II

- Alguns exemplos de tuplas:

```
1 (True, 1)
2 ("Hello world", False)
3 (4, 5, "Seis", True, 'b')
```

- Note, porém, que tuplas de diferentes tipos são *fundamentalmente* diferentes. Por exemplo, não existe um operador cons, como para listas.

Pegando os valores das duplas

- Veremos um mecanismo genérico para obter os elementos armazenados em qualquer estrutura de dados (adivinhou: em breve hehe).
- Por enquanto, podemos ver como tirar valores de duplas: usando as funções `fst` e `snd`:

```
1 Prelude> fst (2, 5)
2 2
3 Prelude> fst (True, "boo")
4 True
5 Prelude> snd (5, "Hello")
6 "Hello"
```

- Essas funções são chamadas de “projeções”.

Tuplas também são genéricas

- Assim como listas, tuplas podem conter elementos de qualquer tipo, inclusive listas e outras tuplas:

```
1 ((2,3), True)
2 ((2,3), [2,3])
3 [(1,2), (3,4), (5,6)]
```

Exercícios

1. Usando `fst` e `snd`, como extrair o `4` de `(("Hello", 4), True)`?
2. Quais desses códigos são válidos?
   - 2.1 `fst [1,2]`
   - 2.2 `1:(2,3)`
   - 2.3 `(2,4):(2,3)`
   - 2.4 `(2,4):[]`
   - 2.5 `(2,4),(5,5),('a','b')]`
   - 2.6 `([2,4],[2,2])`

Assuntos de hoje

Pois é!

- Agora vocês já sabem o básico do básico =).
- Vocês não têm nenhum dever de casa... aproveitem!