Programação Funcional em Haskell

José Romildo Malaquias

BCC222: Programação Funcional

Universidade Federal de Ouro Preto
Departamento de Computação

2016–1

SUMÁRIO

1 Paradigmas de Programação

Resumo

2 O Ambiente Interativo GHCi

Resumo

As atividades de programação serão desenvolvidas usando a linguagem Haskell (`http://www.haskell.org/`).

Nesta aula o aluno irá se familiarizar com o ambiente de programação em Haskell através da avaliação de expressões no ambiente interativo, e edição e compilação de programas. Também ele irá aprender a fazer suas primeiras definições de função.

Sumário

2.1 Haskell

Haskell é uma linguagem de programação funcional pura avançada. É um produto de código aberto de mais de vinte anos de pesquisa de ponta que permite o desenvolvimento rápido de software robusto, conciso e correto. Com um bom suporte para a integração com outras linguagens, concorrência e paralelismo integrados, depuradores, ricas bibliotecas, e uma comunidade ativa, Haskell torna mais fácil a produção de software flexível, de alta qualidade, e de fácil manutenção.

GHC (Glasgow Haskell Compiler, `http://www.haskell.org/ghc/`) é um compilador de código aberto para a linguagem Haskell, disponível para diversas plataformas, incluindo Windows e diversas variedades de Unix (como Linux, Mac OS X e FreeBSD). GHC é a implementação de Haskell mais usada.

GHC compreende um **compilador** de linha de comando (`ghc`) usado para compilar programas gerando código executável, e também um **ambiente interativo** (`GHCi`), que permite a **avaliação de expressões** de forma interativa, muito útil para testes durante o desenvolvimento.

A **Plataforma Haskell** (`http://www.haskell.org/platform/`) é um ambiente de desenvolvimento abrangente e robusto para a programação em Haskell. Ela é formada pelo compilador **GHC** (Glasgow Haskell Compiler: `http://www.haskell.org/ghc/`) e por várias bibliotecas adicionais prontas para serem usadas.

2.2 Instalação do ambiente de desenvolvimento

Para o desenvolvimento de aplicações na linguagem Haskell precisa-se minimamente de um compilador ou interpretador de Haskell, e de um editor de texto para digitação do código fonte. Bibliotecas adicionais e ambientes integrados de desenvolvimento também podem ser úteis.

São recomendados:

- GHC, o compilador de Haskell mais usado atualmente, que oferece também o ambiente interativo GHCi.

- cabal-install, um pacote que fornece a ferramenta de linha de comando `cabal` que simplifica o processo de gerenciamento de software Haskell, automatizando a busca, configuração , compilação e instalação de bibliotecas e programas Haskell.

- Atom, um editor de texto com facilidades para o desenvolvimento de programas.

Atom é um editor de texto de código livre e de código aberto desenvolvido pelo GitHub, com suporte para plugins. Atom é uma aplicação desktop construído usando tecnologias web.

2.2.1 Instalação no Windows

Plataforma Haskell para Windows

É interessante que a Plataforma Haskell seja instalada em seu computador para facilitar o desenvolvimento das atividades de programação deste curso. Ela pode ser instalada a partir do site `https://www.haskell.org/platform/`, que disponibiliza programas de instalação para algumas plataformas incluindo Windows.

A Plataforma Haskell para Windows inclui:

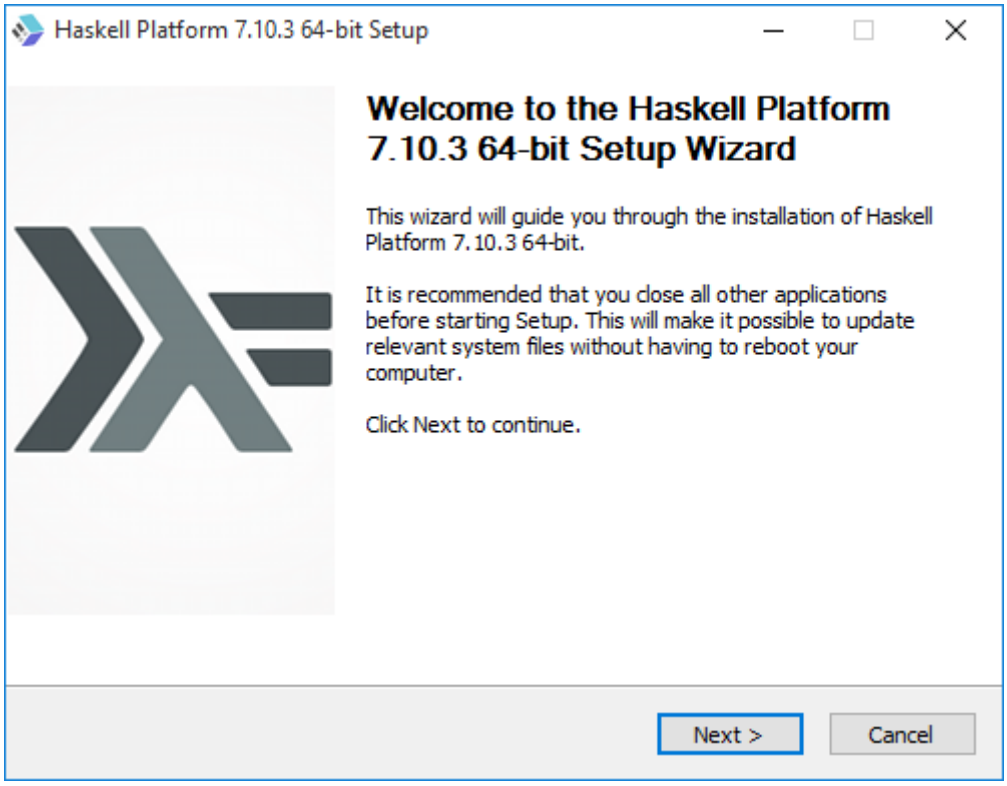
- o compilador GHC, incluindo o ambiente interativo GHCi,

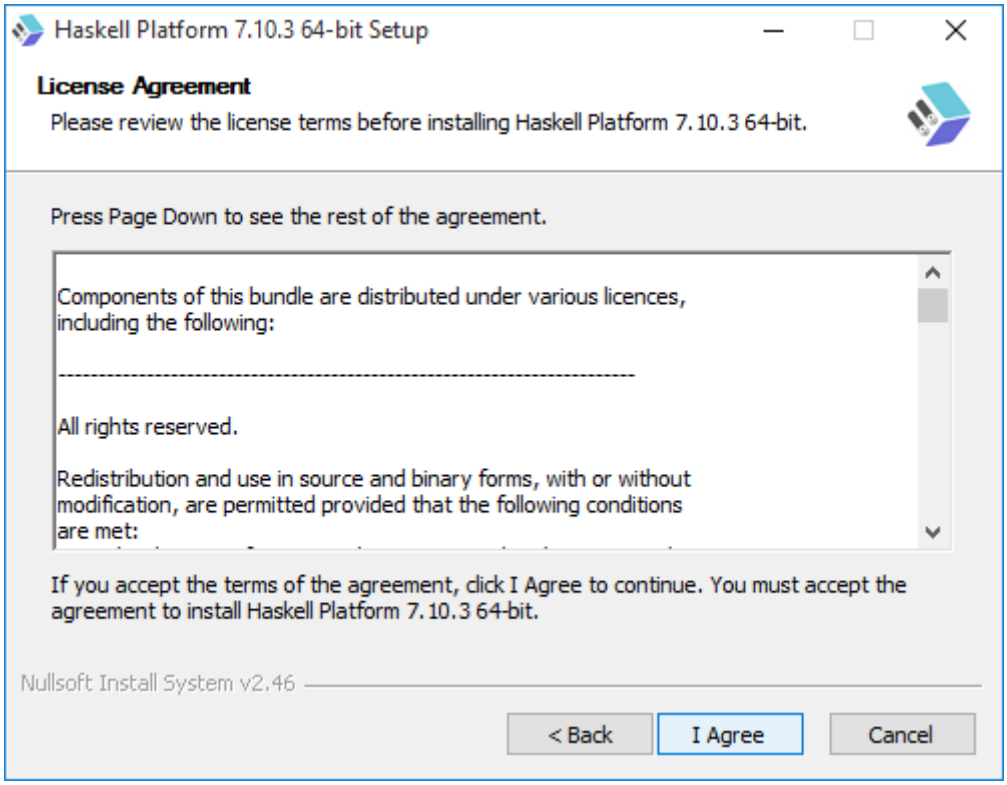
- WinGHCi, uma interface gráfica para o ambiente interativo do GHC,

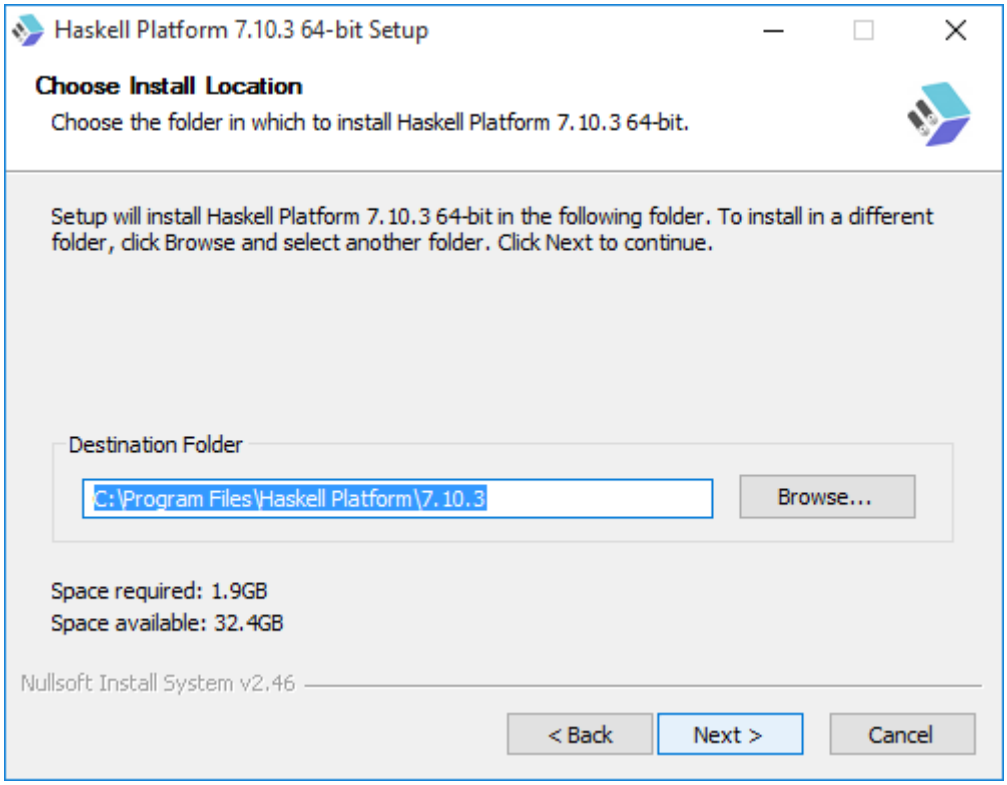
- cabal-install, e

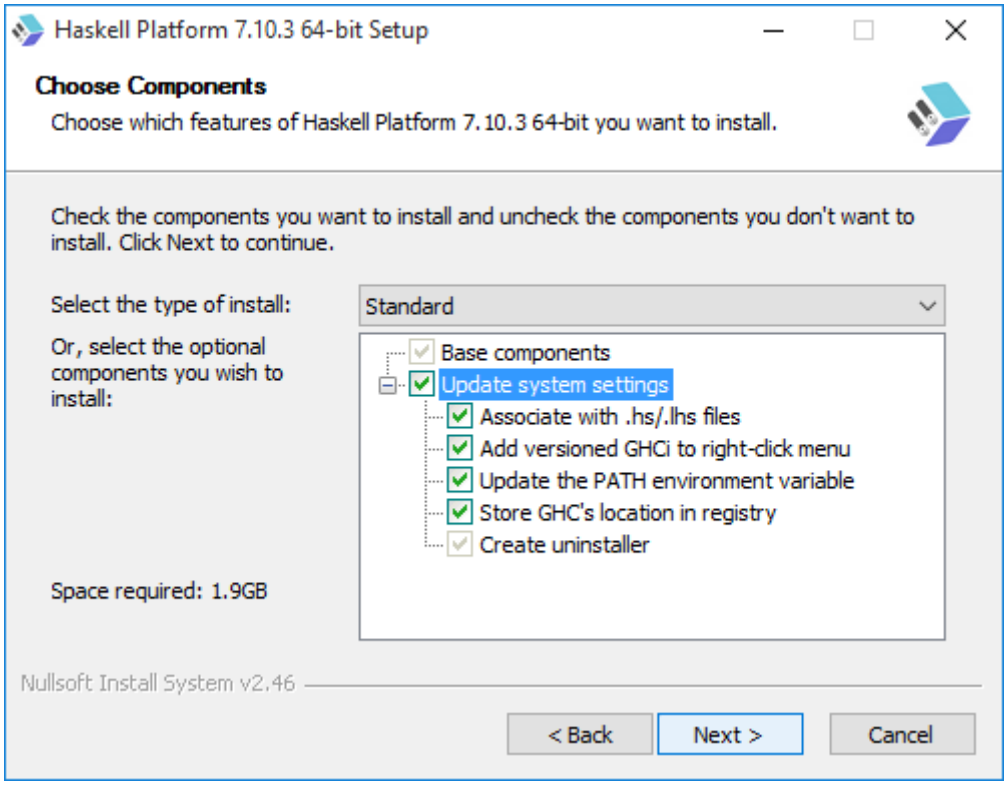
- bibliotecas adicionais.

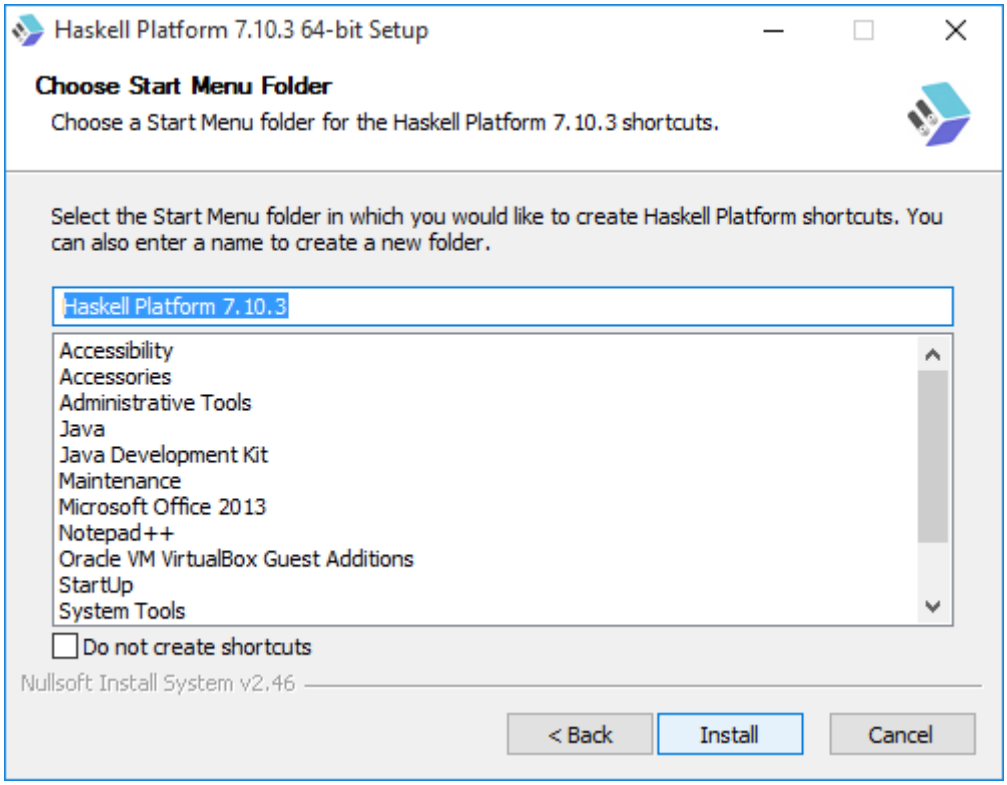
Pode-se obter o programa de instalação da Plataforma Haskell para Windows em `https://www.haskell.org/platform/#windows`. As figuras seguintes ilustram os passos para a instalação no Windows 10.

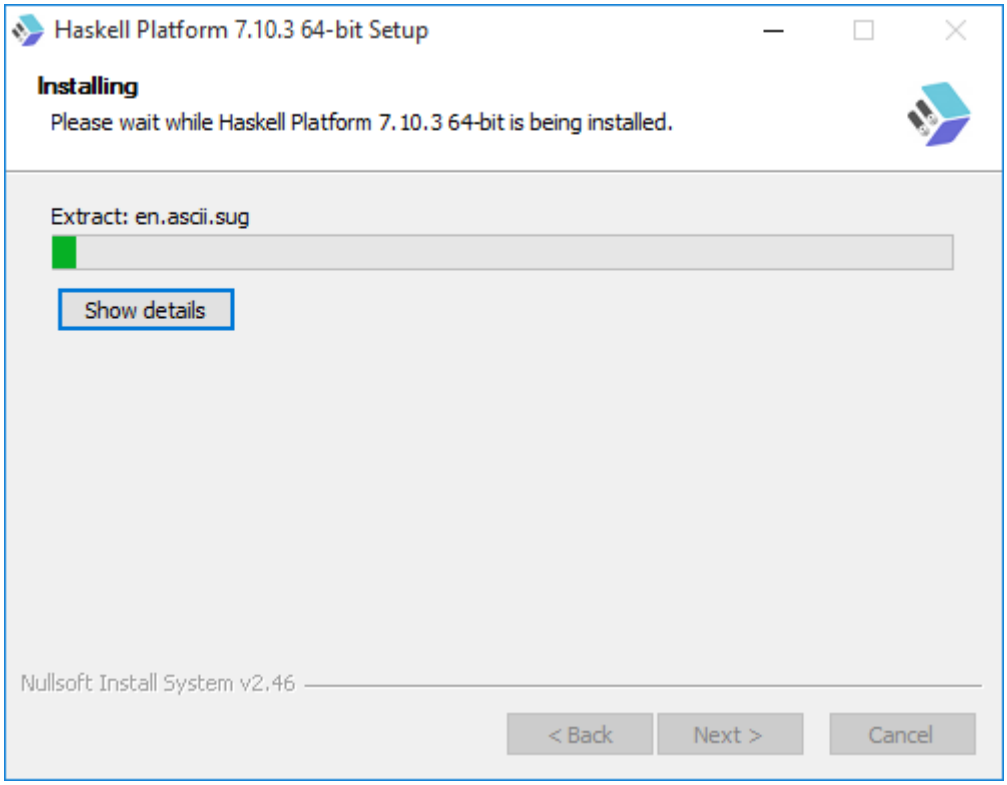



Haskell Platform 7.10.3 64-bit Setup
License Agreement
Please review the license terms before installing Haskell Platform 7.10.3 64-bit.
Press Page Down to see the rest of the agreement.
Components of this bundle are distributed under various licences,
including the following:
All rights reserved.
Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
If you accept the terms of the agreement, click I Agree to continue. You must accept the agreement to install Haskell Platform 7.10.3 64-bit.
Nullsoft Install System v2.46
< Back
I Agree
Cancel


Haskell Platform 7.10.3 64-bit Setup
Choose Install Location
Choose the folder in which to install Haskell Platform 7.10.3 64-bit.
Setup will install Haskell Platform 7.10.3 64-bit in the following folder. To install in a different folder, click Browse and select another folder. Click Next to continue.
Destination Folder
C:\Program Files\Haskell Platform\7.10.3
Browse...
Space required: 1.9GB
Space available: 32.4GB
Nullsoft Install System v2.46
< Back
Next >
Cancel

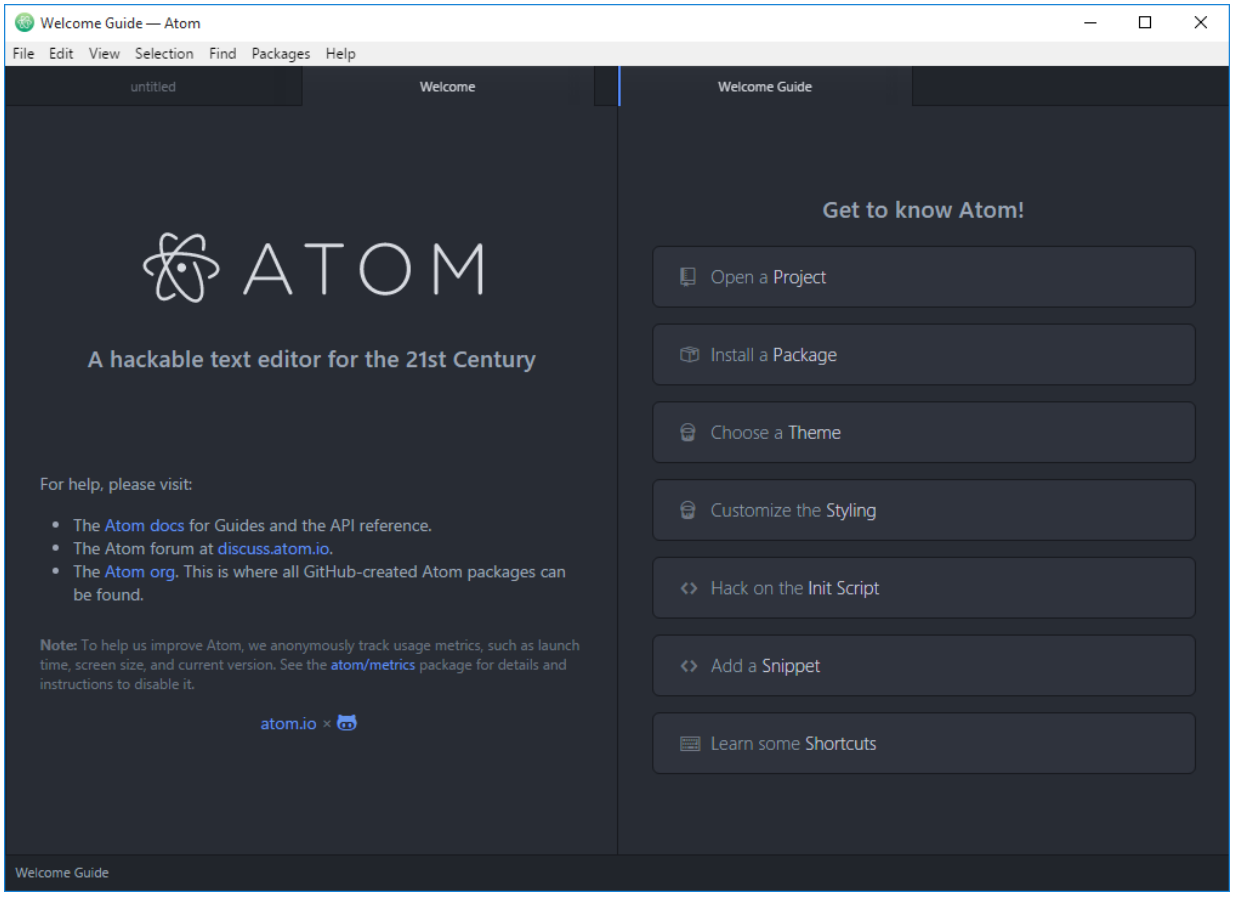

Haskell Platform 7.10.3 64-bit Setup
Choose Components
Choose which features of Haskell Platform 7.10.3 64-bit you want to install.
Check the components you want to install and uncheck the components you don't want to install. Click Next to continue.
Select the type of install:
Standard
Or, select the optional components you wish to install:
Base components
Update system settings
Associate with .hs/.lhs files
Add versioned GHCi to right-click menu
Update the PATH environment variable
Store GHC's location in registry
Create uninstaller
Space required: 1.9GB
Nullsoft Install System v2.46
< Back
Next >
Cancel


Haskell Platform 7.10.3 64-bit Setup
Choose Start Menu Folder
Choose a Start Menu folder for the Haskell Platform 7.10.3 shortcuts.
Select the Start Menu folder in which you would like to create Haskell Platform shortcuts. You can also enter a name to create a new folder.
Haskell Platform 7.10.3
Accessibility
Accessories
Administrative Tools
Java
Java Development Kit
Maintenance
Microsoft Office 2013
Notepad++
Oracle VM VirtualBox Guest Additions
StartUp
System Tools
Do not create shortcuts
Nullsoft Install System v2.46
< Back
Install
Cancel



Instalação do editor Atom no Windows

Os sites `https://atom.io` e `https://github.com/atom/atom/releases/latest` disponibilizam o instalador `AtomSetup.exe` para Windows. Ele irá instalar Atom, adicionar os comandos `atom` e `apm` no `PATH`, criar atalhos na área de trabalho e no menu iniciar, adicionar o item `Open with Atom` no menu de contexto no gerenciador de arquivos, e disponibilizar Atom para associação com arquivos usando o `Open with ...`.



Atom será atualizado automaticamente quando uma nova versão estiver disponível.
Em seguida instale algumas ferramentas necessárias aos plugins relacionados a Haskell:

- stylish-haskell,
- ghc-mod, e
- hlint.

No terminal (prompt de comandos) execute os comandos:

```
cabal update
cabal install stylish-haskell
cabal install hlint
cabal install ghc-mod
```

Finalmente instale os plugins:

- language-haskell
- haskell-ghc-mod
- ide-haskell-cabal
- ide-haskell
- autocomplete-haskell
- ide-haskell-repl

No terminal (prompt de comandos) execute os comandos:

```
apm install language-haskell haskell-ghc-mod ide-haskell-cabal ide-haskell
apm install autocomplete-haskell ide-haskell-repl
```

Em algumas situações o programa de instalação falha em configurar a variável de ambiente `%PATH%` para incluir a pasta contendo o programa `apm`. Neste caso pode-se desinstalar e em seguida reinstalar Atom e ocasionalmente a configuração funciona. Se ainda assim o comando `apm` continua indisponível, pode-se instalar os plugins a partir do próprio Atom, como é explicado em `https://zenagiwa.wordpress.com/2015/02/15/installing-packages-for-atom-on-windows/`.

2.2.2 Instalação no Ubuntu

Para instalar o GHC sugerimos usar o repositório `hvr/ghc` (que dispnibiliza a última versão) para instalar os pacotes ghc e cabal-install. No terminal execute os comandos:

```
sudo apt-get update
sudo apt-get install -y software-properties-common
sudo add-apt-repository -y ppa:hvr/ghc
sudo apt-get update
sudo apt-get install -y cabal-install-1.22 ghc-7.10.3 ghc-7.10.3-htmldocs
echo 'export PATH="$HOME/.cabal/bin:/opt/cabal/1.22/bin:/opt/ghc/7.10.3/bin:$PATH"' | \
  sudo tee -a /etc/profile.d/haskell.sh
export PATH="$HOME/.cabal/bin:/opt/cabal/1.22/bin:/opt/ghc/7.10.3/bin:$PATH"
```

Já para instalar o editor Atom sugerimos usar o repositório webupd8team/atom. No terminal execute os comandos:

```
sudo add-apt-repository -y ppa:webupd8team/atom
sudo apt-get update
sudo apt-get install -y atom
```

Em seguida instale algumas ferramentas necessárias aos plugins relacionados a Haskell:

- stylish-haskell,
- ghc-mod, e
- hlint.

No terminal (prompt de comandos) execute os comandos:

```
cabal update
cabal install stylish-haskell
cabal install hlint
cabal install ghc-mod
```

Finalmente instale os plugins:

- language-haskell
- haskell-ghc-mod
- ide-haskell-cabal
- ide-haskell
- autocomplete-haskell
- ide-haskell-repl

No terminal (prompt de comandos) execute os comandos:

```
apm install language-haskell haskell-ghc-mod ide-haskell-cabal ide-haskell
apm install autocomplete-haskell ide-haskell-repl
```

2.2.3 Configuração do Atom

Considerando que código fonte em Haskell deve ser escrito usando o conjunto de caractees Unicode, e gravado em arquivos codificados em UTF-8, e ainda que o uso de caracteres brancos (espaço, tabulador, mudança de linha) usados para fazer o layout do código são relevantes na sintaxe, recomenda-se as seguintes configurações no editor de texto, acessíveis pelo menu `Edit`→`Preferences` ou pelo atalho de teclado `Ctrl`+`Comma`:

- `File Encoding`: `utf8`
- `Show Invisibles`: *selecionado*
- `Soft Tabs`: *selecionado*
- `Tab Length`: `2`

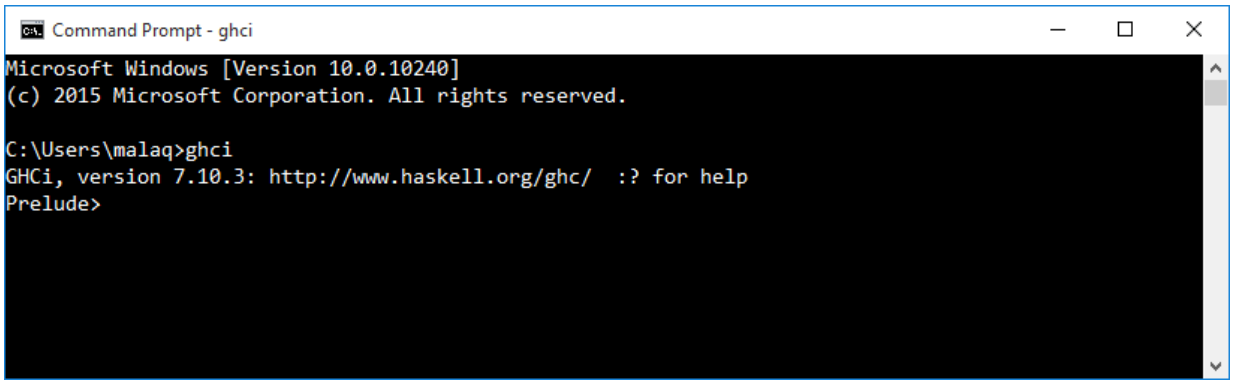
Sugestão de outros plugins interessantes:

- open-terminal-here: `https://atom.io/packages/open-terminal-here`
- open-recent: `https://atom.io/packages/open-recent`

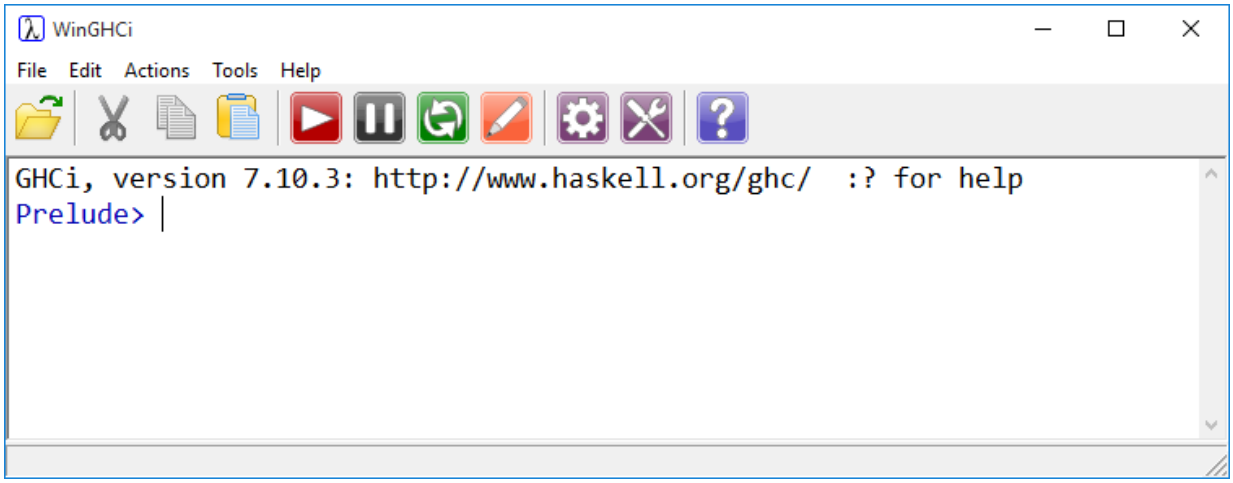
2.3 O ambiente interativo GHCi

O GHCi pode ser iniciado a partir de um terminal simplesmente digitando `ghci`. Isto é ilustrado na figura seguinte, em um sistema Unix.

No Windows pode-se iniciar o GHCi de maneira semelhante, a partir da janela *Prompt de Comandos*.



No Windows o programa `WinGHCi` é uma alternativa para executar o GHCi sem usar um terminal. Este programa tem uma janela própria, além de uma barra de ferramentas e uma barra de menus que podem facilitar algumas operações no ambiente interativo. O `WinGHCi` pode ser iniciado a partir do menu do Windows. *Prompt de Comandos*.



O **prompt** **Prelude>** significa que o sistema GHCi está pronto para avaliar expressões.

Uma aplicação Haskell é formada por um conjunto de módulos contendo definições de tipos, variáveis, funções, etc. À esquerda do prompt é mostrada a lista de módulos abertos (importados) que estão disponíveis. Um **módulo** é formado por definições que podem ser usadas em outros módulos. O módulo

Prelude da biblioteca padrão do Haskell contém várias definições básicas e é importado automaticamente tanto no ambiente interativo quanto em outros módulos.

Na configuração padrão do GHCi o prompt é formado pela lista de módulos abertos seguida do símbolo `>`.

Expressões Haskell podem ser digitadas no prompt. Elas são compiladas e avaliadas, e o seu valor é exibido. Por exemplo:

```
Prelude> 2 + 3 * 4
14
Prelude> (2 + 3) * 4
20
Prelude> sqrt (3^2 + 4^2)
5.0
```

O GHCi também aceita comandos que permitem configurá-lo. Estes comandos começam com o caracter `:` (dois-pontos).

Pode-se obter ajuda no GHCi com os comandos `:help` ou `:?`.

```
% ghci
GHCi, version 7.10.3: http://www.haskell.org/ghc/  :? for help

Prelude> :?
 Commands available from the prompt:

   <statement>                 evaluate/run <statement>
   :                           repeat last command
   :\n ..lines.. \n:\n       multiline command
   :add [*]<module> ...        add module(s) to the current target set
   :browse[!] [[*]<mod>]       display the names defined by module <mod>
                               (!: more details; *: all top-level names)
   :cd <dir>                   change directory to <dir>
   :cmd <expr>                 run the commands returned by <expr>::IO String
   :complete <dom> [<rng>] <s> list completions for partial input string
   :ctags[!] [<file>]          create tags file for Vi (default: "tags")
                               (!: use regex instead of line number)
   :def <cmd> <expr>           define command :<cmd> (later defined command has
                               precedence, ::<cmd> is always a builtin command)
   :edit <file>                edit file
   :edit                       edit last module
   :etags [<file>]             create tags file for Emacs (default: "TAGS")
   :help, :?                   display this list of commands
   :info[!] [<name> ...]       display information about the given names
                               (!: do not filter instances)
   :issafe [<mod>]             display safe haskell information of module <mod>
   :kind[!] <type>             show the kind of <type>
                               (!: also print the normalised type)
   :load [*]<module> ...       load module(s) and their dependents
   :main [<arguments> ...]     run the main function with the given arguments
   :module [+/-] [*]<mod> ...  set the context for expression evaluation
   :quit                       exit GHCi
   :reload                     reload the current module set
   :run function [<arguments> ...] run the function with the given arguments
   :script <filename>          run the script <filename>
   :type <expr>                show the type of <expr>
   :undef <cmd>                undefine user-defined command :<cmd>
   :!<command>                 run the shell command <command>
```

```
-- Commands for debugging:

  :abandon                    at a breakpoint, abandon current computation
  :back                       go back in the history (after :trace)
  :break [<mod>] <l> [<col>]  set a breakpoint at the specified location
  :break <name>               set a breakpoint on the specified function
  :continue                   resume after a breakpoint
  :delete <number>            delete the specified breakpoint
  :delete *                   delete all breakpoints
  :force <expr>               print <expr>, forcing unevaluated parts
  :forward                    go forward in the history (after :back)
  :history [<n>]              after :trace, show the execution history
  :list                       show the source code around current breakpoint
  :list <identifier>          show the source code for <identifier>
  :list [<module>] <line>     show the source code around line number <line>
  :print [<name> ...]         show a value without forcing its computation
  :sprint [<name> ...]        simplified version of :print
  :step                       single-step after stopping at a breakpoint
  :step <expr>                single-step into <expr>
  :steplocal                  single-step within the current top-level binding
  :stepmodule                 single-step restricted to the current module
  :trace                      trace after stopping at a breakpoint
  :trace <expr>               evaluate <expr> with tracing on (see :history)

-- Commands for changing settings:

  :set <option> ...           set options
  :seti <option> ...          set options for interactive evaluation only
  :set args <arg> ...         set the arguments returned by System.getArgs
  :set prog <progname>        set the value returned by System.getProgName
  :set prompt <prompt>        set the prompt used in GHCi
  :set prompt2 <prompt>       set the continuation prompt used in GHCi
  :set editor <cmd>           set the command used for :edit
  :set stop [<n>] <cmd>       set the command to run when a breakpoint is hit
  :unset <option> ...         unset options

 Options for ':set' and ':unset':

   +m            allow multiline commands
   +r            revert top-level expressions after each evaluation
   +s            print timing/memory stats after each evaluation
   +t            print type after evaluation
   -<flags>      most GHC command line flags can also be set here
                       (eg. -v2, -XFlexibleInstances, etc.)
                 for GHCi-specific flags, see User's Guide,
                 Flag reference, Interactive-mode options

-- Commands for displaying information:

  :show bindings              show the current bindings made at the prompt
  :show breaks                show the active breakpoints
  :show context               show the breakpoint context
  :show imports               show the current imports
  :show linker                show current linker state
```

```
   :show modules                 show the currently loaded modules
   :show packages                show the currently active package flags
   :show paths                   show the currently active search paths
   :show language                show the currently active language flags
   :show <setting>               show value of <setting>, which is one of
                                    [args, prog, prompt, editor, stop]
   :showi language               show language flags for interactive evaluation
```

O comando `:quit` pode ser usado para encerrar a sessão interativa no GHCi. A sessão pode ser encerrada também pela inserção do caracter de fim de arquivo `Control-Z` no Windows e `Control-D` no Linux.

Normalmente a entrada para o GHCi deve ser feita em uma única linha. Assim que a tecla **ENTER** é digitada, encerra-se a leitura. Para realizar entrada usando várias linhas, pode-se delimitá-la pelos comandos `:{` e `:}`, colocados cada um em sua própria linha. Por exemplo:

```
Prelude> :{
Prelude| 2 + 3 * 4 ^
Prelude| 5 / (8 - 7)
:}
3074.0
```

As linhas entre os delimitadores `:{` e `:}` são simplesmente unidas em uma única linha que será dada como entrada para o GHCi.

Alternativamente pode-se configurar o GHCi para usar o modo de linhas múltimas por meio do comando `:set +m`. Neste modo o GHCi detecta automaticamente quando o comando não foi finalizado e permite a digitação de linhas adicionais. Uma linha múltipla pode ser terminada com uma linha vazia. Por exemplo:

```
Prelude> :set +m

Prelude> sqrt (2 +
Prelude|    3 * 4)
3.7416573867739413
```

2.4 Bibliotecas

Os 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 o módulo deve ser **importado**. Uma **biblioteca** é formada por uma coleção de módulos relacionados.

A **biblioteca padrão** (`http://www.haskell.org/onlinereport/haskell2010/haskellpa2.html`) é formada por um conjunto de módulos disponível em tdas as implementações de Haskell. Ela contém o módulo **Prelude** (`http://www.haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html`) que é *importado automaticamente por padrão em todas* os programas em Haskell e contém tipos e funções comumente usados.

A **biblioteca padrão do GHC** (`http://www.haskell.org/ghc/docs/latest/html/libraries/index.html`) é uma versão expandida da biblioteca padrão contendo alguns módulos adicionais.

As **bibliotecas da Plataforma Haskell** (`http://www.haskell.org/platform/`) são bibliotecas adicionais incluídas na plataforma Haskell.

Hackage (`http://hackage.haskell.org/`) é uma coleção de **pacotes** contendo bibliotecas disponibilizados pela comunidade. Estes pacotes podem ser *instalados* separadamente.

Todas as definições de um módulo podem ser listadas no ambiente interativo usando o comando `:browse`. Exemplo:

```
Prelude> :browse Prelude
($!) :: (a -> b) -> a -> b
```

```
(!!) :: [a] -> Int -> a
($) :: (a -> b) -> a -> b
(&&) :: Bool -> Bool -> Bool
(++) :: [a] -> [a] -> [a]
(.) :: (b -> c) -> (a -> b) -> a -> c
(=<<) :: Monad m => (a -> m b) -> m a -> m b
data Bool = False | True
⋮
```

Tarefa 2.1

Use o ambiente interativo GHCi para avaliar todas as expressões usadas nos exemplos deste roteiro.

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 simples de expressões são construtores constantes e literais, que representam valores em sua forma mais simples, ou seja, já estão reduzidos à sua forma canônica. 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
```

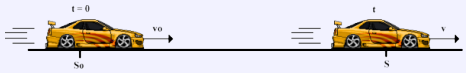
3.4 Avaliando expressões

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 $a = -9.81\,\text{m/s}^2$.

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

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 dentro de um ***script***, um arquivo texto contendo definições (de variáveis, funções, tipos, etc.).

Por convenção, *scripts* 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:

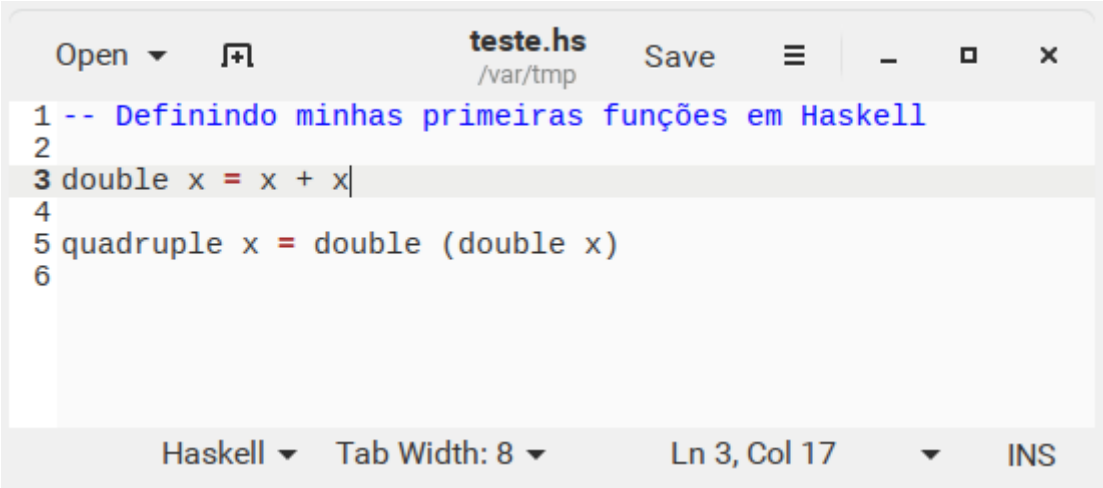
alfanuméricos

- 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'`

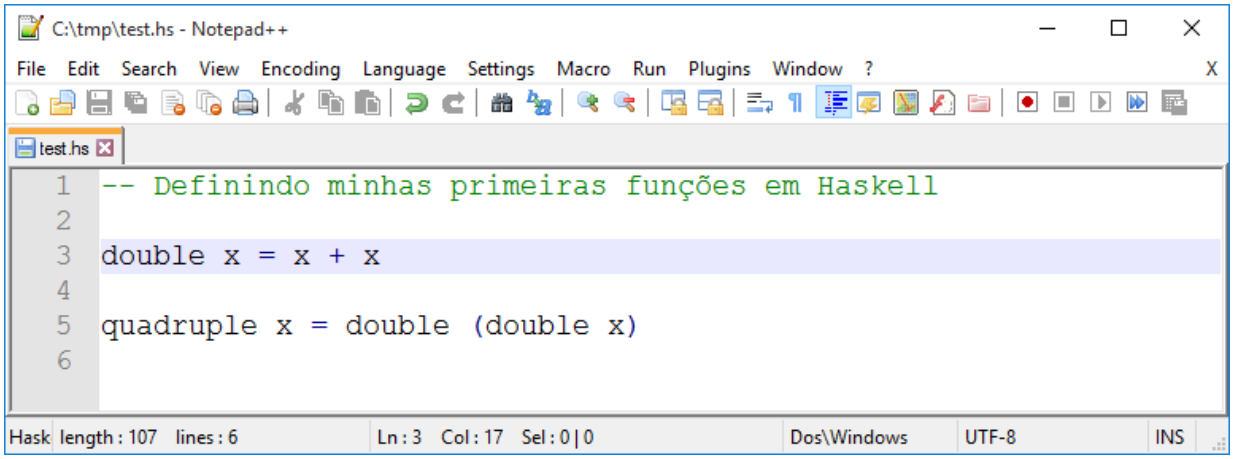
simbólicos

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

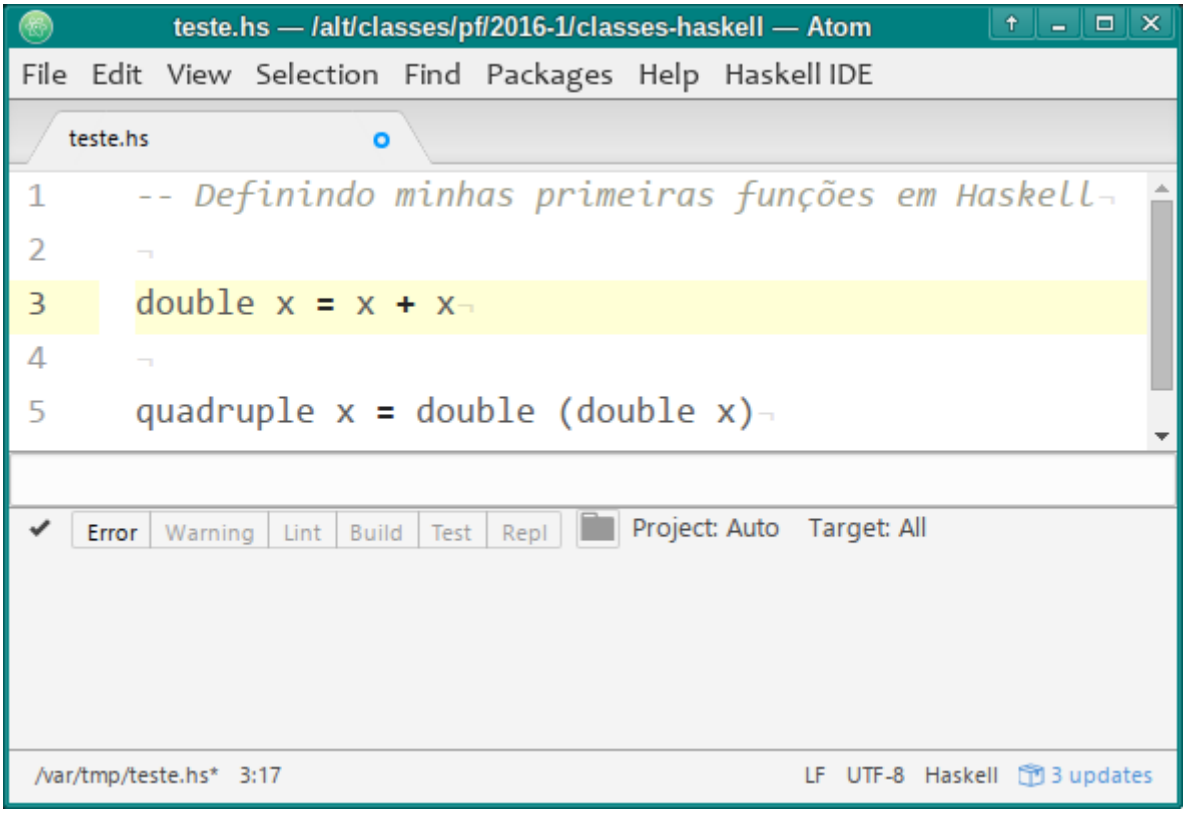
Ao desenvolver um *script* Haskell, é útil manter duas janelas abertas, uma executando um **editor de texto** para editar o *script*, e outra para o **ambiente interativo** (GHCi) em execução. No Linux há vários editores de textos que podem ser usados, como o *gedit*, ou o *kate*. Na figura seguinte vemos o *gedit*.



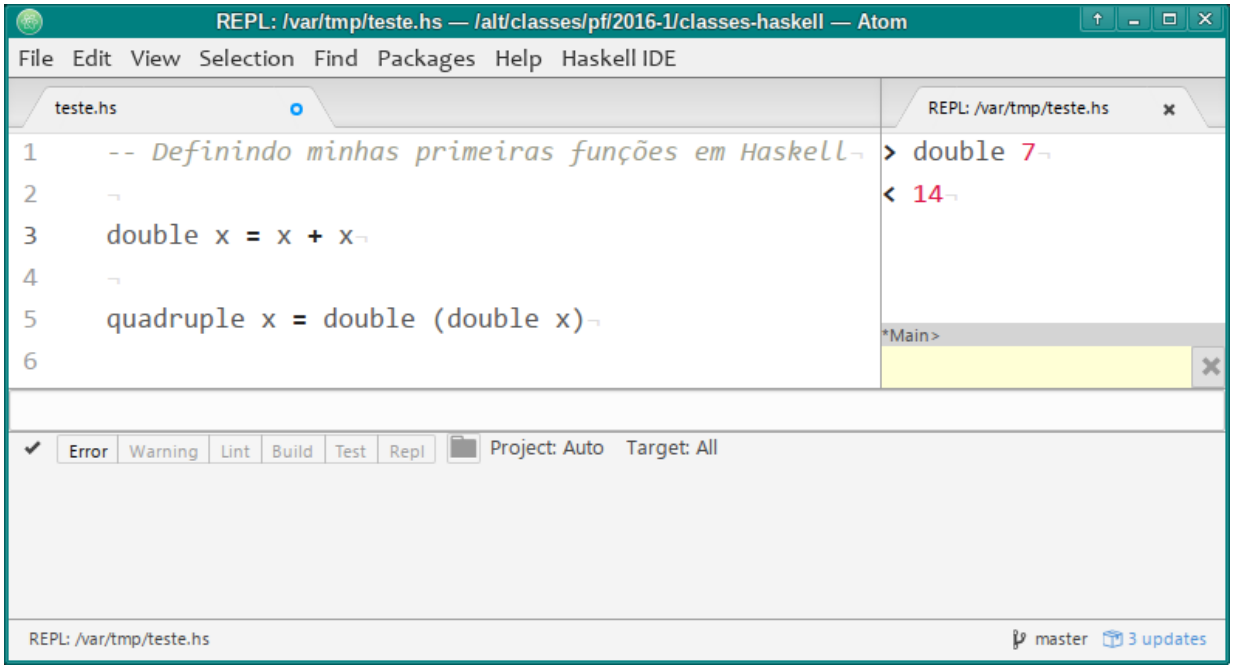
O editor de texto padrão do Windows (*Bloco de Notas*) não é recomendado, pois ele é muito precário para edição de programas. Um editor melhor é o *Notepad*++ (`http://notepad-plus-plus.org/`).



Um editor que tem agradado os desenvolvedores é Atom, disponíveis tanto para Windows como para Linux.



Com os plugins instalados Atom é capaz de marcar a sintaxe do código Haskell, completar palavras, compilar os programas usando cabal, ou avaliar expressões no ambiente interativo diretamente dentro do editor. O menu `Haskell IDE→Open REPL` carrega o arquivo no ambiente interativo integrado (REPL), onde expressões podem ser avaliadas diretamente.



Os seguintes atalho de teclado estão disponíveis no REPL:

- `Shift+Enter`: encerra uma entrada
- `Shift+Control+R`: recarrega o *script*
- `Shift+Up`: entrada anterior
- `Shift+Down`: entrada seguinte

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

Tarefa 3.3: Meu primeiro script

Inicie um editor de texto, digite as seguintes definições de função, e salve o *script* 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 *script*:

```
$ 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 *scripts* 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 script não define o nome do módulo.

Tarefa 3.4: Modificando meu primeiro script

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

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

O GHCi não detecta automaticamente que o *script* foi alterado. Assim ocomando **: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 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 `;`.

```
-- 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
```

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.

3.8 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

3.9 Definindo funções

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

Tarefa 3.5: Encontrando os erros

Identifique e corrija os erros de sintaxe no *script* 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).

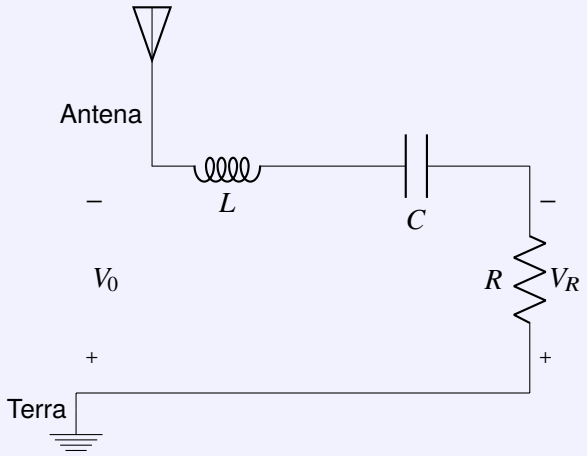
Defina funções para calcular a compressão e a energia potencial armazenada em uma mola, dadas a constante elástica da mola e a força usada para comprimi-la.

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 $C = 0,10nF$.

3.10 Soluções

Tarefa 3.1 on page 3-6: Solução

```
Prelude> let s0 = 100
Prelude> let v0 = 15.0
Prelude> let a = -9.81
Prelude> let t = 8
Prelude> let s = s0 + v0*t + 1/2*a*t^2
Prelude> s
-93.92000000000002
```

Tarefa 3.2 on page 3-6: Solução

```
Prelude> let x = 3.0
Prelude> let y = 4.0
Prelude> 4/3 * pi * sin (x^2) - 1
0.7262778741920746
Prelude> x^2 * y^3 / (x-y)^2
576.0
Prelude> 1/(x^2-y) - exp (-4*x) + (35/y)**(1/3)*(x*y)**(1/2)
7.3382693875428
Prelude> (24 + 4.5^3)/(exp 4.4 - logBase 10 12560)
1.4883284213683803
Prelude> cos (5*pi/6) * sin (7*pi/8) ^ 2 + tan (pi/6*log 8)/(sqrt 7 + 2)
0.28461655551252085
```

Tarefa 3.5 on page 3-12: Solução

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

Tarefa 3.6 on page 3-12: Solução

```
quadradoDoDobro num = (2*num)^2
```

Tarefa 3.7 on page 3-12: Solução

```
dobroDoQuadrado x = 2 * x^2
```

Tarefa 3.8 on page 3-12: Solução

```haskell
ladosTriangulo a b c =
  a + b > c &&
  a + c > b &&
  b + c > a
```

Tarefa 3.9 on page 3-13: Solução

```haskell
compressao k f = f / k

energia k x = 0.5 * k * x^2
```

Tarefa 3.10 on page 3-13: Solução

```haskell
custoEnergia salario energia =
  0.85 * energia * salario/5
```

Tarefa 3.11 on page 3-13: Solução

```haskell
freqRes l c =
  1 / (2 * pi * sqrt (l * c))

f0 = freqRes 0.25e-3 0.10e-9
```

4 Tipos de Dados

Resumo

As linguagens funcionais modernas apresentam um sistema de tipos robusto que permite ao compilador verificar se os operandos usados nas operações são consistentes. Com a inferência de tipos isto pode ser feito sem o programador ter que necessariamente anotar os tipos das variáveis e funções usadas nos programas.

Nesta aula vamos conhecer alguns tipos básicos de Haskell.

Sumário

4.1 Tipos

Um **tipo** é uma *coleção de valores* relacionados. Tipos servem para *classificar os valores* de acordo com as suas características.

Em Haskell **nomes de tipo** são sequências de letras, dígitos decimais, sublinhados e apóstrofo, *começando com uma letra* ***maiúscula***.

Por exemplo, o tipo `Bool` contém os dois valores lógicos `False` e `True`, comumente usados nas operações lógicas.

4.2 Alguns tipos básicos

A tabela 4.1 apresenta alguns tipos básicos de Haskell.

tipo	características	exemplos de valores
`Int`	– inteiros de precisão fixa – limitado (tem um valor mínimo e um valor máximo) – faixa de valores determinada pelo tamanho da palavra da plataforma	`876` `2012`
`Integer`	– inteiros de precisão arbitrária – ilimitado (qualquer número inteiro pode ser representado desde que haja memória suficiente) – menos eficiente que `Int`	`10` `754738748784003045452334209238 2`
`Float`	– aproximação de números reais em ponto flutuante – precisão simples	`4.56` `0.201E10`
`Double`	– aproximação de números reais em ponto flutuante – precisão dupla	`78643` `987.3201E-60`

`Rational`	– números racionais
	– precisão arbitrária
	– representados como uma razão de dois valores do tipo `Integer`
	– os valores podem ser construídos usando o operador `%` do módulo `Data.Ratio` (precedência 7 e associatividade à esquerda)
	`import Data.Ratio`
		`3 % 4`
		`8 % 2`
		`5 % (-10)`

`Bool`	– valores lógicos	`False` `True`

`Char`	– enumeração cujos valores representam caracteres unicode
	– estende o conjunto de caracteres ISO 8859-1 (latin-1), que é uma extensão do conjunto de caracteres ASCII
		`'B'`
		`'!'`
		`'\n'` nova linha
		`'\LF'` nova linha
		`'\^J'` nova linha
		`'\10'` nova linha
		`'\''` aspas simples
		`'\\'` barra invertida

`String`	– sequências de caracteres	`"Brasil"`
		`""`
		`"bom\ndia"`
		`"altura:\10\&199.4"`
		`"primeiro/ \ /segundo"`

Tabela 4.1: Alguns tipos básicos de Haskell.

Alguns literais são **sobrecarregados**. Isto significa que um mesmo literal pode ter mais de um tipo, dependendo do contexto em que é usado. O tipo correto do literal é escolhido pela análise desse contexto.

Em particular:

- os **literais inteiros** podem ser de qualquer tipo numérico, como `Int`, `Integer`, `Float`, `Double` ou `Rational`, e
- os **literais fracionários** podem ser de qualquer tipo numérico fraconário, como `Float`, `Double` ou `Rational`.

Por exemplo:

- o literal inteiro `2016` pode ser de qualquer tipo numérico (como `Int`, `Integer`, `Float`, `Double` ou `Rational`)

- o literal `5.61` pode ser de qualquer tipo fracionário (como `Float`, `Double` ou `Rational`).

No capítulo 5 trataremos da sobrecarga com mais detalhes.

4.3 Tipos função

Nas linguagens funcionais uma **função** é um valor de primeira classe e, assim como os demais valores, tem um tipo. Este tipo é caracterizado pelos tipos dos argumentos e pelo tipo do resultado da função.

Em Haskell um **tipo função** é escrito usando o operador de tipo `->`:

```
t1 -> ... -> tn
```

onde

- $t_1, \ldots, t_{n-1}$ são os tipos dos argumentos
- t_n é o tipo do resultado

Exemplos:

`Bool -> Bool`
: tipo das funções com um argumento do tipo `Bool`, e resultado do tipo `Bool`, como por exemplo a função `not`

`Bool -> Bool -> Bool`
: tipo das funções com dois argumentos do tipo `Bool`, e resultado do tipo `Bool`, como por exemplo as funções `(&&)` e `(||)`

`Int -> Double -> Double -> Bool`
: tipo das funções com três argumentos, sendo o primeiro do tipo `Int` e os demais do tipo `Double`, e o resultado do tipo `Bool`

4.4 Checagem de tipos

Toda expressão sintaticamente correta tem o seu tipo calculado em *tempo de compilação*. Se não for possível determinar o tipo de uma expressão ocorre um **erro de tipo** que é reportado pelo compilador.

A **aplicação** de uma função a um ou mais argumentos de *tipo inadequado* constitui um **erro de tipo**.

Por exemplo:

```
Prelude> not 'A'

<interactive>:6:5:
    Couldn't match expected type ‘Bool’ with actual type ‘Char’
    In the first argument of ‘not’, namely 'A'
    In the expression: not 'A'
    In an equation for ‘it’: it = not 'A'
```

Neste exemplo o erro ocorre porque a função `not`, cujo tipo é `Bool -> Bool`, requer um valor booleano, porém foi aplicada ao argumento `'A'`, que é um caracter.

Haskell é uma linguagem **fortemente tipada**, com um **sistema de tipos** muito avançado. Todos os possíveis *erros de tipo são encontrados em tempo de compilação* (**tipagem estática**). Isto torna os programas *mais seguros* e *mais rápidos*, eliminando a necessidade de verificações de tipo em tempo de execução.

4.5 Assinatura de tipo em definições

Ao fazer uma definição de variável ou função, o seu tipo pode ser anotado usando uma **assinatura de tipo** imediatamente antes da equação. A anotação consiste em escrever o nome e o tipo separados pelo símbolo `::`.

Por exemplo:

```
media2 :: Double -> Double -> Double
media2 x y = (x + y)/2
```

```
notaFinal :: Double
notaFinal = media2 4.5 7.2
```

```
discriminante :: Double -> Double -> Double -> Double
discriminante a b c = b^2 - 4*a*c
```

```
ladosTriangulo :: Float -> Float -> Float -> Bool
ladosTriangulo a b c = a > 0 &&
                       b > 0 &&
                       c > 0 &&
                       a < b + c &&
                       b < a + c &&
                       c < a + b
```

4.6 Assinatura de tipo em expressões

Qualquer expressão pode ter o seu tipo **anotado** junto à expressão. Se e é uma expressão e t é um tipo, então

```
e :: t
```

também é uma expressão. O tipo t especificado deve ser compatível com o tipo da expressão e. O tipo de `e::t` é t. `::` tem *precedência menor* do que todos os operadores de Haskell.

Exemplos de assinatura de tipo em expressões:

```
True                :: Bool     ⇝ True
'a'                 :: Char     ⇝ 'a'
"maria das dores"   :: String   ⇝ "maria das dores"
58                  :: Int      ⇝ 58
50 + 8              :: Double   ⇝ 58.0
4.5                 :: Rational ⇝ 9 % 2
2*(5 - 8) <= 6 + 1  :: Bool     ⇝ True
```

A assinatura de tipo pode ser necessária para resolver *ambiguidades de tipo* devido à *sobrecarga*. No capítulo 5 estudaremos com maior profundidade a questão da sobrecarga de funções, variáveis e literais.

Exemplos:

```haskell
abs (5+4::Float) * (-8)        -- 5+4 é do tipo Float
⇝ -72.0

max (2::Double) 3              -- 2 é do tipo Double
⇝ 3.0

read "34" :: Integer           -- read "34" é do tipo Integer
⇝ 34

read "34" :: Double            -- read "34" é do tipo Double
⇝ 34.0
```

Nos exemplos seguintes temos expressões mal formadas com erros de tipo:

```haskell
7 :: Char                -- 7 não pode ser do tipo Char
'F' :: Bool              -- 'F' não pode ser do tipo Bool
not True :: Float        -- (not True) não pode ser do tipo Float
min (4::Int) 5.0         -- (4::Int) e 5.0 tem tipos incompatíveis
(14::Int) + (6::Float) -- (14::Int) e (6::Float) tem tipos incompatíveis
```

4.7 Consulta do tipo de uma expressão no GHCi

No GHCi, o comando *:type* (ou de forma abreviada *:t*) calcula o *tipo* de uma expressão, sem avaliar a expressão.

Exemplos:

```
Prelude> not False
True

Prelude> :type not False
not False :: Bool

Prelude> :type 'Z'
'Z' :: Char

Prelude> :t 2*(5 - 8) <= 6 + 1
2*(5 - 8) <= 6 + 1 :: Bool

Prelude> :type not
not :: Bool -> Bool

Prelude> :t 69           -- 69 é de qualquer tipo a onde a é um tipo numérico
69 :: Num a => a

Prelude> :t 69::Integer
69::Integer :: Integer

Prelude> :t 69::Float
69::Float :: Float

Prelude> :t 69::Rational
69::Rational :: Rational
```

Tarefa 4.1: Força gravitacional

A lei da gravitação universal, proposta por Newton a partir das observações de Kepler sobre os movimentos dos corpos celestes, diz que:

> Dois corpos quaisquer se atraem com uma força diretamente proporcional ao produto de suas massas e inversamente proporcional ao quadrado da distância entre eles.

Essa lei é formalizada pela seguinte equação:

$$F = G\frac{m_1 m_2}{d^2}$$

onde:

- F é força de atração em Newtons (N),
- G é a constante de gravitação universal ($6.67 \times 10^{-11}\,\mathrm{N\,m^2/kg^2}$),
- m_1 e m_2 são as massas dos corpos envolvidos, em quilos (kg), e
- d é a distância entre os corpos em metros (m).

1. Defina uma variável para denotar a constante de gravitação universal.
2. Defina uma função que recebe as massas dos dois corpos e a distância entre eles, e resulta na força de atração entre esses dois corpos. Use a variável definida em 1.
3. Teste suas definições no ambiente interativo calculando a força de atração entre a terra e a lua sabendo que a massa da terra é 6×10^{24} kg, a massa da lua é 1×10^{23} kg, e a distância entre eles é 4×10^{5} km.

Use anotações de tipo apropriadas para os nomes sendo definidos.

Tarefa 4.2: Salário líquido

Defina uma função que recebe o salário base de um funcionário e resulta no salário líquido a receber, sabendo-se que o funcionário tem gratificação de 10% sobre o salário base e paga imposto de 7% sobre o salário base.

Use uma anotação de tipo para a função.

4.8 Soluções

Tarefa 4.1 on page 4-6: Solução

```
cteGravitacaoUniversal :: Double
cteGravitacaoUniversal = 6.67e-11

forcaGravidade :: Double -> Double -> Double -> Double
forcaGravidade m1 m2 d = cteGravitacaoUniversal * m1 * m2 / d^2
```

```
*Main> forcaGravidade 6e24 1e23 4e5
2.5012499999999998e26
```

Tarefa 4.2 on page 4-6: Solução

```
salario :: Float -> Float       -- poderia ser Double no lugar de Float
salario salBase = salBase + 0.10*salBase - 0.07*salBase
```

5 Sobrecarga

Resumo

Haskell incorpora uma forma de polimorfismo que é a sobrecarga de funções, variáveis e literais. Um mesmo identificador de função pode ser usado para designar funções computacionalmente distintas. A esta caracterísa também se chama polimorfismo *ad hoc*.

Sumário

5.1 Sobrecarga

Alungs tipos possuem operações semelhantes, porém com implementações separadas para cada tipo. Por exemplo, a comparação de igualdade pode ser feita entre dois números inteiros, ou dois números racionais, ou dois caracteres, ou dois valores lógicos, entre outros. Para cada tipo dos argumentos deve haver uma implementação da operação. Para se ter o benefício de uma interface uniforme pode ser desejável que estas operações que são semelhantes entre vários tipos **tenham o mesmo nome**.

Um mesmo nome de variável ou um mesmo nome de função pode estar associado a mais de um valor em um mesmo escopo de um programa, caracterizando a **sobrecarga**, também chamada de **polimorfismo *ad hoc***. Por exemplo o módulo `Prelude` apresenta algumas sobrecargas, como:

- o identificador `pi` é sobrecarregado e denota variáveis dos tipos numéricos com representação em ponto flutuante cujo valor é uma aproximação de π,
- o identificador `abs` é sobrecarregada e denota funções que calculam o valor absoluto, cujo argumento pode ser de qualquer tipo numérico, e cujo resultado é do mesmo tipo que o argumento,

- o operador `(/)` é sobrecarregado e denota funções de divisão fracionária com dois argumentos de qualquer tipo numérico fracionário, e resultado do mesmo tipo dos argumentos.

Para expressar a sobrecarga, Haskell usa classes de tipo. Uma **classe de tipo** é uma coleção de tipos (chamados de **instâncias** da classe) para os quais é definido um conjunto de funções (aqui chamadas de **métodos**) que podem ter diferentes implementações, de acordo com o tipo considerado.

Uma classe especifica uma **interface** indicando o *nome* e a *assinatura de tipo* de cada função. Cada tipo que é **instância** (faz parte) da classe define (implementa) as funções especificadas pela classe.

Por exemplo:

- A classe `Num` é formada por todos os tipos numéricos e sobrecarrega algumas operações aritméticas básicas, como adição. Os tipos `Int` e `Double` são instâncias da classe `Num`. Logo existe uma definição da adição para o tipo `Int` e outra para o tipo `Double`, usando algoritmos diferentes.

- A classe `Eq` é formada por todos os tipos cujos valores podem ser verificados se são iguais ou diferentes, e sobrecarrega os operadores `(==)` e `(/=)`. Logo para cada instância desta classe existe uma definição destes operadores. Todos os tipos básicos apresentados anteriormente são instâncias de `Eq`. Nenhum tipo função é instância de `Eq`, pois de forma geral não é possível comparar duas funções.

Em uma **expressão de tipo** usamos **variáveis de tipo** para denotar um tipo qualquer desconhecido, e um **contexto** para restringi-las aos tipos que são instâncias de classes específicas.

Por exemplo:

- o tipo da função `abs` é `Num a => a -> a`, ou seja, `abs` é uma função que recebe um argumento de um tipo a e resulta em um valor do mesmo tipo a, sendo a qualquer tipo que seja instância da classe `Num`

- o tipo do operador `(*)` é `Num a => a -> a -> a`, ou seja, `(*)` é uma função que recebe dois argumentos de um mesmo tipo a e resulta em um valor deste mesmo tipo a, sendo a qualquer tipo que seja instância da classe `Num`

Quando uma função sobrecarregada é usada, a escolha da implementação adequada baseia-se nos tipos dos argumentos e do resultado da função no **contexto** em que ela é usada. Semelhantemente quando uma variável sobrecarregada é usada, a escolha da implementação é feita de acordo com o contexto. Se o contexto não oferecer informação suficiente pode ser necessário fazer **anotações de tipo**.

Classes de tipos podem ser parecidas com as classes das linguagens orientadas a objetos, mas elas são realmente muito diferentes. Elas são mais parecidas com *interfaces* (como na linguagem Java, por exemplo).

Pode existir uma hierarquia de classes. Se uma classe A possuir uma **superclasse** B, os tipos que são instâncias de A também devem ser instâncias de B. Dizemos também neste caso que A é uma **subclasse** de B.

5.2 Algumas classes de tipo pré-definidas

Haskell tem várias classes predefinidas e o programador pode definir suas próprias classes. Listamos algumas classes a seguir com os seus principais métodos. Você não precisa conhecer todas elas. Use a informação para consulta quando estiver desenvolvendo em Haskell. No entanto é bom que se familiarize com `Eq`, `Ord`, `Num` e `Integral`.

5.2.1 Eq

Valores podem ser comparados quanto à igualdade e desigualdade.

Algumas instâncias

`Bool`, `Char`, `String`, `Int`, `Integer`, `Float`, `Double`, `Rational`

Alguns métodos

(==)	`Eq a => a -> a -> Bool`	igual
(/=)	`Eq a => a -> a -> Bool`	diferente

5.2.2 Ord

Valores podem ser ordenados sequencialmente.

Superclasses

`Eq`

Algumas instâncias

`Bool`, `Char`, `String`, `Int`, `Integer`, `Float`, `Double`, `Rational`

Alguns métodos

(<)	`Ord a => a -> a -> Bool`	menor que
(<=)	`Ord a => a -> a -> Bool`	menor ou igual a
(>)	`Ord a => a -> a -> Bool`	maior que
(>=)	`Ord a => a -> a -> Bool`	maior ou igual a
min	`Ord a => a -> a -> a`	menor de dois valores
max	`Ord a => a -> a -> a`	maior de dois valores

5.2.3 Enum

Valares podem ser enumerados.

Algumas instâncias

`Bool`, `Char`, `Int`, `Integer`, `Float`, `Double`, `Rational`

Alguns métodos

succ	`Enum a => a -> a`	succesor
pred	`Enum a => a -> a`	antecessor
toEnum	`Enum a => Int -> a`	converte de inteiro
fromEnum	`Enum a => a -> Int`	converte para inteiro

5.2.4 Bounded

Algumas instâncias

`Bool`, `Char`, `Int`

Alguns métodos

minBound	`Bounded a => a`	menor valor
maxBound	`Bounded a => a`	maior valor

5.2.5 Show

Valores podem ser convertidos para string.

Algumas instâncias

`Bool`, `Char`, `String`, `Int`, `Integer`, `Float`, `Double`, `Rational`

Alguns métodos

`show`	`Show a => a -> String`	converte para string

5.2.6 Read

Valores podem ser obtidos a partir de strings.

Algumas instâncias

`Bool`, `Char`, `String`, `Int`, `Integer`, `Float`, `Double`, `Rational`

Alguns métodos

`read`	`Read a => String -> a`	converte de string

5.2.7 Num

Valores numéricos.

Algumas instâncias

`Int`, `Integer`, `Float`, `Double`, `Rational`

Alguns métodos

`(+)`	`Num a => a -> a -> a`	adição
`(-)`	`Num a => a -> a -> a`	subtração
`(*)`	`Num a => a -> a -> a`	multiplicação
`negate`	`Num a => a -> a`	mudança de sinal
`abs`	`Num a => a -> a`	valor absoluto (módulo)
`signum`	`Num a => a -> a`	sinal (negativo: -1, nulo: 0, positivo: 1)
`fromInteger`	`Num a => Integer -> a`	converte de inteiro

5.2.8 Real

Números reais.

Superclasses

`Eq`, `Num`

Algumas instâncias

`Int`, `Integer`, `Float`, `Double`, `Rational`

Alguns métodos

`toRational`	`Real a => a -> Rational`	converte para racional

5.2.9 Integral

Números inteiros.

Superclasses

`Real`, `Enum`

Algumas instâncias

`Int`, `Integer`

Alguns métodos

`div`	`Integral a => a -> a -> a`	quociente da divisão inteira em direção a 0
`mod`	`Integral a => a -> a -> a`	resto da divisão inteira
`quot`	`Integral a => a -> a -> a`	quociente da divisão inteira em direação a $-\infty$
`rem`	`Integral a => a -> a -> a`	resto da divisão inteira

5.2.10 Fractional

Números fracionários.

Superclasses

`Num`

Algumas instâncias

`Float`, `Double`, `Rational`

Alguns métodos

`(/)`	`Fractional a => a -> a -> a`	divisão fracionária
`recip`	`Fractional a => a -> a`	recíproco

5.2.11 Floating

Tipos cujos valores são representados como números em ponto flutuante, com as funções trancendentais tradicionais.

Superclasses

`Fractional`

Algumas instâncias

`Float`, `Double`

Alguns métodos

`sqrt`	`Floating a => a -> a`	rais quadrada
`pi`	`Floating a => a`	π
`(**)`	`Floating a => a -> a -> a`	potenciação com expoente fracionário
`exp`	`Floating a => a -> a`	função exponencial (base e)
`log`	`Floating a => a -> a`	logatimo natural
`logBase`	`Floating a => a -> a -> a`	logaritmo
`sin`	`Floating a => a -> a`	seno
`cos`	`Floating a => a -> a`	coseno
`tan`	`Floating a => a -> a`	tangente
`asin`	`Floating a => a -> a`	arco seno
`acos`	`Floating a => a -> a`	arco coseno
`atan`	`Floating a => a -> a`	arco tangente
`sinh`	`Floating a => a -> a`	sen hiperbólico
`cosh`	`Floating a => a -> a`	coseno hiperbólico
`tanh`	`Floating a => a -> a`	tangente hiperbólica
`asinh`	`Floating a => a -> a`	arco seno hiperbólico
`acosh`	`Floating a => a -> a`	arco coseno hiperbólico
`atanh`	`Floating a => a -> a`	arco tangente hiperbólica

5.2.12 RealFrac

Tipos que implementam `Real` e `Fractional`.

Superclasses

`Real`, `Fractional`

Algumas instâncias

`Float`, `Double`, `Rational`

Alguns métodos

`truncate`	`(RealFrac a, Integral b) => a -> b`	trunca
`round`	`(RealFrac a, Integral b) => a -> b`	arredonda
`ceiling`	`(RealFrac a, Integral b) => a -> b`	arredonda para cima
`floor`	`(RealFrac a, Integral b) => a -> b`	arredonda para baixo

5.2.13 RealFloat

Tipos que implementam `RealFrac` e `Floating`, com funções para manipulação de números em ponto flutuante padronizados pela IEEE.

Superclasses

`RealFrac`, `Floating`

Algumas instâncias

`Float`, `Double`

Alguns métodos

`isNaN`	`a -> Bool`	teste para *not a number*
`isInfinite`	`a -> Bool`	teste para infinito
`isNegativeZero`	`a -> Bool`	teste para zero negativo
`atan2`	`a -> a -> a`	arco tangente

5.3 Sobrecarga de literais

Algumas formas de literais são **sobrecarregadas**: *um mesmo literal pode ser considerado de diferentes tipos*. O tipo usado pode ser decidido pelo **contexto** em que o literal é usado ou por **anotações de tipo**. Se não for possível determinar o tipo, o compilador escolhe um **tipo default**.

Literais inteiros

- Podem ser de qualquer **tipo numérico** (como `Int`, `Integer`, `Float`, `Double` e `Rational`).
- Logo o seu tipo mais geral é `Num a => a`.
- O tipo default é `Integer`.

Literais em ponto flutuante

- Podem ser de qualquer **tipo numérico fracionário** (como `Float`, `Double` e `Rational`).
- Logo o seu tipo mais geral é `Fractional a => a`.
- O tipo default é `Double`.

Exemplos:

```
187      :: Num a => a
5348     :: Num a => a
3.4      :: Fractional a => a
56.78e13 :: Fractional a => a
```

5.4 Conversão entre tipos numéricos

Devido ao sistema de tipo rígido de Haskell, não podemos converter entre os tipos numéricos arbitrariamente. Às vezes pode não ser imediatamente claro como converter de um tipo numérico para outro. A tabela 5.1 lista funções que podem ser utilizadas para converter entre os tipos mais comuns.

	`Int`	`Integer`	`Rational`	`Float`	`Double`
`Int`	`id`	`fromIntegral`	`fromIntegral`	`fromIntegral`	`fromIntegral`
`Integer`	`fromIntegral`	`id`	`fromIntegral`	`fromIntegral`	`fromIntegral`
`Rational`	`round`	`round`	`id`	`fromRational`	`fromRational`
`Float`	`round`	`round`	`toRational`	`id`	`realToFrac`
`Double`	`round`	`round`	`toRational`	`realToFrac`	`id`

Tabela 5.1: Funções para conversão entre tipos numéricos.

5.5 Exercícios

Tarefa 5.1

Defina uma função que verifica se uma equação do segundo grau

$$ax^2 + bx + c = 0$$

possui raízes reais. Para tanto é necessário que o discriminante $\Delta = b^2 - 4ac$ seja não negativo.
Determine o tipo mais geral da função e use-o em uma anotação de tipo na sua definição.

Tarefa 5.2: Avaliação de expressões

Determine o valor e o tipo das expressões seguintes caso a expressão esteja correta. Se a expressão estiver incorreta, indique qual é o problema encontrado.

1) `58 /= 58`
2) `abs == negate`
3) `False < True`
4) `"elefante" > "elegante"`
5) `min 'b' 'h'`
6) `max "amaral" "ana"`
7) `show True`
8) `show 2014`
9) `show 'A'`
10) `show "adeus"`
11) `show max`
12) `read "123"`
13) `read "123" :: Int`
14) `mod (read "123") 100`
15) `read "'@'" :: Char`
16) `read "@" :: Char`
17) `read "\"marcos\"" :: String`
18) `read "marcos" :: String`
19) `succ 'M'`
20) `fromEnum 'A'`
21) `toEnum 65 :: Char`
22) `toEnum 0`
23) `not (toEnum 0)`
24) `maxBound :: Int`

```
25) signum (-13)
26) fromInteger 99 :: Double
27) fromInteger 99 :: Rational
28) fromInteger 99
29) toRational (-4.5)
30) fromIntegral 17 :: Double
31) sin (pi/2)
32) floor (3*pi/2)
```

5.6 Inferência de tipos

Toda expressão bem formada tem um **tipo mais geral**, que pode ser calculado *automaticamente* em *tempo de compilação* usando um processo chamado **inferência de tipos**.

A capacidade de inferir tipos automaticamente *facilita a programação*, deixando o programador livre para omitir anotações de tipo ao mesmo tempo que permite a verificação de tipos pelo compilador.

A inferência de tipo é feita usando as **regras de tipagem** de cada forma de expressão.

Literais inteiros

Os literais inteiros são do tipo `Num a => a`.

Literais fracionários

Os literais fracionários são do tipo `Fractional a => a`.

Literais caracteres

Os literais caracteres são do tipo `Char`.

Literais strings

Os literais strings são do tipo `String`.

Construtores constantes

Os consrutores constantes são do tipo que os define. Assim:

- os construtores constantes booleanos `True` e `False` são do tipo `Bool`.

Aplicação de função

Em uma aplicação de função:

- o tipo do argumento deve ser compatível com o domínio da função
- o tipo do resultado deve ser compatível com o contra-domínio da função

$$\frac{\begin{array}{c} f :: a_1 \to \ldots \to a_n \to b \\ x_1 :: a_1 \\ \vdots \\ x_n :: a_n \end{array}}{f\ x_1\ \ldots\ x_n :: b}$$

5.7 Dicas e Sugestões

- Ao definir uma nova função em Haskell, é útil começar por escrever o seu tipo.
- Dentro de um *script*, é uma boa prática indicar o tipo de cada nova função definida.
- Ao indicar os tipos de funções polimórficas que usam números, igualdade, ou ordenações (ou outras restrições), tome o cuidado de incluir as restrições de classe necessárias.

Tarefa 5.3: Área do círculo

Defina uma função que recebe a medida do raio r de um círculo e resulta na área A do círculo, dada por:

$$A = \pi r^2$$

Indique o tipo mais geral da função usando uma anotação de tipo.

Tarefa 5.4: Número de degraus

Defina uma função que recebe a altura dos degraus de uma escada e a altura que o usuário deseja alcançar subindo a escada, e resulta na quantidade mínima de degraus que ele deverá subir para atingir seu objetivo, sem se preocupar com a altura do usuário.

Faça uma anotação do tipo mais geral da função.

Tarefa 5.5

Determine o tipo de cada função definida a seguir.

1) `dobro x = x*2`
2) `aprovado nota = nota >= 6`
3) `myLog x b = log x / log b`

5.8 Soluções

Tarefa 5.1 on page 5-8: Solução

```
possuiRaizesReais :: (Num a, Ord a) => a -> a -> a -> Bool

possuiRaizesReais a b c = b^2 - 4*a*c >= 0
```

Tarefa 5.2 on page 5-8: Solução

1) `58 /= 58`
   `False`
   `Bool`

2) `abs == negate`
   erro de tipo: (==) não está definido para funções

3) `False < True`
   `True`
   `Bool`

4) `"elefante" > "elegante"`
   `False`
   `Bool`

5) `min 'z' 'h'`
   `'h'`
   `Char`

6) `max "amaral" "ana"`
   `"ana"`
   `String`

7) `show True`
   `"True"`
   `String`

8) `show 2014`
   `"2014"`
   `String`

9) `show 'A'`
   `"'A'"`
   `String`

10) `show "adeus"`
    `"\"adeus\""`
    `String`

11) `show max`
    erro de tipo: show não está definida para funções

12) `read "123"`
    erro de tipo: ambiguidade

13) `read "123" :: Int`
```
123
Int
```

14) `mod (read "123") 100`
```
23
(Integral a, Read a) => a
```

15) `read "'@'" :: Char`
```
'@'
Char
```

16) `read "@" :: Char`
erro em tempo de execução: sintaxe

17) `read "\"marcos\"" :: String`
```
"marcos"
String
```

18) `read "marcos" :: String`
erro em tempo de execução: sintaxe

19) `succ 'M'`
```
'N'
Char
```

20) `fromEnum 'A'`
```
65
Int
```

21) `toEnum 65 :: Char`
```
'A'
Char
```

22) `toEnum 0`
erro de tipo: ambiguidade

23) `not (toEnum 0)`
```
True
Bool
```

24) `maxBound :: Int`
```
9223372036854775807
Int
```

25) `signum (-13)`
```
-1
Num a => a
```

26) `fromInteger 99 :: Double`
```
99.0
Double
```

27) `fromInteger 99 :: Rational`
```
99 % 1
Rational
```

28) `fromInteger 99`
erro de tipo: ambiguidade

29) `toRational (-4.5)`
`(-9) % 2`
`Rational`

30) `fromIntegral 17 :: Double`
`17.0`
`Double`

31) `sin (pi/2)`
`1.0`
`Floating a => a`

32) `floor (3*pi/2)`
`4`
`Integral a => a`

Tarefa 5.3 on page 5-10: Solução

```
areaCirculo :: Floating a => a -> a
areaCirculo r = pi * r^2
```

Tarefa 5.4 on page 5-10: Solução

```
numDegraus :: (Integral b, RealFrac a) => a -> a -> b
numDegraus alturaDegrau alturaDesejada =
  ceiling (alturaDesejada / alturaDegrau)
```

Tarefa 5.5 on page 5-10: Solução

---

1) `dobro :: Num a => a -> a`

2) `aprovado :: (Num a, Ord a) => a -> Bool`

3) `myLog :: Floating a => a -> a -> a`

6 Expressão Condicional

Resumo

Expressões condicionais permitem a escolha entre duas alternativas na obtenção do valor da expressão, com base em uma condição (expressão lógica).

Nesta aula vamos nos familiarizar com o uso de expressões condicionais.

Sumário

6.1 Expressão condicional

Uma **expressão condicional** tem a forma

```
if condição then exp1 else exp2
```

onde *condição* é uma *expressão booleana* (chamada **predicado**) e exp_1 (chamada ***consequência***) e exp_2 (chamada ***alternativa***) são expressões de um *mesmo tipo*. O *valor* da expressão condicional é o valor de exp_1 se a condição é verdadeira, ou o valor de exp_2 se a condição é falsa.

Seguem alguns exemplos de expressões condicionais e seus valores.

```
if True then 1 else 2          ⇝ 1
if False then 1 else 2         ⇝ 2
if 2>1 then "OK" else "FAIL"   ⇝ "OK"
if even 5 then 3+2 else 3-2    ⇝ 1
```

A expressão condicional é uma **expressão**, e portanto *sempre tem um valor*. Assim uma expressão condicional pode ser usada *dentro de outra expressão*. Veja os exemplos seguintes.

```
5 * (if True then 10 else 20)           ⇝ 50
5 * if True then 10 else 20             ⇝ 50
length (if 2<=1 then "OK" else "FAIL") ⇝ 4
```

Observe nos exemplos seguintes que uma expressão condicional se *extende à direita* o quanto for possível.

```
(if even 2 then 10 else 20) + 1 ⇝ 11
if even 2 then 10 else 20 + 1   ⇝ 10
```

A cláusula `else` de uma expressão condicional ***não é opcional***. Omiti-la é um *erro de sintaxe*. Se fosse possível omiti-la, qual seria o valor da expressão quando a condição fosse falsa? Não teria nenhum valor neste caso, o que seria um problema. Assim uma expressão condicional *sempre* deve ter as duas alternativas. Por exemplo, a seguinte expressão apresenta um erro de sintaxe, pois foi omitida a cláusula `else`.

```
if True then 10 ⇝ ERRO DE SINTAXE
```

Regra de inferência de tipo

$$\frac{\begin{array}{c} test :: \texttt{Bool} \\ e_1 :: a \\ e_2 :: a \end{array}}{\texttt{if } test \texttt{ then } e_1 \texttt{ else } e_2 :: a}$$

Observe que a *consequência e a alternativa devem ser do mesmo tipo*, que também é o *tipo do resultado*.

Exemplos no ambiente interativo:

```
Prelude> :type if 4>5 then 'S' else 'N'
if 4>5 then 'S' else 'N' :: Char

Prelude> :type if odd 8 then 10 else 20
if odd 8 then 10 else 20 :: Num a => a

Prelude> :type if mod 17 2 == 0 then 12 else 5.1
if mod 17 2 == 0 then 12 else 5.1 :: Fractional a => a
```

```
Prelude> if fromEnum 'A' then "ok" else "bad"
<interactive>:2:4:
    Couldn't match expected type ‘Bool’ with actual type ‘Int’
    In the return type of a call of ‘fromEnum’
    In the expression: fromEnum 'A'
    In the expression: if fromEnum 'A' then "ok" else "bad"

Prelude> if mod 17 2 /= 0 then not True else 'H'
<interactive>:7:37:
    Couldn't match expected type ‘Bool’ with actual type ‘Char’
    In the expression: 'H'
    In the expression: if mod 17 2 /= 0 then not True else 'H'
    In an equation for ‘it’:
        it = if mod 17 2 /= 0 then not True else 'H'
```

Tarefa 6.1

Determine o valor e o tipo das expressões seguintes caso a expressão esteja correta. Se a expressão estiver incorreta, indique qual é o problema encontrado.

a) `if sqrt (abs (10 - 35) * 100) < 5 then "aceito" else "negado"`

b) `if pred 'B' then 10 else 20`

c) `if odd 1 then sqrt 9 else pred 'B'`

d) `4 * if 'B' < 'A' then 2 + 3 else 2 - 3`

e) `signum (if 'B' < 'A' then 2 + 3 else 2) - 3`

6.2 Definição de função com expressão condicional

Como na maioria das linguagens de programação, *funções* podem ser *definidas usando expressões condicionais*. Por exemplo, a função para calcular o valor absoluto de um número inteiro pode ser definida como segue:

```
valorAbsoluto :: Int -> Int
valorAbsoluto n = if n >= 0 then n else -n
```

`valorAbsoluto` recebe um inteiro `n` e resulta em `n` se ele é não-negativo, e `-n` caso contrário.

Expressões condicionais podem ser *aninhadas*, como mostra o exemplo a seguir onde é definida uma função para determinar o sinal de um número inteiro.

```
sinal :: Int -> Int
sinal n = if n < 0
            then -1
            else if n == 0
                   then 0
                   else 1
```

Tarefa 6.2: Maior de três valores

Defina uma função `max3` que recebe três valores e resulta no maior deles. Use expressões condicionais aninhadas.

Faça uma anotação de tipo para a função em seu código, usando o tipo mais geral da mesma.

Teste sua função no ambiente interativo.

6.3 Equações com guardas

Funções podem ser definidas através de **equações com guardas**, onde uma sequência de expressões lógicas, chamadas **guardas**, é usada para *escolher entre vários possíveis resultados*.

Uma **equação com guarda** é formada por uma *sequência de cláusulas* escritas logo após a lista de argumentos. Cada **cláusula** é introduzida por uma barra vertical (`|`) e consiste em uma *condição*, chamada **guarda**, e uma expressão (*resultado*), separados por `=`.

$$
\begin{array}{l}
f\ arg_1\ \ldots\ arg_n \\
\qquad \mid guarda_1 = exp_1 \\
\qquad \vdots \\
\qquad \mid guarda_m = exp_m
\end{array}
$$

Observe que:

- cada guarda deve ser uma *expressão lógica*, e
- os resultados devem ser todos do *mesmo tipo*.

Quando a função é aplicada, as guardas são verificadas na *sequência* em que foram escritas. A *primeira* guarda verdadeira define o resultado.

Como exemplo, considere uma função para calcular o valor absoluto de um número:

```
vabs :: (Num a, Ord a) => a -> a
vabs n | n >= 0 = n
       | n < 0  = - n
```

Nesta definição de `abs`, as guardas são

- `n >= 0`
- `n < 0`

e as expressões associadas são respectivamente

- `n`
- `-n`

Veja um exemplo de aplicação da função:

```
vabs 89
  ?? 89 >= 0
  ?? ⇝ True
⇝ 89
```

Observe que quando o cálculo do valor de uma expressão é escrito passo a passo, indicamos o cálculo das guardas separadamente em linhas que começam com ??.

Veja outra exemplo de aplicação da função:

```
vabs (75 - 2*50)
  ?? 75 - 2*50 >= 0
  ?? ⇝ 75 - 100 >= 0
  ?? ⇝ -25 >= 0
  ?? ⇝ False
  ?? -25 < 0
  ?? ⇝ True
⇝ - (-25)
⇝ 25
```

Note que o argumento `(75 - 2*50)` é avaliado uma única vez, na primeira vez em que ele é necessário (para verificar a primeira guarda). O seu valor não é recalculado quando o argumento é usado novamente na segunda guarda ou no resultado. Esta é uma característica da **avaliação *lazy***:

> Um argumento é avaliado somente se o seu valor for necessário, e o seu valor é guardado caso ele seja necessário novamente.

Logo um argumento nunca é avaliado mais de uma vez.

Observe que na definição de `vabs` o teste `n < 0` pode ser substituído pela constante `True`, pois ele somente será usado se o primeiro teste `n >= 0` for falso, e se isto acontecer, com certeza `n < 0` é verdadeiro:

```
vabs n | n >= 0 = n
       | True   = -n
```

A condição `True` pode também ser escrita como `otherwise`:

```
vabs n | n >= 0    = n
       | otherwise = -n
```

`otherwise` é uma condição que captura todas as outras situações que ainda não foram consideradas. `otherwise` é definida no prelúdio simplesmente como o valor *verdadeiro*:

```
otherwise :: Bool
otherwise  = True
```

Equações com guardas podem ser usadas para tornar definições que envolvem *múltiplas condições* mais *fáceis de ler*, como mostra o exemplo a seguir para determinar o sinal de um número inteiro:

```
sinal :: Int -> Int
sinal n | n < 0     = -1
        | n == 0    = 0
        | otherwise = 1
```

Como outro exemplo temos uma função para análisa o índice de massa corporal:

```
analisaIMC :: (Fractional a, Ord a) => a -> String
analisaIMC imc
  | imc <= 18.5 = "Você está abaixo do peso, seu emo!"
  | imc <= 25.0 = "Você parece normal. Deve ser feio!"
  | imc <= 30.0 = "Você está gordo! Perca algum peso!"
  | otherwise   = "Você está uma baleia. Parabéns!"
```

Uma definição pode ser feita com várias equações. Se todas as guardas de uma equação forem falsas, a *próxima equação* é considerada. Se não houver uma próxima equação, ocorre um *erro em tempo de execução*. Por exemplo:

```
minhaFuncao :: (Ord a, Num b) => a -> a -> b
minhaFuncao x y | x > y = 1
                | x < y = -1
```

```
minhaFuncao 2 3 ⇝ -1
minhaFuncao 3 2 ⇝ 1
minhaFuncao 2 2 ⇝ ERRO
```

Tarefa 6.3: Menor de três valores

Dada a definição de função

```
min3 a b c | a < b && a < c = a
           | b < c          = b
           | otherwise      = c
```

mostre o cálculo passo a passo das expressões:

a) `min3 2 3 4`

b) `min3 5 (4-3) 6`

c) `min3 (div 100 5) (2*6) (4+5)`

Tarefa 6.4

Redefina a função a seguir usando guardas no lugar de expressões condicionais.

```
describeLetter :: Char -> String
describeLetter c =
  if c >= 'a' && c <= 'z'
    then "Lower case"
    else if c >= 'A' && c <= 'Z'
           then "Upper case"
           else "Not an ASCII letter"
```

6.4 Soluções

Tarefa 6.1 on page 6-2: Solução

---

1) `if sqrt (abs (10 - 35) * 100) < 5 then "aceito" else "negado"`
   `"negado"`
   `String`

2) `if pred 'B' then 10 else 20`
   erro de tipo: o test do `if` é do tipo `Char` e deveria ser do tipo `Bool`

3) `if odd 1 then sqrt 9 else pred 'B'`
   erro de tipo: as alternativas do `if` têm tipos incompatíveis: `Floating a => a` e `Char`

4) `4 * if 'B' < 'A' then 2 + 3 else 2 - 3`
   `-4`
   `Num a => a`

5) `signum (if 'B' < 'A' then 2 + 3 else 2) - 3`
   `-2`
   `Num a => a`

Tarefa 6.2 on page 6-3: Solução

```
-- solução usando expressões condicionais
max3 :: Ord a => a -> a -> a -> a
max3 n1 n2 n3 =
  if n1 > n2 && n1 > n3
    then n1
    else if n2 > n3
           then n2
           else n3

-- solução sem usar expressões condicionais
max3' :: Ord a => a -> a -> a -> a
max3' n1 n2 n3 =
  max n1 (max n2 n3)
```

Tarefa 6.3 on page 6-5: Solução

```
min3 2 3 4
  ?? 2 < 3 && 2 < 4
  ?? ⇝ True && 2 < 4
  ?? ⇝ True && True
  ?? ⇝ True
⇝ 2
```

```
min3 5 (4-3) 6
  ?? 5 < (4-3) && 5 < 6
  ?? ⇝ 5 < 1 && 5 < 6
  ?? ⇝ False && 5 < 6
  ?? ⇝ False
  ?? 1 < 6
  ?? ⇝ True
⇝ 1
```

```
min3 (div 100 5) (2*6) (4+5)
  ?? div 100 5 < 2*6 && div 100 5 < 4+5
  ?? ⇝ 20 < 2*6 && 20 < 4+5
  ?? ⇝ 20 < 12 && 20 < 4+5
  ?? ⇝ False && 20 < 4+5
  ?? ⇝ False
  ?? 12 < 4+5
  ?? ⇝ 12 < 9
  ?? ⇝ False
  ?? otherwise
  ?? ⇝ True
⇝ 9
```

Tarefa 6.4 on page 6-5: Solução

```
describeLetter :: Char -> String
describeLetter c
  | c >= 'a' && c <= 'z' = "Lower case"
  | c >= 'A' && c <= 'Z' = "Upper case"
  | otherwise            = "Not an ASCII letter"
```

7 Definições Locais

Resumo

Nesta aula vamos aprender a fazer definições locais a uma equação usando a cláusula `where`, e declarações locais a uma expressão usando a expressão `let`.

Sumário

7.1 Definições locais a uma equação

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 locais** à equação, ou seja, o **escopo** dos nomes definidos em uma cláusula where restringe-se à equação contendo a cláusula where, podendo ser usados:

- nas guardas da equação principal (quando houver)
- nos resultados (expressões que ocorrem no lado direito) da equação principal
- nas próprias definições locais da cláusula where

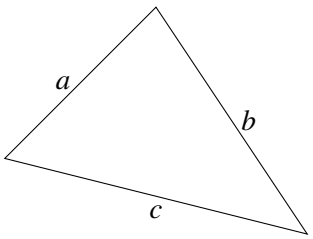
Por exemplo, considere a fórmula de Heron

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

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.




```
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
```

Uma definição mais elaborada para esta função somente calcula a área se os argumentos forem lados de um triângulo (um lado deve ser positivo e menor do que a soma dos outros dois lados):

```
areaTriangulo a b c
  | possivel  = sqrt (s * (s-a) * (s-b) * (s-c))
  | otherwise = 0
  where
    s = (a + b + c)/2
    possivel = a > 0 && b > 0 && c > 0 &&
               a < b + c &&
               b < a + c &&
               c < a + b
```

Veja outro exemplo de definição local:

```
g x y | x <= 10 = x + a
      | x >  10 = x - a
  where
    a = (y+1)^2
```

O escopo de a inclui os dois possíveis resultados determinados pelas guardas.

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

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

```
h 5 ⇝ 320
```

O próximo exemplo mostra uma função para análise do índice de massa corporal.

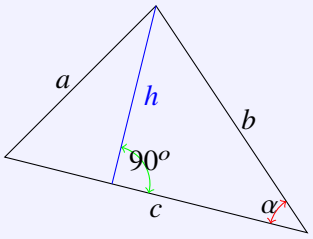
```
analisaIMC peso altura
  | imc <= 18.5 = "Você está abaixo do peso, seu emo!"
  | imc <= 25.0 = "Você parece normal. Deve ser feio!"
  | imc <= 30.0 = "Você está gordo! Perca algum peso!"
  | otherwise   = "Você está uma baleia. Parabéns!"
  where
    imc = peso / altura^2
```

Ou ainda:

```haskell
analisaIMC peso altura
  | imc <= magro  = "Você está abaixo do peso, seu emo!"
  | imc <= normal = "Você parece normal. Deve ser feio!"
  | imc <= gordo  = "Você está gordo! Perca algum peso!"
  | otherwise     = "Você está uma baleia. Parabéns!"
  where
    imc    = peso / altura^2
    magro  = 18.5
    normal = 25.0
    gordo  = 30.0
```

Tarefa 7.1: 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. Caso a , b e c não possam ser lados de um triângulo a função deve resultar em zero.

Dica Faça definições locais para os valores $\cos\alpha$, $\sin\alpha$ e h.

Tarefa 7.2: Número de raízes reais da equação do segundo grau

Defina uma função chamada `numRaizes` que recebe os três coeficientes de uma equação do segundo grau

$$ax^2 + bx + c = 0$$

e calcula a quantidade de raízes reais distintas da equação. Assuma que a equação é não degenerada (isto é, o coeficiente do termo de grau dois não é zero).

Use uma definição local para calcular o discriminante da equação.

$$\Delta = b^2 - 4ac$$

Se Δ for positivo a equação tem duas reais reais e distintas, se for nulo, a equação tem uma raiz real, e se for negativo, a equação não tem raízes reais.

Especifique o tipo mais geral da função.

Tarefa 7.3: Notas de um estudante

A nota final de um estudante é calculada a partir de três notas atribuídas respectivamente a um trabalho de laboratório, a uma avaliação semestral e a um exame final. A média ponderada das três notas mencionadas obedece aos pesos a seguir:

nota	peso
trabalho de laboratório	2
avaliação semestral	3
exame final	5

O programa a seguir, que está incompleto, recebe as três notas e determina e exibe o conceito obtido pelo aluno usando a tabela:

média ponderada	conceito
[8.0 – 10.0]	A
[7.0 – 8.0[	B
[6.0 – 7.0[	C
[5.0 – 6.0[	D
[0.0 – 5.0[	E

Considera-se que as notas digitadas são válidas.

```haskell
module Main where

import System.IO (hSetBuffering, stdout, BufferMode(NoBuffering))

main :: IO ()
main =
  do hSetBuffering stdout NoBuffering
     putStr "Digite a nota do trabalho de laboratório ...: "
     laboratório <- readLn
     putStr "Digite a nota da avaliação semestral .......: "
     semestral <- readLn
     putStr "Digite a nota do exame final ..............: "
     final <- readLn
     putStrLn ""
     putStr "Conceito obtido: "
     putStrLn [conceito laboratório semestral final]

conceito :: Float -> Float -> Float -> Char
-- complete a definição da função
```

Exemplo de execução da aplicação

```
*Main> main
Digite a nota do trabalho de laboratório ...: 7.8
Digite a nota da avaliação semestral .......: 8.0
Digite a nota do exame final ..............: 4.9

Conceito obtido: C
```

Você deve completar a definição do função `conceito`. Use uma definição local para calcular a média, e guardas para selecionar uma das cinco alternativas.

7.2 Definições locais a uma expressão

Também é possível fazer definições locais a uma expressão escrevendo-se uma expressão let. Uma **expressão let** é formada por uma lista de definições mutuamente recursivas, e por um corpo (que é uma expressão), introduzidos pelas palavras chave **let** e **in**:

```
let definições in expressão
```

O **escopo** dos nomes definidos em uma expressão let restringe-se à própria expressão let, podendo ser usados:

- no corpo da expressão let, e
- nas próprias definições locais da expressão let

O **tipo** de uma expressão let é o tipo do seu corpo. O **valor** de uma expressão let é o valor do seu corpo, calculado em um contexto que inclui os nomes introduzidos nas definições locais.

Veja alguns exemplos:

```
let x = 3+2 in x^2 + 2*x - 4                    ⇝ 31
let x = 3+2 ; y = 5-1 in x^2 + 2*x - y          ⇝ 31
let quadrado x = x*x in quadrado 5 + quadrado 3 ⇝ 34
```

Sintaticamente uma expressão let se estende à direita tanto quanto for possível, como é ilustrado a seguir:

```
4 * let x = 5-2 in x * x    ⇝ 36
(let x = 5-2 in x * x) ^ 2  ⇝ 81
```

O exemplo seguinte usa uma expressão let na definição de uma função que calcula a área da superfície de um cilindro, dados o raio da base e a altura do cilindro:

```
areaSuperfCilindro r h = let areaLado = 2 * pi * r * h
                             areaBase = pi * r^2
                         in areaLado + 2*areaBase
```

Esta função também pode ser definida usando uma cláusula where:

```
areaSuperfCilindro r h = areaLado + 2*areaBase
  where
    areaLado = 2 * pi * r * h
    areaBase = pi * r^2
```

7.3 Regra de layout em definições locais

Quando há duas ou mais definições locais, elas podem ser escritas em diferentes estilos.

Na notação básica as definições são separadas por ponto-e-vírgula (`;`) e escritas entre chaves (`{` e `}`). Por exemplo:

```
f x y = (a+1)*(b+2)
  where { a = (x+y)/2; b = (x+y)/3 }
```

Algumas vezes as chaves podem ser omitidas:

```
f x y = (a+1)*(b+2)
  where a = (x+y)/2; b = (x+y)/3
```

Às vezes até os ponto-e-vírgulas podem ser omitidos:

```
f x y = (a+1)*(b+2)
  where a = (x+y)/2
        b = (x+y)/3
```

Neste último exemplo foi usada a regra de *layout*, que dispensa os símbolos `;`, `{` e `}` mas exige que todas as definições comecem na mesma coluna e, quando continuarem em linhas subsequentes, a continuação deve começar em uma coluna mais à direita.

Veja o mesmo exemplo usando uma expressão let sem a regra de *layout* e com a regra de *layout*:

```
f x y = let { a = (x+y)/2; b = (x+y)/3 }
        in (a+1)*(b+2)
```

```
f x y = let a = (x+y)/2
            b = (x+y)/3
        in (a+1)*(b+2)
```

7.4 Diferenças entre `let` e `where`

- Com **where** as definições são colocadas no final, e com **let** elas são colocadas no início.
- **let** é uma expressão e pode ser usada em qualquer lugar onde se espera uma expressão.
- Já ***where*** *não é uma expressão*, podendo ser usada apenas para fazer definições locais em uma equação (definição de variável ou função).

7.5 Soluções

Tarefa 7.1 on page 7-3: Solução

```
areaTri a b c = c*h/2
  where
    cosAlpha = (b^2 + c^2 - a^2)/(2*b*c)
    sinAlpha = sqrt (1 - cosAlpha^2)
    h = b*sinAlpha
```

Tarefa 7.2 on page 7-4: Solução

```
numRaizes :: (Num a, Ord a, Num b) => a -> a -> a -> b
numRaizes a b c | delta > 0  = 2
                | delta == 0 = 1
                | otherwise  = 0
  where delta = b^2 - 4*a*c
```

Tarefa 7.3 on page 7-4: Solução

```
conceito :: Float -> Float -> Float -> Char
conceito notaLaboratorio notaSemestral notaFinal
  | media >= 8 = 'A'
  | media >= 7 = 'B'
  | media >= 6 = 'C'
  | media >= 5 = 'D'
  | otherwise  = 'E'
  where
    media = (2*notaLaboratorio + 3*notaSemestral + 5*notaFinal)/10
```

8 Funções Recursivas

Resumo

Definições recursivas são comuns na programação funcional. Nesta aula vamos aprender a definir funções recursivas.

Sumário

8.1 Recursividade

Recursividade é uma idéia *inteligente* que desempenha um papel central na *programação funcional* e na *ciência da computação* em geral. **Recursividade** é o mecanismo de programação no qual *uma definição* de função ou de outro objeto *refere-se ao próprio objeto* sendo definido. Assim **função recursiva** é uma função que é definida em termos de si mesma. São sinônimos: recursividade, recursão, recorrência.

Recursividade é o mecanismo básico para *repetições* nas linguagens funcionais.

Estratégia para a definição recursiva de uma função:

1. *dividir* o problema em problemas menores do mesmo tipo
2. *resolver* os problemas menores (dividindo-os em problemas ainda menores, se necessário)
3. *combinar* as soluções dos problemas menores para formar a solução final

Ao dividir o problema sucessivamente em problemas menores eventualmente os casos simples são alcançados:

- não podem ser mais divididos
- suas soluções são definidas explicitamente

De modo geral, uma **definição de função recursiva** é dividida em duas partes:

- Há um ou mais **casos base** que dizem o que fazer em situações simples, onde não é necessária nenhuma recursão.
  - Nestes casos *a resposta pode ser dada de imediato*, sem chamar recursivamente a função sendo definida.
  - Isso garante que a recursão eventualmente pode *parar*.
- Há um ou mais **casos recursivos** que são mais gerais, e definem a função em termos de uma *chamada* mais simples *a si mesma*.

Como exemplo de função recursiva, considere o cálculo do fatorial de um número natural. A função que calcula o fatorial de um número natural pode ser definida recursivamente como segue:

```
fatorial :: Integer -> Integer
fatorial n
  | n == 0 = 1
  | n > 0  = fatorial (n-1) * n
```

Nesta definição:

- A primeira guarda estabelece que o fatorial de 0 é 1. Este é o *caso base*.
- A segunda guarda estabelece que o fatorial de um número positivo é o produto deste número e do fatorial do seu antecessor. Este é o *caso recursivo*.

Observe que no caso recursivo o subproblema `fatorial (n-1)` é *mais simples* que o problema original `fatorial n` e está mais próximo do caso base `fatorial 0`.

Aplicando a função fatorial:

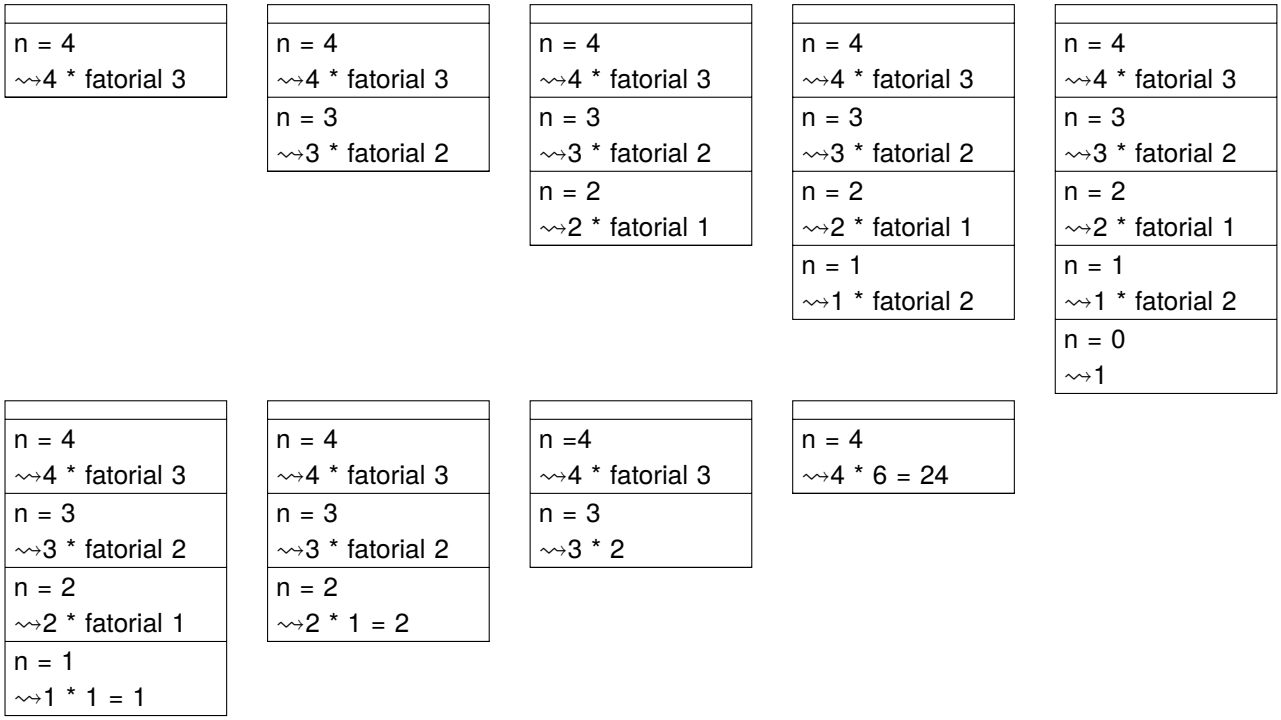
```
fatorial 4
⇝ fatorial 3 * 4
⇝ (fatorial 2 * 3) * 4
⇝ ((fatorial 1 * 2) * 3) * 4
⇝ (((fatorial 0 * 1) * 2) * 3) * 4
⇝ (((1 * 1) * 2) * 3) * 4
⇝ ((1 * 2) * 3) * 4
⇝ (2 * 3) * 4
⇝ 6 * 4
⇝ 24
```

Em muitas implementações de linguagens de programação uma *chamada de função* usa um espaço de memória (chamado de **quadro**, ***frame*** ou **registro de ativação**) em uma área da memória (chamada **pilha** ou *stack*) onde são armazenadas informações importantes, tais como:

- argumentos da função
- variáveis locais
- variáveis temporárias
- endereço de retorno da função

Normalmente para cada nova chamada de função que é realizada, um novo quadro é alocado na pilha. Quando a função retorna, o quadro é desalocado.

De maneira bastante simplificada a sequência de alocação de quadros na pilha de execução na chamada da função `fatorial 6` do exemplo anterior é indicada nas figuras seguintes.



Tarefa 8.1: Aplicando `fatorial`

Digite a função `fatorial` em um arquivo fonte Haskell e carregue-o no ambiente interativo de Haskell.

a) *Mostre* que `fatorial 7 = 5040` usando uma sequência de passos de simplificação.

b) Determine o valor da expressão `fatorial 7` usando o ambiente interativo.

c) Determine o valor da expressão `fatorial 1000` usando o ambiente interativo. Se você tiver uma calculadora científica, verifique o resultado na calculadora.

d) Qual é o valor esperado para a expressão `div (fatorial 1000) (fatorial 999)`? Determine o valor desta expressão usando o ambiente interativo.

e) O que acontece ao se calcular o valor da expressão `fatorial (-2)`?

Vejamos outro exemplo. A função que calcula a potência de dois (isto é, a base é dois) para números naturais pode ser definida recursivamente como segue:

```
potDois :: Integer -> Integer
potDois n
  | n == 0 = 1
  | n > 0  = 2 * potDois (n-1)
```

Nesta definição:

- A primeira cláusula estabelece que $2^0 = 1$. Este é o *caso base*.
- A segunda cláusula estabelece que $2^n = 2 \times 2^{n-1}$, sendo $n > 0$. Este é o *caso recursivo*.

Observe que no caso recursivo o subproblema `potDois (n-1)` é *mais simples* que o problema original `potDois n` e está mais próximo do caso base `potDois 0`.

Aplicando a função potência de dois:

```
potDois 4
⇝ 2 * potDois 3
⇝ 2 * (2 * potDois 2)
⇝ 2 * (2 * (2 * potDois 1))
⇝ 2 * (2 * (2 * (2 * potDois 0)))
⇝ 2 * (2 * (2 * (2 * 1)))
⇝ 2 * (2 * (2 * 2))
⇝ 2 * (2 * 4)
⇝ 2 * 8
⇝ 16
```

Tarefa 8.2

Considere a seguinte definição para a função potência de dois:

```
potDois' :: Integer -> Integer
potDois' n
  | n == 0    = 1
  | otherwise = 2 * potDois' (n-1)
```

O que acontece ao calcular o valor da expressão `potDois' (-5)`?

Vejamos mais um exemplo de definição recursiva. A multiplicação de inteiros está disponível na biblioteca como uma operação primitiva por questões de eficiência. Porém ela pode ser definida usando adição e recursividade em um de seus argumentos:

```
mul :: Int -> Int -> Int
mul m n
  | n == 0    = 0
  | n > 0     = m + mul m (n-1)
  | otherwise = negate (mul m (negate n))
```

Nesta definição:

- A primeira cláusula estabelece que quando o multiplicador é zero, o produto também é zero. Este é o *caso base*.
- A segunda cláusula estabelece que $m \times n = m + m \times (n-1)$, sendo $n > 0$. Este é um *caso recursivo*.
- A terceira cláusula estabelece que $m \times n = -(m \times (-n))$, sendo $n < 0$. Este é outro *caso recursivo*.

Aplicando a função de multiplicação:

```
% mul 7 (-3)
⇝ negate (mul 7 (negate (-3)))
⇝ negate (mul 7 3)
⇝ negate (7 + mul 7 2)
⇝ negate (7 + (7 + mul 7 1))
⇝ negate (7 + (7 + (7 + mul 7 0)))
⇝ negate (7 + (7 + (7 + 0)))
⇝ negate (7 + (7 + 7))
⇝ negate (7 + 14)
⇝ negate 21
⇝ -21
```

A definição recursiva da multiplicação formalisa a idéia de que a multiplicação pode ser reduzida a adições repetidas.

Tarefa 8.3

Mostre que `mul 5 6 = 30`.

Vamos analisar outro exemplo de função recursiva: a sequência de Fibonacci. Na seqüência de Fibonacci

$$0, 1, 1, 2, 3, 5, 8, 13, \ldots$$

os dois primeiros elementos são 0 e 1, e cada elemento subseqüente é dado pela soma dos dois elementos que o precedem na seqüência. A função a seguir calcula o n-ésimo número de Fibonnaci, para $n \geq 0$:

```
fib :: Int -> Int
fib n
  | n == 0 = 0
  | n == 1 = 1
  | n > 1  = fib (n-2) + fib (n-1)
```

Nesta definição:

- A primeira e a segunda cláusulas são os *casos base*.
- A terceira cláusula é o *caso recursivo*.

Neste caso temos **recursão múltipla**, pois a função sendo definida é usada mais de uma vez em sua própria definição.

Aplicando a função de fibonacci:

```
fib 5
⇝ fib 3 + fib 4
⇝ (fib 1 + fib 2) + (fib 2 + fib 3)
⇝ (1 + (fib 0 + fib 1)) + ((fib 0 + fib 1) + (fib 1 + fib 2))
⇝ (1 + (0 + 1)) + ((0 + 1) + (1 + (fib 0 + fib 1)))
⇝ (1 + 1) + (1 + (1 + (0 + 1)))
⇝ 2 + (1 + (1 + 1))
⇝ 2 + (1 + 2)
⇝ 2 + 3
⇝ 5
```

Tarefa 8.4

Mostre que `fib 6 = 8`.

8.2 Recursividade mútua

Recursividade mútua ocorre quando duas ou mais funções são definidas em termos uma da outra. Para exemplificar vamos definir as funções par e ímpar usando recursividade mútua As funções da biblioteca `even` (par) e `odd` (ímpar), que determinam se um número é par ou ímpar, respectivamente, geralmente são definidas usando o resto da divisão por dois:

```
par, impar :: Int -> Bool

par n = mod n 2 == 0

impar n = not (par n)
```

No entanto elas também podem ser definidas usando recursividade mútua:

```haskell
par :: Int -> Bool
par n | n == 0    = True
      | n > 0     = impar (n-1)
      | otherwise = par (-n)

impar :: Int -> Bool
impar n | n == 0    = False
        | n > 0     = par (n-1)
        | otherwise = impar (-n)
```

Nestas definições observamos que:

- Zero é par, mas não é ímpar.
- Um número positivo é par se seu antecessor é ímpar.
- Um número positivo é ímpar se seu antecessor é par.
- Um número negativo é par (ou ímpar) se o seu oposto for par (ou ímpar).

Aplicando as função `par` e `impar`:

```haskell
par (-5)
⇝ par 5
⇝ impar 4
⇝ par 3
⇝ impar 2
⇝ par 1
⇝ impar 0
⇝ False
```

8.3 Recursividade de cauda

Uma função recursiva apresenta **recursividade de cauda** se o *resultado final* da chamada recursiva é o resultado final da própria função. Se o resultado da chamada recursiva deve ser processado de alguma maneira para produzir o resultado final, então a função não apresenta recursividade de cauda.

Por exemplo, a função recursiva a seguir não apresenta recursividade de cauda:

```haskell
fatorial :: Integer -> Integer
fatorial n
  | n == 0 = 1
  | n > 0  = fatorial (n-1) * n
```

No caso recursivo, o resultado da chamada recursiva `fatorial (n-1)` é multiplicado por `n` para produzir o resultado final.

A função recursiva a seguir também não apresenta recursividade de cauda:

```haskell
par :: Integer -> Bool
par n
  | n == 0 = True
  | n > 0  = not (par (n-1))
```

No caso recursivo, a função `not` é aplicada ao resultado da chamada recursiva `par (n-1)` para produzir o resultado final.

Já a função recursiva `pdois'` a seguir apresenta recursividade de cauda:

```haskell
pdois :: Integer -> Integer
pdois n = pdois' n 1

pdois' :: Integer -> Integer -> Integer
pdois' n y
  | n == 0 = y
  | n > 0  = pdois' (n-1) (2*y)
```

No caso recursivo, o resultado da chamada recursiva `pdois' (n-1) (2*y)` é o resultado final.

Tarefa 8.5

Mostre que `pdois 5 = 32`.

Tarefa 8.6

Faça uma definição recursiva da função par usando recursividade de cauda.

Otimização de chamada de cauda

Uma **chamada de cauda** acontece quando uma função chama outra função como sua *última ação*, não tendo mais nada a fazer além desta última ação. O *resultado final* da função é dado pelo resultado da chamada de cauda. Em tais situações o programa não precisa voltar para a função que chama quando a função chamada termina. Portanto, após a chamada de cauda, o programa não precisa manter qualquer informação sobre a função chamadora na pilha.

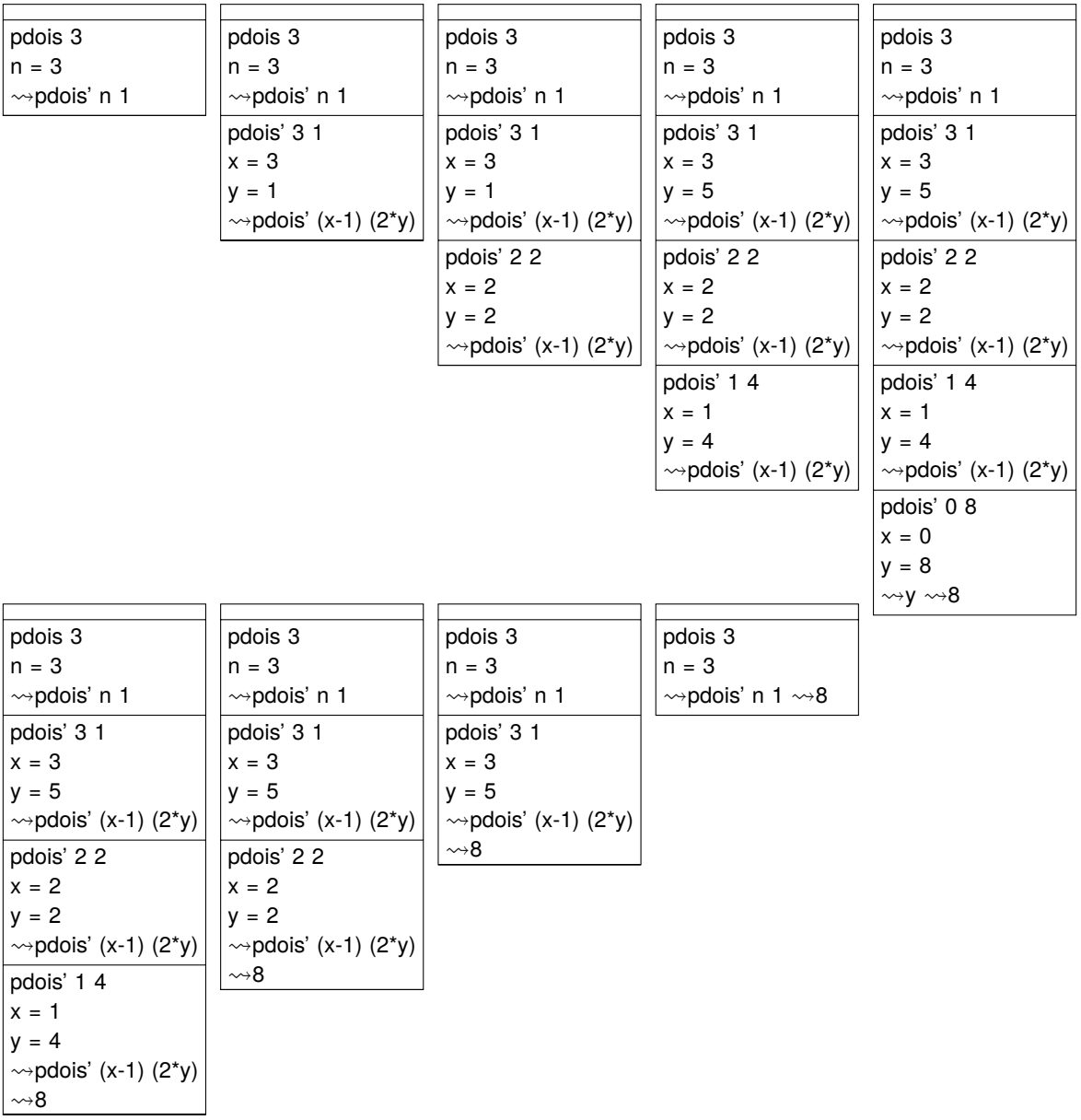
Algumas implementações de linguagem tiram proveito desse fato e na verdade não utilizam qualquer espaço extra de pilha quando fazem uma chamada de cauda, pois o espaço na pilha da própria função chamadora é reutilizado, já que ele não será mais necessário para a função chamadora. Esta técnica é chamada de **eliminação da cauda**, **otimização de chamada de cauda** ou ainda **otimização de chamada recursiva**. A **otimização de chamada de cauda** permite que funções com recursividade de cauda recorram indefinidamente *sem estourar a pilha*.

Considere novamente as funções `pdois` e `pdois'`.

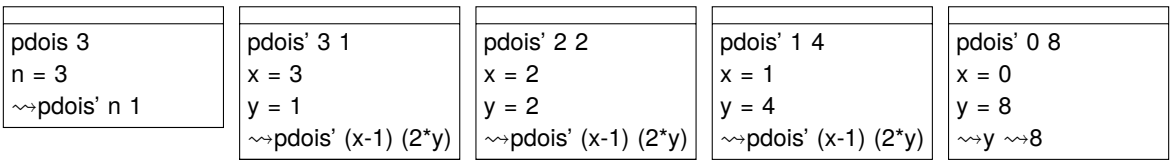
```haskell
pdois :: Integer -> Integer
pdois n = pdois' n 1

pdois' :: Integer -> Integer -> Integer
pdois' n y
  | n == 0 = y
  | n > 0  = pdois' (n-1) (2*y)
```

A figura a seguir ilustra a situação da pilha de execução na avaliação da expressão `pdois 3`, sem otimização de chamada de cauda.



Já com otimização de chamada de cauda, as chamadas recursivas reaproveitam o quadro da função chamadora, não havendo necessidade de crescimento da pilha.



Estruturas de repetição

Muitas linguagens funcionais não possuem *estruturas de repetição* e usam funções recursivas para fazer repetições. Nestes casos a otimização de chamada de cauda é fundamental para uma boa eficiência dos programas.

Em particular a linguagem Haskell não possui estruturas de repetição.

8.4 Vantagens da recursividade

A recursividade permite que:

- muitas funções possam ser naturalmente definidas em termos de si mesmas,

- propriedades de funções definidas usando recursão podem ser provadas usando **indução**, uma técnica matemática simples, mas poderosa.

8.5 Exercícios

Tarefa 8.7: Fatorial duplo

O fatorial duplo de um número natural n é o produto de todos os números de 1 (ou 2) até n, contados de 2 em 2. Por exemplo, o fatorial duplo de 8 é $8 \times 6 \times 4 \times 2 = 384$, e o fatorial duplo de 7 é $7 \times 5 \times 3 \times 1 = 105$.

Defina uma função para calcular o fatorial duplo usando recursividade.

Tarefa 8.8: Multiplicação em um intervalo

Defina uma função recursiva que recebe dois números naturais m e n, e retorna o produto de todos os números no intervalo $[m, n]$:

$$m \times (m+1) \times \cdots \times (n-1) \times n$$

Tarefa 8.9: Fatorial

Usando a função definida no exercício 8.8, escreva uma definição não recursiva para calcular o fatorial de um número natural.

Tarefa 8.10: Adição

Defina uma função recursiva para calcular a soma de dois números naturais, sem usar os operadores `+` e `-`. Utilize as funções `succ` e `pred` da biblioteca, que calculam respectivamente o sucessor e o antecessor de um valor.

Tarefa 8.11: Potenciação

Defina uma função recursiva para calcular a potência de um número, considerando que o expoente é um número natural. Utilize o método das multiplicações sucessivas:

$$x^n = \underbrace{x \times x \times \ldots \times x}_{n \text{ vezes}}$$

Tarefa 8.12: Raiz quadrada inteira

A raiz quadrada inteira de um número inteiro positivo n é o maior número inteiro cujo quadrado é menor ou igual a n. Por exemplo, a raiz quadrada inteira de 15 é 3, e a raiz quadrada inteira de 16 é 4.

Defina uma função recursiva para calcular a raiz quadrada inteira.

Tarefa 8.13: Quociente e resto da divisão inteira

Defina duas funções recursivas que calculam o quociente e o resto da divisão inteira de dois números naturais usando subtrações sucessivas.

Tarefa 8.14: Máximo divisor comum

Defina uma função recursiva para calcular o máximo divisor comum de dois números inteiros não negativos a e b, usando o algoritmo de Euclides:

$$\text{mdc}(a,b) = \begin{cases} a & \text{se } b = 0, \\ \text{mdc}(b, a \bmod b) & \text{se } b > 0, \\ \text{mdc}(a, -b) & \text{se } b < 0 \end{cases}$$

Nota: o prelúdio já tem a função `gcd :: Integral a => a -> a -> a` que calcula o máximo divisor comum de dois números inteiros.

Tarefa 8.15: Fatorial

Considere a seguinte função para calcular o fatorial de um número:

```
fat n = fat' n 1
  where
    fat' n x
      | n == 0 = x
      | n > 0  = fat' (n-1) (n*x)
```

a) Mostre que `fat 6 = 720`.

b) Compare o cálculo de `fat 6` com o cálculo de `fatorial 6` apresentado anteriormente. Qual versão da função fatorial é mais eficiente: `fatorial` ou `fat`? Explique.

Tarefa 8.16: Sequência de Fibonacci

Defina uma função com recursividade de cauda para calcular o n-ésimo ($n \geq 0$) número de Fibonacci.

8.6 Soluções

Tarefa 8.1 on page 8-3: Solução

---

a)
```
fatorial 7
  ⇝ fatorial 6 * 7
  ⇝ (fatorial 5 * 6) * 7
  ⇝ ((fatorial 4 * 5) * 6) * 7
  ⇝ (((fatorial 3 * 4) * 5) * 6) * 7
  ⇝ ((((fatorial 2 * 3) * 4) * 5) * 6) * 7
  ⇝ (((((fatorial 1 * 2) * 3) * 4) * 5) * 6) * 7
  ⇝ ((((((fatorial 0 * 1) * 2) * 3) * 4) * 5) * 6) * 7
  ⇝ ((((((1 * 1) * 2) * 3) * 4) * 5) * 6) * 7
  ⇝ (((((1 * 2) * 3) * 4) * 5) * 6) * 7
  ⇝ ((((2 * 3) * 4) * 5) * 6) * 7
  ⇝ (((6 * 4) * 5) * 6) * 7
  ⇝ ((24 * 5) * 6) * 7
  ⇝ 120 * 6 * 7
  ⇝ 720 * 7
  ⇝ 5040
```

b)
```
*Main> fatorial 7
5040
```

c)
```
*Main> fatorial 1000
402387260077093773543702433923003985719374864210714632543799910429938
512398629020592044208486969404800479988610197196058631666872994808558
901323829669944590997424504087073759918823627727188732519779505950995
276120874975462497043601418278094646496291056393887437886487337119181
045825783647849977012476632889835955735432513185323958463075557409114
262417474349347553428646576611667797396668820291207379143853719588249
808126867838374559731746136085379534524221586593201928090878297308431
392844403281231558611036976801357304216168747609675871348312025478589
320767169132448426236131412508780208000261683151027341827977704784635
868170164365024153691398281264810213092761244896359928705114964975419
909342221566832572080821333186116811553615836546984046708975602900950
537616475847728421889679646244945160765353408198901385442487984959953
319101723355556602139450399736280750137837615307127761926849034352625
200015888535147331611702103968175921510907788019393178114194545257223
865541461062892187960223838971476088506276862967146674697562911234082
439208160153780889893964518263243671616762179168909779911903754031274
622289988005195444414282012187361745992642956581746628302955570299024
324153181617210465832036786906117260158783520751516284225540265170483
304226143974286933061690897968482590125458327168226458066526769958652
682272807075781391858178889652208164348344825993266043367660176999612
831860788386150279465955131156552036093988180612138558600301435694527
224206344631797460594682573103790084024432438465657245014402821885252
470935190620929023136493273497565513958720559654228749774011413346962
715422845862377387538230483865688976461927383814900140767310446640259
899490222221765904339901886018566526485061799702356193897017860040811
889729918311021171229845901641921068884387121855646124960798722908519
296819372388642614839657382291123125024186649353143970137428531926649
875337218940694281434118520158014123344828015051399694290153483077644
569099073152433278288269864602789864321139083506217095002597389863554
277196742822248757586765752344220207573630569498825087968928162753848
```

```
8633969099598262809561214509948717012445164612603790293091208890869420
2851064018215439945715680594187274899809425474217358240106367740459574
1785160829230135358081840096996372524230560855903700624271243416909004
1536901059339838357779394109700277534720000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000
```

d)
```
*Main> div (fatorial 1000) (fatorial 999)
1000
```

e)
```
*Main> fatorial (-2)
*** Exception: fatorial.hs:(2,1)-(4,31): Non-exhaustive patterns in
function fatorial
```

Acontece uma exceção (erro em tempo de execução) porque nenhuma das guardas na definição de `fatorial` tem valor **True**. Logo não é possível determinar o resultado da função.

Tarefa 8.2 on page 8-4: Solução

Vejamos o que acontece quando tentamos calcular `podDois’ (-5)`:

```
podDois’ (-5)
⇝ 2 * podDois’ (-6)
⇝ 2 * 2 * podDois’ (-7)
⇝ 2 * 2 * 2 * podDois’ (-8)
...
```

A chamada recursiva da função calcula a potência de dois do antecessor do argumento original. Porém como o argumento é negativo, o seu antecessor está mais distante de zero, o caso base. Logo a chamada recursiva não converge para o caso base, e o cálculo nunca termina.

Este problema pode ser corrigido acrescentando uma outra alternativa na definição da fun;ão para considerar o caso de argumentos negativos.

Tarefa 8.3 on page 8-5: Solução

```
mul 5 6
⇝ 5 + mul 5 5
⇝ 5 + (5 + mul 5 4)
⇝ 5 + (5 + (5 + mul 5 3))
⇝ 5 + (5 + (5 + (5 + mul 5 2)))
⇝ 5 + (5 + (5 + (5 + (5 + mul 5 1))))
⇝ 5 + (5 + (5 + (5 + (5 + (5 + mul 5 0)))))
⇝ 5 + (5 + (5 + (5 + (5 + (5 + 0)))))
⇝ 30
```

Tarefa 8.4 on page 8-5: Solução

```
fib 6
⇝ fib 4 + fib 5
⇝ (fib 2 + fib 3) + (fib 3 + fib 4)
⇝ ((fib 0 + fib 1) + (fib 1 + fib 2)) + ((fib 1 + fib 2) + (fib 2 + fib 3))
⇝ ((0 + 1) + (1 + (fib 0 + fib 1))) + ((1 + (fib 0 + fib 1)) + ((fib 0 + fib 1) + (fib 1 + fib 2)))
⇝ ((0 + 1) + (1 + (0 + 1))) + ((1 + (0 + 1)) + ((0 + 1) + (1 + (fib 0 + fib 1))))
⇝ ((0 + 1) + (1 + (0 + 1))) + ((1 + (0 + 1)) + ((0 + 1) + (1 + (0 + 1))))
⇝ 8
```

Tarefa 8.5 on page 8-7: Solução

```
pdois 5
⇝ pdois' 5 1
⇝ pdois' 4 2
⇝ pdois' 3 4
⇝ pdois' 2 8
⇝ pdois' 1 16
⇝ pdois' 0 32
⇝ 32
```

Tarefa 8.6 on page 8-7: Solução

```haskell
par n = par' n True

par' n y
  | n == 0    = y
  | otherwise = par' (n-1) (not y)
```

Tarefa 8.7 on page 8-9: Solução

```haskell
fatorialDuplo n
  | n == 0 = 1
  | n == 1 = 1
  | n > 1  = n * fatorialDuplo (n - 2)
```

Tarefa 8.8 on page 8-9: Solução

```haskell
produtoIntervalo m n
  | m > n = 1
  | otherwise = m * produtoIntervalo (m+1) n
```

Tarefa 8.9 on page 8-9: Solução

```haskell
fatorial n = produtoIntervalo 1 n
```

Tarefa 8.10 on page 8-9: Solução

```haskell
soma m n
  | n == 0 = m
  | otherwise = soma (succ m) (pred n)
```

Tarefa 8.11 on page 8-9: Solução

```haskell
potencia x n
  | n == 0 = 1
  | otherwise = x * potencia x (n-1)
```

Tarefa 8.12 on page 8-9: Solução

```haskell
raizInteira n = raizInteira' n n
  where
    raizInteira' n i
      | i^2 > n   = raizInteira' n (i-1)
      | otherwise = i
```

Tarefa 8.13 on page 8-9: Solução

```haskell
quociente p q
  | p < q = 0
  | otherwise = 1 + quociente (p - q) q

resto p q
  | p < q = p
  | otherwise = resto (p - q) q
```

Tarefa 8.14 on page 8-10: Solução

```haskell
mdc a b
  | b == 0 = a
  | b > 0  = mdc b (mod a b)
  | b < 0  = mdc a (negate b)
```

Tarefa 8.15 on page 8-10: Solução

---

a) Cálculo:

```
fat 6
⇝ fat' 6 1
⇝ fat' 5 6
⇝ fat' 4 30
⇝ fat' 3 120
⇝ fat' 2 360
⇝ fat' 1 720
⇝ fat' 0 720
⇝ 720
```

b) `fat` é mais eficiente, pois as operações de multiplicação não ficam pendentes aguardando o retorno das chamadas recursivas.

Tarefa 8.16 on page 8-10: Solução

```haskell
fibonacci n = fibonacci' n 0 1
  where
    fibonacci' n a b
      | n == 0 = a
      | n == 1 = b
      | n >= 2 = fibonacci' (n-1) b (a+b)
```

9 Tuplas, Listas, e Polimorfismo Paramétrico

Resumo

Nesta aula vamos conhecer os tipos tuplas e listas, que são tipos polimórficos pré-definidos em Haskell. Vamos aprender também sobre funções polimórficas.

Sumário

9.1 Tuplas

Tupla é uma estrutura de dados formada por uma sequência de valores **possivelmente de tipos diferentes**. Os componentes de uma tupla são identificados pela **posição** em que ocorrem na tupla.

Em Haskell uma **expressão tupla** é formada por uma sequência de expressões separadas por vírgula e delimitada por parênteses:

$$(\ exp_1 \ , \ \ldots \ , \ exp_n \ )$$

onde $n >= 0$ e $n \neq 1$, e $exp_1, \ldots, exp_n$ são expressões cujos valores são os componentes da tupla.

O tipo de uma tupla é o **produto cartesiano** dos tipos dos seus componentes. Sintaticamente um **tipo tupla** é formado por uma sequência de tipos separados por vírgula e delimitada por parênteses:

$$(\ t_1 \ , \ \ldots \ , \ t_n \ )$$

onde $n >= 0$ e $n \neq 1$, e $t_1, \ldots, t_n$ são os tipos dos respectivos componentes da tupla. Observe que o tamanho de uma tupla (quantidade de componentes) é codificado no seu tipo.

`()` é a **tupla vazia**, do tipo `()`.

Não existe tupla de um único componente.

A tabela a seguir mostra alguns exemplos de tuplas:

tupla	tipo
`('A','t')`	`(Char,Char)`
`('A','t','o')`	`(Char,Char,Char)`
`('A',True)`	`(Char,Bool)`
`("Joel",'M',True,"COM")`	`(String,Char,Bool,String)`
`(True,("Ana",'f'),43)`	`Num a => (Bool,(String,Char),a)`
`()`	`()`
`("nao eh tupla")`	`String`

Vejamos algumas **operações com tuplas** definidas no prelúdio:

- `fst`: seleciona o primeiro componente de um *par*:

```
Prelude> fst ("pedro",19)
"pedro"
```

- `snd`: seleciona o segundo componente de um *par*:

```
Prelude> snd ("pedro",19)
19
```

9.2 Listas

Lista é uma estrutura de dados formada por uma sequência de valores (elementos) **do mesmo tipo**. Os elementos de uma lista são identificados pela **posição** em que ocorrem na lista.

Em Haskell uma **expressão lista** é formada por uma sequência de expressões separadas por vírgula e delimitada por colchetes:

$$[\ exp_1\ ,\ \ldots\ ,\ exp_n\ ]$$

onde $n >= 0$, e $exp_1, \ldots, exp_n$ são expressões cujos valores são os elementos da lista.

Um **tipo lista** é formado pelo tipo dos seus elementos delimitado por colchetes:

$$[\ t\ ]$$

onde t é o tipo dos elementos da lista. Observe que o tamanho de uma lista (quantidade de elementos) não é codificado no seu tipo.

A tabela a seguir mostra alguns exemplos de listas:

lista	tipo
`['O','B','A']`	`[Char]`
`['B','A','N','A','N','A']`	`[Char]`
`[False,True,True]`	`[Bool]`
`[ [False,True], [], [True,False,True] ]`	`[[Bool]]`
`[1,8,6,10.48,-5]`	`Fractional a => [a]`

Estruturalmente uma lista pode ser de duas formas:

- **lista vazia**
  - não contém nenhum elemento
  - é denotada pelo construtor constante `[]`
- **lista não vazia**
  - contém pelo menos um elemento
  - é formada por uma **cabeça** (o primeiro elemento da sequência) e por uma **cauda** (uma lista dos demais elementos da sequência)
  - é construída pelo construtor `:`, um operador binário infixo com associatividade à direita e precedência 5 (imediatamente inferior à precedência dos operadores aditivos `(+)` e `(-)`)
    - o operando da esquerda é a cabeça da lista
    - o operando da direita é a cauda da lista

Por exemplo, a lista formada pela sequência dos valores `1`, `8`, e `6` pode ser escrita como

lista
`[1, 8, 6]`
`1 : [8, 6]`
`1 : (8 : [6])`
`1 : (8 : (6 : []))`
`1 : 8 : 6 : []`

Observe que os parênteses neste exemplo são desnecessários, já que o operador `:` associa-se à direita.

Os exemplos anteriores podem ser reescritos com estes construtores de lista:

notação estendida	notação básica
`['O', 'B', 'A']`	`'O' : 'B' : 'A' : []`
`['B', 'A', 'N', 'A', 'N', 'A']`	`'B' : 'A' : 'N' : 'A' : 'N' : 'A' : []`
`[False, True, True]`	`False : True : True : []`
`[ [False], [], [True, False, True] ]`	`(False : []) : [] : (True : False : True : []) : []`
`[1., 8., 6., 10.48, -5.]`	`1. : 8. : 6. : 10.48 : -5. : []`

Vejamos algumas **operações com listas** definidas na biblioteca padrão:

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

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

- `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, a 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
```

- `lenght`: calcula o tamanho (quantidade de elementos) de uma lista:

```
Prelude> length [1,2,3,4,5]
5
Prelude> length []
0
```

- `(!!)`: seleciona o i-ésimo elemento de uma lista ($0 \leq i < n$, onde n 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
```

- `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]
```

- `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
```

- `(++)`: 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]
```

- `zip`: junta duas listas em uma única lista formada pelos pares dos elementos correspondentes:

```
Prelude> zip ["pedro","ana","carlos"] [19,17,22]
[("pedro",19),("ana",17),("carlos",22)]
```

9.3 Strings

Em Haskell **strings** são **listas de caracteres**. O tipo `String` é um sinônimo para o tipo `[Char]`. A tabela a seguir mostra alguns exemplos de strings:

string	notação de lista
`"ufop"`	`['u','f','o','p']`
`"bom\ndia"`	`['b','o','m','\n','d','i','a']`
`""`	`[]`

9.4 Polimorfismo paramétrico

9.4.1 Operação sobre vários tipos de dados

Algumas funções podem operar sobre vários tipos de dados. Por exemplo: a função `head` recebe uma lista e retorna o primeiro elemento da lista:

```
head ['b','a','n','a','n','a']   ⇝ 'b'
head ["maria","paula","peixoto"] ⇝ "maria"
head [True,False,True,True]      ⇝ True
head [("ana",2.8),("pedro",4.3)] ⇝ ("ana",2.8)
```

Não importa qual é o tipo dos elementos da lista. Qual deve ser o tipo de `head`?

```
head :: [Char] -> Char
head :: [String] -> String
head :: [Bool] -> Bool
head :: [(String,Double)] -> (String,Double)
```

`head` pode ter vários tipos.

9.4.2 Variáveis de tipo

Quando um tipo pode ser *qualquer* tipo da linguagem, ele é representado por uma **variável de tipo**.

No exemplo dado, sendo `a` o tipo dos elementos da lista que é passada como argumento para a função `head`, então

```
head :: [a] -> a
```

`a` é uma *variável de tipo* e pode ser substituída por qualquer tipo. O tipo de `head` estabelece que `head` recebe uma lista com elementos de um tipo qualquer, e retorna um valor deste mesmo tipo.

Em Haskell variáveis de tipo devem começar com uma *letra minúscula*, e são geralmente denominadas `a`, `b`, `c`, etc.

9.4.3 Valor polimórfico

Um valor é chamado **polimórfico** (*de muitas formas*) se o seu tipo contém uma ou mais *variáveis de tipo*.

Por exemplo, o tipo da função `head` pode ser escrito como

```
head :: [a] -> a
```

para qualquer tipo `a`, `head` recebe uma lista de valores do tipo `a` e retorna um valor do tipo `a`.

Já o tipo da função `length`, que recebe uma lista e resulta no tamanho da lista, é dado por:

```
length :: [a] -> Int
```

para qualquer tipo `a`, `length` recebe uma lista de valores do tipo `a` e retorna um inteiro.

A função `fst` é do tipo:

```
fst :: (a,b) -> a
```

para quaisquer tipos `a` *e* `b`, `fst` recebe um par de valores do tipo `(a,b)` e retorna um valor do tipo `a`.

9.4.4 Instanciação de variáveis de tipo

As variaveis de tipo podem ser *instanciadas* para diferentes tipos em diferentes circunstâncias.

Por exemplo, a função `length`

```
length :: [a] -> Int
```

pode ser aplicada em diferentes tipos listas, como mostra a tabela:

expressão	valor	instanciação da variável de tipo
`length [False,True]`	2	`a = Bool`
`length "54321"`	5	`a = Char`
`length ["ana","joel","mara"]`	3	`a = String`
`length [("ana",True)]`	1	`a = (String,Bool)`
`length [(&&),(||)]`	2	`a = Bool -> Bool -> Bool`

9.5 Funções polimórficas predefinidas

Muitas das funções definidas no prelúdio são polimórficas. Algumas delas são mencionadas a seguir:

```
id   :: a -> a                -- função identidade
fst  :: (a,b) -> a            -- seleciona o primeiro elemento de um par
snd  :: (a,b) -> b            -- seleciona o segundo elemento de um par
head :: [a] -> a              -- seleciona o primeiro el. de uma lista
tail :: [a] -> [a]            -- seleciona a cauda de uma lista
take :: Int -> [a] -> [a]     -- seleciona os primeiros el. de uma lista
zip  :: [a] -> [b] -> [(a,b)] -- combina duas listas, elemento a elemento
```

Observe que a lista vazia é polimórfica:

```
[] :: [a]
```

Tarefa 9.1: Tipo de expressões

Verifique se as seguintes expressões são válidas e determine o seu tipo em caso afirmativo.

a) `['a','b','c']`

b) `('a','b','c')`

c) `[(False,'0'),(True,'1')]`

d) `([False,True],['0','1'])`

e) `[tail,init,reverse]`

f) `[[]]`

g) `[[10,20,30],[],[5,6],[24]]`

h) `(10e-2,20e-2,30e-3)`

i) `[(2,3),(4,5.6),(6,4.55)]`

j) `(["bom","dia","brasil"],sum,drop 7 "Velho mundo")`

k) `[sum,length]`

Tarefa 9.2: Tipo de funções

Determine o tipo de cada uma das funções definidas a seguir, e explique o que elas calculam.

a) `second xs = head (tail xs)`

b) `const x y = x`

c) `swap (x,y) = (y,x)`

d) `apply f x = f x`

e) `flip f x y = f y x`

f) `pair x y = (x,y)`

g) `palindrome xs = reverse xs == xs`

h) `twice f x = f (f x)`

i) `mostra (nome,idade) = "Nome: " ++ nome ++ ", idade: " ++ show idade`

Tarefa 9.3: Último

Defina uma função chamada `ultimo` que seleciona o último elemento de uma lista não vazia, usando as funções do prelúdio.

Observação: já existe a função `last` no prelúdio com este propósito.

Tarefa 9.4: Primeiros

Defina uma função chamada `primeiros` que seleciona todos os elementos de uma lista não vazia, exceto o último., usando as funções do prelúdio.

Observação: já existe a função `init` no prelúdio com este propósito.

Tarefa 9.5: Metade

Usando funções da biblioteca, defina a função `metade :: [a] -> ([a],[a])` que divide uma lista em duas metades. Por exemplo:

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

> metade [1,2,3,4,5]
([1,2],[3,4,5])
```

Tarefa 9.6: Equação do segundo grau

Defina uma função para calcular as raízes reais do polinômio

$$ax^2 + bx + c$$

O resultado da função deve ser a lista das raízes reais.

Faça duas versões, usando:

- uma expressão **let** para calcular o discrimante, e
- uma cláusula **where** para calcular o discriminante.

Teste suas funções no GHCi.

Use a função `error :: String ->` a do prelúdio (`error` exibe uma mensagem e termina o programa imediatamente), para exibir uma mensagem quando não houver raízes reais.

9.6 Soluções

Tarefa 9.1 on page 9-6: Solução

---

a) `['a','b','c']`
`[Char]`

b) `('a','b','c')`
`(Char,Char,Char)`

c) `[(False,'0'),(True,'1')]`
`[(Bool,Char)]`

d) `([False,True],['0','1'])`
`([Bool],[Char])`

e) `[tail,init,reverse]`
`[[a] -> [a]]`

f) `[[]]`
`[[a]]`

g) `[[10,20,30],[],[5,6],[24]]`
`Num a => [[a]]`

h) `(10e-2,20e-2,30e-3)`
`(Fractional a, Fractional b, Fractional c) => (a,b,c)`

i) `[(2,3),(4,5.6),(6,4.55)]`
`(Num a, Fractional b) => [(a,b)]`

j) `(["bom","dia","brasil"],sum,drop 7 "Velho mundo")`
`Num a => ([String],[a]->a,String)`

k) `[sum,length]`
`[Int->Int]`

Tarefa 9.2 on page 9-7: Solução

---

a)
```
second xs = head (tail xs)
second :: [a] -> a
```
Recebe uma lista e resulta no segundo elemento da lista.

b)
```
const x y = x
const :: a -> b -> a
```
Recebe dois valores e resulta no segundo valor, ignorando o primeiro.

c)
```
swap (x,y) = (y,x)
swap :: (a,b) -> (b,a)
```
Recebe um par e resulta no par com os seus componentes invertidos.

d)
```
apply f x = f x
apply :: (a -> b) -> a -> b
```
Recebe uma função e um valor e aplica a função no valor.

e)
```
flip f x y = f y x
flip :: (a -> b -> c) -> b -> a -> c
```
Recebe uma função e dois valores e aplica a função nos dois valores invertidos.

f) `pair x y = (x,y)`
`pair :: a -> b -> (a,b)`
Recebe dois valores e resulta em um par formado pelos dois valores.

g) `palindrome xs = reverse xs == xs`
`palindrome :: Eq a => [a] -> Bool`
Recebe uma lista e verifica se ela é uma palíndrome, isto é, se a lista é igual à própria lista invertida.

h) `twice f x = f (f x)`
`twice :: (a -> a) -> a -> a`
Recebe uma função e um valor, e aplica sucessivamente a função no resultado da aplicação da função no valor.

i) `mostra (nome,idade) = "Nome: " ++ nome ++ ", idade: " ++ show idade`
`mostra :: Show a => (String,a) => String`
Recebe um par formado por uma string e um valor e resulta na string contendo o primeiro e o segundo elementos inseridos em uma mensagem.

Tarefa 9.3 on page 9-7: Solução

```haskell
ultimo lista = head (reverse lista)

ultimo' lista = lista !! (length lista - 1)

ultimo'' lista = head (drop (length lista - 1) lista)
```

Tarefa 9.4 on page 9-7: Solução

```haskell
primeiros lista = reverse (tail (reverse lista))
```

Tarefa 9.5 on page 9-7: Solução

```haskell
metade :: [a] -> ([a],[a])
metade lista = ( take k lista, drop k lista )
  where
    k = div (length lista) 2
```

Tarefa 9.6 on page 9-7: Solução

```haskell
raizes :: (Ord a, Floating a) => a -> a -> a -> [a]
raizes a b c | delta > 0  = [ (-b + sqrt delta)/(2*a), (-b - sqrt delta)/(2*a) ]
             | delta == 0 = [ -b/(2*a) ]
             | otherwise  = [ ]
  where
    delta = b^2 - 4*a*c

raizes' :: (Ord a, Floating a) => a -> a -> a -> [a]
raizes' a b c = let delta = b^2 - 4*a*c
                in if delta > 0
                     then [ (-b + sqrt delta)/(2*a), (-b - sqrt delta)/(2*a) ]
                     else if delta == 0
                            then [ -b/(2*a) ]
                            else []
```

10 Casamento de Padrão

Resumo

Linguagens funcionais modernas usam casamento de padrão em várias situações, como por exemplo na seleção de componentes de estruturas de dados, na escolha de alternativas em expressões de seleção múltipla, e na aplicação de funções.

Nesta aula vamos aprender as principais formas de padrão e como funciona a operação casamento de padrão. Vamos ainda conhecer algumas construções de Haskell que usam casamento de padrão, como definições de valores e funções e expressões de seleção múltipla.

Sumário

10.1 Casamento de padrão

10.1.1 Casamento de padrão

Em ciência da computação, **casamento de padrão**[1] é o ato de verificação da presença de um **padrão** em um dado ou em um conjunto de dados. O padrão é rigidamente especificado. O casamento de padrão é usado para testar se o objeto de estudo possui a estrutura desejada.

Algumas linguagens de programação funcionais como Haskell, ML e Mathematica possuem uma sintaxe especial para expressar padrões e uma construção na linguagem para execução condicional baseada no casamento de padrões.

Em Haskell um **padrão** é uma construção da linguagem de programação que permite *analisar* a estrutura de um valor e *associar variáveis* aos seus componentes.

Casamento de padrão é uma operação envolvendo *um padrão* e *uma expressão* que faz a correspondência (*casamento*) entre o padrão e o valor da expressão. Um casamento de padrão pode *suceder* ou *falhar*, dependendo da forma do padrão e da expressão envolvidos. Quando o casamento de padrão sucede as **variáveis** que ocorrem no padrão são *associadas* aos componentes correspondentes do valor.

Por exemplo o padrão `("ana", peso, _)` especifica valores que são triplas onde o primeiro componente é a string `"ana"`, o segundo componente é chamado de `peso`, e o terceiro componente é irrelevante.

[1] *Pattern matching* em inglês.

O valor `("ana", 58.7, 'F')` casa com este padrão e em consequência a variável `peso` é associada ao valor `58.7`. Já o valor `("pedro", 75.3, 'M')` não casa com esse padrão.

Em um casamento de padrão, o padrão e a expressão devem ser do *mesmo tipo*.

Existem várias **formas de padrão**. Na seqüência algumas delas são apresentadas.

10.1.2 Padrão constante

O **padrão constante** é simplesmente uma *constante*. O **casamento** *sucede* se e somente se o padrão for *idêntico* ao valor. Nenhuma associação de **variável** é produzida.

Veja os exemplos seguintes, onde ✓ indica sucesso e × indica falha:

padrão	valor	casamento
`10`	`10`	✓
`10`	`28`	×
`10`	`'P'`	*erro de tipo*
`'P'`	`'P'`	✓
`'P'`	`'q'`	×
`'P'`	`True`	*erro de tipo*
`True`	`True`	✓
`True`	`False`	×
`True`	`65`	*erro de tipo*
`GT`	`GT`	✓
`GT`	`EQ`	×

✓: sucede

×: falha

10.1.3 Padrão variável

O **padrão variável** é simplesmente um identificador de *variável* de valor (e como tal deve começar com letra minúscula). O **casamento** *sucede sempre*. A **variável** é associada ao *valor*.

Exemplos:

padrão	valor	casamento
`x`	`10`	✓ `x` ↦ `10`
`alfa`	`563.1223`	✓ `alfa` ↦ `563.1223`
`letra`	`'K'`	✓ `letra` ↦ `'K'`
`nomeCliente`	`"Ana Maria"`	✓ `nomeCliente` ↦ `"Ana Maria"`
`pessoa`	`("Ana",'F',16)`	✓ `pessoa` ↦ `("Ana",'F',16)`
`notas`	`[5.6,7.1,9.0]`	✓ `notas` ↦ `[5.6,7.1,9.0]`

10.1.4 Padrão curinga

O **padrão curinga** é escrito como um sublinhado (`_`). O **casamento** *sucede* sempre. *Nenhuma* associação de **variável** é produzida. `_` é também chamado de **variável anônima**, pois, assim como a variável, casa com qualquer valor, porém não nomeia o valor.

Exemplos:

padrão	valor	casamento
`_`	`10`	✓
`_`	`28`	✓
`_`	`'P'`	✓
`_`	`()`	✓
`_`	`(18,3,2012)`	✓
`_`	`"Ana Maria"`	✓
`_`	`[5.6,7.1,9.0]`	✓

10.1.5 Padrão tupla

Uma **tupla** de padrões também é um padrão:

$$(\mathit{padrão}_1, \ldots, \mathit{padrão}_n)$$

O **casamento** *sucede* se e somente se cada um dos padrões casar com o componente correspondente do valor. Se as *aridades* (tamanhos) do padrão tupla e do valor tupla forem *diferentes*, então ocorre um *erro de tipo*.

Exemplos de padrão tupla:

padrão	valor	casamento
`(18,True)`	`(18,True)`	✓
`(97,True)`	`(18,True)`	×
`(18,False)`	`(18,True)`	×
`(18,'M')`	`(18,True)`	*erro de tipo*
`(18,True,'M')`	`(18,True)`	*erro de tipo*
`()`	`()`	✓
`(x,y)`	`(5,9)`	✓ `x` ↦ 5, `y` ↦ 9
`(d,_,a)`	`(5,9,2012)`	✓ `d` ↦ 5, `a` ↦ 2012
`(x,y,z)`	`(5,9)`	*erro de tipo*
`(18,m,a)`	`(18,3,2012)`	✓ `m` ↦ 3, `a` ↦ 2012
`(d,5,a)`	`(18,3,2012)`	×
`(nome,sexo,_)`	`("Ana",'F',18)`	✓ `nome` ↦ `"Ana"`, `sexo` ↦ `'F'`
`(_,_,idade)`	`("Ana",'F',18)`	✓ `idade` ↦ 18
`(_,(_,fam),9)`	`('F',("Ana","Dias"),9)`	✓ `fam` ↦ `"Dias"`
`(_,(_,fam),5)`	`('F',("Ana","Dias"),9)`	×

10.1.6 Padrões lista

Estruturalmente uma lista pode ser vazia ou não vazia:

- **padrão lista vazia**

  `[]`

  - é um padrão constante
  - o casamento sucede se e somente se o valor for a lista vazia

- **padrão lista não vazia**

  $$\mathit{pad}_1 \ : \ \mathit{pad}_2$$

  - é formado por dois padrões pad_1 e pad_2
  - o casamento sucede se e somente se o valor for uma lista não vazia cuja cabeça casa com pad_1 e cuja cauda casa com pad_2
  - `:` é o construtor de lista não vazia, um operador binário infixo associativo à direita com precedência 5 (logo abaixo dos operadores aritméticos aditivos `(+)` e `(-)`).

Exemplos de padrões lista:

- O padrão `[]` casa somente com a lista vazia.

padrão	valor	casamento
`[]`	`[]`	✓
`[]`	`[1,2,3]`	×

- O padrão `x:xs` casa com qualquer lista não vazia, associando as variáveis `x` e `xs` com a cabeça e com a cauda da lista, respectivamente.

padrão	valor	casamento
`x:xs`	`[]`	×
`x:xs`	`[1,2,3,4]`	✓ `x` ↦ `1`, `xs` ↦ `[2,3,4]`
`x:xs`	`['A']`	✓ `x` ↦ `'A'`, `xs` ↦ `[]`

- O padrão `x:y:_` casa com qualquer lista que tenha pelo menos dois elementos, associando as variáveis `x` e `y` ao primeiro e segundo elementos da lista, respectivamente.

padrão	valor	casamento
`x:y:_`	`[]`	×
`x:y:_`	`["ana"]`	×
`x:y:_`	`[1,2]`	✓ `x` ↦ `1`, `y` ↦ `2`
`x:y:_`	`[1,2,3,4]`	✓ `x` ↦ `1`, `y` ↦ `2`

- O padrão `x:_:z:[]` casa com qualquer lista que tenha exatamente três elementos, associando as variáveis `x` e `z` ao primeiro e ao terceiro elementos da lista, respectivamente.

padrão	valor	casamento
`x:_:z:[]`	`[]`	×
`x:_:z:[]`	`["ana"]`	×
`x:_:z:[]`	`[1,2,3]`	✓ `x` ↦ `1`, `z` ↦ `3`
`x:_:z:[]`	`[1,2,3,4,5]`	×

- O padrão `0:a:_` casa com qualquer lista de números que tenha pelo menos dois elementos, sendo o primeiro igual a zero, associando a variável `a` ao segundo elemento da lista.

padrão	valor	casamento
`0:a:_`	`[]`	×
`0:a:_`	`[0]`	×
`0:a:_`	`[0,2,3]`	✓ `a` ↦ `2`
`0:a:_`	`[0,10,6,3]`	✓ `a` ↦ `10`
`0:a:_`	`[7,0,8]`	×

- O padrão `(m,_):_` casa com qualquer lista não vazia de pares, associando a variável `m` ao primeiro componente do primeiro elemento da lista.

padrão	valor	casamento
`(m,_):_`	`[]`	×
`(m,_):_`	`[("fim",True)]`	✓ `m` ↦ `"fim"`
`(m,_):_`	`[(10,'M'),(20,'F')]`	✓ `m` ↦ `10`

10.1.7 Padrão lista na notação especial

O padrão

$$[\ padrão_1\ ,\ \ldots\ ,\ padrão_n\ ]$$

é uma *abreviação sintática* para

$$padrão_1\ :\ \ldots\ :\ padrão_n\ :\ []$$

cujo casamento sucede somente se o valor for uma lista com exatamente n elementos.

Exemplos: o padrão `[1,alfa]` casa com qualquer lista de dois números que começa com `1`, associando a variável `alfa` ao segundo elemento da lista.

padrão	valor	casamento
`[1,alfa]`	`[]`	×
`[1,alfa]`	`[1]`	×
`[1,alfa]`	`[1,5]`	✓ `alfa` ↦ `5`
`[1,alfa]`	`[9,5]`	×
`[1,alfa]`	`[1,2,3]`	×

10.2 Definição de função usando padrões

10.2.1 Definindo funções com casamento de padrão

Uma **definição de função** é formada por uma *seqüência de equações*. Os **parâmetros** usados em uma equação para representar os argumentos são *padrões*. Em uma **aplicação de função** o resultado é dado pela primeira equação cujos parâmetros *casam* com os respectivos argumentos, e cuja guarda (se houver) é *verdadeira*. Se em todas as equações os casamentos de padrão *falharem* ou todas as guardas forem *falsas*, ocorre um *erro de execução*.

Geralmente o uso de padrões para especificar os argumentos torna a definição da função mais *clara* e *concisa*.

Exemplo: a função `not` mapeia `False` a `True`, e `True` a `False`:

```
not :: Bool -> Bool
not False = True
not True  = False
```

```
not False    ⇝ True
not (even 6) ⇝ False
```

Exemplo: a função `(&&)` calcula a conjunção (e lógico) de dois valores lógicos:

```
(&&) :: Bool -> Bool -> Bool
True  && True  = True
True  && False = False
False && True  = False
False && False = False
```

```
True && True  ⇝ True
False && True ⇝ False
2>3 && odd 4  ⇝ False
```

Exemplo: outra possível definição para `(&&)`:

```
(&&) :: Bool -> Bool -> Bool
True && True = True
_    && _    = False
```

```
True && True  ⇝ True
False && True ⇝ False
2>3 && 2<3    ⇝ False
```

Exemplo: outra definição para `(&&)`:

```haskell
(&&) :: Bool -> Bool -> Bool
True  && b = b
False && _ = False
```

```haskell
True && True  ⇝ True
2>3 && 2<3    ⇝ False
2<3 && even 5 ⇝ False
```

Exemplo: de novo outra definição para (&&):

```haskell
(&&) :: Bool -> Bool -> Bool
b && b = b
_ && _ = False
```

Esta definição está *incorreta*, pois *não é possível usar uma variável mais de uma vez nos padrões* (**princípio da linearidade**).

Outros exemplos:

```haskell
fst      :: (a,b) -> a
fst (x,_) = x
```

```haskell
snd      :: (a,b) -> b
snd (_,y) = y
```

```haskell
fst (1+2,1-2)         ⇝ 3
snd (div 5 0,even 9)  ⇝ False
```

Tarefa 10.1: Disjunção lógica

Dê três definições diferentes para o operador lógico ou (||), utilizando casamento de padrão.

Tarefa 10.2: Conjunção lógica

Redefina a seguinte versão do operador lógico e (&&) usando expressões condicionais ao invés de casamento de padrão:

```haskell
True && True = True
_    && _    = False
```

Tarefa 10.3: Conjunção lógica

Redefina a seguinte versão do operador lógico e (&&) usando expressões condicionais ao invés de casamento de padrão:

```haskell
True  && b = b
False && _ = False
```

Comente sobre o diferente número de expressões condicionais necessárias em comparação com o exercício 10.2.

Tarefa 10.4: Distância entre dois pontos

A distância entre dois pontos (x_1, y_1, z_1) e (x_2, y_2, z_2) no espaço é dada por

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$$

Defina uma função que recebe dois pontos no espaço e retorna a distância entre eles. Considere que um ponto no espaço é representado por uma tripla de números que são as coordenadas do ponto. Use casamento de padrão.

Tarefa 10.5

Estude a seguinte definição e apresente uma definição alternativa mais simples desta função, usando padrões.

```
opp :: (Int,(Int,Int)) -> Int
opp z = if fst z == 1
        then fst (snd z) + snd (snd z)
        else if fst z == 2
             then fst (snd z) - snd (snd z)
             else 0
```

Tarefa 10.6

Defina uma função usando casamento de padrão que retorna o sucessor do primeiro elemento de uma lista, se houver, e zero caso contrário.

Tarefa 10.7

Usando casamento de padrão, defina uma função que, dada uma lista de números, retorna

- a soma dos dois primeiros elementos se a lista tiver pelo menos dois elementos,
- a cabeça da lista se ela contiver apenas um elemento, e
- zero caso contrário.

Tarefa 10.8

Resolva os exercícios 10.6 e 10.7 usando funções da biblioteca ao invés de casamento de padrão.

Tarefa 10.9: Média das notas

A seguir é mostrada uma aplicação (incompleta) para fechamento das notas de uma disciplina. Cada aluno recebe três notas nas atividades desenvolvidas. O usuário deverá informar a quantidade de alunos na turma, e em seguida as notas de cada aluno. A aplicação calcula e exibe a média da turma.

Complete a aplicação definindo as funções `somaNotas` e `mediaTurma`.

A função `somaNotas` recebe uma lista com as três notas de cada aluno, e resulta na soma das médias das três notas de todos os alunos. Esta função deverá ser recursiva:

- caso base: lista vazia; neste caso a soma é zero (elemento neutro da adição)
- caso recursivo: lista não vazia; neste caso a soma é obtida somando a média das notas do primeiro elemento da lista com a soma do resto da lista

A função `mediaTurma` recebe uma lista com as três notas de cada aluno, e resulta na média de todas as notas. Esta função deve usar a função `somaNotas` e a função `length`. Como o resultado de `length` é do tipo `Int`, será necessário convertê-lo para o tipo `Float` para calcular a média. Use

a função `fromIntegral` do prelúdio para converter um número de um tipo integral para qualquer tipo numérico que for apropriado no contexto.

```haskell
module Main where

import System.IO (hSetBuffering, stdout, BufferMode(NoBuffering))

main :: IO ()
main = do hSetBuffering stdout NoBuffering
          putStrLn "Fechamento de notas"
          putStrLn "==================================="
          putStr "Quantidade de alunos: "
          qtdeAlunos <- readLn
          notas <- leNotas qtdeAlunos 1
          let media = mediaTurma notas
          putStrLn ("Média da turma: " ++ show media)

leNotas :: Int -> Int -> IO [(Float, Float, Float)]
leNotas n i
  | i > n     = return []
  | otherwise = do putStrLn ("aluno " ++ show i)
                   putStr "  nota 1: "
                   n1 <- readLn
                   putStr "  nota 2: "
                   n2 <- readLn
                   putStr "  nota 3: "
                   n3 <- readLn
                   resto <- leNotas n (i+1)
                   return ((n1,n2,n3):resto)

somaNotas :: [(Float, Float, Float)] -> Float
-- complete a definição da função

mediaTurma :: [(Float, Float, Float)] -> Float
-- complete a definição da função
```

Exemplo de execução da aplicação

```
*Main> main
Fechamento de notas
===================================
Quantidade de alunos: 5
aluno 1
  nota 1: 10
  nota 2: 10
  nota 3: 10
aluno 2
  nota 1: 4
  nota 2: 6
  nota 3: 8
aluno 3
  nota 1: 6
  nota 2: 7
  nota 3: 6
aluno 4
  nota 1: 9
  nota 2: 3
  nota 3: 6
aluno 5
  nota 1: 0
  nota 2: 7
  nota 3: 5
Média da turma: 6.466667
```

10.3 Casamento de padrão em definições

Definições podem ocorrer um módulo (globais ao módulo), em uma cláusula where (locais a uma equação), ou em uma expressão let (locais a uma expressão).

O lado esquerdo de uma equação pode ser um padrão. Neste caso o lado direito deve ser uma expressão. O valor da expressão e o padrão devem casar. Caso o casamento de padrão falhe ocorre um erro em tempo de execução.

Por exemplo, na equação

```
(prefixo,sufixo) = splitAt 6 "Hello World!"
```

são definidas duas variáveis, `prefixo` e `sufixo`, correspondentes ao primeiro e segundo componentes do par resultante da função `splitAt`, que divide uma lista em duas partes, em uma dada posição.

Definições locais com **`where`** não são compartilhadas entre diferentes equações de uma definição principal. Por exemplo:

```haskell
saudacao :: String -> String

saudacao "Joana"   = saudacaoLegal ++ " Joana!"

saudacao "Ferando" = saudacaoLegal ++ " Fernando!"

saudacao nome      = saudacaoInfeliz ++ " " ++ nome
  where
    saudacaoLegal   = "Olá! Que bom encontrar você, "
    saudacaoInfeliz = "Oh! Pfft. É você, "
```

Esta definição de função está incorreta. Para corrigi-la, transforme as definições locais de `saudacaoLegal` e `saudacaoInfeliz` em definições globais.

10.4 Problema: validação de números de cartão de crédito

Alguma vez você já se perguntou como os sites validam o número do seu cartão de crédito quando você faz compras *online*? Eles não verifiquem um enorme banco de dados de números, e muito menos usam magia. Na verdade, a maioria dos provedores de crédito dependem de uma fórmula de verificação de soma (*checksum*) para distinguir entre números válidos de cartões e sequências aleatórias de dígitos (ou erros de digitação).

Nesta atividade você vai implementar o algoritmo de validação para cartões de crédito. A validação segue as etapas seguintes:

- Dobre o valor de cada segundo dígito começando pela direita. Ou seja, o último dígito não é alterado, o penúltimo dígito é dobrado, o antepenúltimo não é alterado, e assim por diante. Por exemplo, `[1,3,8,6]` torna-se `[2,3,16,6]`.

- Adicione os dígitos dos valores dobrados e não dobrados a partir do número original. Por exemplo, `[2,3,16,6]` torna-se `2 + 3 + 1 + 6 + 6 = 18`.

- Calcule o resto da divisão desta soma por 10. No exemplo acima, o resto é 8.

- O número é válido se e somente se o resultado for igual a 0.

Tarefa 10.10

Precisamos primeiramente encontrar os dígitos de um número natural para processarmos o número do cartão de crédito.

Defina as funções

```
toDigits    :: Integer -> [Integer]
toDigitsRev :: Integer -> [Integer]
```

`toDigits` deve converter um número natural na lista dos dígitos que formam este número. Para números negativos, `toDigits` deve resultar na lista vazia.

`toDigitsRev` deve fazer o mesmo, mas com os dígitos em ordem invertida.

Exemplos:

```
toDigits 1234    ⇝ [1,2,3,4]
toDigitsRev 1234 ⇝ [4,3,2,1]
toDigits 0       ⇝ [0]
toDigits (-17)   ⇝ []
```

Para obter os dígitos decimais que formam um número natural você deverá considerar os casos:

- Se o número for menor que 10, a lista dos dígitos é uma lista unitária contendo o próprio número.
- Caso contrário, divida o número por 10. O *resto* desta divisão é o dígito menos significativo. Os demais dígitos são obtidos de maneira similar usando o quociente desta divisão. Por exemplo, se o número é 538, então o dígito menos significativo é 8 (o resto da divisão de 538 por 10), e os demais dígitos são obtidos a partir de 53 (o quociente da divisão de 538 por 10).

Lembre-se de considerar o caso do número negativo.

Dica: Primeiro defina `toDigitsRev`, e depois `toDigits` usando `toDigitsRev`.

Tarefa 10.11

Uma vez obtidos os dígitos na ordem correta, precisamos dobrar um dígito não e outro sim, contando de trás para frente (ou seja, da direita para a esquerda, ou ainda, do dígito menos significativo para o dígito mais significativo).

Defina a função

```
doubleEveryOther :: [Integer] -> [Integer]
```

para este propósito. Lembre-se de que `doubleEveryOther` deve dobrar os números da lista de dois em dois, começando pelo penúltimo de trás para frente.

Exemplos:

```
doubleEveryOther [9,4,8,7,6,5] ⇝ [18,4,16,7,12,5]
doubleEveryOther [4,8,7,6,5]   ⇝ [4,16,7,12,5]
doubleEveryOther [1,2,3]       ⇝ [1,4,3]
```

Dica: Defina uma função auxiliar que dobra os elementos de uma lista de dois em dois, *do começo para o fim*. Isto é, o primeiro elemento não é dobrado, o segundo é, o terceiro não é, o quarto é, e assim por diante. Depois use esta função para definir `doubleEveryOther`. Neste caso será necessário inverter a lista antes e depois de chamar a função auxiliar.

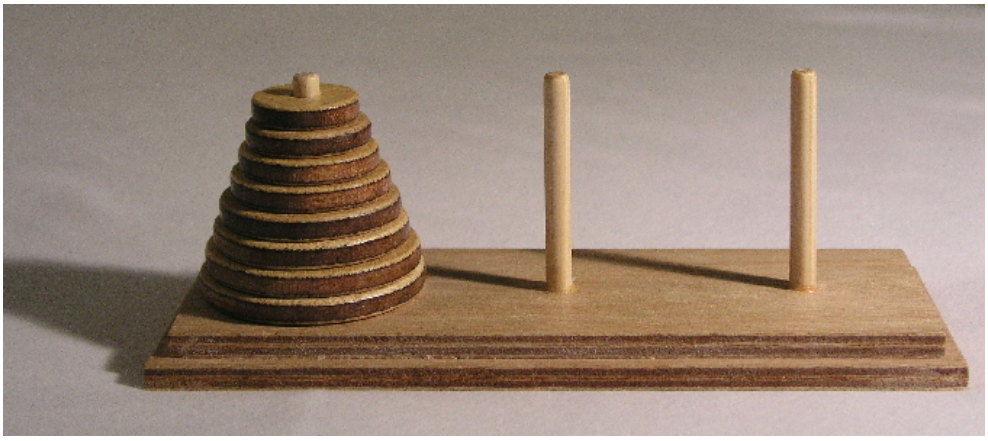


Figura 10.1: Torres de Hanoi.

Tarefa 10.12

O resultado de `doubleEveryOther` pode conter números de um dígito ou de dois dígitos. Defina a função

```
sumDigits :: [Integer] -> Integer
```

para calcular a soma de todos os dígitos.

Exemplos:

```
sumDigits [16,7,12,5] ⇝ 1 + 6 + 7 + 1 + 2 + 5 ⇝ 22
```

Dicas:

- Observe que são os dígitos dos números que devem ser somados, e não os próprios números.
- Faça uma definição recursiva que soma os elementos da lista. Divida cada elemento da lista por dez e considere tanto o quociente quanto o resto ao efetuar a soma.

Tarefa 10.13

Defina a função

```
validate :: Integer -> Bool
```

que indica se um número inteiro pode ser um número válido de cartão de crédito. Sua definição usará todas as funções definidas nos ítens anteriores.

Exemplos:

```
validate 4012888888881881 ⇝ True
validate 4012888888881882 ⇝ False
```

10.5 Problema: torres de Hanoi

Torres de Hanoi (figura 10.1) é um quebra-cabeça clássico com uma solução que pode ser descrita de forma recursiva. Vários discos de tamanhos diferentes são empilhados em três cavilhas. O objetivo é obter, a partir de uma configuração inicial com todos os discos empilhados sobre a primeira cavilha, uma configuração que termina com todos os discos empilhados sobre a última cavilha, como mostrado na figura 10.2.

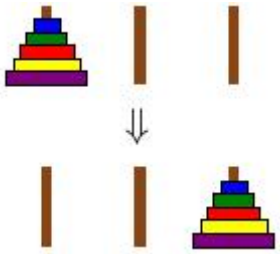


Figura 10.2: As torres de Hanoi.

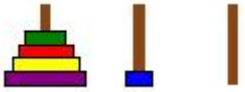


Figura 10.3: Um primeiro movimento válido.

As únicas regras são:

1. somente um disco pode ser movido de cada vez, e

2. um disco maior nunca pode ser empilhado em cima de um menor.

Por exemplo, como o primeiro passo tudo o que você pode fazer é mover o disco menor que se encontra no topo do pino, para outro pino diferente, uma vez que apenas um disco pode ser movido de cada vez. A partir desta situação, é ilegal mover para a configuração mostrada nas figura 10.4, porque você não tem permissão para colocar o disco verde em cima do disco azul, que é menor.

Para mover n discos (empilhados em ordem crescente de tamanho) de um pino a para um pino b usando um pino c como armazenamento temporário:

1. mova $n-1$ discos de a para c usando b como armazenamento temporário,

2. mova o disco no topo de a para b, e

3. mova $n-1$ discos de c para b usando a como armazenamento temporário.

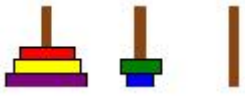


Figura 10.4: Uma configuração ilegal.

Tarefa 10.14: Torres de Hanoi

Defina uma função `hanoi` do tipo especificado a seguir.

```
type Pino = String
type Movimento = (Pino,Pino)

hanoi :: Integer -> Pino -> Pino -> Pino -> [Movimento]
```

Dados o número de discos e os nomes dos pinos para os três pinos, `hanoi` deve resultar na lista dos movimentos que devem ser realizados para mover a pilha de discos do primeiro pino para o segundo pino, usando o terceiro pino como armazenamento temporário.

Note que uma declaração de tipo usando a palavra chave **type**, como em **type Pino = String**, define um *sinônimo de tipo*. Neste caso **Pino** é declarado como um sinônimo para **String**, e os dois nomes de tipo **Pino** e **String** podem agora ser usado um no lugar do outro. Nomes descritivos para tipos dados desta maneira podem ser usados para dar nomes mais curtos para tipos complicados, ou para simplesmente ajudar na documentação, como foi feito aqui.

Exemplo:

```
hanoi 2 "a" "b" "c" ⇝ [("a","c"), ("a","b"), ("c","b")]
```

10.6 Soluções

Tarefa 10.1 on page 10-6: Solução

```
ou1 True  True  = True
ou1 True  False = True
ou1 False True  = True
ou1 False False = False

ou2 False False = False
ou2 _     _     = True

ou3 True  _ = True
ou3 False x = x
```

Tarefa 10.2 on page 10-6: Solução

```
e a b = a == if a
             then if b
                  then True
                  else False
             else False
```

Tarefa 10.3 on page 10-6: Solução

```
e a b = a == if a
             then b
             else False
```

Tarefa 10.4 on page 10-7: Solução

```
distancia :: Floating a => (a,a,a) -> (a,a,a) -> a

distancia (x1,y1,z1) (x2,y2,z2) =
  sqrt ( (x1-x2)^2 + (y1-y2)^2 + (z1-z2)^2 )
```

Tarefa 10.5 on page 10-7: Solução

```
opp :: (Int,(Int,Int)) -> Int

opp (1,(m,n)) = m + n
opp (2,(m,n)) = m - n
opp _         = 0
```

Tarefa 10.6 on page 10-7: Solução

```haskell
sucPrimLista :: (Num a, Enum a) => [a] -> a

sucPrimLista [] = 0
sucPrimLista (x:_) = succ x
```

Tarefa 10.7 on page 10-7: Solução

```haskell
flist :: Num a => [a] -> a

flista (x:y:_) = x + y
flista [x] = x
flista _ = 0
```

Tarefa 10.8 on page 10-7: Solução

```haskell
sucPrimLista' lista = succ (head lista)

flista' lista | null lista        = 0
              | null (tail lista) = head lista
              | otherwise         = head lista + head (tail lista)
```

Tarefa 10.9 on page 10-7: Solução

```haskell
module Main where

import System.IO (hSetBuffering, stdout, BufferMode(NoBuffering))

main :: IO ()
main = do hSetBuffering stdout NoBuffering
          putStrLn "Fechamento de notas"
          putStrLn "==================================="
          putStr "Quantidade de alunos: "
          qtdeAlunos <- readLn
          notas <- leNotas qtdeAlunos 1
          let media = mediaTurma notas
          putStrLn ("Média da turma: " ++ show media)

leNotas :: Int -> Int -> IO [(Float, Float, Float)]
leNotas n i
  | i > n     = return []
  | otherwise = do putStrLn ("aluno " ++ show i)
                   putStr "  nota 1: "
                   n1 <- readLn
                   putStr "  nota 2: "
                   n2 <- readLn
                   putStr "  nota 3: "
                   n3 <- readLn
                   resto <- leNotas n (i+1)
                   return ((n1,n2,n3):resto)

somaNotas :: [(Float, Float, Float)] -> Float
somaNotas [] = 0
somaNotas ( (n1,n2,n3) : resto ) = (n1+n2+n3)/3 + somaNotas resto

mediaTurma :: [(Float, Float, Float)] -> Float
mediaTurma notas = somaNotas notas / fromIntegral (length notas)
```

Tarefa 10.14 on page 10-14: Solução

```haskell
toDigitsRev :: Integer -> [Integer]
toDigitsRev n | n < 0 = []
              | n < 10 = [ n ]
              | otherwise = mod n 10 : toDigitsRev (div n 10)

toDigits :: Integer -> [Integer]
toDigits n = reverse (toDigitsRev n)

-- toDigits = reverse . toDigitsRev

doubleEveryOther' :: [Integer] -> [Integer]
doubleEveryOther' [] = []
doubleEveryOther' [x] = [x]
doubleEveryOther' (x:y:resto) = x : 2*y : doubleEveryOther' resto

doubleEveryOther :: [Integer] -> [Integer]
doubleEveryOther lista = reverse (doubleEveryOther' (reverse lista))

-- doubleEveryOther = reverse . doubleEveryOther' . reverse

somaDigitos :: [Integer] -> Integer
somaDigitos [] = 0
somaDigitos (n:ns) | n < 10 = n + somaDigitos ns
                   | otherwise = div n 10 + mod n 10 + somaDigitos ns

validate :: Integer -> Bool
validate num = mod (sumDigits (doubleEveryOther (toDigits num))) 10 == 0
```

11 Expressão de Seleção Múltipla

Resumo

Linguagens funcionais modernas usam casamento de padrão para selecionar componentes de estruturas de dados e também para selecionar alternativas em expressões `case` e em aplicações de funções definidas com várias equações.

Nesta aula vamos aprender as principais formas de padrão. Vamos também aprender como funciona o casamento de padrão. Vamos ainda conhecer algumas construções de Haskell que usam casamento de padrão, como a expressão `case` e definições usando padrões.

Sumário

11.1 Expressão case

Expressão case é uma forma de expressão que permite *selecionar* um entre vários resultados possíveis baseando-se no casamento de padrões. Uma expressão **case** é formada por:

- uma *expressão de controle*, cujo valor é usado para escolher uma das alternativas
- uma sequência de *alternativas*, onde cada alternativa é formada por:
  - um *padrão*, usado para decidir se a alternativa será escolhida
  - uma *expressão*, usada para dar o resultado caso a alternativa seja escolhida

Exemplo:

```
case calculo x (div y 2) of
  { 0 -> x^2 + 5*x + 6;
    1 -> 4*y - 8;
    n -> (x^2 + y^2) / n
  }
```

11.2 Forma e regras de tipo da expressão case

Uma expressão **case** é da forma:

```
case  exp of
   {  padrão_1 -> res_1 ;
      ⋮
      padrão_n -> res_n
   }
```

onde:

- a *expressão de controle* é *exp*
- os *resultados alternativos* são dados pelas expressões $res_1, \ldots, res_n$, selecionados pelos respectivos *padrões* $padrão_1, \ldots, padrão_n$

Regras de tipo:

- a expressão de controle *exp* e os padrões $padrão_1, \ldots, padrão_n$ devem ser todos de um *mesmo tipo*
- os resultados $res_1, \ldots, res_n$ devem ser todos do *mesmo tipo*, o que determina o *tipo* da expressão case (ou seja, o tipo do resultado final)

11.3 Regra de *layout* para a expressão case

A *regra de* layout pode ser aplicada para uma expressão `case`, permitindo a omissão dos sinais de pontuação `{`, `;` e `}`:

- Todas as alteranativas devem estar alinhadas (ou seja, devem começar na mesma coluna: a coluna de alinhamento).
- Se uma alternativa precisar se extender nas linhas seguintes, estas linhas devem começar em uma coluna mais à direita em relação à coluna de alinhamento.
- Uma linha que começa em uma coluna mais à esquerda em relação à coluna de alinhamento encerra a sequencia de alternativas (e não faz parte da expressão `case`)

Exemplo:

```
case calculo x (div y 2) of
  0 -> x^2 + 5*x + 6
  1 -> 4*y - 8
  n -> (x^2 + y^2) / n
proxima_expressao
```

é traduzido para

```
case calculo x (div y 2) of
  { 0 -> x^2 + 5*x + 6;
    1 -> 4*y - 8;
    n -> (x^2 + y^2) / n
  }
proxima_expressao
```

11.4 Avaliação de expressões case

- É feito o *casamento de padrão* do valor de exp com os padrões, na seqüência em que foram escritos, até que se obtenha sucesso ou se esgotem os padrões
- O *primeiro* padrão cujo casamento suceder é escolhido
- O *resultado* final da expressão case é dado pela expressão associada ao padrão escolhido
- O resultado é avaliado em um *ambiente estendido* com as associações de variáveis resultantes do casamento de padrão
- Se a expressão *não casar* com nenhum padrão, a avaliação da expressão case resulta em um *erro*

11.5 Exemplos de expressões case

A expressão

```
case 3 - 2 + 1 of
  0 -> "zero"
  1 -> "um"
  2 -> "dois"
  3 -> "tres"
```

resulta em `"dois"`, pois o valor da expressão `3-2+1` é `2`, que casa com o terceiro padrão `2`, selecionando `"dois"` como resultado.

A expressão

```
case 23 > 10 of
  True  -> "beleza!"
  False -> "oops!"
```

resulta em `"beleza!"`, pois o valor da expressão `23 > 10` é `True`, que casa com o primeiro padrão `True`, selecionando `"beleza!"` como resultado.

A expressão

```
case toUpper (head "masculino") of
  'F' -> 10.2
  'M' -> 20.0
```

resulta em `20.0`, pois o valor da expressão `toUpper (head "masculino")` é `'M'`, que casa com o segundo padrão `'M'`, selecionando `20.0` como resultado.

A expressão

```
case head "masculino" of
  'F' -> 10.2
  'M' -> 20.0
```

resulta em um erro em tempo de execução, pois o valor da expressão `head "masculino"` não casa com nenhum dos padrões.

A expressão

```
case toUpper (head "masculino") of
  'F' -> "mulher"
  'M' -> 20.0
```

está incorreta, pois os resultados `"mulher"` e `20.0` não são do mesmo tipo.

A expressão

```
case head "Masculino" == 'F' of
  True  -> "mulher"
  1     -> "homem"
```

está incorreta, pois os padrões `True` e `1` não são do mesmo tipo.

A expressão

```
case head "Masculino" of
  True  -> "mulher"
  False -> "homem"
```

está incorreta, pois a expressão `head "Masculino"` e os padrões **`True`** e **`False`** não são do mesmo tipo.

A expressão

```
case toUpper (head "masculino") of
   'F' -> 10.0
 'M' -> 20.0
```

está incorreta, uma vez que não segue a regra de *layout* (os padrões não estão na mesma coluna).

A expressão

```
case 3 - 2 + 1 of
  x -> 11 * x
```

resulta em `22`, pois o valor da expressão `3 - 2 + 1` é `2`, que casa com o primeiro padrão `x`, associando a variável `x` com o valor `2`, e selecionando `11 * x` como resultado

A expressão

```
case mod 256 10 of
  7 -> 0
  n -> n * 1000
```

resulta em `6000`, pois o valor da expressão `mod 256 10` é `6`, que casa com o segundo padrão `n`, associando a variável `n` com o valor `6`, e selecionando `n * 1000` como resultado

A expressão

```
case mod 257 10 of
  7 -> 0
  n -> n * 1000
```

resulta em `0`, pois `7` é o primeiro padrão que casa com o valor da expressão `mod 257 10`.

Já a expressão

```
case mod 257 10 of
  n -> n * 1000
  7 -> 0
```

resulta em `7000`, pois `n` é o primeiro padrão que casa com o valor da expressão `mod 257 10`.

A expressão

```
case 46 - 2*20 of
  0 -> "zero"
  1 -> "um"
  2 -> "dois"
  3 -> "tres"
  4 -> "quatro"
  _ -> "maior que quatro"
```

resulta em `"maior que quatro"`, pois `_` é o primeiro padrão que casa com o valor da expressão `46 - 2*20`.

A expressão

```
case (3+2,3-2) of
  (0,0) -> 10
  (_,1) -> 20
  (x,2) -> x^2
  (x,y) -> x*y - 1
```

resulta em `20`, pois `(_,1)` é o primeiro padrão que casa com o valor da expressão `(3+2,3-2)`.

A expressão

```
case tail [10] of
  [] -> "vazia"
  _  -> "nao vazia"
```

resulta em `"vazia"`, pois o valor da expressão `tail [10]` casa com o padrão para lista vazia `[]`.

A expressão

```
case [10,20,30,40] of
  []   -> "lista vazia"
  x:xs -> "cabeca: " ++ show x ++ " cauda: " ++ show xs
```

resulta em `"cabeca: 10 cauda: [20,30,40]"`, pois a lista `[10,20,30,40]` casa com o padrão para lista não vazia `x:xs`, associando `x` com `10` e `xs` com `[20,30,40]`.

A expressão

```
case [10..20] of
  x:y:z:_ -> x + y + z
  _       -> 0
```

resulta em `33`, pois a lista `[10..20]` casa com o padrão `x:y:z:_`, associando `x` com `10`, `y` com `11` e `z` com `12`.

A expressão

```
case [10,20] of
  x:y:z:_ -> x + y + z
  _       -> 0
```

resulta em `0`, pois a lista `[10,20]` não casa com o primeiro padrão `x:y:z:_`, mas casa com o segundo `_`. Observe que o primeiro padrão casa somente com listas que tenham *pelo menos três elementos*.

A expressão

```
case [10,20,30] of
  [x1,_,x3] -> x1 + x3
  _         -> 0
```

resulta em `40`, pois a lista `[10,20,30]` casa com o primeiro padrão `[x1,_,x3]`. Observe que este padrão casa somente com listas que tenham *exatamente três elementos*.

A expressão

```
case [100,20,3] of
  a:b:xs | a > b  -> b:a:xs
         | a == b -> a:xs
  xs              -> xs
```

resulta em `[20,100,3]`, pois a lista `[100,20,3]` casa com o primeiro padrão `a:b:xs` *e* o primeiro elemento é maior do que o segundo.

11.6 Expressão case com guardas

- Em uma expressão case cada padrão pode ser acompanhado de uma seqüência de *cláusulas*.
- Cada cláusula é introduzida por uma barra vertical (`|`) e consiste em uma *condição* (**guarda**) e uma expressão (*resultado*), separados por `->`.

- Para que o resultado de uma cláusula seja escolhido é necessário que o *casamento de padrão suceda*, e que *a guarda correspondente seja verdadeira*.

No exemplo a seguir, a expressão

```
case ("Paulo Roberto",'M',28,69.3) of
  (_,_,idade,peso)    | idade < 18 -> 2*peso
                      | idade < 21 -> 3*peso
  (_,'F',_,peso)                   -> peso
  (_,'M',idade,peso) | idade < 40 -> peso + 10
                      | otherwise  -> 0.9 * peso
```

resulta em `79.3`, pois a tupla `("Paulo Roberto",'M',28,69.3)`

- casa com o primeiro padrão, porém nenhuma guarda é satisfeita
- não casa com o segundo padrão
- casa com o terceiro padrão, e a primeira guarda é satisfeita, logo o resultado é dado por `peso + 10`

Tarefa 11.1: Seleção de um prefixo de uma lista

Defina a função `prefixo :: Int -> [a] -> [a]` que recebe um número inteiro n e uma lista l e resulta na lista dos n primeiros elementos de l.

Exemplos:

```
prefixo 0 [10, 20, 30, 40, 50] ⇝ []
prefixo 2 [10, 20, 30, 40, 50] ⇝ [10, 20]
prefixo 9 [10, 20, 30, 40, 50] ⇝ [10, 20, 30, 40, 50]
```

Sua definição deve consistir de uma única equação sem usar casamento de padrão ou guardas para obter o resulado. Porém o corpo da função deverá usar uma expressão case, na qual deve-se usar casamento de padrão.

Observação A função `take` do prelúdio é similar a esta função.

11.7 Soluções

Tarefa 11.1 on page 11-6: Solução

```haskell
prefixo :: Int -> [a] -> [a]
prefixo n lista = case (n,lista) of
                    (0,_)    -> []
                    (_,[])   -> []
                    (_,x:xs) -> x : prefixo (n-1) xs

prefixo' 0 _      = []
prefixo' _ []     = []
prefixo' n (x:xs) = x : prefixo' (n-1) xs
```

12 Programas Interativos

Resumo

Programas interativos se comunicam com o usuário recebendo dados e exibindo resultados. Nesta aula vamos aprender como desenvolver programs funcionais que interagem com o usuário.

Sumário

12.1 Interação com o *mundo*

12.1.1 Programas interativos

Programas interativos podem exibir mensagens para o usuário e obter valores informados pelo usuário. De forma geral um programa poderá *trocar informações* com o *restante do sistema computacional* para obter dados do sistema computacional e gravar dados no sistema computacional.

Em *linguagens imperativas* as operações de entrada e saída produzem **efeitos colaterais**, refletidos na *atualização de variáveis globais* que representam o *estado* do sistema de computação.

Exemplo de programa interativo em C Programa que obtém dois caracteres digitados pelo usuário e exibe-os em maiúsculas na tela:

```c
#include <stdio.h>
#include <ctype.h>

int main(void)
{
  char x = getchar();
  char y = getchar();
  printf("%c%c\n", toupper(x), toupper(y));
  return 0;
}
```

Supondo que o usuário informe os caracteres `'A'` e `'b'`, a execução do programa produzirá a seguinte interação:

```
Ab
AB
```

A aplicação de função `getchar()` *retorna valores diferentes mesmo quando chamada com os mesmos argumentos* (nenhum argumento, neste caso). A primeira chamada retorna `'A'` e a segunda chamada retorna `'b'`. Isto acontece porque `getchar()` utiliza uma variável global representando o dispositivo de entrada padrão (`stdin`). Durante a chamada da função esta variável é atualizada (*efeito colateral*), removendo o próximo caracter disponível na entrada e retornando-o como resultado. Assim, quando a função `getchar()` é chamada novamente, o próximo caracter disponível na entrada padrão (representada pela variável global `stdin`) é o segundo caracter digitado pelo usuário.

12.1.2 Linguagens puras

Em **linguagens puras** o valor retornado por uma função *depende única e exclusivamente dos argumentos* especificados na aplicação da função. Portanto toda vez que uma função é aplicada em um dado argumento, *o resultado é o mesmo*. Assim não é possível implementar uma função que lê um caracter da mesma maneira que em linguagens impuras, como C.

Exemplo:

```
let x = getchar ()
    y = getchar ()
in ...
```

Em uma linguagem pura os valores de `x` e `y` serão iguais, uma vez que são definidos aplicando a função `getchar` ao mesmo argumento (a tupla vazia).

12.1.3 O mundo

Para interagir com o usuário, precisamos de uma *representação do sistema de computação* onde o programa está sendo executado: o **mundo** (*world*). O **mundo** é formado por todas as informações no contexto de execução da aplicação, incluindo:

- dispositivo de entrada padrão (o teclado)
- dispositivo de saída padrão (a tela)
- sistema de arquivos (arquivos em disco)
- conexões de rede
- gerador de números pseudo-aleatórios (usa uma semente que depende do sistema, como por exemplo o horário atual)

12.1.4 Modificando o mundo

Em **linguagens impuras** o mundo (ou parte dele) corresponde a uma *variável global atualizável*. Uma função impura que interage com o mundo pode alterar esta variável, de foma que uma aplicação posterior da função ao mesmo argumento pode retornar um valor diferente.

Em uma **linguagem pura** não há a possibilidade de alterar uma variável. Uma função pura que interage com o mundo tem um *argumento* e um *resultado* adicionais que representam *o mundo antes* e *o mundo depois* da interação.

12.1.5 Ações de entrada e saída

Uma **ação de entrada e saída** (E/S) é um valor que representa uma interação com o mundo. Uma ação de E/S pode ser *executada* pelo sistema computacional para interagir com o mundo e *retornar um valor* obtido através desta interação.

Em Haskell `IO a` é o **tipo das ações de entrada e saída** que interagem com o mundo e retornam um valor do tipo `a`. `IO a` é um *tipo abstrato*, logo sua representação não está disponível nos programas.

Haskell provê algumas ações de entrada e saída primitivas, e um mecanismo para combinar ações de entrada e saída.

12.2 Ações de saída padrão

A função putChar

```
putChar :: Char -> IO ()
```

`putChar` é uma função que recebe um caracter e resulta em uma ação de E/S que, quando executada, interage com o mundo inserindo o caracter na saída padrão e retorna a tupla vazia `()`.

Quando executada, a ação `putChar` x apenas insere x na saída padrão e não há nenhum valor interessante para ser retornado. Como toda ação deve retornar um valor quando executada, a tupla vazia `()` é usada.

Exemplo: o valor da expressão

```
putChar 'H'
```

é uma ação de E/S que, quando executada, interage com o mundo inserindo o caracter `'H'` na saída padrão e retorna e tupla vazia.

A função putStr

```
putStr :: String -> IO ()
```

A função `putStr` recebe uma string e resulta em uma ação de E/S que, quando executada, interage com o mundo inserindo a string na saída padrão e retorna a tupla vazia.

A função putStrLn

```
putStrLn :: String -> IO ()
```

A função `putStrLn` recebe uma string e resulta em uma ação de E/S que, quando executada, interage com o mundo inserindo a string seguida do caracter `'\n'` na saída padrão e retorna a tupla vazia.

A função print

```
print :: Show a => a -> IO ()
```

A função `print` recebe um valor e resulta em uma ação de E/S que, quando executada, insere na saída padrão o valor convertido para string, seguido de mudança de linha, e retorna a tupla vazia.

A conversão para string é feita usando a função `show :: Show a => a -> String`. Portanto o tipo do valor deve ser instância da classe `Show`.

12.3 Ações de entrada padrão

A ação `getChar`

```
getChar :: IO Char
```

A ação de E/S `getChar`, quando executada, interage com o mundo extraindo o próximo caracter disponível da entrada padrão e retorna este caracter.

A ação `getChar` levanta uma exceção (que pode ser identificada pelo predicado `isEOFError` do módulo `IO`) se for encontrado fim de arquivo na entrada padrão.

A ação `getLine`

```
getLine :: IO String
```

A ação de E/S `getLine`, quando executada, interage com o mundo extraindo a próxima linha disponível na entrada padrão e retorna esta linha. A ação `getLine` pode falhar com uma exceção se encontrar o fim de arquivo ao ler o primeiro caracter.

A ação `getContents`

```
getContents :: IO String
```

A ação de E/S `getContents`, quando executada, interage com o mundo extraindo todos os caracteres da entrada padrão e retorna a string formada pelos caracteres.

A ação `readLn`

```
readLn :: Read a => IO a
```

A ação de E/S `readLn`, quando executada, interage com o mundo extraindo a próxima linha disponível na entrada padrão e retorna um valor obtido dessa string.

A conversão da string para o valor é feita usando uma função similar à função `read`, com a diferença de que se a conversão falhar o programa não termina, mas uma exceção é levantada no sistema de E/S. Portanto o tipo do valor deve ser instância da classe `Read`.

12.4 Programa em Haskell

Já que Haskell é uma linguagem pura, você pode estar perguntando quando é que uma ação de entrada e saída é executada.

> Um *programa* em Haskell é uma *ação de E/S*.

Quando o *sistema operacional* executa o programa, a ação de E/S é executada. **Executar** o programa implica em executar a ação de E/S que o constitui. Logo não é Haskell que é responsável pela interação com o mundo, mas o sistema operacional. Desta forma a linguagem continua sendo pura.

Um **programa** é organizado como uma *coleção de módulos*. Um dos módulos deve ser chamado `Main` e deve exportar a variável `main`, do tipo `IO t`, para algum tipo `t`. Quando o programa é *executado* pelo sistema operacional, a ação `main` é executada, e o seu resultado (do tipo `t`) é descartado.

Exemplo de programa em Haskell

Exibir um caracter.

```
module Main (main) where

main :: IO ()
main = putChar 'A'
```

Quando o programa é **executado**:

1. `main` recebe (automaticamente) como argumento o mundo existente antes de sua execução,
2. realiza ações de entrada e saída,
3. resultando em uma tupla vazia (nenhum valor interessante é produzido), e
4. produzindo um novo mundo que reflete o efeito das ações de entrada e saída realizadas.

Tarefa 12.1: Preparando e executando um programa em Haskell

1. *Grave o código fonte* do programa em um arquivo texto, digamos `putchar-a.hs`.

```
module Main (main) where

main :: IO ()
main = putChar 'A'
```

2. *Compile o programa* (por exemplo usando o Glasgow Haskell Compiler em um terminal):

```
$ ghc --make putchar-a
[1 of 1] Compiling Main       ( putchar-a.hs, putchar-a.o )
Linking putchar-a ...
```

3. *Execute* o programa já compilado:

```
$ ./putchar-a
A
```

12.5 Combinando ações de entrada e saída

Sendo `IO a` um tipo abstrato, como poderíamos combinar duas ações em *sequência*? Por exemplo, como exibir os caracteres 'A' e 'B' em sequência?

Haskell tem uma forma de expressão (expressão **do**) que permite combinar ações de entrada e saída a serem executadas em sequência. Exemplo:

```
do { putChar 'A'; putChar 'B' }
```

Uma **expressão do** permite combinar várias ações de E/S de forma *sequencial*. Uma **expressão do** é da *forma*

$$\texttt{do } \{ \ \textit{ação}_1 \ ; \ \ldots ; \ \textit{ação}_n \ ; \ \textit{expressao} \ \}$$

onde $n \geq 0$, e *expressão* é uma ação de E/S.

Cada $\textit{ação}_i$ pode ser da forma:

- `expressao`
  uma ação de E/S cujo retorno é ignorado

- `padrao <- expressao`
  uma ação de E/S cujo retorno é casado com o padrão indicado. O escopo das variáveis introduzidas no casamento de padrão estende-se até o final da expressão do. Se o casamento falhar, toda a ação falha.

- `let declaracoes`
  permite fazer declarações cujo escopo se estende até o final da expressão do. É semelhante à expressão

  ```
  let declaracoes in expressao
  ```

  porém sem o corpo (expressão).

O *valor* da **expressão do** é uma ação de E/S formada pela combinação sequencial das ações de E/S que a compõem. Quando a **expressão do** *é executada*, as ações que a compõem são executadas em sequência, e o *valor retornado* pela expressão do é o valor retornado pela última ação.

Exemplo de expressão do em um programa

Exibe três caracteres na saída padrão.

```
module Main (main) where

main :: IO ()
main = do { putChar 'F' ; putChar 'i' ; putChar 'm' }
```

O código seguinte é idêntico ao anterior, porém com um layout diferente:

```
module Main (main) where

main :: IO ()
main = do { putChar 'F'
          ; putChar 'i'
          ; putChar 'm'
          }
```

Regra de *layout* com a expressão do

A expressão `do` pode usar a **regra de *layout*** da mesma maneira que `let`, `where` e `case`. Assim as chaves `{` e `}` e os pontos-e-vírgula `;` podem ser omitidos, sendo substituídos por uso de indentação adequada. Neste caso, cada ação que compõe a expressão `do` deve começar na mesma coluna e, se continuar em linhas subsequentes, deve sempre ocupar as colunas à direita da coluna onde iniciou. Uma linha que começa em uma coluna mais à esquerda da coluna de referência encerra a expressão `do`.

Exemplo: exibe três caracteres na saída padrão.

```
module Main (main) where

main :: IO ()
main = do putChar 'F'
          putChar 'i'
          putChar 'm'
```

12.6 Exemplos de programas interativos

Exemplo: ler um caracter

Obter um caracter da entrada padrão.

```
module Main (main) where

main :: IO Char
main = getChar
```

Exemplo: ler e exibir um caracter

Obter um caracter da entrada padrão e exibi-lo na saída padrão.

```
module Main (main) where

main :: IO ()
main = do caracter <- getChar
          putChar caracter
```

Exemplo: ler e exibir um caracter (v2)

Ler um caracter e exibi-lo em minúsculo e em maiúsculo.

```
module Main (main) where

import Data.Char (toLower, toUpper)

main :: IO ()
main = do letra <- getChar
          putChar (toLower letra)
          putChar (toUpper letra)
```

Exemplo: saudação

Ler o nome do usuário e exibir uma saudação.

```
module Main (main) where

main :: IO ()
main = do putStrLn "Qual é o seu nome? "
          nome <- getLine
          putStr nome
          putStrLn ", seja bem vindo(a)!"
```

Tarefa 12.2: Palíndromes

Escreva um programa em Haskell que solicita ao usuário para digitar uma frase, lê a frase (uma linha) da entrada padrão e testa se a string lida é uma palíndrome, exibindo uma mensagem apropriada.

Exemplo de execução da aplicação

```
Digite uma frase:
abcddcba
É uma palíndrome.
```

Exemplo de execução da aplicação

```
Digite uma frase:
ABCdCBA
É uma palíndrome.
```

Exemplo de execução da aplicação

```
Digite uma frase:
ouro preto
Não é uma palíndrome.
```

Exemplo: soma de dois números

Ler dois números e exibir a soma dos mesmos.

```haskell
module Main (main) where

main :: IO ()
main = do putStrLn "Digite um número: "
          s1 <- getLine
          putStrLn "Digite outro número: "
          s2 <- getLine
          putStr "Soma dos números digitados: "
          putStrLn (show (read s1 + read s2))
```

Exemplo: soma de dois números (v2)

Ler dois números e exibir a soma dos mesmos.

```haskell
module Main (main) where

main :: IO ()
main = do putStrLn "Digite um número: "
          n1 <- readLn
          putStrLn "Digite outro número: "
          n2 <- readLn
          putStr "Soma dos números digitados: "
          putStrLn (show (n1 + n2))
```

Tarefa 12.3

Escreva um programa que solicita ao usuário três números em ponto flutuante, lê os números, e calcula e exibe o produto dos números.

Dica

Provavelmente será necessária uma anotação de tipo para que o programa funcione com números em ponto flutuante, pois a operação de multiplicação é definida para todos os tipos numéricos e, não havendo informações no contexto suficientes para decidir o tipo numérico a ser usado, o tipo `Integer` é escolhido. A anotação de tipo pode ser feita em qualquer subexpressão do programa.

Exemplo de execução da aplicação

```
Digite um número:
10
Digite outro número:
2.3
Digite outro número:
5
Produto dos números digitados: 115.0
```

12.7 Saída bufferizada

Reconsidere o exemplo do programa para calcular e exibir a soma de dois números, onde o usuário deve digitar cada número na mesma linha da mensagem que solicita o número.

```
module Main (main) where

main :: IO ()
main =
  do putStr "Digite um número: "    -- observe que não há uma mudança de linha
     s1 <- getLine
     putStr "Digite outro número: " -- observe que não há uma mudança de linha
     s2 <- getLine
     putStr "Soma dos números digitados: "
     putStrLn (show (read s1 + read s2))
```

Execução do programa onde o usuário informa os números 34 e 17:

```
34
17
Digite um número: Digite outro número: Soma dos números digitados: 51
```

O que aconteceu de errado?

A saída para o dispositivo padrão de saída é *bufferizada*. Isto significa que o sistema operacional mantém uma área da memória (chamada de **buffer**) onde armazena os caracteres a serem enviados para o dispositivo de saída. Geralmente os caracteres enviados para a saída padrão somente são transferidos para o dispositivo de saída quando o **buffer** estiver cheio. Este mecanismo reduz o número de acesso aos dispositivos de saída (que são muito mais lentos que o processador), melhorando o desempenho da aplicação. Por este motivo as mensagens não aparecem imediatamente quando o programa anterior é executado.

A função `hFlush` (definida no módulo `System.IO`) recebe um manipulador de arquivo (*handle*) e resulta em uma ação de E/S que, quando executada, faz com que os itens armazenados no *buffer* de saída do manipulador sejam *enviados imediatamente* para a saída.

```haskell
hFlush :: Handle -> IO ()
```

O tipo `Handle` (definido no módulo `System.IO`) é um tipo abstrato que representa um dispositivo de E/S internamente para o Haskell.

O módulo `System.IO` define variáveis que representam alguns dispositivos padrões:

```haskell
stdin  :: Handle  -- entrada padrão
stdout :: Handle  -- saída padrão
stderr :: Handle  -- saída de erro padrão
```

Para que o exemplo dado funcione corretamente é necessário esvaziar o buffer da saída padrão antes de fazer a entrada de dados, como mostra a nova versão do programa.

```haskell
module Main (main) where

import System.IO (stdout, hFlush)

main :: IO ()
main = do putStr "Digite um número: "
          hFlush stdout                    -- esvazia o buffer de saída
          s1 <- getLine
          putStr "Digite outro número: "
          hFlush stdout                    -- esvazia o buffer de saída
          s2 <- getLine
          putStr "Soma dos números digitados: "
          putStrLn (show (read s1 + read s2))
```

Execução do programa onde o usuário informa os números 34 e 17:

```
Digite um número: 34
Digite outro número: 17
Soma dos números digitados: 51
```

A função `hSetBuffering` (definida no módulo `System.IO`) pode ser utilizada para configurar o modo de *bufferização* de um dispositivo.

```haskell
hSetBuffering :: Handle -> BufferMode -> IO ()
```

O tipo `BufferMode` (definido no módulo `System.IO`) representa um modo de *bufferização*:

- sem buferização: `NoBuffering`
- buferização por linha: `LineBuffering`
- buferização por bloco: `BlockBuffering`

Normalmente a saída para o dispositivo padrão é feita com *buferização por linha*. A expressão

```haskell
hSetBuffering hdl mode
```

é uma ação que, quando executada, configura o modo de *bufferização* para o *handler* `hdl`.

Então podemos corrigir o problema no exemplo dado anteriormente adicionando a ação

```haskell
hSetBuffering stdout NoBuffering
```

no começo da sequência de ações para desabilitar a bufferização da saída padrão.

```haskell
module Main (main) where

import System.IO (stdout, hSetBuffering, BufferMode(NoBuffering))

main :: IO ()
main = do hSetBuffering stdout NoBuffering -- desabilita a bufferização
          putStr "Digite um número: "
          s1 <- getLine
          putStr "Digite outro número: "
          s2 <- getLine
          putStr "Soma dos números digitados: "
          putStrLn (show (read s1 + read s2))
```

Execução do programa onde o usuário informa os números 34 e 17:

```
Digite um número: 34
Digite outro número: 17
Soma dos números digitados: 51
```

Tarefa 12.4: Conversão de temperaturas

Escreva um programa em Haskell que solicita ao usuário uma temperatura na escala Fahrenheit, lê esta temperatura, converte-a para a escala Celsius, e exibe o resultado.

Para fazer a conversão, defina uma função `celsius :: Double -> Double` que recebe a temperatura na escala Fahrenheit e resulta na temperatura correspondente na escala Celsius. Use a seguinte equação para a conversão:

$$C = \frac{5}{9} \times (F - 32)$$

onde F é a temperatura na escala Fahrenheit e C é a temperatura na escala Celsius.

Use a função `celsius` na definição de `main`.

A digitação da temperatura em Fahrenheit deve ser feita na mesma linha onde é exibida a mensagem que a solicita.

Exemplo de execução da aplicação

```
Temperatura em Fahrenheit: 100
Temperatura em Celsius: 37.77777777777778
```

12.8 Exemplos

Peso ideal

Escrever um programa em Haskell que recebe a altura e o sexo de uma pessoa e calcula e mostra o seu peso ideal, utilizando as fórmulas constantes na tabela a seguir.

sexo	peso ideal
masculino	$72.7 \times h - 58$
feminino	$62.1 \times h - 44.7$

onde h é a altura da pessoa.

```haskell
module Main (main) where

import System.IO (stdout, hSetBuffering, BufferMode(NoBuffering))
import Data.Char (toUpper)

main :: IO ()
main =
  do hSetBuffering stdout NoBuffering
     putStr "Altura: "
     h <- readLn
     putStr "Sexo (f/m): "
     s <- getLine
     case s of
       [x] | toUpper x == 'F' -> putStrLn ("Peso ideal: " ++ show (62.1 * h - 44.7))
           | toUpper x == 'M' -> putStrLn ("Peso ideal: " ++ show (72.7 * h - 58))
       _                      -> putStrLn "Sexo inválido"
```

Situação do aluno

Faça um programa que receba três notas de um aluno, e calcule e mostre a média aritmética das notas e a situação do aluno, dada pela tabela a seguir.

média das notas	**situação**
menor que 3	reprovado
entre 3 (inclusive) e 7	exame especial
acima de 7 (inclusive)	aprovado

```haskell
module Main (main) where
import System.IO (stdout, hSetBuffering, BufferMode(NoBuffering))

prompt :: Read a => String -> IO a
prompt msg = do putStr msg
                readLn

main :: IO ()
main = do hSetBuffering stdout NoBuffering
          n1 <- prompt "Nota 1: "
          n2 <- prompt "Nota 2: "
          n3 <- prompt "Nota 3: "
          let media = (n1 + n2 + n3)/3
          putStrLn ("Média: " ++ show media)
          putStr "Situação: "
          if media < 3
            then putStrLn "reprovado"
            else if media < 7
                   then putStrLn "exame especial"
                   else putStrLn "aprovado"
```

Raízes da equação do segundo grau

Faça um programa que leia os coeficientes de uma equação do segundo grau e calcule e mostre suas raízes reais, caso existam.

```haskell
module Main (main) where
import System.IO (stdout, hSetBuffering, BufferMode(NoBuffering))

raizes2grau a b c
  | d > 0     = [ (-b + sqrt d)/(2*a), (-b - sqrt d)/(2*a) ]
  | d == 0    = [ -b/(2*a) ]
  | otherwise = [ ]
  where d = b^2 - 4*a*c

prompt mensagem = do { putStr mensagem; readLn }

main = do hSetBuffering stdout NoBuffering
          putStrLn "Cálculo das raízes da equação do segundo grau"
          putStrLn "a x^2 + b x + c = 0"
          a <- prompt "Coeficiente a: "
          b <- prompt "Coeficiente b: "
          c <- prompt "Coeficiente c: "
          case raizes2grau a b c of
            [r1,r2] -> putStrLn ("Raízes: " ++ show r1 ++ " e " ++ show r2)
            [r]     -> putStrLn ("Raíz: " ++ show r)
            []      -> putStrLn "Não há raízes reais"
```

12.9 Problemas

Tarefa 12.5: Linha de crédito

A prefeitura de Contagem abriu uma linha de crédito para os funcionários estatutários. O valor máximo da prestação não poderá ultrapassar 30% do salário bruto.

Fazer um programa que permita entrar com o salário bruto e o valor da prestação, e informar se o empréstimo pode ou não ser concedido.

Exemplo de execução da aplicação

```
Análise de crédito
-------------------------------------------
Salário bruto: 1000
Valor da prestação: 20
O empréstimo pode ser concedido
```

Exemplo de execução da aplicação

```
Análise de crédito
-------------------------------------------
Salário bruto: 1000
Valor da prestação: 430.23
O empréstimo não pode ser concedido
```

Tarefa 12.6: Classe eleitoral

Crie um programa que leia a idade de uma pessoa e informe a sua classe eleitoral:

não eleitor abaixo de 16 anos;

eleitor obrigatório entre 18 (inclusive) e 65 anos;

eleitor facultativo de 16 até 18 anos e acima de 65 anos (inclusive).

Exemplo de execução da aplicação

```
Classe eleitoral
----------------------------------------
Digite a idade da pessoa: 11
não eleitor
```

Exemplo de execução da aplicação

```
Classe eleitoral
----------------------------------------
Digite a idade da pessoa: 17
eleitor facultativo
```

Exemplo de execução da aplicação

```
Classe eleitoral
----------------------------------------
Digite a idade da pessoa: 20
eleitor obrigatório
```

Exemplo de execução da aplicação

```
Classe eleitoral
----------------------------------------
Digite a idade da pessoa: 73
eleitor facultativo
```

Tarefa 12.7: Impostos

Faça um programa que apresente o menu a seguir, permita ao usuário escolher a opção desejada, receba os dados necessários para executar a operação, e mostre o resultado.

```
--------------------------------
Opções:
--------------------------------
1. Imposto
2. Novo salário
3. Classificação
--------------------------------
Digite a opção desejada:
```

Verifique a possibilidade de opção inválida.

Na **opção 1** receba o salário de um funcionário, calcule e mostre o valor do imposto sobre o salário usando as regras a seguir:

salário	taxa de imposto
Abaixo de R$500,00	5%
De R$500,00 a R$850,00	10%
Acima de R$850,00	15%

Na **opção 2** receba o salário de um funcionário, calcule e mostre o valor do novo salário, usando as regras a seguir:

salário	aumento
Acima de R$1.500,00	R$25,00
De R$750,00 (inclusive) a R$1.500,00 (inclusive)	R$50,00
De R$450,00 (inclusive) a R$750,00	R$75,00
Abaixo de R$450,00	R$100,00

Na **opção 3** receba o salário de um funcionário e mostre sua classificação usando a tabela a seguir:

salário	classificação
Até R$750,00 (inclusive)	mal remunerado
Acima de R$750,00	bem remunerado

Exemplo de execução da aplicação

```
Análise de salário
-------------------------------
Opções:
-------------------------------
1. Imposto
2. Novo salário
3. Classificação
-------------------------------
Digite a opção desejada: 1

Cálculo do imposto
Digite o salário: 700
Imposto calculado: 70.0
```

Exemplo de execução da aplicação

```
Análise de salário
-------------------------------
Opções:
-------------------------------
1. Imposto
2. Novo salário
3. Classificação
-------------------------------
Digite a opção desejada: 2

Cálculo do novo salário
Digite o salário: 700
Novo salário: 775
```

Exemplo de execução da aplicação

```
Análise de salário
--------------------------------
Opções:
--------------------------------
1. Imposto
2. Novo salário
3. Classificação
--------------------------------
Digite a opção desejada: 3

Classificação do salário
Digite o salário: 700
Classificação obtida: mal remunerado
```

Exemplo de execução da aplicação

```
Análise de salário
--------------------------------
Opções:
--------------------------------
1. Imposto
2. Novo salário
3. Classificação
--------------------------------
Digite a opção desejada: 4

Opção inválida!
```

Tarefa 12.8: Terno pitagórico

Em Matemática um **terno pitagórico** (ou trio pitagórico, ou ainda tripla pitagórica) é formado por três números a, b e c tais que $a^2 + b^2 = c^2$. O nome vem do *teorema de Pitágoras* que afirma que se as medidas dos lados de um triângulo retângulo são números inteiros, então elas formam um terno pitagórico.

Codifique um programa que leia três números positivos e verifique se eles formam um terno pitagórico.

Exemplo de execução da aplicação

```
Verificação de ternos pitagóricos
-----------------------------------------
Digite o primeiro número positivo .....: 3
Digite o segundo número positivo ......: 4
Digite o terceiro número positivo .....: 5
Os números formam um terno pitagórico
```

Exemplo de execução da aplicação

```
Verificação de ternos pitagóricos
-----------------------------------------
Digite o primeiro número positivo .....: 6
Digite o segundo número positivo ......: 5
Digite o terceiro número positivo .....: 4
Os números não formam um terno pitagórico
```

Exemplo de execução da aplicação

```
Verificação de ternos pitagóricos
-----------------------------------------
Digite o primeiro número positivo .....: 3
Digite o segundo número positivo ......: -4
Digite o terceiro número positivo .....: 0
Números inválidos
```

12.10 Soluções

Tarefa 12.2 on page 12-8: Solução

Tarefa 12.3 on page 12-9: Solução

```haskell
module Main (main) where

main :: IO ()
main = do putStrLn "Digite um número: "
          n1 <- readLn :: IO Double
          putStrLn "Digite outro número: "
          n2 <- readLn
          putStrLn "Digite outro número: "
          n3 <- readLn
          putStr "Produto dos números digitados: "
          print (n1 * n2 * n3)
```

Tarefa 12.4 on page 12-11: Solução

```haskell
module Main (main) where

import System.IO (stdout, hSetBuffering, BufferMode(NoBuffering))

converte :: Double -> Double
converte f = 5/9 * (f - 32)

main :: IO ()
main = hSetBuffering stdout NoBuffering >>
       putStr "Temperatura em Fahrenheit: " >>
       getLine >>= \str ->
       let f = read str
           c = converte f
       in  putStrLn ("Temperatura em Celsius: " ++ show c)
```

Tarefa 12.5 on page 12-13: Solução

```haskell
module Main (main) where

import System.IO (stdout, hSetBuffering, BufferMode(NoBuffering))

main =
  do hSetBuffering stdout NoBuffering
     putStrLn "Análise de crédito"
     putStrLn "-----------------------------------------"
     putStr "Salário bruto: "
     salario <- readLn
     putStr "Valor da prestação: "
     prestação <- readLn
     if prestação <= 0.3 * salario
       then putStrLn "O empréstimo pode ser concedido"
       else putStrLn "O empréstimo não pode ser concedido"
```

Tarefa 12.6 on page 12-14: Solução

```haskell
module Main (main) where

import System.IO (stdout, hSetBuffering, BufferMode(NoBuffering))

classeEleitoral idade
  | idade < 16                = "não eleitor"
  | idade >= 18 && idade < 65 = "eleitor obrigatório"
  | otherwise                 = "eleitor facultativo"

main =
  do hSetBuffering stdout NoBuffering
     putStrLn "Classe eleitoral"
     putStrLn "-----------------------------------------"
     putStr "Digite a idade da pessoa: "
     idade <- readLn
     putStrLn (classeEleitoral idade)
```

Tarefa 12.7 on page 12-14: Solução

```haskell
module Main (main) where

import System.IO (stdout, hSetBuffering, BufferMode(NoBuffering))

main =
  do hSetBuffering stdout NoBuffering
     putStrLn "Análise de salário"
     putStrLn "-------------------------------"
     putStrLn "Opções:"
     putStrLn "-------------------------------"
     putStrLn "1. Imposto"
     putStrLn "2. Novo salário"
     putStrLn "3. Classificação"
     putStrLn "-------------------------------"
     putStr "Digite a opção desejada: "
     opção <- readLn
     putStrLn ""
     case opção of
       1 -> do putStrLn "Cálculo do imposto"
               putStr "Digite o salário: "
               salário <- readLn
               let taxa | salário < 500 = 5
                        | salário < 850 = 10
                        | otherwise     = 15
                   imposto = taxa * salário / 100
               putStrLn ("Imposto calculado: " ++ show imposto)
       2 -> do putStrLn "Cálculo do novo salário"
               putStr "Digite o salário: "
               salário <- readLn
               let aumento | salário > 1500 = 25
                           | salário >= 750 = 50
                           | salário >= 450 = 75
                           | otherwise      = 100
                   novoSalário = salário + aumento
               putStrLn ("Novo salário: " ++ show novoSalário)
       3 -> do putStrLn "Classificação do salário"
               putStr "Digite o salário: "
               salário <- readLn
               let classificação | salário <= 750 = "mal remunerado"
                                 | otherwise      = "bem remunerado"
               putStrLn ("Classificação obtida: " ++ classificação)
       _ -> putStrLn "Opção inválida!"
```

Tarefa 12.8 on page 12-17: Solução

```haskell
module Main (main) where

import System.IO (stdout, hSetBuffering, BufferMode(NoBuffering))

main =
  do hSetBuffering stdout NoBuffering
     putStrLn "Verificação de ternos pitagóricos"
     putStrLn "----------------------------------------"
     n1 <- prompt "Digite o primeiro número positivo .....: "
     n2 <- prompt "Digite o segundo número positivo ......: "
     n3 <- prompt "Digite o terceiro número positivo .....: "
     if n1 <= 0 || n2 <= 0 || n3 <= 0
       then putStrLn "Números inválidos"
       else if ternoPitagórico n1 n2 n3
              then putStrLn "Os números formam um terno pitagórico"
              else putStrLn "Os números não formam um terno pitagórico"

prompt mensagem = do putStr mensagem
                     readLn

ternoPitagórico n1 n2 n3 = n1^2 == n2^2 + n3^2 ||
                           n2^2 == n1^2 + n3^2 ||
                           n3^2 == n1^2 + n2^2
```

13 Ações de E/S Recursivas

Resumo

Nesta aula vamos aprender a definir ações de entrada e saída recursivas, que nos permitirá definir ações repetitivas através de recursividade.

Sumário

13.1 A função `return`

Às vezes é necessário escrever uma ação de E/S que não faz nenhuma interação com o mundo e retorna um valor previamante especificado. Isto é possível usando a função `return`.

```
return :: a -> IO a
```

A função `return` recebe um valor e resulta em uma ação de E/S que não interage com o mundo e retorna o valor.

A função `return` é muito utilizada nos casos bases de definições recursivas.

13.2 Exemplo: exibir uma sequência

Faça um programa que exiba todos os números naturais pares menores ou iguais a 30.

```
module Main (main) where

mostraLista :: Show a => [a] -> IO ()
mostraLista []     = return ()
mostraLista (x:xs) = do print x
                        mostraLista xs

main :: IO ()
main = mostraLista [0,2..30]
```

13.3 Exemplo: somar uma sequência

Escreva um programa que obtém uma sequência de números inteiros até encontrar o valor zero, e mostra a soma dos números lidos.

```haskell
-- solução sem recursividade de cauda

module Main (main) where

main = do putStrLn "Digite uma sequência de números (um por linha)"
          putStrLn "Para terminar digite o valor zero"
          soma <- lerESomar
          putStr "A soma dos números digitados é "
          print soma

lerESomar = do n <- readLn
               if n == 0
                 then return 0
                 else do somaResto <- lerESomar
                         return (n + somaResto)
```

```haskell
-- solução com recursividade de cauda

module Main (main) where

main = do putStrLn "Digite uma sequência de números (um por linha)"
          putStrLn "Para terminar digite o valor zero"
          soma <- lerESomar 0
          putStr "A soma dos números digitados é "
          print soma

lerESomar total = do n <- readLn
                     if n == 0
                       then return total
                       else lerESomar (total + n)
```

```haskell
-- solução que separa a leitura do processamento
-- sem recursividade de cauda

module Main (main) where

main = do putStrLn "Digite uma sequência de números (um por linha)"
          putStrLn "Para terminar digite o valor zero"
          lista <- lerLista
          putStr "A soma dos números digitados é "
          print (sum lista)

lerLista = do x <- readLn
              if x == 0
                then return []
                else do resto <- lerLista
                        return (x:resto)
```

```haskell
-- solução que separa a leitura do processamento
-- com recursividade de cauda

module Main (main) where

main = do putStrLn "Digite uma sequência de números (um por linha)"
          putStrLn "Para terminar digite o valor zero"
          lista <- lerLista []
          putStr "A soma dos números digitados é "
          print (sum lista)

lerLista xs = do x <- readLn
                 if x == 0
                   then return (reverse xs)
                   else lerLista (x:xs)
```

13.4 Problemas

Tarefa 13.1: Soma de uma sequência de números

Faça um programa que leia um número natural n, e então leia outros n números e calcule e exiba a soma destes números.

Exemplo de execução da aplicação

```
Quantidade de números: 4
Digite um número: 10
Digite um número: -5
Digite um número: 1
Digite um número: 20
Soma dos números digitados: 26
```

Exemplo de execução da aplicação

```
Quantidade de números: -6
Soma dos números digitados: 0
```

Tarefa 13.2: Média aritmética de uma sequência de números

Faça um programa que leia uma seqüência de números não negativos e determine a média aritmética destes números. A entrada dos números deve ser encerrada com um número inválido (negativo).

Exemplo de execução da aplicação

```
Cálculo da média aritmética
---------------------------------------
Digite uma sequência de números (um por linha)
Para terminar digite um valor negativo
10
9
8
9.2
-1
A média dos números digitados é 9.171428571428573
```

Exemplo de execução da aplicação

```
Cálculo da média aritmética
---------------------------------------
Digite uma sequência de números (um por linha)
Para terminar digite um valor negativo
-5
Sequência vazia
```

Tarefa 13.3: Perda de massa por radioatividade

Um elemento químico radioativo perde sua massa de acordo com a função

$$m(t) = m_0 e^{-kt}$$

onde, t é o tempo (em segundos), m_0 é a++ massa inicial (em gramas) e k é a constante 5×10^{-2}.

Faça uma aplicação que, dada a massa inicial desse elemento, calcule a perda de massa durante um minuto, exibindo as massas resultantes em intervalos de 10 segundos.

Tarefa 13.4: Cálculo aproximado de π

A série abaixo converge para o número π quando $n \to \infty$.

$$4 \sum_{i=0}^{n} \frac{(-1)^i}{2i+1}$$

Codifique um programa que solicita ao usuário o número de parcelas da série e calcula e exibe o valor aproximado de π usando o número solicitado de parcelas.

Tarefa 13.5: Aumento salarial

Um funcionário de uma empresa recebe aumento salarial anualmente. O primeiro aumento é de 1,5% sobre seu salário inicial. Os aumentos subsequentes sempre correspondem ao dobro do percentual de aumento do ano anterior. Faça uma aplicação onde o usuário deve informar o salário inicial do funcionário, o ano de contratação e o ano atual, e calcula e exibe o seu salário atual.

Tarefa 13.6: Fechamento de notas de uma disciplina

Faça uma aplicação para fechamento das notas de uma disciplina. Cada aluno recebe uma nota para cada uma das três atividades desenvolvidas. O usuário deverá informar a quantidade de alunos na turma, e em seguida as notas de cada aluno. Calcule e exiba:

- a média aritmética das três notas de cada aluno,
- a situação do aluno, dada pela tabela seguinte

média aritmética	**situação**
até 3	reprovado
entre 3 (inclusive) e 7	exame especial
acima de 7 (inclusive)	aprovado

- a média da turma
- o percentual de alunos aprovados
- o percentual de alunos em exame especial
- o percentual de alunos reprovados

Dicas Primeiramente obtenha os dados armazenando-os em uma lista e posteriormente processe os dados para calcular e exibir cada um dos ítens solicitados. Faça uma função para calcular a resposta em cada caso, e use a função na definição de `main`.

Tarefa 13.7: Correção de provas de múltipla escolha

Faça um programa para corrigir provas de múltipla escolha que foram aplicadas em uma turma de alunos. O usuário deverá informar:

- o gabarito (as respostas corretas de cada questão) da prova
- a matrícula e as respostas de cada aluno da turma

As notas devem ser normalizadas na faixa de zero a dez. Assim para calcular a nota obtida em uma prova, divida a soma dos pontos obtidos (um ponto para cada resposta correta) pelo número de questões na prova, e multiplique o resulado por dez.

Calcule e mostre:

1. a matrícula e a nota de cada aluno
2. a taxa (em porcentagem) de aprovação, sabendo-se que a nota mínima para aprovação é sete.

Dicas

- Faça uma ação de E/S para obter os dados do gabarito (uma string onde caracter é a resposta correta de uma questão).
- Faça uma ação de E/S para obter os dados das provas dos alunos (uma lista de pares, onde o primeiro componente do par é a matrícula do aluno (um número inteiro), e o segundo componente do par são as respostas do aluno (uma string)).
- Faça uma função que recebe o gabarito e a lista das provas dos alunos e resulta na lista dos resultados, formada por pares contendo a matrícula e a nota do aluno.
- Faça uma função que recebe a lista dos resultados e resulta na porcentagem de aprovação.
- Use estas funções para montar a aplicação.

13.5 Soluções

Tarefa 13.1 on page 13-3: Solução

Primeira versão: sem recursividade de cauda

```
module Main (main) where

import System.IO (stdout, hSetBuffering, BufferMode(NoBuffering))

main = do hSetBuffering stdout NoBuffering
          putStr "Quantidade de números: "
          n <- readLn
          s <- lerESomar n
          putStr "Soma dos números digitados: "
          print s

lerESomar :: Integer -> IO Integer
lerESomar n | n <= 0 = return 0
            | n > 0  = do putStr "Digite um número: "
                          x <- readLn
                          s <- lerESomar (n-1)
                          return (x+s)
```

Segunda versão: com recursividade de cauda

```
module Main (main) where

import System.IO (stdout, hSetBuffering, BufferMode(NoBuffering))

main = do hSetBuffering stdout NoBuffering
          putStr "Quantidade de números: "
          n <- readLn
          s <- lerESomar n 0
          putStr "Soma dos números digitados: "
          print s

lerESomar n s | n <= 0 = return s
              | n > 0  = do putStr "Digite um número: "
                            x <- readLn
                            lerESomar (n-1) (x+s)
```

Terceira versão: entrada dos dados e cálculos separados

```haskell
module Main (main) where

import System.IO (stdout, hSetBuffering, BufferMode(NoBuffering))

main = do hSetBuffering stdout NoBuffering
          putStr "Quantidade de números: "
          n <- readLn
          lista <- leLista n
          let soma = sum lista
          putStrLn ("Soma dos números digitados: " ++ show soma)

leLista n | n <= 0    = return []
          | otherwise = do putStr "Digite um número: "
                           x <- readLn
                           xs <- leLista (n-1)
                           return (x:xs)
```

Tarefa 13.2 on page 13-4: Solução

```haskell
module Main (main) where

main = do putStrLn "Cálculo da média aritmética"
          putStrLn "--------------------------------------"
          putStrLn "Digite uma sequência de números (um por linha)"
          putStrLn "Para terminar digite um valor negativo"
          lista <- lerLista
          case lista of
            [] -> putStrLn "Sequência vazia"
            _  -> do putStr "A média dos números digitados é "
                     print (sum lista / fromIntegral (length lista))

lerLista = do x <- readLn
              if x < 0
                then return []
                else do xs <- lerLista
                        return (x:xs)
```

4 Argumentos da Linha de Comando e Arquivos

Resumo

Nesta aula vamos aprender a escrever aplicações que obtém dados de arquivos e que gravam os resultados calculados em arquivos. Vamos também aprender a usar argumentos passados para um programa na linha de comando.

Sumário

14.1 Argumentos da linha de comando

Quando um programa é iniciado, ele pode receber argumentos através da linha de comando, e estes argumentos podem ser usados durante a execução do programa. O módulo `System.Environment` exporta algumas definições que podem ser usadas para acessar estes argumentos.

A ação `getArgs`

```
getArgs :: IO [String]
```

A ação de E/S `getArgs` (definida no módulo `System.Environment`), quando executada, retorna uma lista formada pelos argumentos da linha de comando do programa.

A ação `getProgName`

```
getProgName :: IO String
```

A ação de E/S `getProgName` (definida no módulo `System.Environment`), quando executada, retorna o nome do programa.

Exemplo: argumentos do programa

`args.hs`

```haskell
module Main (main) where

import System.Environment (getArgs, getProgName)

main =
  do progName <- getProgName
     putStr "The program name is ...: "
     print progName
     args <- getArgs
     putStr "The arguments are......: "
     print args
```

Exemplo de execução da aplicação

```
$ ghc --make args
[1 of 1] Compiling Main             ( args.hs, args.o )
Linking args ...

$ ./args a b c -o test
The program name is ...: "args"
The arguments are......: ["a","b","c","-o","test"]
```

Quando o programa for executado no ambiente interativo, o nome do programa e os argumentos a serem usados pelo programa podem ser especificados usando os comandos `:set prog` e `:set args`, respectivamente.

Exemplo de execução da aplicação

```
*Main> main
The program name is ...: "<interactive>"
The arguments are......: []
```

Exemplo de execução da aplicação

```
*Main> :set prog test-args

*Main> :set args -pdf entrada.txt saida.txt

*Main> main
The program name is ...: "test-args"
The arguments are......: ["-pdf","entrada.txt","saida.txt"]
```

14.2 Encerrando o programa explicitamente

O módulo `System.Exit` exporta algumas definições úteis para encerrar a execução do programa explicitamente.

O tipo `ExitCode`

Quando a execução de um programa termina, o ambiente onde o programa foi executado (normalmente o sistema operacional) recebe um código (*status*) de retorno. Este código pode ser inspecionado pelo ambiente de execução para verificar em que condições a execução do programa terminou. Tipicamente o valor zero indica sucesso, e um valor diferente de zero indica falha.

O tipo `ExitCode` define códigos de saída que podem ser retornados por um programa quando ele é encerrado. Este tipo possui dois construtores de dados:

- `ExitSuccess`: indica término com sucesso.

- `ExitFailure Int`: indica falha com um código de saída; a interpretação exata do código é dependente do sistema operacional.

A função `exitWith`

```haskell
exitWith :: ExitCode -> IO a
```

A função `exitWith` pode ser usada para criar uma ação de E/S que, quando executada, termina o programa com o código de saída especificado.

A ação de E/S `exitFailure`

```haskell
exitFailure :: IO ()
```

A ação de E/S `exitFailure`, quando executada, termina o programa com um código de falha que é dependente da implementação.

A ação de E/S `exitSuccess`

```haskell
exitSuccess :: IO ()
```

A ação de E/S `exitSuccess`, quando executada, termina o programa com uma indicação de sucesso.

Exemplo: validando os argumentos da linha de comando

```haskell
module Main (main) where

import System.Environment (getArgs, getProgName)
import System.Exit (exitFailure)

main =
  do args <- getArgs
     case args of
       [input,output] ->
         do putStrLn ("Entrada: " ++ input)
            putStrLn ("Saída  : " ++ output)
       _ ->
         do progName <- getProgName
            putStrLn ("Chamada inválida do programa " ++ progName)
            putStrLn ("Uso: " ++ progName ++ " <arquivo-entrada> <arquivo-saída>")
            exitFailure
```

Exemplo de execução da aplicação

```
$ ./validate-args arquivo1.txt arquivo2.txt
Entrada: arquivo1.txt
Saída  : arquivo2.txt
```

Exemplo de execução da aplicação

```
$ ./validate-args -pdf arquivo1.txt arquivo2.txt fim
Chamada inválida do programa validate-args
Uso: validate-args <arquivo-entrada> <arquivo-saída>
```

14.3 Formatando dados com a função `printf`

O módulo `Text.Printf` oferece a possibilidade de formatação de textos usando formatadores semelhantes àqueles disponíveis em C através da função `printf`.

```
printf :: PrintfType r => String -> r
```

A função `printf` (definida no módulo `Text.Printf`) formata um número variável de argumentos usando uma string de formatação no estilo da função `printf` da linguagem C. O resultado pode ser de qualquer tipo que seja instância da classe `PrintfType`, que inclui os tipos `String` e `IO a`.

A string de formatação consiste de caracteres comuns e especificações de conversão, que podem especificar como formatar um dos argumentos na string de saída. Uma especificação de formato é introduzida pelo caracter % e termina com um caracter de formato que é a principal indicação de como o valor deve ser formatado. Use %% para inserir o próprio caracter % na string de formatação. O restante da specificação de conversão é opcional, podendo ser caractares de *flag*, espeficador de tamanho, specificador de precisão, e caracteres modificadores de tipo, nesta ordem.

Caracteres de *flag*

-	alinhamento à esquerda (o padrão é à direita)
+	sempre use um singal (+ ou -) para conversão com sinal
espaço	espaço na frente de números positivos para conversão com sinal
`0`	complete com zeros (o padrão é espaços)
#	use uma forma alternativa (veja abaixo)

Formas alternativas

%o	prefixa com um `0` se necessário
%x	prefixa com um `0x` se diferente de zero
%X	prefixa com um `0X` se diferente de zero
%b	prefixa com um `0b` se diferente de zero
%[eEfFgG]	garante que o número contém um ponto decimal

Tamanho de campo

num	largura mínima do campo
*	largura mínima do campo tomada da lista de argumentos

Precisão

.num	precisão
.	o mesmo que `.0`
.*	precisão tomada da lista de argumentos

O signifcado da precisão depende do tipo de conversão:

`Integral`	número mínimo de dígitos a serem exibidos
`RealFloat`	número de dígitos depois do ponto decimal
`String`	número máximo de caracteres

Modificadores de tamanho

hh	`Int8`
h	`Int16`
l	`Int32`
ll	`Int64`
L	`Int64`

Caracteres de formatação

c	caracter	`Integral`
d	decimal	`Integral`
o	octal	`Integral`
x	hexadecimal	`Integral`
X	hexadecimal	`Integral`
b	binário	`Integral`
u	decimal sem sinal	`Integral`
f	ponto flutuante	`RealFloat`
F	ponto flutuante	`RealFloat`
g	ponto flutuante geral	`RealFloat`
G	ponto flutuante geral	`RealFloat`
e	ponto flutuante com expoente	`RealFloat`
E	ponto flutuante com expoente	`RealFloat`
s	string	`String`
v	padrão	qualquer tipo

Exemplos

```
Text.Printf> printf "%d\n" 23
23

Text.Printf> printf ":%7d:%-7d:%+7d:%7d:%07d:\n" 2014 2015 2016 2017 2018
:   2014:2015   :  +2016:   2017:0002018:

Text.Printf> printf "%d %o %x %b %#b" 123 123 123 123 123
123 173 7b 1111011 0b1111011

Text.Printf> printf "/%d/%7d/%*d/%4f/%.2f/" 745 745 10 745 pi pi
/745/    745/       745/3.141592653589793/3.14/

Text.Printf> printf "%s %s!\n" "Hello" "World"
Hello World!

Text.Printf> printf "sin(%.2f) = %f\n" pi (sin pi)
sin(3.14) = 0.00000000000000012246467991473532
```

Veja a documentação completa da função `printf` em `http://hackage.haskell.org/package/base-4.7.0.1/docs/Text-Printf.html`

14.4 Arquivos

Haskell possui várias definições para manipular arquivos definidas no módulo `System.IO`. Algumas delas são mencionadas a seguir.

```
type FilePath = String
```

Tipo usado para representar o caminho de um arquivo, incluindo o seu nome.

```
readFile :: FilePath -> IO String
```

Lê o conteúdo de um arquivo como uma única string.

```
writeFile :: FilePath -> String -> IO ()
```

Grava uma string em um arquivo.

```
appendFile :: FilePath -> String -> IO ()
```

Acrescenta uma string no final de um arquivo.

14.5 As funções lines e unlines, e words e unwords

A função lines

```
lines :: String -> [String]
```

A função `lines` divide uma string em uma lista de strings nas mudanças de linha. As strings resultantes não contem o caracter de mudança de linha.

Por exemplo:

```
lines "aa\nbb\nbb\n\nzz\n"
⇝ ["aa","bb","bb","","zz"]

lines "1234 Pedro 1.5 1.7\n1111 Carla 6.2 7.0\n2121 Rafael 8.1 8.8"
⇝ ["1234 Pedro 1.5 1.7","1111 Carla 6.2 7.0","2121 Rafael 8.1 8.8"]
```

A função unlines

```
unlines :: [String] -> String
```

A função `unlines` é uma operação inversa de `lines`. Ela junta as strings da lista dada após acrescentar o caracter de mudança de linha no final de cada uma delas.

Por exemplo:

```
unlines ["aa","bb","bb","zz"]
⇝ "aa\nbb\nbb\nzz\n"
```

A função words

```
words :: String -> [String]
```

A função `words` divide uma string em uma lista de strings nos caracteres brancos (espaço, tabulação, mudança de linha, etc). As strings resultantes não contem caracteres brancos.

Por exemplo:

```
words "aa bb\tbb        zz"
⇝["aa","bb","bb","zz"]
```

A função unwords

```
unwords :: [String] -> String
```

A função `unwords` é uma operação inversa de `lines`. Ela junta as strings da lista dada acrescentando um espaço entre elas.

Por exemplo:

```
unlines ["aa","bb","bb","zz"]
⇝ "aa bb bb zz"
```

14.6 Exemplo: processar notas em arquivo

Tarefa 14.1

Criar um programa para ler de um arquivo os dados dos alunos de uma turma (a matrícula, o nome, a nota na primeira avaliação, e a nota na segunda avaliação), calcular a média aritmética das notas das duas avaliações, e determinar a situação de cada aluno, gravando os resultados em outro arquivo.

A situação do aluno é dada pela tabela seguinte

média aritmética das notas	**situação**
até 3	reprovado
entre 3 (inclusive) e 7	exame especial
acima de 7 (inclusive)	aprovado

Os nomes dos arquivos de entrada e saída devem ser informados como argumentos da linha de comando.

Exemplo de arquivo de entrada:

```
1234    Pedro   1.5     1.7
1111    Carla   6.2     7.0
2121    Rafael  8.1     8.8
4321    Ivan    5.0     5.2
```

Arquivo de saída correspondente:

```
1234 Pedro 1.5 1.7 1.6 reprovado
1111 Carla 6.2 7.0 6.6 exame especial
2121 Rafael 8.1 8.8 8.45 aprovado
4321 Ivan 5.0 5.2 5.1 exame especial
```

```haskell
module Main (main) where

import System.Environment (getArgs)
import System.Exit (exitFailure)

main =
  do args <- getArgs
     case args of
       [nome1,nome2] ->
         do str <- readFile nome1
            writeFile nome2 (processa str)
       _ -> exitFailure

processa s =
  unlines (processaAlunos (lines s))

-- processaAlunos [] = []
-- processaAlunos (x:xs) = processaAluno x : processaAlunos xs

-- processaAlunos lista = map processaAluno lista

processaAlunos = map processaAluno

processaAluno s =
  case words s of
    [mat,nome,nota1,nota2] ->
      let media = (read nota1 + read nota2)/2
          situacao | media < 3 = "reprovado"
                   | media < 7 = "exame especial"
                   | otherwise = "aprovado"
      in unwords [mat,nome,nota1,nota2,show media,situacao]
    _ -> ""
```

14.7 Problemas

Tarefa 14.2: Popularidade de nomes próprios

O arquivos texto `boynames.txt` e `girlnames.txt`, que estão disponiveis no sítio da disciplina, contêm uma lista dos 1.000 nomes de garotos e garotas mais populares nos Estados Unidos para o ano de 2003 como compilados pela Administração do Segurança Social.

Estes arquivos consitem dos nomes mais populares listados por linha, onde o nome mais popular é listada em primeiro lugar, o segundo nome mais popular é listada em segundo lugar, e assim por diante, até o 1000 nome mais popular, que é listada por último. Cada linha é composta pelo primeiro nome seguido de um espaço em branco e, em seguida, do número de nascimentos registrados usando esse nome no ano. Por exemplo, o arquivo `girlnames.txt` inicia com

```
Emily 25494
Emma 22532
Madison 19986
```

Isso indica que entre as garotas Emily foi o nome mais popular em 2003, com 25.494 nomes registrados, Emma foi o segundo mais popular, com 22.532 registros, e Madison foi o terceiro mais popular, com 19.986 registros.

Escreva um programa que lê os arquivos com os dados dos garotas e das garotas e em seguida, permita que o usuário insira um nome. O programa deve pesquisar ambas as listas de nomes. Se houver uma correspondência, então ele deve emitir o *classificação* de popularidade e o número de nascimentos registrados com este nome. O programa deve também indicar se não houver correspondência.

Por exemplo, se o usuário digita o nome *Justice*, o programa deve produzir a saída

```
Justice é classificado como 456 em popularidade entre garotas com 655 registros.
Justice é classificado como 401 em popularidade entre garotos com 653 registros.
```

Se o usuário digitar o nome *Walter*, o programa deve produzir a saída

```
Walter não está classificado entre os 1000 nomes mais populares de garotas.
Walter é classificado como 356 em popularidade entre garotos com 775 registros.
```

O programa deve terminar quando o usuário digitar um nome em branco.

Dicas:

1. Defina uma função recursiva `tabela` para fazer a análise de um texto (string) contendo uma tabela de nomes, obtendo como resultado uma lista de triplas onde cada tripla é formada pela posição, pelo nome, e pela quantidade de registros.
   Por exemplo:

   ```
   tabela "ana 1234\npaula 561\nbeatriz 180"
   ⇝ [(1,"ana",1234),(2,"paula",561),(3,"beatriz",180)]
   ```

2. Defina uma função `pesquisa` que recebe uma string descrevendo a tabela de nomes usada, uma lista de triplas formadas pela posição, pelo nome (a tabela de nomes), e pela quantidade de registros de uma tabela de nomes, e um nome (string) a ser pesquisado na tabela. O resultado da função deve ser uma ação de E/S que, quando executada, pesquisa o nome na lista e exibe o resultado na saída padrão, retornando a tupla vazia.
   Por exemplo:

   ```
   pesquisa
     "garotas"
     [(1,"ana",1234),(2,"paula",561),(3,"beatriz",180)]
     "paula"
   ↪
   ana e classificado como 2 em popularidade entre garotas com 561 registros
   ```

   ```
   pesquisa
     "garotas"
     [(1,"ana",1234),(2,"paula",561),(3,"beatriz",180)]
     "Maria"
   ↪
   Maria nao esta classificado entre os 3 nomes mais populares de garotas.
   ```

3. Defina uma função `go` que recebe as duas tabelas de nomes e resulta em uma ação de E/S que, quando executada:
   - solicita ao usuário para digitar o nome a ser pesquisado,
   - lê o nome,
   - analisa o nome lido
     - se for a string vazia, retorna a tupla vazia
     - caso contrário:

        * pesquisa o nome entre os garotos e exibe o resultado da pesquisia,
        * pesquisa o nome entre as garotas e exibe o resultado da pesquisia, e
        * chama `go` recursivamente para continuar a interação com o usuário.

4. Defina a ação `main` para fazer o sequenciamento:
    - desligar a bufferização da saída padrão,
    - ler o arguivo com a tabela de nomes de garotos,
    - ler o arguivo com a tabela de nomes de garotas, e
    - interagir com o usuário usando a função `go`.

14.8 Soluções

15 Valores Aleatórios

Resumo

A geração de valores pseudo-aleatórios em aplicações em Haskell pode ser feita através de ações de E/S. Nesta aula vamos aprender a desenvolver aplicações que usam valores aleatórios.

Estes tópicos serão usados na implementação do jogo *adivinha o número*.

Sumário

15.1 Instalação do pacote `random`

A biblioteca `random`, que usaremos para geração de valores aleatórios, não faz parte da Plataforma Haskell e muito provavelmente precisa ser instalada separadamente em seu sistema. Para tanto pode-se usar a ferramenta `cabal` (um gerenciador de pacotes do Haskell), ou um gerenciador de pacotes nativo do seu sistema operacional. Neste último caso serão necessários privlégios de administrador para instalar a biblioteca.

A instalação da biblioteca `random` usando `cabal` é feita por meio dos seguintes comands que devem ser executados em um terminal:

```
$ cabal update
$ cabal install random
```

O primeiro comando acessa o repositório de pacotes (`hackage.haskell.org`) e obtém uma lista atualizada dos pacotes disponíveis. O segundo comando instala o pacote `random`, acessando o repositório para fazer o download do seu código fonte, que é em seguida compilado e instalado no sistema.

No Ubuntu basta executar o comando seguinte para instalar a biblioteca usando o gerenciador de pacotes (o que requer privilégios de administrador):

```
$ sudo apt-get install libghc-random-dev
```

15.2 Valores aleatórios

A biblioteca `random`, que define o módulo `System.Random`, lida com a tarefa comum de geração de valores pseudo-aleatórios em Haskell.

Atavés da classe `Random` é possível obter valores aleatórios de uma variedade de tipos. Esta classe fornece uma maneira de extrair valores de um gerador de números aleatórios. Por exemplo, a instância `Float` da classe `Random` permite gerar valores aleatórios do tipo `Float`.

A geração de números aleatórios pode ser feita através da manipulação explícita de um gerador de números aleatórios, ou através de um gerador global acessível através de ações de entrada e saída. Vamos considerar apenas o segundo caso.

A classe `Random` define dois métodos para geração de números aleatórios usando o gerador global:

- `randomIO`:

```
randomIO :: Random a => IO a
```

  `randomIO` é uma ação de E/S que, quando executada, extrai o próximo valor aleatório do tipo `a` do gerador global de números aleatórios (disponível no sistema de computação), e retorna este valor.

  A faixa de possíveis valores normalmente é:

  - para tipos limitados: todo o tipo.
  - para tipos fracionários: o intervalo semi-fechado $[0, 1)$.
  - para o tipo `Integer`: a faixa de `Int`.

- `randomRIO`:

```
randomRIO :: Random a => (a, a) -> IO a
```

  Esta função recebe um par de valores (inf, sup) e resulta em uma ação de E/S que, quando executada, extrai o próximo valor aleatório do tipo `a`, uniformemente distribuído no intervalo fechado $[inf, sup]$, do gerador global de números aleatórios (disponível no sistema de computação), e retorna este valor.

Exemplo: lançamento de dados

`lancadados.hs`

```
module Main (main) where

import System.Random (randomRIO)

main :: IO ()
main =
  do putStrLn "Lançamento de dois dados"
     x <- randomRIO (1,6::Int)
     y <- randomRIO (1,6)
     putStrLn ("Faces obtidas: " ++ show x ++ " e " ++ show (y::Int))
```

Exemplo de execução da aplicação

```
$ ./lancadados
Lancamento de dois dados
Faces obtidas: 3 e 5
```

Exemplo de execução da aplicação

```
$ ./lancadados
Lancamento de dois dados
Faces obtidas: 4 e 1
```

15.3 Jogo: adivinha o número

Ao executar as tarefas que se segem você estará escrevendo uma aplicação para jogar o jogo ***adivinhe o número***, como explicado a seguir.

1. O programa escolhe um número a ser adivinhado pelo jogador (usuário) selecionando um número inteiro aleatório no intervalo de 1 a 1000.

2. O programa exibe a mensagem *Adivinhe um número entre 1 e 1000*.

3. O jogador informa o seu palpite.

4. Se o palpite do jogador estiver incorreto:

    - o programa exibe a mensagem *Muito alto* ou *Muito baixo* convenientemente para ajudar o jogador a acertar o número nas próximas jogadas.
    - o jogo continua com o programa solicitando o próximo palpite e analisando a resposta do usuário.

5. Quando o jogador insere a resposta correta:

    - o programa exibe a mensagem *Parabéns, você adivinhou o número*, e
    - permite que o usuário escolha se quer jogar novamente, e joga novamente em caso afirmativo.

Exemplo de execução da aplicação

```
$ ./advinha
Adivinha o número v1.0
======================================
Digite um número entre 1 e 1000: 444
Muito grande
Tente novamente

Digite um número entre 1 e 1000: 200
Muito grande
Tente novamente

Digite um número entre 1 e 1000: 111
Muito pequeno
Tente novamente

Digite um número entre 1 e 1000: 157
Muito grande
Tente novamente

Digite um número entre 1 e 1000: 138
Muito grande
Tente novamente

Digite um número entre 1 e 1000: 123
Muito pequeno
Tente novamente

Digite um número entre 1 e 1000: 130
Muito grande
Tente novamente

Digite um número entre 1 e 1000: 125
Muito pequeno
Tente novamente

Digite um número entre 1 e 1000: 128
Muito pequeno
Tente novamente

Digite um número entre 1 e 1000: 129
Parabéns, você acertou

Deseja jogar novamente? n
```

Tarefa 15.1

Em um arquivo `adivinha.hs` defina o módulo `Main` exportando a variável `main`.

Tarefa 15.2

Defina `main` como uma ação de E/S que, quando executada:

- configura o sistema para não realizar **bufferização** da saída de dados padrão, e
- exibe uma mensagem identificando o programa e sua versão

Exemplo de execução da aplicação

```
*Main> main
Adivinha o número v1.0
======================================
```

Tarefa 15.3

Defina uma função `simOuNao` que recebe uma string e resulta em uma ação de E/S que, quando executada:

- exibe a string na saída padrão (com o objetivo de fazer uma pergunta do tipo *sim ou não* ao usuário)
- lê a resposta do usuário
- verifica se a resposta é
  - *s* ou *S*, retornando verdadeiro
  - *n* ou *N*, retornando falso
  - qualquer outra coisa, chamando `simOuNao` novamente para que o usuário responda corretamente.

  Use uma expressão `case`.

Exemplo de execução da aplicação

```
*Main> simOuNao "Quer jogar novamente?"
Quer jogar novamente? talvez
Quer jogar novamente? k
Quer jogar novamente? S
True

*Main> simOuNao "Você é inteligente?"
Você é inteligente? com certeza
Você é inteligente?
Você é inteligente? acho que sim
Você é inteligente? n
False
```

Esta função deve ser usada em `jogar` (veja a tarefa 15.5) para verificar se o usuário deseja continuar jogando ou não.

Tarefa 15.4

Defina uma função `acertar` que recebe um número a ser adivinhado e resulta em uma ação de E/S que, quando executada:

- exibe uma mensagem solicitando um número entre 1 e 1000
- lê o número informado pelo usuário
- compara o número informado com o número a ser adivinhado:
  - se forem iguais, exibe uma mensagem parabenizando o usuário por ter adivinhado o número
  - caso contrário
    - exibe uma mensagem informando que o número é muito pequeno ou muito grande, adequadamente
    - exibe uma mensagem solicitando ao usuário uma nova tentativa
    - faz uma nova tentativa através de uma chamada recursiva de `acertar`

Exemplo de execução da aplicação

```
*Main> acertar 119
Digite um número entre 1 e 1000: 600
Muito grande
Tente novamente

Digite um número entre 1 e 1000: 23
Muito pequeno
Tente novamente

Digite um número entre 1 e 1000: 119
Parabéns, você acertou
```

A função `acertar` deverá ser usada na definição de `jogar` (veja a tarefa 15.5).

Tarefa 15.5

O programa deve permitir ao usuário jogar várias vezes, o que nos leva à necessidade do uso de recursão.

Defina uma ação de E/S `jogar` que, quando executada

- gera um número inteiro aleatório entre 1 e 1000, inclusive
- interage com o usuário até que o usuário acerte o número (veja a tarefa 15.4)
- verifica se o usuário deseja jogar novamente (veja a tarefa 15.3)
  - se sim, executa `jogar` recursivamente
  - se não, não faz nada

Para gerar um número aleatório, utilize a função `randomRIO` do módulo `System.Random`. A classe `Random` é formada pelos tipos para os quais pode-se gerar valores aleatórios. Os tipos inteiros `Int` e `Integer` são instâncias desta classe.

A função `randomRIO :: Random a => (a, a) -> IO a` recebe um par de valores como argumento e resulta em uma ação de E/S que, quando executada, gera e retorna um número pseudo-aleatório no intervalo fechado definido pelo par.

Exemplo de execução da aplicação

```
*Main> jogar
Digite um número entre 1 e 1000: 509
Muito pequeno
Tente novamente

Digite um número entre 1 e 1000: 780
Muito grande
Tente novamente

Digite um número entre 1 e 1000: 640
Muito pequeno
Tente novamente

Digite um número entre 1 e 1000: 700
Muito pequeno
Tente novamente

Digite um número entre 1 e 1000: 744
Muito grande
Tente novamente

Digite um número entre 1 e 1000: 730
Muito grande
Tente novamente

Digite um número entre 1 e 1000: 720
Muito pequeno
Tente novamente

Digite um número entre 1 e 1000: 725
Muito pequeno
Tente novamente

Digite um número entre 1 e 1000: 728
Parabéns, você acertou

Deseja jogar novamente?  n
```

A ação `jogar` deve ser usada em `main` para que o usuário possa joagar o jogo.

Tarefa 15.6

Modifique o programa `adivinha.hs` de forma que o usuário possa especificar o intervalo a ser utilizado para adivinhar o número através de dois argumentos na linha de comando.

Tarefa 15.7

Modifique o programa `adivinha.hs` para que seja exibida o número de tentativas feitas pelo usuário.

15.4 Soluções

16 Expressão Lambda

Resumo

Expressões lambdas são funções anônimas que podem ser usadas como qualquer outro valor de primeira classe. Nesta aula vamos aprender sobre expressões lambda.

Sumário

16.1 Valores de primeira classe

16.1.1 Valores de primeira classe

- Tipo de **primeira classe**: não há restrições sobre como os seus valores podem ser usados.
- São valores de primeira classe:
  - números
  - caracteres
  - tuplas

- listas
- *funções*

entre outros

16.1.2 Valores de primeira classe: Literais

- Valores de vários tipos podem ser escritos *literalmente*, sem a necessidade de dar um nome a eles:

valor	tipo	descrição
`True`	`Bool`	o valor lógico *verdadeiro*
`'G'`	`Char`	o caracter *G*
`456`	`Num a => a`	o número 456
`2.45`	`Fractional a => a`	o número em ponto flutuante 2.45
`"haskell"`	`String`	a cadeia de caracteres *haskell*
`[1,6,4,5]`	`Num a => [a]`	a lista dos números 1, 6, 4, 5
`("Ana",False)`	`([Char],Bool)`	o par formado por *Ana* e *falso*

- Funções também podem ser escritas sem a necessidade de receber um nome:

valor	tipo	descrição
`\x -> 3*x`	`Num a => a -> a`	função que calcula o triplo
`\n -> mod n 2 == 0`	`Integral a => a -> Bool`	função que verifica se é par
`\(p,q) -> p+q`	`Num a => (a,a) -> a`	função que soma par

16.1.3 Valores de primeira classe: Variáveis

- Valores de vários tipos podem ser *nomeados*:

```
matricula   = 456
sexo        = 'M'
aluno       = ("Ailton Mizuki Sato",101408,'M',"com")
disciplinas = ["BCC222","BCC221","MTM153","PRO300"]
livroTexto  = ("Programming in Haskell","G. Hutton",2007)
```

- Funções também podem ser nomeadas:

```
triplo = \x -> 3*x
```

Esta equação define a variável `triplo`, associando-a a um valor que é uma função.

Haskell permite escrever esta definição de forma mais sucinta:

```
triplo x = 3 * x
```

16.1.4 Valores de primeira classe: Argumentos

- Valores de vários tipos podem ser *argumentos* de funções:

```
sqrt 2.45
not True
length [1,6,4,5]
take 5 [1,8,6,10,23,0,0,100]
```

- Funções também podem ser argumentos de outras funções:

```
map triplo [1,2,3] ⇝ [3,6,9]
```

A função `triplo` é aplicada a cada elemento da lista `[1,2,3]`, resultando na lista `[3,6,9]`

16.1.5 Valores de primeira classe: Resultado

- Valores de vários tipos podem ser *resultados* de funções:

```
not False        ⇝ True
length [1,6,4,5] ⇝ 4
snd ("Ana",'F')  ⇝ 'F'
tail [1,6,4,5]   ⇝ [6,4,5]
```

- Funções também podem ser resultados de outras funções:

```
(abs . sin) (3*pi/2) ⇝ 1.0
(sqrt . abs) (-9)    ⇝ 3.0
```

O operador binário infixo `(.)` faz a composição de duas funções.

16.1.6 Valores de primeira classe: Componentes

- Valores de vários tipos podem ser *componentes* de outros valores:

```
("Ana",'F',18)
["BCC222","BCC221","MTM153","PRO300"]
[("Ailton",101408),("Lidiane",102408)]
```

- Funções também podem ser componentes de outros valores:

```
map (\g -> g (-pi)) [abs,sin,cos]
⇝ [3.141592653589793,-1.2246467991473532e-16,-1.0]
```

O segundo argumento de `map` é a lista das funções `abs`, `sin` e `cos`.

16.2 Expressão lambda

16.2.1 Expressões lambda

- Da mesma maneira que um número inteiro, uma string ou um par podem ser escritos sem ser nomeados, uma função também pode ser escrita sem associá-la a um nome.

- **Expressão lambda** é uma função anônima (sem nome), formada por uma seqüência de padrões representando os argumentos da função, e um corpo que especifica como o resultado pode ser calculado usando os argumentos:

```
\padrão1 ... padrãon -> expressao
```

- O termo *lambda* provém do cálculo lambda (teoria de funções na qual as linguagens funcionais se baseiam), introduzido por Alonzo Church nos anos 1930 como parte de uma investigação sobre os fundamentos da Matemática.
- No cálculo lambda expressões lambdas são introduzidas usando a letra grega λ. Em Haskell usa-se o caracter `\`, que se assemelha-se um pouco com λ.

16.2.2 Exemplos de expressões lambda

Função anônima que calcula o dobro de um número:

```
\x -> x + x
```

O tipo desta expressão lambda é `Num a => a -> a`
Função anônima que mapeia um número x a $2x + 1$:

```
\x -> 2*x + 1
```

cujo tipo é `Num a => a -> a`
Função anônima que calcula o fatorial de um número:

```
\n -> product [1..n]
```

cujo tipo é `(Enum a, Num a) => a -> a`
Função anônima que recebe três argumentos e calcula a sua soma:

```
\a b c -> a + b + c
```

cujo tipo é `Num a => a -> a -> a -> a`
Definições de função usando expressão lambda:

```
f       = \x -> 2*x + 1
somaPar = \(x,y) -> x + y
fatorial = \n -> product [1..n]
```

é o mesmo que

```
f x           = 2*x + 1
somaPar (x,y) = x + y
fatorial n    = product [1..n]
```

16.2.3 Uso de expressões lambda

- Apesar de não terem um nome, funções construídas usando *expressões lambda podem ser usadas da mesma maneira que outras funções*.
- **Exemplos** de aplicações de função usando expressões lambda:

```
(\x -> 2*x + 1) 8
⇝ 17
```

```
(\a -> (a,2*a,3*a)) 5
⇝ (5,10,15)
```

```
(\x y -> sqrt (x*x + y*y)) 3 4
⇝ 5.0
```

```
(\xs -> let n = div (length xs) 2 in (take n xs,drop n xs)) "Bom dia"
⇝ ("Bom", " dia")
```

```
(\(x1,y1) (x2,y2) -> sqrt ((x2-x1)^2 + (y2-y1)^2)) (6,7) (9,11)
⇝ 5.0
```

16.2.4 Exercícios

Tarefa 16.1

Escreva uma função anônima que recebe uma tripla formada pelo nome, peso e altura de uma pessoa e resulta no seu índice de massa corporal, dado pela razão entre o peso e o quadrado da altura da pessoa.

Tarefa 16.2

Escreva uma expressão para selecionar (filtrar) os elementos múltiplos de 3 em uma lista de números. Utilize a função `filter :: (a -> Bool) -> [a] -> [a]` do prelúdio. Especifique a função que determina a propriedade a ser satisfeita pelos elementos selecionados usando uma expressão lambda.

Tarefa 16.3

Determine o tipo mais geral da seguinte expressão:

```
\a (m,n) -> if a then (m+n)^2 else (m+n)^3
```

Dica: do prelúdio tem-se
`(^) :: (Num a, Integral b) => a -> b -> a`.

Tarefa 16.4

Composição de funções é uma operação comum em Matemática, que a define como

$$(f \circ g)(x) = f(g(x))$$

Em Haskell podemos definir uma função para compor duas outras funções dadas como argumentos. O resultado é uma função: a função composta.

Definia a função `composta` que recebe duas funções como argumentos e resulta na função composta das mesmas. Use uma definição local para definir a função resultante:

```
composta f g = ···
  where
    ···
```

Tarefa 16.5

1. Escreva outra definição para a função `composta` usando uma expressão lambda para determinar o seu resultado. Nesta versão não use definições locais.
2. Determine o tipo mais geral da função `composta`.
3. Teste a função `composta` calculando o tipo e o valor da expressão

   ```
   (composta even length) "linguagens modernas"
   ```

Tarefa 16.6

O módulo **Prelude** define o operador binário `(.)` para fazer composição de funções. Este operador tem precedência 9 e associatividade à direira:

```
infixr 9 .
(.) :: (b -> c) -> (a -> b) -> a -> c
```

Determine o tipo e o valor das seguintes expressões que usam composição de funções e expressões lambda:

1. `(toUpper . head) ["maria","jose","silva"]`
2. `(not . odd . length) "felicidade"`
3. `(isLetter . head . head . reverse) ["maria","silva","pereira"]`
4. `(even . (\x -> x*2 + 3) . (\x -> div x 2) . snd) (9+4,9-4)`

16.3 Aplicação parcial de funções

16.3.1 Aplicação parcial de funções

- Uma função com *múltiplos argumentos* pode também ser considerada como uma função que retorna outra *função como resultado*.

16.3.2 Aplicação parcial de funções: exemplos

- Seja a seguinte função:

```
f    :: Int -> Int -> Int
f x y = 2*x + y
```

A função `f` recebe dois argumentos inteiros `x` e `y` e resulta na soma `2*x + y`.

- Alternativamente esta função pode ser definida em duas etapas:

```
f'  :: Int -> (Int -> Int)
f' x = h
  where h y = 2*x + y
```

A função `f'` recebe um argumento inteiro `x` e resulta na função `h`, que por sua vez recebe um argumento inteiro `y` e calcula `2*x + y`.

- Aplicando a função:

```
f' 2 3
⇝ (f' 2) 3
⇝ h 3
⇝ 2*2 + 3
⇝ 7
```

- As funções `f` e `f'` produzem o mesmo resultado final, mas `f` foi definida de uma forma mais breve.

- Podemos ainda definir a função usando uma expressão lambda:

```
f''  :: Int -> (Int -> Int)
f'' x =
  \y -> 2*x + y
```

Da mesma forma que `f'`, a função `f''` recebe um argumento inteiro `x` e resulta em uma função. Esta função recebe um argumento inteiro `y` e calcula `2*x + y`.

- Aplicando a função:

```
f'' 2 3
⇝ (f'' 2) 3
⇝ (\y -> 2*2 + y) 3
⇝ 2*2 + 3
⇝ 7
```

- Podemos ainda definir a função usando duas expressões lambda:

```
f''' :: Int -> (Int -> Int)
f'''  =
  \x -> (\y -> 2*x + y)
```

- Aplicando a função:

```
f''' 2 3
⇝ (\x -> (\y -> 2*x + y)) 2 3
⇝ (\y -> 2*2 + y) 3
⇝ 2*2 + 3
⇝ 7
```

- Todas as versões apresentadas para a função `f` (`f`, `f'`, `f''` e `f'''`) são equivalentes.

- Portanto a função `f` pode ser considerada como uma função que recebe um argumento e resulta em outra função que, por sua vez, recebe outro argumento e resulta na soma do dobro do primeiro argumento com o segundo argumento.

- Isto permite a *aplicação parcial* da função:

```haskell
let g = f 5 in (g 8, g 1)
⇝ (18,11)

map (f 2) [1,8,0,19,5]
⇝ [5,12,4,23,9]

(f 2 . length) "entendeu?"
⇝ 13

filter (not . even . f 10) [1,8,0,19,5]
⇝ [1,19,5]
```

- Outro exemplo: multiplicação de três números:

```haskell
mult      :: Int -> Int -> Int -> Int
mult x y z = x * y * z
```

A função `mult` recebe três argumentos e resulta no produto destes argumentos.

- Na verdade `mult` recebe um argumento de cada vez. Ou seja, `mult` recebe um inteiro `x` e resulta em uma função que por sua vez recebe um inteiro `y` e resulta em outra função, que finalmente recebe um inteiro `z` e resulta no produto `x * y * z`.
- Este entendimetno fica claro quando usamos expressões lambda para definir a função de maneira alternativa:

```haskell
mult' :: Int -> (Int -> (Int -> Int))
mult'  = \x -> \y -> \z -> x * y * z
```

16.4 *Currying*

16.4.1 Funções *curried*

- Outra opção para passar vários argumentos em uma aplicação de função é formar uma estrutura de dados com os dados desejados e passar a estrutura como argumento.
- Neste caso fica claro que haverá um único argumento, que é a estrutura de dados.
- Exemplo: usando uma tupla:

```haskell
somaPar :: (Int,Int) -> Int
somaPar (x,y) = x + y
```

A função `somaPar` recebe *um único* argumento que é um par, e resulta na soma dos componentes do par.

- Evidentemente este mecanismo não permite a aplicação parcial da função.
- Funções que *recebem os seus argumentos um por vez* são chamadas de **funções *curried***[1], celebrando o trabalho de **Haskell Curry** no estudo de tais funções.
- Funções com mais de um argumento *curried*, resultando em funções aninhadas.

[1] Funções *curried* às vezes são chamadas de **funções currificadas** em português.

16.4.2 Por que *currying* é útil?

- Funções *curried* são mais flexíveis do que as funções com tuplas, porque muitas vezes funções úteis podem ser obtidas pela *aplicação parcial* de uma função *curried*.
- Por exemplo:

```
take 5 :: [a] -> [a]            -- função que seleciona os 5
                                -- primeiros elementos de uma lista

drop 5 :: [a] -> [a]            -- função que descarta os 5
                                -- primeiros elementos de uma lista

div 100 :: Integral a => a -> a -- função que divide 100 pelo seu argumento

elem 'a' :: String -> String    -- função que verifica se 'a' é
                                -- elemento de uma lista
```

16.4.3 Convenções sobre *currying*

- Para evitar excesso de parênteses ao usar funções *curried*, duas regras simples foram adotadas na linguagem Haskell:
- A seta `->` (construtor de tipos função) *associa-se à direita*.
- Exemplo:

```
Int -> Int -> Int -> Int
```

  significa

```
Int -> (Int -> (Int -> Int))
```

- A *aplicação de função* tem *associatividade à esquerda*.
- Exemplo:

```
mult x y z
```

  significa

```
((mult x) y) z
```

- A menos que seja explicitamente necessário o uso de tuplas, todas as funções em Haskell são normalmente definidas na forma *curried*.

16.5 Seções de operadores

16.5.1 Operadores

- Um **operador binário infixo** é uma *função de dois argumentos* escrita em *notação infixa*, isto é, entre os seus (dois) argumentos, ao invés de precedê-los.

- Por exemplo, a função `(+)` do prelúdio, para somar dois números, é um operador infixo, portanto deve ser escrita entre os operandos:

```
3 + 4
```

- Lexicalmente, operadores consistem inteiramente de *símbolos*, em oposição aos identificadores normais que são *alfanuméricos*.

- Haskell não tem **operadores prefixos**, com exceção do menos (`-`), que pode ser tanto infixo (subtração) como prefixo (negação).

- Por exemplo:

```
3 - 4 ⇝ -1  {- operador infixo: subtração -}
- 5   ⇝ -5  {- operador prefixo: negação  -}
```

- Um identificador alfanumérico pode ser usado como operador infixo quando escrito entre sinais de crase (`'`).

- Por exemplo, a função `div` do prelúdio calcula o quociente de uma divisão inteira:

```
div 20 3 ⇝ 6
```

  Usando a notação de operador infixo:

```
20 `div` 3 ⇝ 6
```

- Um operador infixo (escrito entre seus dois argumentos) pode ser convertido em uma função *curried* normal (escrita antes de seus dois argumentos) usando *parênteses*.

- **Exemplos:**

  - `(+)` é a função que soma dois números.

```
1 + 2   ⇝ 3
(+) 1 2 ⇝ 3
```

  - `(>)` é a função que verifica se o primeiro argumento é maior que o segundo.

```
100 > 200   ⇝ False
(>) 100 200 ⇝ False
```

  - `(++)` é a função que concatena duas listas.

```
[1,2] ++ [30,40,50]   ⇝ [1,2,30,40,50]
(++) [1,2] [30,40,50] ⇝ [1,2,30,40,50]
```

16.5.2 Seções de operadores

- Como os operadores infixos são de fato funções, eles podem ser aplicados parcialmente.
- Haskell oferece uma notação especial para a *aplicação parcial de um operador infixo*, chamada de **seção** do operador. Uma seção de um operador é escrita colocando o operador e o argumento desejado entre parênteses.
- **Exemplo:**

```
(1+)
```

é a função que incrementa (soma um) ao seu argumento. É o mesmo que

```
\x -> 1 + x
```

```
(1+) 8 ⇝ 9
```

- **Exemplo:**

```
(*2)
```

é a função que dobra (multiplica por 2) o seu argumento. É o mesmo que

```
\x -> x * 2
```

```
(*2) 8 ⇝ 16
```

- **Exemplo:**

```
(100>)
```

é a função que verifica se 100 é maior que o seu argumento. É o mesmo que

```
\x -> 100 > x
```

```
(100>) 8 ⇝ True
```

- **Exemplo:**

```
(<0)
```

é a função que verifica se o seu argumento é negativo. É o mesmo que

```
\x -> x < 0
```

```
(<0) 8 ⇝ False
```

- Outros **Exemplos** de aplicação de seções de operador:

```
(1+) 2 ⇝ 3
(+1) 2 ⇝ 3
```

```
(100>) 200 ⇝ False
(>100) 200 ⇝ True
```

```
([1,2]++) [30,40,50] ⇝ [1,2,30,40,50]
(++[1,2]) [30,40,50] ⇝ [30,40,50,1,2]
```

- Em geral, se $\oplus$ é um operador binário infixo, então as formas

$$(\oplus)$$
$$(x\ \oplus)$$
$$(\oplus\ y)$$

são chamados de **seções**.

- Seções são equivalentes às definições com expressões lambdas:

```
(⊕) = \x y -> x ⊕ y
```

```
(x ⊕) = \y -> x ⊕ y
```

```
(⊕ y) = \x -> x ⊕ y
```

- **Nota**:
  - Como uma exceção, o operador binário `-` para *subtração* não pode formar uma seção direita

    ```
    (-x)
    ```

    porque isso é interpretado como negação unária na sintaxe Haskell.
  - A função `subtract` do prelúdio é fornecida para este fim. Em vez de escrever `(-x)`, você deve escrever

    ```
    (subtract x)
    ```

    ```
    (subtract 8) 10 ⇝ 2
    ```

16.6 Utilidade de expressões lambda

16.6.1 Por que seções são úteis?

- *Funções úteis* às vezes podem ser construídas de uma forma simples, utilizando seções.
- **Exemplos:**

seção	descrição
`(1+)`	função sucessor
`(1/)`	função recíproco
`(*2)`	função dobro
`(/2)`	função metade

- Seções são necessárias para anotar o tipo de um operador.
- **Exemplos:**

```
(&&) :: Bool -> Bool -> Bool
(+)  :: Num a => a -> a -> a
(:)  :: a -> [a] -> [a]
```

- Seções são necessárias para passar operadores como argumentos para outras funções.
- **Exemplo:**
  A função `and` do prelúdio, que verifica se todos os elementos de uma lista são verdadeiros, pode ser definida como:

```
and :: [Bool] -> Bool
and  = foldr (&&) True
```

  onde `foldr` é uma função do prelúdio que reduz uma lista de valores a um único valor aplicando uma operação binária aos elementos da lista.

16.6.2 Utilidade de expressões lambda

- Expressões lambda podem ser usadas para dar um sentido formal para as funções definidas usando currying e para a *aplicação parcial de funções*.
- **Exemplo**:
  A função

```
soma x y = x + y
```

  pode ser entendida como

```
soma = \x -> (\y -> x + y)
```

  isto é, `soma` é uma função que recebe um argumento `x` e resulta em uma função que por sua vez recebe um argumento `y` e resulta em `x+y`.

```
soma
⇝ \x -> (\y -> x + y)
```

```
soma 2
⇝ (\x -> (\y -> x + y)) 2
⇝ \y -> 2 + y
```

```
soma 2 3
⇝ (\x -> (\y -> x + y)) 2 3
⇝ (\y -> 2 + y) 3
⇝ 2 + 3
⇝ 5
```

- Expressões lambda também são úteis na definição de *funções que retornam funções como resultados*.
- **Exemplo:**
  A função `const` definida na biblioteca retorna como resultado uma função constante, que sempre resulta em um dado valor:

```
const    :: a -> b -> a
const x _ = x
```

```
const 6 0  ⇝ 6
const 6 1  ⇝ 6
const 6 2  ⇝ 6
const 6 9  ⇝ 6
const 6 75 ⇝ 6
```

```
h = const 6 ⇝ \_ -> 6
```

```
h 0  ⇝ 6
h 4  ⇝ 6
h 75 ⇝ 6
```

A função `const` pode ser definida de uma maneira mais natural usando expressão lambda, tornando explícito que o resultado é uma função:

```
const  :: a -> (b -> a)
const x = \_ -> x
```

- Expressões lambda podem ser usadas para evitar a nomeação de funções que são *referenciados apenas uma vez*.
- **Exemplo:**
  A função

```
impares n = map f [0..n-1]
  where
    f x = x*2 + 1
```

que recebe um número n e retorna a lista dos n primeiros números ímpares, pode ser simplificada:

```
impares n = map (\x -> x*2 + 1) [0..n-1]
```

16.6.3 Exercícios

Tarefa 16.7

Para cada uma das seguintes funções:

- descreva a função
- determine o tipo mais geral da função
- reescreva a função usando expressões lambda ao invés de seções de operadores

a) `('c':)`

b) `(:"fim")`

c) `(==2)`

d) `(++"\n")`

e) `(^3)`

f) `(3^)`

g) `(`elem` "AEIOU")`

Tarefa 16.8

Determine o valor da expressão:

```
let pares = [(1,8),(2,5),(0,1),(4,4),(3,2)]
    h = sum . map (\(x,y) -> x*y-1) . filter (\(x,_) -> even x)
in h pares
```

Tarefa 16.9

Mostre como a definição de função *curried*

```
mult x y z = x * y * z
```

pode ser entendida em termos de expressões lambda.
Dica: Redefina a função usando expressões lambda.

16.7 Soluções

17 Funções de Ordem Superior

Resumo

Uma função é conhecida como *função de ordem superior* quando ela tem uma função como argumento ou resulta em uma função. Nesta aula vamos aprender sobre funções de ordem superior.

Sumário

17.1 Funções de Ordem Superior

Uma função de ordem superior é uma função que

- tem outra *função* como *argumento*, ou
- produz uma *função* como *resultado*.

17.2 Um operador para aplicação de função

O operador `($)` definido no prelúdio se destina a substituir a aplicação de função normal, mas com uma precedência e associatividade diferente para ajudar a evitar parênteses. O operador `($)` tem precedência zero e associa-se à direita. Já a aplicação de função normal tem precedência maior que todos os operadores e associa-se à esquerda. O operador `($)` é usado principalmente para eliminar o uso de parênteses nas aplicações de funções.

Exemplos de aplicação de função com `($)`

```
sqrt 36                          ⇝ 6.0
sqrt $ 36                        ⇝ 6.0
($) sqrt 36                      ⇝ 6.0
head (tail "asdf")               ⇝ 'a'
head $ tail $ "asdf"             ⇝ 'a'
head $ tail "asdf"               ⇝ 'a'
even (succ (abs (negate 36)))  ⇝ False
even $ succ $ abs $ negate 36  ⇝ False
```

Definição de `($)`

```
infixr 0 $

($) :: (a -> b) -> a -> b

f $ x = f x
```

17.3 Composição de funções

Composição de funções é uma operação comum na Matemática. Dadas duas funções f e g, a função composta $f \circ g$ é definida por

$$(f \circ g)(x) = f(g(x))$$

Ou seja, quando a função composta $f \circ g$ é aplicada a um argumento x, primeiramente g é aplicada a x, e em seguida f é aplicada a este resultado gx.

A operação de composição de funções faz parte do prelúdio de Haskell. A função `(.)` recebe duas funções como argumento e resulta em uma terceira função que é a *composição* das duas funções dadas. A função `(.)` é um operador binário infixo de precedência 9 e associatividade à esquerda.

Observe que a operação `(.)` é uma função de ordem superior, pois recebe duas funções como argumento e resulta em outra função.

Exemplos de composição de funções

```
sqrt . abs                    ⇝ a função composta de sqrt e abs
(sqrt . abs) 9                ⇝ 3
(sqrt . abs) (16 - 25)        ⇝ 3
(sqrt . abs . sin) (3*pi/2)   ⇝ 1.0
(not . null) "abc"            ⇝ True
(sqrt . abs . snd) ('Z',-36) ⇝ 6
```

Definição de `(.)`

```
infixr 9 .

(.) :: (b -> c) -> (a -> b) -> a -> c

f . g = h
  where h x = f (g x)
```

17.4 A função `filter`

A função `filter` do prelúdio recebe uma função e uma lista como argumentos, e seleciona (*filtra*) os elementos da lista para os quais a função dada resulta em verdadeiro.

Note que `filter` é uma função de ordem superior, pois recebe outra função como argumento.

Exemplos de aplicação de `filter`

```
filter even [1,8,10,48,5,-3]           ⇝ [8,10,48]
filter odd [1,8,10,48,5,-3]            ⇝ [1,5,-3]
filter isDigit "A186 B70"              ⇝ "18670"
filter (not . null) ["abc","","ok",""] ⇝ ["abc", "ok"]
```

Importando um módulo

A função `isDigit` não faz parte do módulo **`Prelude`**, mas está definida no módulo **`Data.Char`**. Para usar `isDigit` é necessário *importar* o módulo **`Data.Char`**:

- no ambiente interativo use o comando `:module` (ou simplesmente `:m`):

```
:m + Data.Char
```

- em um script e no ambiente interativo use a declaração

```
import Data.Char
```

Definição de `filter`

```
filter                    :: (a -> Bool) -> [a] -> [a]

filter _ []                = []
filter f (x:xs) | f x      = x : filter f xs
                | otherwise = filter f xs
```

17.5 A função `map`

A função `map` do prelúdio recebe uma função e uma lista como argumentos, e aplica a função a cada um dos elementos da lista, resultando na lista dos resultados. `map` é uma função de ordem superior, pois recebe outra função como argumento.

Exemplos de aplicação de `map`

```
map sqrt [0,1,4,9]                          ⇝ [0.0,1.0,2.0,3.0]
map succ "HAL"                              ⇝ "IBM"
map head ["bom","dia","turma"]              ⇝ "bdt"
map even [8,10,-3,48,5]                     ⇝ [True,True,False,True,False]
map isDigit "A18 B7"                        ⇝ [False,True,True,False,False,True]
map length ["ciência", "da", "computação"] ⇝ [6,2,10]
map (sqrt.abs.snd) [('A',100),('Z',-36)]    ⇝ [10,6]
```

Definição de `map`

```
map         :: (a -> b) -> [a] -> [b]

map _ []     = []
map f (x:xs) = f x : map f xs
```

17.6 A função `zipWith`

`zipWith` recebe uma função binária e duas listas e retorna a lista formada pelos resultados da aplicação da função aos elementos correspondentes da listas dadas. Se as listas forem de tamanhos diferentes, o tamanho do resultado é o menor tamanho.

Observe que `zipWith` é uma função de ordem superior, pois recebe outra função como argumento.

Exemplos de aplicação de `zipWith`

```
zipWith (+) [] []                       ⇝ []
zipWith (+) [1,2,3,4,5] [3,3,4,1,5]     ⇝ [4,5,7,5,10]
zipWith (++) ["AB","cde"] ["?","123"] ⇝ ["AB?","cd123"]
zipWith (^) [5,6,7,8] [2,3,4,5]         ⇝ [25,216,2401,32768]
zipWith (*) [5,6,7,8] [2,3]             ⇝ [10,18]
```

Definição de `zipWith`

```
zipWith                 :: (a -> b -> c) -> [a] -> [b] -> [c]

zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith _ _      _      = []
```

17.7 As funções `foldl` e `foldr`, `foldl1` e `foldr1`

17.7.1 `foldl`

`foldl` reduz uma lista, usando uma função binária e um valor inicial, de forma associativa à esquerda.

```
foldl (⊕) e [x0,x1,...,xn-1]
≡
(...((e ⊕ x0) ⊕ x1) ...) ⊕ xn-1
```

Exemplos de aplicação de `foldl`

```
foldl (+) 0 []                                ⇝ 0
foldl (+) 0 [1]                               ⇝ 1
foldl (+) 0 [1,2]                             ⇝ 3
foldl (+) 0 [1,2,4]                           ⇝ 7
foldl (*) 1 [5,2,4,10]                        ⇝ 400
foldl (&&) True [2>0,even 6,odd 5,null []]    ⇝ True
foldl (||) False [2>3,even 6,odd 5,null []] ⇝ True
```

Definição

```
foldl           :: (a -> b -> a) -> a -> [b] -> a

foldl f z []     = z
foldl f z (x:xs) = foldl f (f z x) xs
```

17.7.2 foldr

`foldr` reduz uma lista, usando uma função binária e um valor inicial, de forma associativa à direita.

```
foldr (⊕) e [x0,...,xn-2,xn-1]
≡
x0 ⊕ (... (xn-2 ⊕ (xn-1 ⊕ e)) ...)
```

Exemplos de aplicação de `foldr`

```
foldr (+) 0 []                                  ⇝ 0
foldr (+) 0 [1]                                 ⇝ 1
foldr (+) 0 [1,2]                               ⇝ 3
foldr (+) 0 [1,2,4]                             ⇝ 7
foldr (*) 1 [5,2,4,10]                          ⇝ 400
foldr (&&) True [2>0,even 6,odd 5,null []]      ⇝ True
foldr (||) False [2>3,even 6,odd 5,null []]     ⇝ True
```

Definição

```
foldr           :: (a -> b -> b) -> b -> [a] -> b

foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

17.7.3 foldl1

`foldl1` reduz uma lista não vazia usando uma função binária, de forma associativa à esquerda.

`foldl1` é uma variante de `foldl` que não tem valor inicial, e portanto deve ser aplicada a listas não-vazias.

Exemplos de aplicação de `foldl1`

```
foldl1 (+) []                                   ⇝ erro
foldl1 (+) [1]                                  ⇝ 1
foldl1 (+) [1,2,4]                              ⇝ 7
foldl1 (*) [5,2,4,10]                           ⇝ 400
foldl1 (&&) [2>0,even 6,odd 5,null []]          ⇝ True
foldl1 max [1,8,6,10,-48,5]                     ⇝ 10
```

Definição

```
foldl1          :: (a -> a -> a) -> [a] -> a

foldl1 f (x:xs) = foldl f x xs
```

17.7.4 foldr1

`foldr1` reduz uma lista não vazia usando uma função binária, de forma associativa à esquerda.

`foldr1` é uma variante de `foldr` que não tem valor inicial, e portanto deve ser aplicada a listas não-vazias.

Exemplos de aplicação de `foldr1`

```
foldr1 (+) []                          ⇝ erro
foldr1 (+) [1]                         ⇝ 1
foldr1 (+) [1,2,4]                     ⇝ 7
foldr1 (*) [5,2,4,10]                  ⇝ 400
foldr1 (&&) [2>0,even 6,odd 5,null []] ⇝ True
foldr1 max [1,8,6,10,-48,5]            ⇝ 10
```

Definição

```
foldr1          :: (a -> a -> a) -> [a] -> a

foldr1 _ [x]    = x
foldr1 f (x:xs) = f x (foldr1 f xs)
```

17.8 *List comprehension*

17.8.1 *List comprehension*

Em Matemática a **notação de compreensão** pode ser usada para construir novos conjuntos a partir de conjuntos já conhecidos. Por exemplo,

$$\{x^2 | x \in [1...5]\}$$

é o conjunto $\{1, 4, 9, 16, 25\}$ de todos os números x^2 tal que x é um elemento do conjunto $\{1, 2, 3, 4, 5\}$.

Em Haskell também há uma notação de compreensão similar que pode ser usada para construir novas listas a partir de listas conhecidas. Por exemplo

```
[ x^2 | x <- [1..5] ]
```

é a lista `[1,4,9,16,25]` de tdos os números `x^2` tal que `x` é um elmento da lista `[1,2,3,4,5]`.

A frase `x <- [1..5]` é chamada **gerador**, já que ela informa como gerar valores para a variável `x`. Compreensões podem ter múltiplos geradores, separados por vírgula. Por exemplo:

```
[(x,y) | x <- [1,2,3], y <- [4,5]] ⇝ [(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
```

Se a ordem dos geradores for trocada, a ordem dos elementos na lista resultante também é trocada. Por exemplo:

```
[(x,y) | y <- [4,5], x <- [1,2,3]] ⇝ [(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)]
```

Geradores múltiplos são semelhantes a *loops* aninhados: os últimos geradores são como *loops* mais profundamente aninhados cujas variáveis mudam mais freqüentemente. No exemplo anterior, como `x <- [1,2,3]` é o último gerador, o valor do componente `x` de cada par muda mais frequentemente.

Geradores posteriores podem depender de variáveis introduzidas em geradores anteriores. Por exemplo:

```
[(x,y) | x <- [1..3], y <- [x..3]] ⇝ [(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
```

é a lista `[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]` de todos os pares de números `(x,y)` tal que `x` e `y` são elementos da lista `[1..3]` e `y >= x`.

Como exemplo, usando geradores dependentes pode-se definir a função que concatena uma lista de listas:

```
concat :: [[a]] -> [a]
concat xss = [x | xs <- xss, x <- xs]
```

```
concat [[1,2,3],[4,5],[6]] ⇝ [1,2,3,4,5,6]
```

List comprehensions podem usar **guardas** para restringir os valores produzidos por geradores anteriores. Por exemplo:

```
[x | x <- [1..10], even x] ⇝ [2,4,6,8,10]
```

é a lista de todos os números `x` tal que `x` é um elemento da lista `[1..10]` e `x` é par.

Como exemplo, usando uma guarda podemos definir uma função para calcular a lista de divisores de um número inteiro positivo:

```
divisores  :: Int -> [Int]
divisores n = [x | x <- [1..n], mod n x == 0]
```

Exemplos de aplicação da função:

```
divisores 15  ⇝ [1,3,5,15]
divisores 120 ⇝ [1,2,3,4,5,6,8,10,12,15,20,24,30,40,60,120]
```

Um número inteiro positivo é *primo* se seus únicos divisores são 1 e ele próprio. Assim, usando `divisores`, podemos definir uma função que decide se um número é primo:

```
primo  :: Int -> Bool
primo n = divisores n == [1,n]
```

Exemplos de aplicação da função:

```
primo 15 ⇝ False
primo 7  ⇝ True
```

Usando um guarda agora podemos definir uma função que retorna a *lista de todos os números primos* até um determinado limite:

```
primos  :: Int -> [Int]
primos n = [x | x <- [2..n], primo x]
```

Exemplos de aplicação da função:

```
primos 40 ⇝ [2,3,5,7,11,13,17,19,23,29,31,37]
primos 12 ⇝ [2,3,5,7,11]
```

17.8.2 *List comprehension* e funções de ordem superior

List comprehension nada mais é que uma abreviação sintática que é traduzida em aplicações das funções `map` e `filter`. Os exemplos a seguir ilustram como esta tradução é feita.

```
[ x^2 | x <- [1..5] ] ⇝ [1,4,9,16,25]
map (^2) [1..5]       ⇝ [1,4,9,16,25]

[(x,y) | x <- [1,2,3], y <- [4,5]]      ⇝ [(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
map (\x->map (\y->(x,y)) [4,5]) [1,2,3] ⇝ [(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]

primos  n = [x | x <- [2..n], primo x]
primos' n = filter primo [2..n]
```

17.9 Cupom fiscal do supermercado

Nas tarefas que se seguem temos por objetivo desenvolver uma aplicação em Haskell para **automatizar o caixa de um supermercado** usando técnicas de manipulação de listas empregando funções de ordem superior.

Um leitor de código de barras é usado no caixa de um supermercado para produzir uma lista de códigos de barras a partir dos produtos que se encontram em um carrinho de compras contendo os produtos comprados. Usando os códigos de barra cria-se uma nota descritiva da compra. Considere por exemplo a seguinte lista de códigos de barra:

```
[1234,4719,3814,1112,1113,1234]
```

Esta lista deve ser convertida para uma conta como mostra a figura a seguir:

```
Haskell Stores

Dry Sherry, 1lt.....5.40
Fish Fingers.....1.21
Orange Jelly.....0.56
Hula Hoops (Giant).....1.36
Unknown Item.....0.00
Dry Sherry, 1lt.....5.40

Total.....13.90
```

Primeiro devemos decidir como modelar os objetos envolvidos. Códigos de barra e preços (**em centavos**) podem ser representados por números inteiros, e nomes de mercadorias podem ser representados por strings. Então usaremos os seguintes tipos:

```
type Nome   = String
type Preco  = Int
type Codigo = Int
```

A conversão dos códigos de barras será baseada em um banco de dados que relaciona códigos de barras, nomes de mercadorias, e preços. Usaremos uma lista para representar o banco de dados de mercadorias:

```haskell
type Mercadorias = [ (Codigo, Nome, Preco) ]
```

O banco de dados para o exemplo dado é:

```haskell
tabelaMercadorias :: Mercadorias
tabelaMercadorias = [ (4719, "Fish Fingers",       121 )
                    , (5643, "Nappies",            1010)
                    , (3814, "Orange Jelly",       56  )
                    , (1111, "Hula Hoops",         21  )
                    , (1112, "Hula Hoops (Giant)", 133 )
                    , (1234, "Dry Sherry, 1lt",    540 )
                    ]
```

O objetivo do programa é primeiramente converter uma lista de códigos de barra em uma lista de pares (`Nome`,`Preco`) por meio de uma consulta à tabela de mercadorias. Em seguida esta lista de pares deve ser convertida em uma string para exibição na tela. Usaremos as seguintes definições de tipo:

```haskell
type Carrinho = [Codigo]
type Conta    = [(Nome,Preco)]
```

para representar um carrinho de compras e uma conta (cupom fiscal) corresponde a uma compra.

Tarefa 17.1: Formatação do preço em reais

Defina uma função `formataCentavos :: Preco -> String` que recebe o preço em centavos e resulta em uma string representando o preço em reais.

Por exemplo:

```haskell
formataCentavos 1023  ⇝ "10.23"
formataCentavos 56015 ⇝ "560.15"
formataCentavos 780   ⇝ "7.80"
formataCentavos 309   ⇝ "3.09"
formataCentavos 15    ⇝ "0.15"
formataCentavos 5     ⇝ "0.05"
```

Use as funções `div`, `mod` e `show`. Observe que ao dividir o preço em centavos por 100, o quociente corresponde à parte inteira do preço em reais, e o resto corresponde à parte fracionária do preço em reais. Preste atenção no caso do resto menor do que 10: deve-se inserir um 0 à esquerda explicitamente.

Tarefa 17.2: Formatação de uma linha do cupom fiscal

Defina uma função `formataLinha :: (Nome,Preco) -> String` que recebe um par formado pelo nome e preço de uma mercadoria e resulta em uma string representando uma linha da conta do supermercado.

Por exemplo:

```
formataLinha ("Dry Sherry, 1lt",540)⇝ "Dry Sherry, 1lt.....5.40\n"
formataLinha ("Nappies, 1lt",1010)  ⇝ "Nappies.....10.10\n"
```

O tamanho de uma linha em uma conta deve ser 30. Use a variável abaixo para representar este valor.

```
tamanhoLinha :: Int
tamanhoLinha = 30
```

Use as funções `(++)`, `show`, `length` e `replicate` do prelúdio, e a função `formataCentavos` da tarefa 17.1.

A função `replicate :: Int -> a -> [a]` recebe um número inteiro n e um valor x e resulta em uma lista de comprimento n onde todos os elementos são x. Por exemplo:

```
replicate 5 13  ⇝ [13,13,13,13,13]
replicate 8 '.' ⇝ "........"
```

Tarefa 17.3: Formatação de várias linhas do cupom fiscal

Defina a função `formataLinhas :: [(Nome,Preco)] -> String` que recebe uma lista de pares formados pelos nomes das mercadorias e seus respectivos preços em uma compra, e resulta na string correspondente ao corpo da conta do supermercado.

Por exemplo:

```
formataLinhas [ ("Dry Sherry, 1lt",     540)
              , ("Fish Fingers",        121)
              , ("Orange Jelly",        056)
              , ("Hula Hoops (Giant)", 136)
              , ("Unknown Item",        000)
              , ("Dry Sherry, 1lt",     540)
              ]
⇝
"Dry Sherry, 1lt.....5.40\n\
\Fish Fingers.....1.21\n\
\Orange Jelly.....0.56\n\
\Hula Hoops (Giant).....1.36\n\
\Unknown Item.....0.00\n\
\Dry Sherry, 1lt.....5.40\n"
```

Use a função `formataLinha` da tarefa 17.3 para obter as linhas correspondentes a cada produto, e concatene estas linhas usando a função `(++)` do prelúdio. Não use recursividade explícita, mas use as funções `map` e `foldr` ou `foldl` do prelúdio. Alternativamente você poderá usar *list comprehension*.

Tarefa 17.4: Formatação do total

Defina a função `formataTotal :: Preco -> String` que recebe o valor total da compra, e resulta em uma string representado a parte final da conta do supermercado.

Por exemplo:

```
formataTotal 1390 ⇝ "\nTotal.....................13.90"
```

Use as dicas da tarefa 17.2.

Tarefa 17.5: Formatação do cupom fiscal

Defina a função `formataConta :: Conta -> String` que recebe a lista dos itens comprados e resulta na string representando a conta do supermercado, já formatada.

Por exemplo:

```
formataConta [ ("Dry Sherry, 1lt",     540)
             , ("Fish Fingers",        121)
             , ("Orange Jelly",        056)
             , ("Hula Hoops (Giant)", 136)
             , ("Unknown Item",        000)
             , ("Dry Sherry, 1lt",     540)
             ]
```

resulta na string que é exibida pela função `putStr` como

```
Haskell Stores

Dry Sherry, 1lt...........5.40
Fish Fingers..............1.21
Orange Jelly..............0.56
Hula Hoops (Giant)........1.36
Unknown Item..............0.00
Dry Sherry, 1lt...........5.40

Total.....................13.90
```

Use as funções definadas nas tarefas 17.3 e 17.4.

Tarefa 17.6: Cálculo do valor total da compra

Defina a função `calculaTotal :: Conta -> Preco` que recebe uma conta (lista de pares formados pelo nome e preço das mercadorias de uma compra), e resulta no preço total da compra.

Por exemplo:

```
calculaTotal [("a",540),("b",121),("c",12)]  ⇝ 673
calculaTotal [("vinho",3540),("carne",7201)] ⇝ 10741
calculaTotal []                              ⇝ 0
```

Não use recursividade explícita, mas use as funções `map` e `sum` do prelúdio.

Tarefa 17.7: Pesquisa do código de um produto

Defina uma função `procuraCodigo :: Mercadorias -> Codigo -> (Nome,Preco)` que recebe o banco de dados com os nomes e preços das mercadorias disponíveis no supermercado e o código de barras da mercadoria comprada, e resulta no par formado pelo nome e pelo preço da mercadoria, de acordo com o banco de dados. Se o código de barras não constar no banco de dados, o resultado deve ser o par `("Unknown Item",0)`.

Por exemplo:

```
procuraCodigo tabelaMercadorias 5643  ⇝ ("Nappies", 1010)
procuraCodigo tabelaMercadorias 9999  ⇝ ("Unknown Item", 0)
```

Use recursão explícita.

Tarefa 17.8: Criação da conta da compra

Defina a função `criaConta :: Mercadorias -> Carrinho -> Conta` que recebe o banco de dados com os nomes e preços das mercadorias disponíveis no supermercado, e a lista de códigos de barra correspondente a uma compra, e resulta na lista dos pares `(Nome,Preco)` para as mercadorias compradas.

Por exemplo:

```
criaConta tabelaMercadorias [3814, 5643]
⇝ [("Orange Jelly", 56), ("Nappies", 1010)]
```

Use uma aplicação parcial da função `procuraCodigo` definida na tarefa 17.7 e a função `map` do prelúdio. Não use recursão explícita.

Tarefa 17.9: Criação do cupom fiscal

Defina a função `fazCompra :: Mercadorias -> Carrinho -> String` que recebe o banco de dados com os nomes e preços das mercadorias disponíveis no supermercado, e a lista de códigos de barra correspondente a uma compra, e resulta na string correspondente à nota da compra.

Use a função `criaConta` (definida na tarefa 17.8) para criar a conta a partir dos argumentos, e a função `formataConta` (definida na tarefa 17.5) para converter a conta para string. Use composição de funções.

Tarefa 17.10: Ação `main`

Defina a variável `main :: IO ()` como uma ação de entrada e saída que interage com o usuário. Quando `main` for executada, o usuário deve digitar os códigos de barras das mercadorias compradas e em seguida a conta do supermercado deve ser exibida na tela.

Para fazer a entrada de um valor (digitado pelo usuário) você pode usar a função `readLn` do prelúdio.

Tarefa 17.11

Complete a aplicação com a definição do módulo `Main` contendo as definições feitas anteriormente, e exportando a variável `main`. Compile a aplicação gerando um programa executável. Teste a aplicação.

Se necessário, importe `stdout`, `hSetBuffering`, `BufferMode` e `NoBuffering` do módulo `System.IO` e cancele a *bufferização* da saída padrão.

17.10 Soluções

18 Tipos Algébricos

Resumo

Um tipo algébrico é um tipo onde são especificados a forma de cada um dos seus elementos. *Algébrico* se refere à propriedade de que um tipo algébrico é criado por operações *algébricas*. A álgebra aqui é *somas* e *produtods*:

- *soma* é a alternância: $A|B$ significa A ou B, mas não ambos, e
- *produto* é a combinação: AB significa A e B juntos.

Somas e produtos podem ser combinados repetidamente em estruturas arbitrariamente largas.

Nesta aula vamos aprender como definir e usar tipos algébricos (ou seja, estruturas de dados), em Haskell.

Sumário

18.1 Novos tipos de dados

- **Tipos básicos**:
  - `Bool`
  - `Char`
  - `Int`
  - `Integer`
  - `Float`
  - `Double`
- **Tipos Compostos**:
  - tuplas: $(t_1, t_2, \ldots, t_n)$
  - listas: $[t]$

  - funções: t_1 `->` t_2

- **Novos tipos**: como definir?
  - dias da semana
  - estações do ano
  - figuras geométricas
  - árvores
  - tipos cujos elementos são inteiros ou strings
  - ...

18.2 Tipos algébricos

Uma **declaração de tipo algébrico** é da forma:

$$\texttt{data}\ cx\ \texttt{=>}\ T\ u_1\ \cdots\ u_k\ \texttt{=}\ C_1\ t_{11}\ \cdots\ t_{1n_1}$$
$$\vdots$$
$$\mid\ C_m\ t_{m2}\ \cdots\ t_{mn_m}$$

onde:

- cx é um contexto
- $u_1\ \cdots\ u_k$ são variáveis de tipo
- T é o **construtor de tipo**
- $T\ u_1\ \cdots\ u_k$ é um novo tipo introduzido pela declaração `data`
- $C_1, \ldots, C_m$ são **construtores de dados**
- t_{ij} são tipos
- Construtores de tipo e construtores de dados são *identificadores alfanuméricos* começando com letra maiúscula, ou *identificadores simbólicos*.
- Um **construtor de dados** é utilizado para
  - construir valores do tipo definido, funcionando como uma função (eventualmente, constante) que recebe argumentos (do tipo indicado para o construtor), e constrói um valor do novo tipo de dados;
  - decompor um valor do tipo em seus componentes, através de casamento de padrão
- Construtores de dados são funções especiais, pois não tem nenhuma definição (algoritmo) associada.

18.3 Exemplo: formas geométricas

- Definição de um novo tipo para representar formas geométricas:

```
data Figura = Circulo Double
            | Retangulo Double Double
```

- O **construtor de tipo** é `Figura`.

- Os **construtores de dados** deste tipo são:

```
Circulo   :: Double -> Figura
Retangulo :: Double -> Double -> Figura
```

e com eles é possível construir todo e qualquer valor do tipo `Figura`:

```
a :: Figura
a = Circulo 2.3  -- um círculo de raio 2.3
```

```
b :: Figura
b = Retangulo 2.8 3.1  -- um retângulo de base 2.8 e altura 3.1
```

```
lfig :: [Figura]
lfig = [Retangulo 5 3, Circulo 5.7, Retangulo 2 2]
```

- Expressões como `Circulo 2.3` ou `Retangulo 2.8 3.1` *não podem ser reduzidas*, pois já estão em sua forma mais simples.

- Os construtores são utilizados em casamento de *padrões* para acessar os componentes de um valor do tipo algébrico.

- Podemos definir *funções* envolvendo os tipos algébricos.

```
eRedondo :: Figura -> Bool
eRedondo (Circulo _)     = True
eRedondo (Retangulo _ _) = False
```

```
eRedondo (Circulo 3.2)     ⇝ True
eRedondo (Retangulo 2 5.1) ⇝ False
```

```
area :: Figura -> Double
area (Circulo r)     = pi * r^2
area (Retangulo b a) = b * a
```

```
area (Circulo 2.5)     ⇝ 19.634954084936208
area (Retangulo 2 5.1) ⇝ 10.2
```

```
quadrado :: Double -> Figura
quadrado lado = Retangulo lado lado
```

```
area (quadrado 2.5) ⇝ 6.25
```

18.4 Exemplo: sentido de movimento

- Definição de um novo tipo para representar direções de movimento:

```
data Sentido = Esquerda | Direita | Acima | Abaixo
```

- O **construtor de tipo** é `Sentido`.
- Os **construtores de dados** deste tipo, todos constantes, são:

```
Esquerda :: Sentido
Direita  :: Sentido
Acima    :: Sentido
Abaixo   :: Sentido
```

- Quando os construtores de dados são **constantes**, (ou seja, não tem argumentos), dizemos que o tipo é uma **enumeração**.
- Neste exemplo os únicos valores do tipo `Sentido` são `Direita`, `Esquerda`, `Acima` e `Abaixo`.
- Podemos definir funções envolvendo o tipo algébrico:

```
type Pos = (Double,Double)
```

```
move :: Sentido -> Pos -> Pos
move Esquerda (x,y) = (x-1,y  )
move Direita  (x,y) = (x+1,y  )
move Acima    (x,y) = (x  ,y+1)
move Abaixo   (x,y) = (x  ,y-1)
```

```
moves :: [Sentido] -> Pos -> Pos
moves []     p = p
moves (s:ss) p = moves ss (move s p)
```

```
moves [Direita,Acima,Acima,Abaixo,Acima,Direita,Acima] (0,0)
⇝ (2.0,3.0)
```

Definição alternativa usando funções de ordem superior:

```
moves :: [Sentido] -> Pos -> Pos
moves sentidos pontoInicial = foldl (flip move) pontoInicial sentidos
```

```
flipSentido :: Sentido -> Sentido
flipSentido Direita = Esquerda
flipSentido Esquerda = Direita
flipSentido Acima = Abaixo
flipSentido Abaixo = Acima
```

```
flipSentido Direita ⇝ erro:
No instance for (Show Sentido) arising from a use of ‘print’
```

Oops!

- A princípio Haskell não sabe como exibir valores dos novos tipos.
- O compilador pode definir automaticamente funções necessárias para exibição:

```
data Sentido = Esquerda | Direita | Acima | Abaixo
             deriving (Show)
```

- A cláusula **deriving** permite declarar as classes das quais o novo tipo será instância, automaticamente.
- Logo, segundo a declaração dada, o tipo `Sentido` é uma instância da classe `Show`, e a função `show` é sobrecarregada para o tipo `Sentido`.

```
show Direita        ⇝ "Direita"
flipSentido Direita ⇝ Esquerda
```

18.5 Exemplo: cor

- Definição de um novo tipo para representar cores:

```
data Cor = Azul | Amarelo | Verde | Vermelho
```

- O **construtor de tipo** é `Cor`.
- Os **construtores de dados** deste tipo são:

```
Azul     :: Cor
Amarelo  :: Cor
Verde    :: Cor
Vermelho :: Cor
```

- Podemos agora definir funções envolvendo cores:

```
fria :: Cor -> Bool
fria Azul  = True
fria Verde = True
fria _     = False
```

```
fria Amarelo ⇝ False
```

```
quente :: Cor -> Bool
quente Amarelo  = True
quente Vermelho = True
quente _        = False
```

```
quente Amarelo ⇝ True
```

18.6 Exemplo: coordenadas cartesianas

- Definição de um novo tipo para representar coordenadas cartesianas:

```
data Coord = Coord Double Double
```

- O **construtor de tipo** é `Coord`.
- O **construtor de dados** deste tipo é:

```
Coord :: Double -> Double -> Coord
```

- Podemos agora definir funções envolvendo coordenadas:

```
somaVet :: Coord -> Coord -> Coord
somaVet (Coord x1 y1) (Coord x2 y2) = Coord (x1+x2) (y1+y2)
```

18.7 Exemplo: horário

- Definição de um novo tipo para representar horários:

```
data Horario = AM Int Int Int | PM Int Int Int
```

- Os **construtores do tipo** `Horario` são:

```
AM :: Int -> Int -> Int -> Horario
PM :: Int -> Int -> Int -> Horario
```

  e podem ser vistos como uma *etiqueta* (*tag*) que indica de que forma os argumentos a que são aplicados devem ser entendidos.

- Os valores `AM 5 10 30`, `PM 5 10 30` e `(5,10,30)` não contém a mesma informação. Os construtores `AM` e `PM` tem um papel essencial na interpretação que fazemos destes termos.
- Podemos agora definir funções envolvendo horários:

```
totalSegundos :: Horario -> Int
totalSegundos (AM h m s) = (h*60 + m)*60 + s
totalSegundos (PM h m s) = ((h+12)*60 + m)*60 + s
```

18.8 Exemplo: booleanos

- O tipo `Bool` da biblioteca padrão é um tipo algébrico:

```
data Bool = True | False
```

- O **construtor de tipo** é `Bool`.

- Os **construtores de dados** deste tipo são:

```
True  :: Bool
False :: Bool
```

- Exemplos de uso do tipo:

```
infixr 3 &&
(&&) :: Bool -> Bool -> Bool
True && True = True
_    && _    = False
```

```
infixr 3 ||
(||) :: Bool -> Bool -> Bool
False || False = False
_     || _     = True
```

```
not :: Bool -> Bool
not True  = False
not False = True
```

18.9 Exemplo: listas

- Um tipo algébrico pode ser **polimórfico**.
- O tipo `Lista a` é um tipo algébrico polimórfico:

```
data Lista a = Nil | Cons a (Lista a)
```

- Os **construtores de dados** são:
  - `Nil :: Lista a`
    um construtor constante representando a lista vazia
  - `Cons :: a -> Lista a -> Lista a`
    um construtor para listas não vazias, formadas por uma cabeça e uma cauda.
- **Exemplo**: a lista do tipo `Lista Int` formada pelos elementos 3, 7 e 1 é representada por `Cons 3 (Cons 7 (Cons 1 Nil))`.
- O **construtor de tipo** `Lista` está parametrizado com uma variável de tipo `a`, que poderá ser substituída por um tipo qualquer. É neste sentido que se diz que `Lista` é um construtor de tipo.
- Operações com lista:

```
comprimento :: Lista a -> Int
comprimento Nil = 0
comprimento (Cons _ xs) = 1 + comprimento xs
```

```
elemento :: Eq a => a -> Lista a -> Bool
elemento _ Nil         = False
elemento x (Cons y xs) = x == y || elemento x xs
```

- O tipo `Lista a` deste exemplo é similar ao tipo `[a]` da bilioteca padrão do Haskell:

```haskell
data [a] = [] | a : [a]
```

- Observe apenas que Haskell usa:
  - uma notação especial para o construtor de tipo: `[a]`
  - uma notação especial para o construtor de lista vazia: `[]`
  - um identificador simbólico com status de operador infixo para o construtor de lista não vazia: `(:)`

18.10 Exercícios básicos

Tarefa 18.1: Perímetro de uma figura

Defina uma função para calcular o perímetro de uma forma geométrica do tipo `Figura`. Qual é o tipo desta função?

Tarefa 18.2: Item do supermercado

Considere a seguinte definição de tipo para produtos em um supermercado:

```haskell
-- nome, quantidade e preço unitáro de um item
type ShopItem = (String, Int, Double)
```

1. Redefina este tipo como um novo tipo, ao invés de um tipo sinônimo.
2. Defina uma função que recebe uma lista de ítens como argumento e resulta no valor total a ser pago pelos ítens na lista. Escreva a assinatura de tipo da função.

Tarefa 18.3: Adicionando triângulos às figuras

Adicione um novo construtor de dados ao tipo `Figura` para triângulos, e extenda as funções `eRedondo`, `area` e `perimetro` para incluir triângulos.
Dicas:

- Um triângulo pode ser representando pelas medidas dos seus lados.
- A área de um triângulo pode ser calculada pela fórmula de Heron:

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

sendo p o seu semi-perímetro:

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

e a, b e c as medidas dos lados.

Tarefa 18.4: Figuras regulares

Defina uma função para verificar se uma figura é regular. São figuras regulares: o círculo, o quadrado, o triângulo equilátero.

Tarefa 18.5: Endereçamento

Algumas casas tem um número; outras tem um nome.

1. Como você implementaria o tipo de *strings ou números* usados como parte de um endereço para identificar uma casa?
2. Escreva uma função que receba uma identifcação de casa (de acordo com o item anterior) e dê a sua representação textual (isto é, a função deve converter para uma string).
3. Dê a definição de um tipo para endereçamento contendo o nome e o endereço do destinatário. Use o tipo que você definiu.

18.11 Números naturais

Tarefa 18.6: Um tipo para os números naturais

Defina um tipo algébrico `Nat` para representar números naturais. Um número natural pode ser:

- zero, ou
- positivo, sendo neste caso o sucessor de outro número natural

O seu tipo deve ter dois construtores de dados: `Zero`, um construtor constante, para representar o valor zero, e `Succ`, um construtor de aridade um, para representar um número positivo.

Observe que o tipo `Nat` deve ser recursivo, já que ele deverá ser usado em sua própria definição. Use derivação automática da classe `Show`.

Tarefa 18.7: Alguns numeros naturais

Defina as variávies `um`, `dois` e `tres` do tipo `Nat` cujos valores são os números naturais 1, 2 e 3, respectivamente.

Tarefa 18.8: Convertendo de natural para inteiro

Defina a função `nat2integer :: Nat -> Integer` que converte um número natural em um número inteiro. Faça uma definição recursiva onde o caso base corresponde 0, e o caso recursivo corresponde aos números positivos.

Tarefa 18.9: Convertendo de inteiro para natural

Defina a função `integer2nat :: Integer -> Nat` que converte um número inteiro em um número natural.

Tarefa 18.10: Adição de números naturais

Defina a função `natAdd :: Nat -> Nat -> Nat` que recebe dois números naturais e resulta na soma dos números. A função deve ser recursiva no segundo argumento.

Tarefa 18.11: Subtração de números naturais

Defina a função `natSub :: Nat -> Nat -> Nat` que recebe dois números naturais e resulta na diferença dos números. A função deve ser recursiva no segundo argumento.

Tarefa 18.12: Multiplicação de números naturais

Defina a função `natMul :: Nat -> Nat -> Nat` que recebe dois números naturais e resulta no produto dos números. A função deve ser recursiva no segundo argumento.

Tarefa 18.13: Divisão de números naturais

Defina as funções `natDiv :: Nat -> Nat -> Nat` e `natMod :: Nat -> Nat -> Nat`, que recebem dois números naturais e resultam no quociente e no resto dos números, respectivamente.

Tarefa 18.14: Comparação de números naturais

Defina a função `natLt :: Nat -> Nat -> Bool` que recebe dois números naturais e verifica se o primeiro é menor que o segundo.

18.12 Árvores binárias

Tarefa 18.15: Um tipo para árvores binárias

Defina um construtor de tipo algébrico `BinTree` para representar árvores binárias de busca. Uma árvore binária de busca pode ser

- vazia
- não vazia (nó), formada por um valor qualquer (uma informação armazenada no nó da árvore) e duas sub-árvores.

O tipo `BinTree a` será o tipo das árvores binárias de busca que armazenam valores do tipo `a` em seus nós. Observe que este tipo será **polimórfico** e **recursivo**. Observe ainda que o construtor de tipo `BinTree` tem aridade um, ou seja, ele espera um argumento de tipo (correspondente ao tipo dos valores armazenados nos nós da árvore).

Use derivação automática da classe `Show`.

Tarefa 18.16: Tamanho de uma árvore

Defina uma função `btLength :: BinTree a -> Int` que recebe uma árvore binária de busca e resulta no número de elementos armazenados na árvore (tamanho da árvore).

Tarefa 18.17: Profundidade de uma árvore

Defina uma função `btDepth :: BinTree a -> Int` que recebe uma árvore binária de busca e resulta na profundidade da árvore.

Tarefa 18.18: Verificar se um valor é elemento de uma árvore

Defina uma função `btElem :: a -> BinTree a -> Bool` que recebe um valor e uma árvore, e verifica se o valor é um elemento da árvore.

18.13 O construtor de tipo `Maybe`

O prelúdio define o tipo `Maybe a` que pode ser usado para indicar um valor opcional. A definição de `Maybe` é

```
data Maybe a = Nothing | Just a
```

O construtor de tipo é `Maybe`, de aridade um, que espera um argumento de tipo representando o tipo do dado encapsulado pelo construtor de dados `Just`.

Os **construtores de dados** deste tipo são:

```
Nothing :: Maybe a
Just    :: a -> Maybe a
```

Os valores do tipo `Maybe a` podem ser de duas formas possíveis:

- `Nothing`, uma constante que indica que o valor opcional não foi informado, e
- `Just x`, onde o valor opcional `x` foi informado.

O tipo `Maybe a` também pode ser usado para indicar sucesso ou falha de alguma operação:

- `Nothing` indica falha, e
- `Just x` indica sucesso, resultando no valor `x`.

Exemplo: divisão segura

```
safediv :: Double -> Double -> Maybe Double
safediv _ 0 = Nothing
safediv x y = Just (x / y)

test :: IO ()
test =
  do putStrLn "digite dois números"
     a <- readLn
     b <- readLn
     case safediv a b of
       Nothing -> do putStrLn "divisão por zero"
                     putStrLn "tente novamente"
                     test
       Just z  -> putStrLn ("resposta: " ++ show z)
```

Tarefa 18.19: Conversão para string com segurança

A função `readMaybe :: Read a => String -> Maybe a`, definida no módulo `Text.Read`, converte uma string em um valor do tipo `a` (que deve ser instância da classe `Read`). A conversão sucede se e somente se há exatamente um resultado válido.

Faça um programa que leia uma temperatura na escala Celsius e calcula e exibe a temperatura correspondente na escala Fahrenheit. O programa deve verificar se a entrada de dados sucede ou falha. Uma nova entrada deve ser feita enquanto a leitura for inválida.

18.14 Expressão booleana

Tarefa 18.20: Avaliando expressões booleanas

1. Defina um módulo `ExpBool` onde será definido um tipo e algumas funções para expressões booleanas, como solicitado a seguir.
2. Defina um tipo algébrico para representar uma expressão booleana. Uma expressão boolena pode ser
   - uma constante booleana (verdadeiro ou falso)
   - uma variável

- a negação de uma expressão booleana
- a conjunção de duas expressões booleanas
- a disjunção de duas expressões booleanas

Defina o tipo das expressões booleanas usando o nome **ExpBool** para o construtor de tipo, com os seguintes construtores de dados:

```
Cte :: Bool -> ExpBool                  -- constantes
Var :: String -> ExpBool                -- variáveis
Neg :: ExpBool -> ExpBool               -- negação
Con :: ExpBool -> ExpBool -> ExpBool    -- conjunção (e)
Dis :: ExpBool -> ExpBool -> ExpBool    -- disjunção (ou)
```

3. Defina um tipo para representar uma *memória*, isto é, um mapeamento de identificadores a valores booleanos.

   Dica: Use listas de associações. Uma lista de associação é uma lista de pares.

4. Defina uma função que recebe uma memória e uma expressão boolena e calcula o valor da expressão booleana usando a memória. Considere que o valor de uma variável indefinida é falso.

   Dica: Utilize a função `lookup` do prelúdio para encontrar o valor associado a uma chave em uma lista de associações.

5. Defina uma função que recebe uma expressão booleana e resulta na lista das variáveis que ocorrem na expressão. Cada variável deve ocorrer uma única vez na lista.

   Dica: Use a função `sort :: Ord a => [a] -> [a]` do prelúdio. Esta função recebe uma lista e resulta em uma lista com os mesmos elementos da lista recebida, porém ordenados de forma crescente, sem repetições.

6. Defina um tipo **TabelaVerdade** para representar a tabela verdade de uma expressão booleana. Você pode usar uma lista de pares onde cada elemento da lista representa uma linha da tabela verdade. Cada par é formada por uma lista das variáveis que ocorrem na expressão e seus valores, e pelo valor da expressão booleana correspondente a estes valores das variáveis.

7. Defina uma função para converter uma tabela verdade para string.

8. Defina uma função que recebe uma expressão booleana e resulta na tabela verdade para a expressão booleana.

   Dica: Para gerar as permitações dos valores booleanos a fim de montar a tabela verdade, utilize a função `replicateM` do módulo **Control.Monad**. Veja alguns exemplos do uso desta função:

```
replicateM 2 "abc" ⇝
["aa","ab","ac","ba","bb","bc","ca","cb","cc"]

replicateM 3 [True,False] ⇝
[ [True,  True,  True  ]
, [True,  True,  False ]
, [True,  False, True  ]
, [True,  False, False ]
, [False, True,  True  ]
, [False, True,  False ]
, [False, False, True  ]
, [False, False, False ]
]
```

9. Em um módulo chamado `Main` defina uma ação de E/S `main :: IO ()` que, quando executada, solicita ao usuáro para digitar uma expressão boolena, e em seguida exibe a tabela verdade desta expressão booleana.

   Você pode utilizar o módulo a seguir para converter uma string em uma expressão booleana.

```haskell
module ParseExpBool where

import Text.Parsec
import ExpBool

type Parser = Parsec String () ExpBool

parseExpBool :: String -> Maybe ExpBool
parseExpBool str =
  case parse pExp "" str of
    Left _ -> Nothing
    Right e -> Just e

lexeme :: Parsec String () a -> Parsec String () a
lexeme p = do spaces
              x <- p
              spaces
              return x

pExp :: Parser
pExp = pDis

pAtm :: Parser
pAtm = try (lexeme (string "1") >> return (Cte True))
       <|>
       try (lexeme (string "0") >> return (Cte False))
       <|>
       try (fmap Var (lexeme (many1 letter)))
       <|>
       try (between (lexeme (string "(")) (lexeme(string ")")) pExp)
       <|>
       (lexeme (string "~") >> (fmap Neg pAtm))

pCon :: Parser
pCon = fmap (foldl1 Con) (many1 pAtm)

pDis :: Parser
pDis = fmap (foldl1 Dis) (sepBy1 pCon (lexeme (string "+")))
```

18.15 Soluções

19 Classes de Tipos

Resumo

Nesta aula vamos aprender a definir classes de tipos e instâncias de classes de tipos.

Sumário

19.1 Polimorfismo *ad hoc* (sobrecarga)

Além do polimorfismo paramétrico, Haskell tem uma outra forma de polimorfismo que é a **sobrecarga de nomes**. Um mesmo identificador de variável (o que inclui função) pode ser usado para designar valores computacionalmente distintos. Esta característica também é chamada **polimorfismo *ad hoc***.

Por exemplo:

- O operador `(+)` tem sido usado para somar tanto valores inteiros como valores fracionários.
- O operador `(==)` pode ser usado para comparar inteiros, caractres, listas de inteiros, strings, booleanos, ...

Afinal, qual é o tipo de `(+)`? E de `(==)`? A sugestão

```
(+)  :: a -> a -> a
(==) :: a -> a -> Bool
```

não serve, pois são tipos demasiado *genéricos* e fariam com que fossem aceitas expressões como

```
'a' + 'b'
True + False
"está" + "errado"
div == mod
```

e estas expressões resultariam em erro, pois estas operações não estão definidas para trabalhar com valores destes tipos.

Em Haskell esta situação é resolvida através de **tipos qualificados** (*qualified types*), fazendo uso da noção de **classe de tipos**.

19.2 Tipos qualificados

Conceitualmente um **tipo qualificado** pode ser visto como um tipo polimórfico, só que, em vez da quantificação universal da forma

> para todo tipo a, ...

vai-se poder dizer

> para todo tipo a que pertence à classe C, ...

Uma **classe** pode ser vista como um conjunto de tipos. Por exemplo: Sendo `Num` uma classe (a classe dos tipos numéricos) que tem como elementos os tipos:

`Int`, `Integer`, `Float`, `Double`, `Rational`, ...,

pode-se dar a `(+)` o tipo

$$\forall a \in \mathtt{Num}.a \to a \to a$$

o que em Haskell é escrito como:

```
(+) :: Num a => a -> a -> a
```

e lê-se

para todo o tipo `a` que pertence à classe `Num`, `(+)` tem tipo `a -> a -> a`.

Desta forma uma classe surge como uma forma de classificar tipos quanto às funcionalidades a ele associadas. Neste sentido as classes podem ser vistas como os tipos dos tipos.

Os tipos que pertencem a uma classe são chamados de **instâncias** da classe.

A capacidade de qualificar tipos polimórficos é uma característica inovadora de Haskell.

19.3 Classes e Instâncias

Uma **classe** estabelece um conjunto de assinaturas de variáveis (o que inclui funções): os **métodos** da classe. Deve-se definir os métodos de uma classe para cada um dos tipos que são instâncias desta classe.

Como exemplo, considere a seguinte **declaração de classe** simplificada:

```
class Num a where
  (+) :: a -> a -> a
  (*) :: a -> a -> a
```

Todo tipo `a` da classe `Num` deve ter as operações `(+)` e `(*)` definidas. Para declarar `Int` e `Float` como elementos da classe `Num`, tem que se fazer as seguintes **declarações de instância**:

```
instance Num Int where
  (+) = primPlusInt
  (*) = primMulInt
```

```haskell
instance Num Float where
  (+) = primPlusFloat
  (*) = primMulFloat
```

Neste caso as funções `primPlusInt`, `primMulInt`, `primPlusFloat` e `primMulFloat` são funções primitivas da linguagem. Se `x::Int` e `y::Int`, então `x + y` ≡ `primPlusInt x y`. Se `x::Float` e `y::Float`, então `x + y` ≡ `primPlusFloat x y`.

19.4 Tipo principal

O **tipo principal** de uma expressão ou de uma função é o tipo mais geral que lhe é possível associar, de forma que todas as possíveis instâncias desse tipo constituam ainda tipos válidos para a expressão ou função. Qualquer expressão ou função válida tem um tipo principal único.

Haskell **infere** sempre o tipo principal das expressões e funções, mas é sempre possível associar tipos mais específicos (que são instâncias do tipo principal). Por exemplo, o tipo principal inferido por haskell para o operador `(+)` é

```haskell
(+) :: Num a => a -> a -> a
```

mas,

```haskell
(+) :: Int -> Int -> Int
(+) :: Float -> Float -> Float
```

também são tipos válidos, dado que tanto `Int` como `Float` são instâncias da classe `Num`, e portanto podem substituir a variável de tipo `a`.

Note que `Num a` não é um tipo, mas antes uma restrição sobre um tipo. Diz-se que `Num a` é o **contexto** para o tip apresentado.

Como outro exemplo, considere:

```haskell
sum []     = 0
sum (x:xs) = x + sum xs
```

O tipo principal da função `sum` é

```haskell
sum :: Num a => [a] -> a
```

`sum :: [a] -> a` seria um tipo demasiado geral. Porquê? Qual será o tipo principal da função `product`?

19.5 Definição padrão

Considere a função pré-definida `elem`:

```haskell
elem _ []     = False
elem x (y:ys) = (x == y) || elem x ys
```

- Qual é o seu tipo?
- É necessário que `(==)` esteja definido para o tipo dos elementos da lista.

A classe pre-definida `Eq` é formada pelos tipos para os quais existem operações de comparação de igualdade e desigualdade:

```haskell
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  -- Minimal complete difinition: (==) or (/=)
  x == y = not (x /= y)
  x /= y = not (x == y)
```

Esta classe introduz as funções (==) e (/=), e também fornece definições padrão para estes métodos, chamados **métodos *default***.

Caso a definição de uma função seja omitida numa declaração de instância, o sistema assume a definição padrão feita na classe. Se existir uma nova definição do método na declaração de instância, esta definição será usada.

19.6 Exemplos de instâncias

- O tipo **Cor** é uma instância da classe **Eq** com (==) definido como segue:

```haskell
data Cor = Azul | Verde | Amarelo | Vermelho

instance Eq Cor where
  Azul     == Azul     = True
  Verde    == Verde    = True
  Amarelo  == Amarelo  = True
  Vermelho == Vermelho = True
  _        == _        = False
```

  O método (/=) utiliza a definição padrão dada na classe **Eq**.

- O tipo **PontoCor** abaixo também pode ser declarado como instância da classe **Eq**:

```haskell
data PontoCor = Pt Double Double Cor

instance Eq PontoCor where
  (Pt x1 y1 c1) == (Pt x2 y2 c2) = (x1 == x2) &&
                                   (y1 == y2) &&
                                   (c1 == c2)
```

- O tipo **Nat** também pode ser declarado como instância da classe **Eq**:

```haskell
data Nat = Zero | Succ Nat

instance Eq Nat where
  Zero     == Zero     = True
  (Succ m) == (Succ n) = m == n
  _        == _        = False
```

19.7 Instâncias com restrições

- Considere a seguinte definição de tipo para árvores binárias:

```haskell
data BinTree a = V
               | N a (BinTree a) (BinTree a)
```

- Como podemos fazer o teste de igualdade para árvores binárias?
- Duas árvores são iguais se tiverem a mesma estrutura (a mesma forma) e se os valores que estão nos nós também forem iguais.
- Portanto, para fazer o teste de igualdade para o tipo `BinTree a`, necessariamente tem que se saber como testar a igualdade entre os valores que estão nos nós.
- Só poderemos declarar `BinTree a` como instância da classe `Eq` se `a` também for uma instância de `Eq`.
- Este tipo de **restrição** pode ser colocado na declaração de instância.

```haskell
instance (Eq a) => Eq (BinTree a) where
  V            == V            = True
  (N x1 l1 r1) == (N x2 l2 r2) = x1 == x2 && l1 == l2 && r1 == r2
  _            == _            = False
```

19.8 Derivação de instâncias

- Os testes de igualdade definidos nos exemplos anteriores implementam a **igualdade estrutural**: dois valores são iguais quando resultam da aplicação do mesmo construtor de dados a argumentos também iguais.
- Nestes casos o compilador pode gerar sozinho a definição da função a partir da definição do tipo.
- Para tanto basta acrescentar a instrução **`deriving Eq`** no final da declaração do tipo:

```haskell
data BinTree a = V
               | N a (BinTree a) (BinTree a)
              deriving (Eq)
```

- Instâncias de algumas outras classes também podem ser derivadas automaticamente.

19.8.1 Herança

- O sistema de classes de Haskell também suporta a noção de **herança**, onde uma classe pode herdar todos os métodos de uma outra classe, e ao mesmo tempo ter seus próprios métodos.
- **Exemplo**: a classe `Ord`:

```haskell
class (Eq a) => Ord a where
  (<), (<=), (>), (>=) :: a -> a -> Bool
  min, max             :: a -> a -> a
```

- `Eq` é uma **superclasse** de `Ord`.
- `Ord` é uma **subclasse** de `Eq`.
- `Ord` **herda** todos os métodos de `Eq`.
- Todo tipo que é instância de `Ord` tem que ser necessariamente instância de `Eq`.
- Haskell suporta **herança múltipla**: uma classe pode ter mais do que uma superclasse.

19.9 Alguma classes do prelúdio

19.9.1 A classe Show

- Define métodos para conversão de um valor para string.
- `Show` pode ser derivada.
- Definição completa mínima: `showsPrec` ou `show`.

```
type ShowS = String -> String
```

```
class  Show a  where
    show      :: a   -> String
    showsPrec :: Int -> a -> ShowS
    showList  :: [a] -> ShowS
```

```
shows :: (Show a) => a -> ShowS
shows =  showsPrec 0
```

- Exemplo:

```
data Horario = AM Int Int Int
             | PM Int Int Int

instance Show Horario where
  show (AM h m s) = show h ++ ":" ++ show m ++ ":" ++ show s
                    ++ " am"
  show (PM h m s) = show h ++ ":" ++ show m ++ ":" ++ show s
                    ++ " pm"
```

19.9.2 A classe Eq

- Define igualdade (==) e desigualdade (/=).
- Todos os tipos básicos exportados por `Prelude` são instâncias de `Eq`.
- `Eq` pode ser derivada para qualquer tipo cujos constituintes são instâncias de `Eq`.
- Definição completa mínima: == ou /=.

```
class Eq a where
    (==)   :: a -> a -> Bool
    (/=)   :: a -> a -> Bool
```

19.9.3 A classe Ord

- Tipos com ordenação total.
- `Ord` pode ser derivada para qualquer tipo cujos constituintes são instâncias de `Ord`. A ordenação dos valores é determinada pela ordem dos construtores na declaração do tipo.

- Definição completa mínima: `compare` ou `<=`.
- `compare` pode ser mais eficiente para tipos complexos.

```
data Ordering = LT | EQ | GT
```

```
class  (Eq a) => Ord a  where
    compare              :: a -> a -> Ordering
    (<), (<=), (>), (>=) :: a -> a -> Bool
    max, min             :: a -> a -> a
```

19.9.4 A classe Enum

- Define operações em tipos sequencialmante ordenados (enumerações).
- **Enum** pode ser derivada para qualquer tipo enumerado (os construtores de dados são todos constantes). Os construtores são numerados da esquerda para a direita começando com 0.
- Definição completa mínima: `toEnum` e `fromEnum`.

```
class  Enum a   where
    succ                 :: a -> a
    pred                 :: a -> a
    toEnum               :: Int -> a
    fromEnum             :: a -> Int
    enumFrom             :: a -> [a]
    enumFromThen         :: a -> a -> [a]
    enumFromTo           :: a -> a -> [a]
    enumFromThenTo       :: a -> a -> a -> [a]
```

- As operações da classe **Enum** permitem construir sequências aritméticas.

```
take 5 (enumFrom 'c')       ⇒ "cdefg"
take 5 (enumFromThen 7 10) ⇒ [7,10,13,16,19]
enumFromTo 'A' 'Z'          ⇒ "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
enumFromThenTo 5 10 38      ⇒ [5,10,15,20,25,30,35]
```

- As sequências aritméticas são abreviações sintáticas para estas operações:

```
take 5 ['c'..]     ⇒ "cdefg"
take 5 [7, 10 ..] ⇒ [7,10,13,16,19]
['A' .. 'Z']       ⇒ "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
[5,10 .. 38]       ⇒ [5,10,15,20,25,30,35]
```

19.9.5 A classe Num

- Define operações numéricas básicas.
- **Num** não pode ser derivada.
- Definição completa mínima: todos, exceto `negate` ou `(-)`.

```
class  (Eq a, Show a) => Num a  where
    (+), (-), (*)       :: a -> a -> a
    negate              :: a -> a
    abs                 :: a -> a
    signum              :: a -> a
    fromInteger         :: Integer -> a
```

- Um literal inteiro representa a aplicação da função `fromInteger` ao valor apropriado do tipo `Integer`. Portanto o tipo destes literais é `(Num a) => a`.
- Exemplo: `35` é na verdade `fromInteger 35`

```
:t 35
35 :: Num a => a
```

```
35 :: Double   ⇒ 35.0
35 :: Rational ⇒ 35 % 1
```

19.10 Exercícios

Tarefa 19.1

Em JavaScript e em algumas outras linguagens com tipagem dinâmica o tipo do teste em uma expressão condicional pode ser de quase qualquer tipo. Por exemplo, em JavaScript, você pode escrever expressões como

```
if (0) alert("YEAH!") else alert("NO!")
```

```
if ("") alert("YEAH!") else alert("NO!")
```

```
if (false) alert("YEAH!") else alert("NO!")
```

Todos estes exemplos emitem um alerta `"NO!"`.

Já os códigos seguintes emitem um alerta `"YEAH!"`:

```
if ("WHAT") alert("YEAH!") else alert("NO!")
```

```
if (2+3) alert("YEAH!") else alert("NO!")
```

```
if (true) alert("YEAH!") else alert("NO!")
```

Embora em Haskell a limitação do tipo do teste em uma expressão condicional ao tipo `Bool` funcione melhor, vamos implementar um comportamento semelhante ao encontrado no JavaScript, *just for fun*!

1. Vamos começar com uma declaração de classe de tipo. Defina uma classe `YesNo` contendo um método chamado `yesno` que permita converter um valor de qualquer tipo que seja instância desta classe para o tipo `Bool`.

   Em ouras palavras, a classe `YesNo` deve introduzir uma função `yesno` que recebe um valor de um tipo que seja instância da classe, resultando em um valor do tipo `Bool`. O argumento

de `yesno` pode ser interpretado como tendo algum conceito de veracidade, e a função `yesno` nos diz claramente se ele é verdadeiro ou falso.

Lembre-se que na definição da classe enumeramos os seus métodos com as respectivas assinaturas de tipo.

2. O próximo passo é definir algumas instâncias. Com certeza o tipo **`Bool`** pode ser uma instância da classe **`YesNo`**. Faça a definição de instância da classe **`YesNo`** para o tipo **`Bool`**.

3. A lista vazia é considerada como falso, enquanto que listas não vazias são consideradas como verdadeiro. Defina uma instância da classe **`YesNo`** para o tipo das listas `[a]`.

4. Um valor do tipo **`Maybe`** a pode ser interpretado como um indicativo de sucesso ou falha de uma computação, tendo um valor agregado do tipo `a` em caso de sucesso. Com certeza o tipo **`Maybe`** a também pode ser uma instância da classe **`YesNo`**. Defina esta instância.

5. Para números vamos assumir que (como no JavaScript) qualquer número que não seja zero é verdadeiro e zero é falso. Defina uma instância da classe **`YesNo`** para o **`Int`**.

6. Defina um tipo algébrico **`Semaforo`** para representar os possíveis estados de um semáforo de trânsito: verde, amarelo e vermelho. Use os respectivos construtores de dados **`Verde`**, **`Amarelo`** e **`Vermelho`**.

   Um sinal do semáforo também pode ser um valor *yesno*. Se ele for vermelho, o condutor deve parar. Se for verde, o condutor pode seguir. E se for amarelo? Eh! Eu geralmente acelero e passo no amarelo, porque gosto de viver com adrenalina!

   Defina uma instância do tipo **`Semaforo`** para a classe **`YesNo`**.

7. Agora que temos algumas instâncias da classe **`YesNo`**, podemos brincar. Determine o valor das seguintes expressões:

   (a) `yesno $ length []`
   (b) `yesno "bom dia"`
   (c) `yesno ""`
   (d) `yesno (Just 12.4)`
   (e) `yesno True`
   (f) `yesno []`
   (g) `yesno [("ana",10), ("pedro",12), ("beatriz", 9)]`
   (h) `yesno Vermelho`

   Assim você pode confirmar que `yesno` é uma função sobrecarregada, com uma versão específica para cada um dos tipos que são instâncias da classe **`YesNo`**.

8. Agora vamos fazer uma função chamada `yesnoIf` que imita a expressão **`if`** do Haskell, mas que aceita expressões de qualquer tipo que seja instância da classe **`YesNo`** como sendo o teste. Defina a função `yesnoIf`, indicando também o seu tipo.

   A função `yesnoIf` recebe um valor **`YesNo`** e dois outros valores de um determinado tipo. Se o primeiro valor corresponder ao conceito de *yes*, o resultado deve ser o primeiro dos outros dois valores; caso contrário a função resulta no segundo dos outros dois valores.

9. Finalmente teste a sua função `yesnoIf` com as seguintes expressões:

   (a) `yesnoIf [] "YEAH!" "NO!"`
   (b) `yesnoIf [2,3,4] "YEAH!" "NO!"`
   (c) `yesnoIf True "YEAH!" "NO!"`
   (d) `yesnoIf (Just ("carla",34,174)) "YEAH!" "NO!"`
   (e) `yesnoIf Nothing "YEAH!" "NO!"`
   (f) `yesnoIf Verde "YEAH!" "NO!"`

Tarefa 19.2

Complete as seguintes declarações de instância:

1. `instance (Ord a, Ord b) => Ord (a,b) where ...`
2. `instance (Ord a) => Ord [a] where ...`

onde pares e listas devem ser ordenadas lexicographicamente, como palavras em um dicionário.

Tarefa 19.3

Considere a seguinte declaração de tipo para representar números naturais:

```
data Nat = Zero | Succ Nat
```

1. Defina uma instância da classe `Eq` para o tipo `Nat`.
2. Defina uma instância da classe `Ord` para o tipo `Nat`.
3. Defina uma instância da classe `Num` para o tipo `Nat`.
4. Defina uma instância da classe `Enum` para o tipo `Nat`.
5. Defina uma instância da classe `Show` para o tipo `Nat`. A string resultante da aplicação da função `show` deve ser da form $\#i$, onde i é o número escrito em notação decimal. Por exemplo:

```
show Zero                    ⇝ #0
show (Succ Zero)             ⇝ #1
show (Succ (Succ Zero))      ⇝ #2
show (Succ (Succ (Succ Zero))) ⇝ #3
```

6. Defina uma instância da classe `Read` para o tipo `Nat`. O argumento da função `read` deverá ser uma string da form $\#i$, onde i é o número escrito em notação decimal. Por exemplo:

```
read "#0" ⇝ #0  -- Zero
read "#1" ⇝ #1  -- Succ Zero
read "#2" ⇝ #2  -- Succ (Succ Zero)
read "#3" ⇝ #3  -- Succ (Succ (Succ Zero))
```

Tarefa 19.4

Considere o tipo

```
data BinTree a = V | N a (BinTree a) (BinTree a)
```

para representar árvores binárias de busca.

1. Defina uma função que verifica se uma árvore binária é vazia ou não.
2. Defina uma função que recebe um valor e uma árvore binária e insere o valor na árvore binária mantendo-a a ordenada, resultando na nova árvore assim obtida.
3. Defina uma função que recebe um valor e uma árvore binária e verifica se o valor é um elemento da árvore.
4. Modifique a definição do tipo para que sejam criadas automaticamente istâncias desse tipo para as classes **Read** e **Show**.
5. Declare uma instância de **BinTree** a para a classe **Eq**.
6. Declare uma instância de **BinTree** a para a classe **Ord**.
7. Declare uma instância de **BinTree** a para a classe **Functor**. A classe functor tem apenas um método chamado fmap que permite mapear uma função aos elementos de uma estrutura de dados, resultando em uma estrutura de dados similar contendo os resultados obtidos pela aplicação da função.

   ```
   class Functor f where
     fmap :: (a -> b) -> f a -> f b
   ```

19.11 Soluções

20 Mônadas

Resumo

Mônada é uma abstração para computações que podem ser realizadas de forma sequencial, como por exemplo ações de entrada e saída. Neste capítulo vamos estudar as operações que caracterizam uma mônada. Vamos estudar também algumas mônadas específicas.

Sumário

20.1 Mônadas

- **Mônadas** em Haskell podem ser entendidas como descrições de *computações* que podem ser *combinadas sequencialmente*.
- Uma mônada pode ser **executada** a fim de *realizar* a computação por ela representada e produzir um valor como *resultado*.
- Cada mônada pode representar uma forma diferente de computação.

20.1.1 Operações monádicas básicas

- `return`
  - `return x` é uma computação que, quando executada, apenas *produz o resultado* `x`.
- `(>>=)`
  - O operador binário `(>>=)` é usado para fazer a *combinação sequencial* de duas computações.
  - `(>>=)` combina duas computações de forma que, quando a computação combinada é executada, a primeira computação é executada e *o seu resultado é passado para a segunda computação*, que então é executada usando (ou dependendo de) o resultado da primeira.

20.1.2 Outras operações monádicas

- `(>>)`
  - O operador binário `(>>)` também é usado para fazer a *combinação sequencial* de duas computações, porém o resultado da primeira computação é ignorado.
  - `(>>)` combina duas computações de forma que, quando a computação combinada é executada, a primeira computação é executada (e seu resultado é ignorado) e em seguida a segunda computação é executada (sem depender do resultado da primeira).
- `fail`
  - `fail msg` é uma computação que, quando executada, indica algum tipo de falha descrita pela mensagem `msg`.

20.1.3 A classe Monad

Pode-se dizer que **mônada** é qualquer *construtor de tipo* de aridade 1 que suporta as *operações monádicas básicas* mencionadas anteriormente.

Em Haskell, a classe de tipos `Monad` introduz as operações monádicas:

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
  (>>)   :: m a -> m b -> m b
  fail   :: String -> m a

  p >> q = p >>= \_ -> q

  fail msg = error msg
```

Nesta classe a variável restrita `m` representa um *construtor de tipo (de aridade um)*, e não um tipo! Dizemos que este construtor de tipo é uma **mônada**.

Um construtor de tipo de aridade 1 é uma **mônada** se existirem as operações `return` e `(>>=)` que permitem combinar valores desse tipo em sequência.

20.1.4 Leis das mônadas

Além de implementar os métodos da classe `Monad`, todas as mônadas devem obedecer as seguintes leis, dadas pelas equações:

```
return a >>= k  =  k a
```

```
m >>= return  =  m
```

```
m >>= (\x -> k x >>= h)  =  (m >>= k) >>= h
```

20.2 Entrada e saída

- Uma ação de entrada e saída é representada pelo tipo `IO a`.
- `IO a` é um tipo abstrato em que pelo menos as seguintes operações estão disponíveis:

– Operações primitivas, como por exemplo:

```
putChar :: Char -> IO ()
getChar :: IO Char
```

– Operações para combinar ações de entrada e saída:

```
return :: a -> IO a
```

* `return x` é uma ação de E/S que quando executada não interage com o mundo e retorna `x`.

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

* `p >>= f` é uma ação de E/S que quando executada, executa primeira a ação `p` e em seguida aplica a função `f` no resultado de `p`.
* O corpo da função `f` é a segunda ação da sequência.
* O resultado da primeira ação pode ser usado na segunda ação porque ele é passado como argumento para a função `f`.

- As funções `return` e `(>>=)` caracterizam a classe de tipos chamada **Monad**.
- As demais funções primitivas são específicas de ações de entrada e saída.
- Assim `IO` é uma mônada e `x :: IO a` é uma ação monádica.

Exemplo de entrada e saída

Ler um caracter e exibi-lo em maiúscula:

```
module Main (main) where
main :: IO ()
main = getChar >>= (\c -> putChar (toUpper c))
```

A expressão lambda não precisa estar entre parênteses. Lembre-se de que uma expressão lambda se extende o máximo possível para a direita.

```
module Main (main) where
main :: IO ()
main = getChar >>= \c -> putChar (toUpper c)
```

20.3 Expressão do

20.3.1 Notação do

- Tipicamente computações monádicas complexas são construídas a partir de *longos encadeamentos* de computações mais simples combinadas usando os operadores `(>>)` e `(>>=)`.
- Haskell oferece a **expressão** `do`, que *permite combinar várias computações a serem executadas em sequência* usando uma *notação mais conveniente*.
- Uma expressão `do` é uma *extensão sintática* do Haskell e sempre pode ser reescrita como uma expressão mais básica usando os operadores de sequenciamento `(>>)` e `(>>=)` e a expressão `let`.

Exemplo de notação do

Um programa para ler dois números e exibir a sua soma:

```
module Main (main) where

main :: IO ()

main = do { putStrLn "Digite um número:";
            s1 <- getLine;
            putStrLn "Digite outro número:";
            s2 <- getLine;
            putStr "Soma: ";
            putStrLn (show (read s1 + read s2))
          }
```

20.3.2 Regra de layout com a notação do

- A expressão **do** pode usar ***layout*** em sua estrutura sintática, de maneira semelhate à expressão **let** e às cláusulas **where**.
- As *regras de* layout, por meio do uso de *indentação* adequada, permitem omitir as chaves { e } usadas para delimitar o corpo da expressão do e os pontos-e-vírgula ; usados para separar as ações que compõem o seu corpo.
- Neste caso todas as ações devem começar na *mesma coluna*, e se continuarem nas linhas seguintes, não podem usar colunas menores que esta coluna.

Exemplo de notação do usando *layout*

Um programa para ler dois números e exibir a sua soma:

```
module Main (main) where

main :: IO ()

main = do putStrLn "Digite um número:"
          s1 <- getLine
          putStrLn "Digite outro número:"
          s2 <- getLine
          putStr "Soma: "
          putStrLn (show (read s1 + read s2))
```

20.3.3 Tradução da expressão do

Código escrito usando a notação do é *transformado automaticamente pelo compilador* em expressões ordinárias que usam as funções `(>>=)` e `(>>)` da classe **Monad**, e a expressão **let**.

Quando houver uma única ação no corpo:

```
do { ação }
≡
ação
```

Exemplo:

```
do putStrLn "Bom dia, galera!"
≡
putStrLn "Bom dia, galera!"
```

Observe que neste caso não é permitido usar as formas

```
padrão <- ação
```

e

```
let declaracões
```

pois não há outras ações que poderiam usar variáveis instanciadas pelos casamentos de padrão.

Quando houver duas ou mais ações sem casamento de padrão na primeira ação:

```
do { ação ; resto }
≡
ação >> do { resto }
```

Exemplo:

```
do putStrLn "um" ; putStrLn "dois"
≡
putStrLn "um" >> do putStrLn "dois"
≡
putStrLn "um" >> putStrLn "dois"
```

Quando houver duas ou mais ações com casamento de padrão na primeira ação:

```
do { padrão <- ação ; resto }
≡
ação >>= ( \padrão -> do { resto } )
```

Exemplo:

```
do x <- getLine ; putStrLn ("Você digitou: " ++ x)
≡
getLine >>= ( \x -> do putStrLn ("Você digitou: " ++ x) )
≡
getLine >>= ( \x -> putStrLn ("Você digitou: " ++ x) )
≡
getLine >>= \x -> putStrLn ("Você digitou: " ++ x)
```

Quando houver duas ou mais ações com declaração local na primeira ação:

```
do { let declarações ; resto }
≡
let declarações in do { resto }
```

Exemplo:

```
do let f xs = xs ++ xs ; putStrLn (f "abc")
≡
let f xs = xs ++ xs in do putStrLn (f "abc")
≡
let f xs = xs ++ xs in putStrLn (f "abc")
```

Exemplo sem usar a notação do

Um programa para ler dois números e exibir a sua soma, com uso explícito dos operadores de sequenciamento:

```
module Main (main) where

main :: IO ()

main = putStrLn "Digite um número:" >>
       ( getLine >>=
         ( \s1 ->
             putStrLn "Digite outro número:" >>
             ( getLine >>=
               ( \s2 ->
                   putStr "Soma: " >>
                   putStrLn (show (read s1 + read s2))
               )
             )
         )
       )
```

Considerando que os operadores (>>) e (>>=) são associativos à direita, e que uma expressão lambda extende-se o máximo que for possível, os parênteses usados na versão anterior não são necessários.

```
module Main (main) where

main :: IO ()

main = putStrLn "Digite um número:" >>
       getLine >>= \s1 ->
       putStrLn "Digite outro número:" >>
       getLine >>= \s2 ->
       putStr "Soma: " >>
       putStrLn (show (read s1 + read s2))
```

Tarefa 20.1

Faça um programa em Haskell que receba dois números e mostre o menor.

1. Utilize a notação do para sequenciamento das ações de E/S.
2. Utilize explicitamente os operadores (>>=) e (>>) para sequenciamento das ações de E/S.

Tarefa 20.2

Faça um programa em Haskell que receba quatro notas de um aluno, calcule e mostre a média das notas e a mensagem de aprovado ou reprovado, considerando para aprovação média 7.

1. Utilize a notação do para sequenciamento das ações de E/S.
2. Utilize explicitamente os operadores (>>=) e (>>) para sequenciamento das ações de E/S.

20.4 Computações que podem falhar

- O construtor de tipo **Maybe** é uma mônada que representa *computações que podem falhar*.

- Declaração:

```haskell
data Maybe a = Nothing | Just a
```

```haskell
instance Monad Maybe where
  return = Just

  Nothing >>= f = Nothing
  Just x  >>= f = f x
```

Exemplo: agenda telefônica

Uma agenda telefônica pode ser representada por uma *lista de associações*:

```haskell
agenda :: [(String,String)]

agenda = [ ("Bob",   "01788 665242"),
           ("Fred",  "01624 556442"),
           ("Alice", "01889 985333"),
           ("Jane",  "01732 187565") ]
```

A função `lookup` do prelúdio pesquisa uma chave em uma lista de associações:

```haskell
lookup :: Eq a => a -> [(a,b)] -> Maybe b

lookup x []                     = Nothing
lookup x ((y,v):ys) | x == y    = Just v
                    | otherwise = lookup x ys
```

Queremos procurar dois itens na agenda:

- se algum dos itens não for encontrado, a operação falha
- se ambos os itens forem encontrados, resulta no par formado pelos valores correspondentes

```haskell
lookup2 :: Eq a => a -> a -> [(a,b)] -> Maybe (b,b)
```

Inspecionando diretamente a estrutura de dados:

```haskell
lookup2 k1 k2 lst = case lookup k1 lst of
                      Nothing -> Nothing
                      Just v1 -> case lookup k2 lst of
                                   Nothing -> Nothing
                                   Just v2 -> Just (v1,v2)
```

Usando operações monádicas sem a notação do:

```haskell
lookup2 k1 k2 lst = lookup k1 lst >>= \v1 ->
                    lookup k2 lst >>= \v2 ->
                    return (v1,v2)
```

Usando operações monádicas com a notação do:

```haskell
lookup2 k1 k2 lst = do v1 <- lookup k1 lst
                       v2 <- lookup k2 lst
                       return (v1,v2)
```

20.5 Expressões aritméticas

20.5.1 Expressões aritméticas

- Considere o tipo **Exp** que representa uma expressão aritmética:

```haskell
data Exp = Cte Integer
         | Som Exp Exp
         | Sub Exp Exp
         | Mul Exp Exp
         | Div Exp Exp
         deriving (Read, Show)
```

- Uma expressão aritmética é:
  - uma constante inteira, ou
  - a soma de duas expressões, ou
  - a diferença de duas expressões, ou
  - o produto de duas expressões, ou
  - o quociente de duas expressões.

Exemplos de expressões aritméticas

```haskell
-- 73/(3+3) * 8
expOk = Mul (Div (Cte 73)
                 (Som (Cte 3) (Cte 3)))
            (Cte 8)

-- 73/(3-3) * 8
expProblema = Mul (Div (Cte 73)
                       (Sub (Cte 3) (Cte 3)))
                  (Cte 8)
```

20.5.2 Avaliação de expressões aritméticas

- Um avaliador simples de expressões aritméticas:

```haskell
avalia :: Exp -> Integer

avalia (Cte x) = x
avalia (Som a b) = avalia a + avalia b
avalia (Sub a b) = avalia a - avalia b
avalia (Mul a b) = avalia a * avalia b
avalia (Div a b) = div (avalia a) (avalia b)
```

- Avaliando as expressões anteriores:

```
*Main> avalia exp0k
96

*Main> avalia expProblema
*** Exception: divide by zero
```

- A segunda expressão não pode ser avaliada, pois a divisão por zero leva a um resultado indefinido.

Tarefa 20.3

Redefina a função `avalia` para que ela não produza uma exceção quando em sua estrutura houver divisão por zero.

A função deve retornar uma indicação de falha ou sucesso, juntamente com o seu valor em caso de sucesso. Use o construtor de tipo `Maybe`.

Inspecione os resultados das avaliações das subexpressões diretamente usando análise de casos.

Tarefa 20.4

Modifique a função `avalia` do exercício anterior para usar operações monádicas ao invés de inspecionar os resultados das avaliações das subexpressões diretamente usando análise de casos.

Use explicitamente os operadores `(>>=)` e `(>>)` para sequenciamento das ações de E/S.

Não use a notação do.

Tarefa 20.5

Modifique a função `avalia` do exercício anterior para usar a notação do para realizar as operações monádicas.

Não use explicitamente os operadores `(>>=)` e `(>>)` para sequenciamento das ações de E/S.

20.6 Computações que produzem log

- Frequentemente é desejável para uma computação produzir um fluxo de dados adicional, além dos valores computados.
- *Logging* e *tracing* são os exemplos mais comuns nos quais dados são gerados durante uma computação, e queremos retê-los, mas eles não são os resultados principais da computação.
- Definição:

```haskell
newtype Out a = Out (a,String)
```

Exemplo

Redefina a função a `avalia` para calcular o valor de uma expressão aritmética e simultaneamente gerar um *trace* da avaliação de cada subexpressão.

```haskell
avalia :: Exp -> Out Integer
```

20.7 Soluções