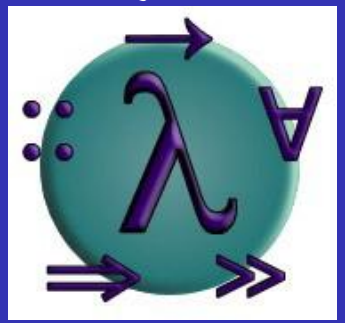
Programação Funcional



Capítulo 2

Primeiros Passos

José Romildo Malaquias

Departamento de Computação
Universidade Federal de Ouro Preto

2018.1

Tópicos

Glasglow Haskell Compiler

GHC (Glasglow Haskell Compiler):

- http://www.haskell.org/ghc/
- É um compilador e um ambiente interativo para a linguagem funcional Haskell.
- É um software livre, distribuído com a licença GPL.
- Suporta toda a linguagem Haskell 2010 mais uma grande variedade de extensões.
- Tem suporte particularmente bom para concorrência e paralelismo.
- Gera código rápido, principalmente para programas concorrentes.
- Funciona em várias plataformas, incluindo Windows, Mac, Linux, a maioria das variedades de Unix, e várias arquiteturas de processadores diferentes.
- Tem capacidades de otimização, incluindo otimização entre módulos.

Glasglow Haskell Compiler (cont.)

- GHC compila código Haskell
  - diretamente para código nativo ou
  - pode usar LLVM como um *back-end*, ou
  - pode gerar código C como código intermediário para portar para novas plataformas.
- O ambiente interativo **GHCi** compila para *bytecode* (representação intermediária entre linguagem de alto nível e linguagem de máquina), e suporta a execução mista de *bytecode* e código nativo.
- GHC suporta *profiling*.
- GHC vem com várias bibliotecas: `http://www.haskell.org/ghc/docs/latest/html/libraries/index.html`.
- Muitas outras bibliotecas estão disponíveis no repositório Hackage em `http://hackage.haskell.org/packages/hackage.html`.

Plataforma Haskell

- A **Plataforma Haskell** (http://www.haskell.org/platform/) é um ambiente de desenvolvimento abrangente e robusto para a programação em Haskell.
- A plataforma contém o compilador GHC e várias bibliotecas prontas para serem usadas.

O ambiente interativo **ghci**

- GHCi é um ambiente interativo do GHC em que expressões Haskell podem ser avaliadas de forma interativa e os programas podem ser interpretados.
- GHCi tem suporte para carregar código compilado de forma interativa.
- GHCi inclui um depurador interativo.
- O GHCi pode ser iniciado a partir do prompt simplesmente digitando `ghci`. Em um sistema Unix:

```
$ ghci
GHCi, version 8.2.2: http://www.haskell.org/ghc/  :? for help
Loaded GHCi configuration from /home/romildo/.ghc/ghci.conf

Prelude>
```

O ambiente interativo **ghci** (cont.)

- O prompt `>` significa que o sistema GHCi está pronto para avaliar expressões.
- À esquerda do prompt é mostrada a lista de módulos abertos.
- Expressões Haskell podem ser digitadas no prompt.
- Por exemplo:

```
Prelude> 2 + 3 * 4
14

Prelude> (2 + 3) * 4
20

Prelude> sqrt (3^2 + 4^2)
5.0
```

Tópicos

Módulos

- Programas em Haskell são organizados em módulos.
- Um **módulo** é formado por um conjunto de definições (tipos, variáveis, funções, etc.).
- Para que as definições de um módulo possam ser usadas em um programa, o módulo deve ser importado.
- A **biblioteca padrão** é formada por um conjunto de módulos disponível automaticamente para todas os programas em Haskell.
- A **Plataforma Haskell** inclui várias bibliotecas adicionais.
- Além das bibliotecas padrão, estão disponíveis vários outros módulos que podem ser instalados separadamente.

standard prelude

- O arquivo de biblioteca `Prelude.hs` oferece um grande número de funções definidas no *padrão da linguagem* através do módulo **`Prelude`**.
- O módulo **`Prelude`** é importado automaticamente em todos os módulos de uma aplicação Haskell.
- O módulo **`Prelude`** oferece várias funções para manipulação de números e outros dados.
- Todas as definições do módulo **`Prelude`** podem ser listadas usando o seguinte comando no GHCi:

```
Prelude> :browse Prelude
```

standard prelude (cont.)

- Normalmente uma **aplicação de função** usa a notação prefixa: escreve-se a função seguida dos argumentos.
- Exemplos:

```
Prelude> negate 78
-78

Prelude> abs (-7412)
7412

Prelude> sqrt 2.56
1.6

Prelude> cos pi
-1.0

Prelude> log 100
4.605170185988092
```

standard prelude (cont.)

```
Prelude> exp 2.0
7.38905609893065

Prelude> sqrt (abs (- 16))
4.0
```

standard prelude (cont.)

- No entanto algumas funções são definidas como operadores binários infixos.
- Os operadores binários apresentam a precedência e associatividade usuais da Matemática.
- O módulo `Prelude` oferece as funções aritméticas familiares, como `+`, `-`, `*`, `div`, `mod`, `/`.
- Exemplos:

```
Prelude> 7 * 8
56

Prelude> 1 + 2 * 3
7

Prelude> (1 + 2) * 3
9

Prelude> div (7*8) 3
18
```

standard prelude (cont.)

```
Prelude> mod (7*8) 3
2

Prelude> (7*8) / 3
18.666666666666668
```

standard prelude (cont.)

- A função que calcula o inverso aditivo (simétrico) de um número é o operador unário prefixo `-`, que tem a mesma definição que a função `negate`.
- Exemplos:

```
Prelude> negate (9 - 5)
-4

Prelude> negate (5 - 9)
4

Prelude> - (5 - 9)
4

Prelude> - 5
-5
```

standard prelude (cont.)

- Além das funções aritméticas, o módulo `Prelude` também oferece muitas funções úteis para a manipulação de **listas** e outras **estruturas de dados**.
- Uma **lista** pode ser escrita como uma sequência de expressões separadas por vírgula e delimitada por colchetes.
- **Exemplo**:

```
Prelude> [12, div 78 10, 0, 5 - 9]
[12,7,0,-4]

Prelude> []
[]
```

- Algumas operações com listas:

standard prelude (cont.)

- `null` : verifica se uma lista é vazia:

```
Prelude> null []
True

Prelude> null [1,2,3,4,5]
False
```

standard prelude (cont.)

- `head` : seleciona a **cabeça** (primeiro elemento) de uma lista:

```
Prelude> head [1,2,3,4,5]
1
Prelude> head []
*** Exception: Prelude.head: empty list
```

- `tail` : seleciona a **cauda** da lista, ou seja, uma lista formada por todos os elementos exceto o primeiro:

```
Prelude> tail [1,2,3,4,5]
[2,3,4,5]
Prelude> tail [5*4, 6*5]
[30]
Prelude> tail [8-1]
[]
Prelude> tail []
*** Exception: Prelude.tail: empty list
```

standard prelude (cont.)

- `lenght` : calcula o tamanho de uma lista:

```
Prelude> length [1,2,3,4,5]
5

Prelude> length []
0
```

standard prelude (cont.)

- `(!!)` : seleciona o n-ésimo elemento de uma lista ($0 \leq n < \texttt{len}$, onde `len` é o comprimento da lista):

```
Prelude> [1,2,3,4,5] !! 2
3

Prelude> [1,2,3,4,5] !! 0
1

Prelude> [1,2,3,4,5] !! 10
*** Exception: Prelude.(!!): index too large

Prelude> [1,2,3,4,5] !! (-4)
*** Exception: Prelude.(!!): negative index
```

standard prelude (cont.)

- `take` : seleciona os primeiros *n* elementos de uma lista:

```
Prelude> take 3 [1,2,3,4,5]
[1,2,3]
```

- `drop` : remove os primeiros *n* elementos de uma lista:

```
Prelude> drop 3 [1,2,3,4,5]
[4,5]
```

standard prelude (cont.)

- `sum` : calcula a soma dos elementos de uma lista de números:

```
Prelude> sum [1,2,3,4,5]
15
```

- `product` : calcula o produto dos elementos de uma lista de números:

```
Prelude> product [1,2,3,4,5]
120
```

standard prelude (cont.)

- `(++)` : concatena duas listas:

```
Prelude> [1,2,3] ++ [4,5]
[1,2,3,4,5]
```

- `reverse` : inverte uma lista:

```
Prelude> reverse [1,2,3,4,5]
[5,4,3,2,1]
```

Tópicos

Applicação de função

- Em Matemática, **aplicação de função** é denotada usando parênteses, e a **multiplicação** é muitas vezes denotada usando justaposição ou espaço.
- Exemplo:

$$f(a, b) + cd$$

aplica a função f aos argumentos a e b, e adiciona o resultado ao produto de c e d.

- Em Haskell, a **aplicação de função** é denotada usando espaço, e a **multiplicação** é indicada pelo operador *.
- Exemplo:

```
f a b + c * d
```

aplica a função `f` aos argumentos `a` e `b`, e adiciona o resultado ao produto de `c` e `d`.

Applicação de função (cont.)

- Aplicação de função tem precedência maior do que todos os outros operadores.
- Assim

```
f a + b
```

significa

```
(f a) + b
```

em vez de

```
 f (a + b)
```

Applicação de função (cont.)

- Exemplos:

Matemática	Haskell
$f(x)$	`f x`
$f(x, y)$	`f x y`
$f(g(x))$	`f (g x)`
$f(x, g(y))$	`f x (g y)`
$f(x)g(y)$	`f x * g y`

Tópicos

Programas fontes

- Além de poder usar as funções do prelúdio, o programador pode também definir e usar suas próprias funções.
- Novas funções são definidas em arquivos de programa (às vezes chamado de ***script***), um arquivo texto contendo uma seqüência de definições.
- Por convenção, *scripts* Haskell normalmente têm a extensão `.hs` em seu nome. Isso não é obrigatório, mas é útil para fins de identificação.
- Definições de variáveis e funções são feitas usando equações.

Meu primeiro *script*

- Ao desenvolver um *script* Haskell, é útil manter duas janelas abertas, uma executando um editor para editar o script, e outra para o ambiente interativo (GHCi) em execução.
- **Exemplo**:
  Inicie um editor de texto, digite as seguintes definições de função, e salve o *script* com o nome `teste.hs`:

```
dobro x = x + x

quadruplo x = dobro (dobro x)
```

Meu primeiro *script* (cont.)

- Deixando o editor aberto, em outra janela execute o GHCi carregando o novo *script*:

```
$ ghci teste.hs
GHCi, version 8.2.2: http://www.haskell.org/ghc/  :? for help
package flags have changed, resetting and loading new packages...
Loaded GHCi configuration from /home/romildo/.ghc/ghci.conf
[1 of 1] Compiling Main             ( teste.hs, interpreted )
Ok, modules loaded: Main.
*Main>
```

Meu primeiro *script* (cont.)

- Agora, tanto `Prelude.hs` como `teste.hs` são carregados, e as funções de ambos os *scripts* podem ser usadas:

```
*Main> quadruplo 10
40

*Main> take (dobro 2) [1,2,3,4,5,6]
[1,2,3,4]
```

Meu primeiro *script* (cont.)

- Deixando o GHCi aberto, volte para o editor, adicione as seguintes definições ao *script* `teste.hs`, e salve:

```
fatorial n = product [1..n]

media ns  = div (sum ns) (length ns)

mediaFracionaria ns = sum ns / fromIntegral (length ns)
```

Meu primeiro *script* (cont.)

- GHCi não detecta automaticamente que o *script* foi alterado. Assim um comando **reload** deve ser executado para que as novas definições possam ser usadas:

```
*Main> :reload
[1 of 1] Compiling Main             ( teste.hs, interpreted )
Ok, modules loaded: Main.
```

```
*Main> fatorial 10
3628800

*Main> fatorial 50
30414093201713378043612608166064768844377641568960512000000000000

*Main> media [1,2,3,4,5]
3

*Main> mediaFracionaria [1,2,3,4,5,11]
4.333333333333333
```

Meu primeiro *script* (cont.)

- A aplicação de uma função de dois argumentos pode ser escrita usando a notação de operador infixo: basta escrever o nome da função entre aspas invertidas (crase)
- **Exemplo**:

```
media ns  = sum ns `div` length ns
```

- Observe que
  - `div` é colocado entre aspas simples invertidas (crase)
  - `x `f` y` é apenas uma abreviação sintática para `f x y`.

Identificadores

- Nomes de **função** e **variáveis** devem começar com uma letra minúscula e podem conter letras, dígitos, sublinhado e apóstrofo (aspa simples).
- **Exemplos**:

```
myFun
fun1
arg_2
x'
```

- Por convenção, uma lista de elementos normalmente têm um sufixo `s` em seu nome, que indica plural.
- **Exemplos**:

```
xs
ns
nss
pesos
idades
```

Comentários

- **Comentários** são usados para fazer anotações no programa que podem ajudar a entender o funcionamento do mesmo.
- Os comentários são ignorados pelo compilador.
- **Comentário de linha**: é introduzido por `--` e se estendem até o final da linha.
- **Comentário de bloco**: é delimitado por `{-` e `-}`. Pode ser aninhado.

Tópicos

A regra de *layout*

Em uma **seqüência de definições**, cada definição deve começar precisamente na mesma coluna:

```
a = 10
b = 20
c = 30
```

```
a = 10
  b = 20
c = 30
```

```
  a = 10
b = 20
   c = 30
```

A regra de *layout* (cont.)

Se uma definição for escrita em mais de uma linha, as linhas subsequentes à primeira devem começar em uma coluna mais à direita da coluna que caracteriza a sequência de definições.

```
a = 10 + 20 + 30 +
      40 + 50 + 60
b = sum [10,20,30]
```

```
   a = 10 + 20 + 30 +
40 + 50 + 60
   b = sum [10,20,30]
```

```
a = 10 + 20 + 30 +
40 + 50 + 60
b = sum [10,20,30]
```

A regra de *layout* (cont.)

A regra de *layout* evita a necessidade de uma sintaxe explícita para indicar o agrupamento de definições usando `{`, `}` e `;`

```
{- agrupamento implícito -}
a = b + c
    where
      b = 1
      c = 2
d = a * 2
```

significa

```
{- agrupamento explícito -}
a = b + c
     where { b = 1 ; c = 2 }
d = a * 2
```

A regra de *layout* (cont.)

- Para evitar problemas com a regra de *layout*, é recomendado não utilizar caracteres de tabulação para indentação do código fonte, uma vez que um único caracterizar de tabulação pode ser apresentado na tela como vários espaços.
  O texto do programa vai aparentar estar alinhado na tela do computador, mas na verdade pode não estar devido ao uso do tabulador.
- No editor **notepad++** você deve desabilatar o uso de tabulação.
  Para tanto marque a opção para substituir tabulações por espaço, acessando o menu Configurações -> Preferências -> Menu de Linguagens/Configuração de Abas -> Substituir por espaço antes de editar o arquivo.

Tópicos

Comandos úteis do GHCi

comando	abrev	**significado**
`:load` *name*	`:l`	carrega o script *name*
`:reload`	`:r`	recarrega o script atual
`:edit` *name*	`:e`	edita o script *name*
`:edit`	`:e`	edita o script 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

Exercícios

Exercise 1

Experimente os exemplos apresentados nos slides usando o GHCi.

Exercise 2

Identifique e orrija os erros de sintaxe no *script* que se segue, e teste sua solução usando o GHCi.

```
N = a 'div' length xs
    where
       a = 10
      xs = [1,2,3,4,5]
```

Exercícios (cont.)

Exercise 3

Defina uma função para calcular o quadrado do dobro do seu argumento. Teste sua função no GHCi.

Exercise 4

Defina uma função para calcular o dobro do quadrado do seu argumento. Teste sua função no GHCi.

Exercícios (cont.)

Exercise 5

Mostre como a função de biblioteca `last` que seleciona o último elemento de uma lista não vazia pode ser definida usando as funções do prelúdio introduzidas neste capítulo.

Exercise 6

Você pode pensar em outra possível definição para a função `last`?

Exercise 7

Da mesma forma, mostrar como a função de biblioteca `init`, que remove o último elemento de uma lista não vazia pode ser definida de duas maneiras diferentes.

Fim