Programação Funcional em Haskell

José Romildo Malaquias

BCC222: Programação Funcional

Universidade Federal de Ouro Preto
Departamento de Computação

27 de setembro de 2023

Sumário

3 Expressões e Definições

Resumo

Neste capítulo são apresentados alguns elementos básicos da linguagem Haskell que permitirão a construção de expressões envolvendo constantes, variáveis e funções.

Sumário

3.1 Constantes

As formas mais básicas de expressões são os construtores constantes e os literais, que representam valores em sua forma mais simples, ou seja, já estão reduzidos à sua forma canônica. Uma expressão está na **forma canônica** quando ela não pode mais ser simplificada.

Os **literais** são expressões com sintaxe especial para escrever alguns valores. Já **construtores constantes** são identificadores começando com letra maiúscula.

Veja alguns exemplos de construtores constantes e literais na tabela a seguir.

descrição			exemplo
literais numéricos	inteiros	em decimal	`8743`
		em octal	`0o7464`
			`0O103`
		em hexadecimal	`0x5A0FF`
			`0xE0F2`
	fracionários	em decimal	`140.58`
			`8.04e7`
			`0.347E+12`
			`5.47E-12`
			`47e22`
literais caracter			`'H'`
			`'\n'`
			`'\x65'`
literais string			`"bom dia"`
			`"ouro preto\nmg"`
construtores booleanos			`False`
			`True`

Os literais numéricos são sempre positivos.

3.2 Aplicação de função

Aplicação de função é uma das formas de expressões mais comuns na programação funcional, uma vez que os programas são organizados em funções.

Sintaticamene uma aplicação de função em **notação prefixa** consiste em escrever a função seguida dos argumentos, se necessário *separados por caracteres brancos* (espaços, tabuladores, mudança de linha, etc.).

Exemplos:

```
Prelude> sqrt 25
5.0

Prelude> cos 0
1.0

Prelude> tan pi
-1.2246467991473532e-16

Prelude> exp 1
2.718281828459045

Prelude> logBase 3 81
4.0

Prelude> log 10
2.302585092994046
```

```
Prelude> mod 25 7
4

Prelude> negate 7.3E15
-7.3e15

Prelude> not True
False
```

Observe que, diferentemente de várias outras linguagens de programação, **os argumentos não são escritos entre parênteses e nem separados por vírgula**.

Parênteses podem ser usados para **agrupar subexpressões**. Por exemplo:

```
Prelude> sqrt (logBase 3 81)
2.0

Prelude> logBase (sqrt 9) 81
4.0
```

Aplicações de função também podem ser escritas em **notação infixa**, onde a função é escrita entre os seus argumentos. Neste caso dizemos que as funções são **operadores infixos**. Exemplos:

```
Prelude> 2 + 3
5

Prelude> 10 / 4
2.5

Prelude> (12 - 7) * 6
30

Prelude> 5 * sqrt 36
30.0

Prelude> 6 <= 17
True

Prelude> 'A' == 'B'
False

Prelude> 'A' /= 'B'
True

Prelude> True || False
True

Prelude> True && False
False
```

Assim como na Matemática e em outras linguagens de programação, os operadores possuem um **nível de precedência** (ou prioridade) e uma **associatividade**. Parênteses podem ser usados para agrupar subexpressões dentro de expressões maiores *quebrando* a precedência ou associatividade dos operadores.

O nível de precedência de um operador é dado por um número entre 0 e 9, inclusive. Se dois operadores disputam um operando, o operador de maior precedência é escolhido.

A tabela 3.1 lista os operadores definidos no prelúdio.

precedência	**associatividade**	**operador**	**descrição**
9	esquerda	`!!`	índice de lista
	direita	`.`	composição de funções
8	direita	`^`	potenciação com expoente inteiro não negativo
		`^^`	potenciação com expoente inteiro
		`**`	potenciação com expoente em ponto flutuante
7	esquerda	`*`	multiplicação
		`/`	divisão fracionária
		`` `div` ``	quociente inteiro truncado em direção a $-\infty$
		`` `mod` ``	módulo inteiro satisfazendo `(div x y)*y + (mod x y) == x`
		`` `quot` ``	quociente inteiro truncado em direção a 0
		`` `rem` ``	resto inteiro satisfazendo `(quot x y)*y + (rem x y) == x`
6	esquerda	`+`	adição
		`-`	subtração
5	direita	`:`	construção de lista não vazia
		`++`	concatenção de listas
4	não associativo	`==`	igualdade
		`/=`	desigualdade
		`<`	menor que
		`<=`	menor ou igual a
		`>`	maior que
		`>=`	maior ou igual a
		`` `elem` ``	pertinência de lista
		`` `notElem` ``	negação de pertinência de lista
3	direita	`&&`	conjunção (*e* lógico)
2	direita	`||`	disjunção (*ou* lógico)
1	esquerda	`>>=`	composição de ações sequenciais
		`>>`	composição de ações sequenciais (ignora o resultado da primeira)
0	direita	`$`	aplicação de função
		`$!`	aplicação de função estrita
		`` `seq` ``	avaliação estrita

Tabela 3.1: Precedências e associatividades dos operadores do `Prelude`.

Exemplos:

```
Prelude> 2 + 3 * 4        -- * tem maior precedência que +
14

Prelude> 5 ^ 2 - 10       -- ^ tem maior precedência que -
15

Prelude> 2 ^ 3 ^ 2        -- ^ associa-se à direita
512
```

Aplicações de função em notação prefixa tem prioridade maior do que todos os operadores. Exemplos:

```
Prelude> abs 10 - 20             -- abs tem precedência maior que -
-10

Prelude> abs (10 - 20)
10

Prelude> succ 9 + max 5 4 * 3    -- succ e max tem precedência maior que + e *
25

Prelude> 2 * logBase (8/2) 256 + 1000
1008.0
```

Um operador pode ser associativo à esquerda, associativo à direita, ou não-associativo. Quando dois operadores com a mesma precedência disputam um operando,

- se eles forem associativos à esquerda, o operador da esquerda é escolhido,
- se eles forem associativos à direita, o operador da direta é escolhido,
- se eles forem não associativos, a expressão é mal formada e contém um erro de sintaxe,

Exemplos:

```
Prelude> 15 - 4 - 6   -- - associa-se à esquerda
5

Prelude> 15 - (4 - 6)
17

Prelude> 10 - 2 + 5   -- + e - tem a mesma precedência e associam-se à esquerda
13

Prelude> 10 - (2 + 5)
3

Prelude> 2^3^4                -- ^ associa-se à direita
2417851639229258349412352

Prelude> (2^3)^4
```

```
4096

Prelude> 3 > 4 > 5            -- > é não associativo
erro de sintaxe
```

O símbolo - merece atenção especial, pois ele pode tanto ser a função de subtração (operador infixo) como a função de inversão de sinal (operador prefixo).

```
Prelude> 6 - 2
4

Prelude> - 5
-5

Prelude> - (5 - 9)
4

Prelude> negate (5 - 9)
4

Prelude> 4 * (-3)
-12

Prelude> 4 * -3
erro de sintaxe
```

A notação prefixa é usada com nomes de funções que são identificadores alfanuméricos: formados por uma sequência de letras, dígitos decimais, sublinhado (_) e apóstrofo (') começando com letra minúscula ou sublinhado (e que não seja uma palavra reservada).

Já a notação infixa é usada com nomes de funções simbólicos: formados por uma sequência de símbolos especiais (! # $ % & + . / < = > ? @ | \ ^ - ~ :) que não começa com :.

Qualquer operador pode ser usado em notação prefixa, bastando escrevê-lo entre parênteses. Exemplos:

```
Prelude> (+) 4 5
9

Prelude> (/) 18.2 2
9.1

Prelude> (>=) 10 20
False

Prelude> sqrt ((+) 4 5)
3
```

Qualquer função prefixa de dois argumentos pode ser usada em notação infixa, bastando escrevê-la entre apóstrofos invertidos (sinal de crase: `), com precedência padrão 9 e associatividade à esquerda. Exemplos:

```
Prelude> 20 `div` 3
6
```

```
Prelude> 20 `mod` 3
2

Prelude> 20 `mod` 3 == 0
False

Prelude> 3 `logBase` 81
4.0

Prelude> (3 `logBase` 81) ^ 2
16.0

Prelude> 3 `logBase` (81 ^ 2)
8.0

Prelude> 3 `logBase` 81 ^ 2
16.0

Prelude> (20 `div` 3) ^ 2
36

Prelude> 20 `div` 3 ^ 2
2
```

3.3 Nomeando valores

Quando uma expressão é avaliada diretamente no ambiente interativo, uma variável chamada `it` é automaticamente definida para denotar o valor da expressão. Exemplo:

```
Prelude> 2 + 3 * 4
14

Prelude> it
14

Prelude> 7*(it - 4)
70

Prelude> it
70
```

Uma **declaração `let`** pode ser usada para definir uma variável no ambiente interativo. Por exemplo:

```
Prelude> let idade = 2 + 3 * 4

Prelude> idade
14
```

```
Prelude> 7*(idade - 4)
70
```

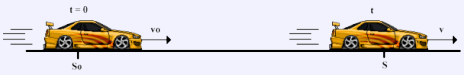
3.4 Avaliando expressões

Tarefa 3.1: Movimento Retilínio Uniformementne Variado

A posição s de um corpo em movimento retilínio uniformemente variado, em função do tempo t, é dado pela equação

$$s = s_0 + v_0 t + \frac{1}{2} a t^2$$

onde s_0 é a posição inicial do corpo, v_0 é a sua velocidade inicial, e a é a sua acelaração.



Utilize o **ambiente interativo GHCi** para calcular a posição de uma bola em queda livre no instante $t = 8\,\text{s}$, considerando que a posição inicial é $s_0 = 100\,\text{m}$, a velocidade inicial é $v_0 = 15\,\text{m/s}$ e a acelaração da gravidade é $a = -9.81\,\text{m/s}^2$.

Dicas: Use a declaração `let` para criar variáveis correspondentes aos dados e em seguida avalie a expressão correspondente à função horária do movimento usando estas variáveis.

Tarefa 3.2: Expressões matemáticas

Utilize o **ambiente interativo** para avaliar as expressões aritméticas seguintes, considerando que $x = 3$ e $y = 4$.

a) $\frac{4}{3}\pi \sin x^2 - 1$

b) $\frac{x^2 y^3}{(x-y)^2}$

c) $\frac{1}{x^2 - y} - e^{-4x} + \sqrt[3]{\frac{35}{y}}\sqrt{xy}$

d) $\frac{24 + 4.5^3}{e^{4.4} - log_{10} 12560}$

e) $\cos\frac{5\pi}{6} \sin^2\frac{7\pi}{8} + \frac{\tan\left(\frac{\pi}{6}\ln 8\right)}{\sqrt{7}+2}$

3.5 Definindo variáveis e funções

Além de poder usar as funções das bibliotecas, o programador também pode *definir* e *usar* suas próprias funções. Novas funções são definidas em arquivos texto geralmente chamdos de **código fonte** ou **programa fonte** ou ainda ***script***. Um programa fonte contém definições (de variáveis, funções, tipos, etc.) usadas para estruturar o código da aplicação.

Por convenção, *arquivos de programas fonte* em Haskell normalmente tem a *extensão* `.hs` em seu nome. Isso não é obrigatório, mas é útil para fins de identificação.

Variáveis e *funções* são definidas usando **equações**. No lado esquerdo de uma equação colocamos o nome da variável ou o nome da função seguido de seus parâmetros formais. No lado direito colocamos uma expressão cujo valor será o valor da variável ou o resultado da função quando a função for aplicada em seus argumentos.

Nomes de **funções** e **variáveis** podem ser alfanuméricos ou simbólicos:

identificadores alfanuméricos

- começam com uma letra minúscula ou sublinhado e podem conter letras, dígitos decimais, sublinhado (_) e apóstrofo (aspa simples ')
- são normalmente usados em notação prefixa
- exemplos:
  `myFun`
  `fun1`
  `arg_2`
  `x'`

identificadores simbólicos

- formados por uma sequência de símbolos e não podem começar com dois pontos (:)
- são normalmente usados em notação infixa
- exemplos:
  `<+>`
  `===`
  `$*=*$`
  `+=`

Ao desenvolver um programa pode ser útil manter duas janelas abertas, uma executando um **editor de texto** para editar o código, e outra para o **ambiente interativo** (GHCi) em execução. Na seção2.2 são apresentadas algumas sugestões de editores de texto que poderão ser usados para escrever seus programas.

> Os arquivos de programas em Haskell sempre devem ser salvos usando a codificação de caracteres `UTF-8`.

Tarefa 3.3: Meu primeiro programa fonte

Inicie um editor de texto, digite as seguintes definições de função, e salve o arquivo com o nome `test.hs`.

```
-- calcula o dobro de um número
dobro x = x + x

-- calcula o quádruplo de um número
quadruplo x = dobro (dobro x)
```

Deixando o editor aberto, em outra janela execute o GHCi carregando o novo arquivo fonte:

```
$ ghci test.hs
GHCi, version 7.10.3: http://www.haskell.org/ghc/  :? for help
[1 of 1] Compiling Main             ( test.hs, interpreted )
Ok, modules loaded: Main.

*Main>
```

Agora, tanto `Prelude.hs` como `test.hs` são carregados, e as funções de ambos os arquivos fonte podem ser usadas:

```
*Main> quadruplo 10
40

*Main> 5*(dobro 2) - 3
17
```

Observe que o GHCi usa o nome de módulo `Main` se o arquivo fonte não define um nome para o módulo.

Tarefa 3.4: Modificando meu primeiro programa fonte

Deixando o GHCi aberto, volte para o editor, adicione as seguintes definições ao arquivo fonte `test.hs`, e salve-o.

```
areaCirculo r = pi * r^2
```

O GHCi não detecta automaticamente que o arquivo fonte foi alterado. Assim o comando **:reload** deve ser executado para que as novas definições possam ser usadas:

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

```
*Main> areaCirculo 5
78.53981633974483
```

3.6 Comentários

Comentários são usados para fazer anotações no programa que podem ajudar a entender o funcionamento do mesmo. Os comentários são ignorados pelo compilador.

Um **Comentário de linha** é introduzido por `--` e se estende até o final da linha.

Um **Comentário de bloco** é delimitado por `{-` e `-}`. Comentários de bloco podem ser aninhados.

3.7 Definições locais em equações

Em Haskell **equações** são usadas para definir variáveis e funções, como discutido anteriormente. Em muitas situações é desejável poder definir valores e funções auxiliares em uma definição principal. Isto pode ser feito escrevendo-se uma **cláusula where** ao final da equação. Uma cláusula where é formada pela palavra chave `where` seguida das definições auxiliares.

A cláusula where faz **definições que são locais** à equação, ou seja, o **escopo** dos nomes definidos em uma cláusula where restringe-se à menor equação contendo a cláusula where.

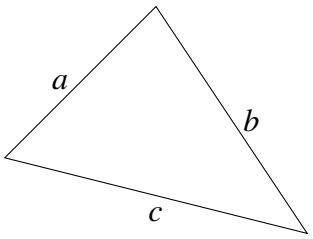
Por exemplo, considere a fórmula de Heron

$$A = \sqrt{s(s-a)(s-b)(s-c)}$$

para calcular a área de um triângulo com lados a, b e c, sendo

$$s = \frac{a+b+c}{2}$$

o semiperímetro do triângulo.



Como s aparece várias vezes na fórmula, podemos defini-lo localmente uma única vez e usá-lo quantas vezes forem necessárias na equação.

```
areaTriangulo a b c = sqrt (s * (s-a) * (s-b) * (s-c))
  where
    s = (a + b + c)/2
```

Esta definição assume que os argumentos da função são valores válidos para os lados de um triângulo.

```
areaTriangulo 5 6 8
⇝ sqrt (s * (s-5) * (s-6) * (s-8))
   where
     s = (5 + 6 + 8)/2
         ⇝ 9.5
⇝ sqrt (9.5 * (9.5-5) * (9.5-6) * (9.5-8))
⇝ sqrt 224.4375
⇝ 14.981238266578634
```

Tanto funções como variáveis podem ser definidas localmente. A ordem das equações locais é irrelevante. Por exemplo:

```
minhaFuncao x = 3 + f x + f a + f b
  where
    f x = x + 7*c
    a = 3*c
    b = f 2
    c = 10
```

```
minhaFuncao 5
⇝ 3 + f 5 + f a + f b
   where f x = x + 7*c
         a = 3*c
             ⇝ 3*10
             ⇝ 30
         b = f 2
             ⇝ 2 + 7*10
             ⇝ 2 + 70
             ⇝ 72
         c = 10
⇝ 3 + (5 + 7*10) + (30 + 7*10) + (72 + 7*10)
⇝ 3 + (5 + 70) + (30 + 70) + (72 + 70)
⇝ 3 + 75 + 100 + 142
⇝ 320
```

3.8 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
```

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 começa a sequência de definições.

```
a = 10 + 20 + 30 +
      40 + 50 + 60 +
      70 + 80
b = sum [10,20,30]
```

```
   a = 10 + 20 + 30 +
40 + 50 + 60 +
70 + 80
   b = sum [10,20,30]
```

```
a = 10 + 20 + 30 +
40 + 50 + 60 +
70 + 80
b = sum [10,20,30]
```

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

```haskell
-- agrupamento implícito

a = b + c
    where
      b = 1
      c = 2

d = a * 2
```

significa

```haskell
-- agrupamento explícito

a = b + c
    where { b = 1 ; c = 2 }

d = a * 2
```

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.

3.9 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

3.10 Exercícios

Nas tarefas seguintes, quando for solicitado para definir funções, elas devem ser definidas em um arquivo fonte e testadas no GHCi.

Tarefa 3.5: Encontrando os erros

Identifique e corrija os erros de sintaxe no programa fonte que se segue.

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

Tarefa 3.6

Defina uma função para calcular o quadrado do dobro do seu argumento.

Tarefa 3.7

Defina uma função para calcular o dobro do quadrado do seu argumento.

Tarefa 3.8: Lados de um triângulo

Os lados de qualquer triângulo respeitam a seguinte restrição:

> A soma dos comprimentos de quaisquer dois lados de um triângulo é superior ao comprimento do terceiro lado.

Escreva uma função que receba o comprimento de três segmentos de reta e resulte em um valor lógico indicando se satisfazem esta restrição.

Tarefa 3.9: Energia armazenada em uma mola

A força requerida para comprimir uma mola linear é dada pela equação

$$F = kx$$

onde F é a força em N (Newton), x é a compressão da mola em m (metro), e k é a constante da mola em N/m.

A energia potencial armazenada na mola comprimida é dada pela equação

$$E = \frac{1}{2}kx^2$$

onde E é a energia em J (joule).

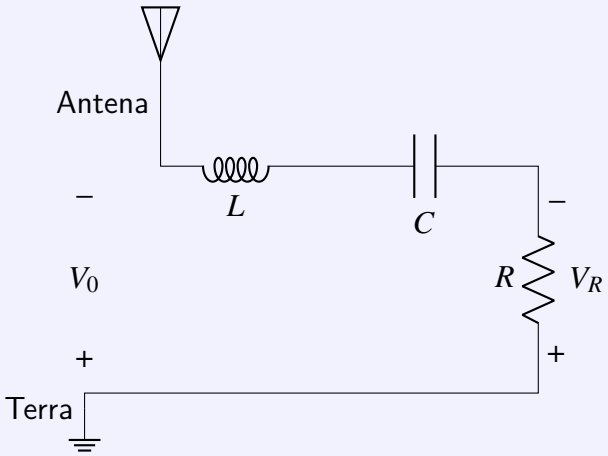
Defina funções para calcular a compressão e a energia potencial armazenada em uma mola, dadas a constante elástica da mola e a força usada para comprimi-la.

Tarefa 3.10: Custo da energia elétrica

Sabe-se que o quilowatt de energia elétrica custa um quinto do salário mínimo. Defina uma função que receba o valor do salário mínimo e a quantidade de quilowatts consumida por uma residência, e resulta no valor a ser pago com desconto de 15%.

Tarefa 3.11: Receptor de rádio

Uma versão simplificada da parte frontal de um receptor de rádio AM é apresentada na figura abaixo. Esse receptor é composto por um circuito que contém um resistor R, um capacitor C e um indutor L conectados em série. O circuito é conectado a uma antena externa e aterrado conforme mostra a figura.



O circuito permite que o rádio selecione uma estação específica dentre as que transmitem na faixa AM. Na frequência de resonância do circuito, essencialmente todo o sinal V_0 da antena vai até o resistor, que representa o resto do rádio. Em outras palavras, o rádio recebe seu sinal mais forte na frequência de ressonância. A frequência de ressonância do circuito indutor-capacitor é dada pela equação

$$f_0 = \frac{1}{2\pi\sqrt{LC}}$$

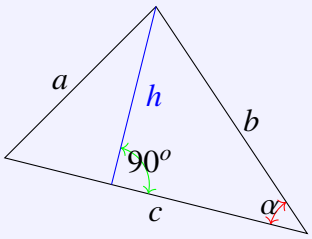
onde L é a indutância em H (henry) e C é a capcitância em F (farad).

Defina uma função que receba a indutância L e a capacitância C, e resulta na frequência de ressonância desse aparelho de rádio

Teste seu programa pelo cálculo da frequência do rádio quando $L = 0,25mH$ e $C = 0,10nF$.

Tarefa 3.12: Área de um triângulo usando relações métricas

A área de um triângulo de lados a, b e c pode ser calculada usando relações métricas em um triângulo qualquer.



Pela lei dos cossenos temos:

$$a^2 = b^2 + c^2 - 2bc\cos\alpha \implies \cos\alpha = \frac{b^2 + c^2 - a^2}{2bc}$$

Pela relação fundamental da trigonometria temos:

$$\sin^2\alpha + \cos^2\alpha = 1 \implies \sin\alpha = \sqrt{1 - \cos^2\alpha}$$

Pela definição de seno temos:

$$\sin\alpha = \frac{h}{b} \implies h = b\sin\alpha$$

Pela definição da área de um triângulo temos:

$$A = \frac{ch}{2}$$

Defina uma função para calcular a área de um triângulo de lados a, b e c usando as esquações apresentadas.

Dicas: Use variáveis locais para calcular os valores $\cos\alpha$, $\sin\alpha$ e h.