José Augusto N. G. Manzano

PROGRAMAÇÃO DE COMPUTADORES COM LINGUAGEM julia

PRIMEIROS PASSOS

José Augusto N. G. Manzano

ORCID: 0000-0001-9248-7765

PROGRAMAÇÃO DE COMPUTADORES COM LINGUAGEM

PRIMEIROS PASSOS

São Paulo
2022 - Propes Vivens

ISBN 978-65-00-59016-6

9 786500 590166

CC BY-NC-ND 4.0 - 2022 - José Augusto N. G. Manzano

Esta obra está licenciada sob a licença:

- *Atribuição-NãoComercial-SemDerivações 4.0 Internacional Creative Commons*.
- *https://creativecommons.org/licenses/by-nc-nd/4.0/deed.pt_BR.*

É permitido:

- copiar, distribuir e comunicar publicamente a obra.

Sob as seguintes condições:

(BY) Reconhecimento: devem ser dados os créditos da obra de forma específica ao autor.

(NC) Não-comercial: não pode utilizar esta obra com finalidades comerciais.

(ND) Sem-derivações: se você alterar ou transformar este trabalho, não poderá distribuí-lo.

Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

Manzano, José Augusto N. G.
Programação de computadores com linguagem Julia [livro eletrônico] : primeiros passos / José Augusto N. G. Manzano. -- São Paulo : Ed. do Autor, 2022.
PDF

ISBN 978-65-00-59016-6

1. Algoritmos de computadores 2. Linguagem de programação (Computadores) 3. Julia (Linguagem de programação para computadores) I. Título.

22-139518 CDD-005.13

Índices para catálogo sistemático:

1. Linguagens de programação : Computadores : Processamento de dados 005.13

Inajara Pires de Souza - Bibliotecária - CRB PR-001652/O

Produção e Editoração: José Augusto Navarro Garcia Manzano
Imagens: canva.com (quadro de título) e flaticon.com (olho – observação)
Logo Julia: Stefan Karpinski - 23 de dezembro de 2012 - Domínio público - Arquivo: Julia prog language.svg
Edição: Propes Vivens

FERRAMENTAS UTILIZADAS

Produto: **Julia - Linguagem de Programação - [Versão 1.7.3 – 06/05/2022]**
Sítio: **https://julialang.org/**

Produto: **SQLite - [Versão 3.39.2 – 21/07/2022]**
Sítio: **https://www.sqlite.org/**

Produto: **Git bash - [Versão 2.37.2 – 12/08/2022]**
Sítio: **https://git-scm.com/**

AGRADECIMENTOS

À minha esposa Sandra e à minha filha Audrey, motivos de constante inspiração ao meu trabalho. Pela paciência nos momentos de ausência que tenho, quando estou absorto em escrever, dedicando-me ao ensino;

Há ainda uma pessoa que merece grande atenção, você, leitor, que me prestigia adquirindo esta e outras obras de minha autoria, sempre de forma legal, na livraria de sua confiança. Devido à sua postura tenho incentivo para continuar escrevendo, mesmo em um País em que está enraizada a cultura da pirataria, das famigeradas apostilas e tutoriais e da cópia ilegal do trabalho alheio. Muito obrigado por sua confiança e respeito.

A todos os alunos que passaram e passam por minhas mãos, que acreditaram e acreditam na minha pessoa e seguiram e seguem as orientações passadas; por me incentivarem continuamente quando me questionam sobre temas que ainda não conheço, por me levarem a um patamar maior por exigirem assim que eu pesquise mais.

Vida longa e próspera.

APOIO, PATROCÍNIO OU DOAÇÃO

Esta obra é distribuída gratuitamente a partir do site de seu autor na forma eletrônica (em formato PDF) e disponibilizada comercialmente na forma impressa junto as plataformas de auto publicação "AgBook/Clube de autores" respectivamente nos endereços **https://agbook.com.br/** e **https://clubedeautores.com.br/**.

A forma impressa é destinada aos leitores que preferem esta modalidade para leitura. Neste sentido, existem custos de produção que impedem a distribuição deste material de forma gratuita a menos que hajam patrocinadores que venham a auxiliar este tipo de produção. Havendo interesse em realizar patrocínio, por favor, entre em contato por meio do e-mail: **certificado@mail.com**.

O material impresso comercializado tem seu valor de venda revertido como pagamento de direito autoral que é reaplicado no desenvolvimento de novos trabalhos ampliando o projeto de desenvolvimento de livros de tecnologia em computação distribuídos de forma gratuita e de qualidade a toda comunidade brasileira. Caso deseje contribuir, além da aquisição impressa, poderá fazer doações de qualquer valor a partir dos meios de pagamentos indicados a seguir.

BITCOIN	PIX: EMAIL

BITCOIN: bc1qe2p4yqgnerwwma765wjrvwm0x408uq0h3q7l9m
EMAIL: bb@manzano.pro.br

Grato pela colaboração.

SUMÁRIO

Capítulo 1 - Linguagem Julia
1.1 - Breve histórico da linguagem 19
1.2 - Obtenção e instalação 20
1.2.1 - Instalação: Windows 21
1.2.2 - Instalação: Linux 23
1.2.3 - Instalação: macOS 24
1.3 - Cálculos matemáticos 26
1.3.1 - Constantes matemáticas 26
1.3.2 - Operadores aritméticos 29
1.3.3 - Funções matemáticas 31
1.4 - Variáveis 34
1.5 - Tipos de dados básicos (primitivos) 36
1.6 - Pacotes externos 41
1.7 - Ambiente interativo 44
1.8 - Programa "Alo, mundo!" e a linha de comando 45
1.9 - Palavras-chave 47

Capítulo 2 - Programação estruturada e modular
2.1 - Entrada e saída de dados 49
2.2 - Tomada de decisão 54
2.2.1 - Operadores relacionais 54
2.2.2 - Operadores lógicos 56
2.2.3 - Desvio condicional simples 58
2.2.4 - Desvio condicional composto 59
2.2.5 - Desvio condicional então-se 59
2.2.6 - Desvio com operador condicional 60
2.2.7 - Tratamento de exceções (introdução) 61
2.3 - Laços para repetições de blocos 62
2.3.1 - Laço condicional 62
2.3.2 - Laço incondicional 64
2.3.3 - Interrupção de laço 65
2.3.4 - Desvio incondicional 66
2.3.5 - Formatações 68
2.4 - Sub-rotinas 70
2.4.1 - Procedimento 70
2.4.2 - Escopo e visibilidade de variáveis 73
2.4.3 - Passagem de parâmetro 76
2.4.4 - Função 78
2.5 - Símbolos e Macros 82
2.6 - Recursividade 87
2.6.1 - Simples 88
2.6.2 - Cauda 88
2.7 - Terminal ANSI 89
2.8 - Documentação interna de código 93
2.9 - Regras básicas de estilo 94

Capítulo 3 - Coleções de dados

3.1 - Arranjos 97
3.1.1 - Arranjo com uma dimensão: vetor 98
3.1.2 - Arranjo com duas dimensões: matriz 117
3.2 - Tuplas 125
3.3 - Dicionários 128
3.4 - Conjuntos 132
3.5 - Estruturas (Classes) 135
3.5.1 - Ser ou não ser orientada a objetos, eis a questão 135
3.5.2 - Classe e objeto (simulados) 136
3.5.3 - Encapsulamento 139
3.5.4 - Método construtor 140
3.5.5 - Composição com tipo abstrato e polimorfismo 141
3.5.6 - Generalização como polimorfismo 142
3.6 - Cadeias de caracteres 146

Capítulo 4 - Programação (quase) funcional

4.1 - Ser ou não ser funcional, eis a questão 155
4.2 - Comportamento funcional 156
4.3 - Simulação de casamento de padrões 157
4.4 - Funções anônimas (lambda) 159
4.5 - Funções de primeira classe e ordem superior 160

Capítulo 5 - Operações auxiliares

5.1 - Randomização 163
5.2 - Data e hora 167
5.3 - Números racionais e complexos 178
5.4 - Conversões e bases numéricas 184
5.5 - Estatística 188
5.6 - Gráficos com arranjos 192
5.7 - Gráficos com funções 201
5.8 - Quadros de dados 212
5.9 - Conjuntos de dados "R" 232

Capítulo 6 - Uso de arquivos

6.1 - Tipos de arquivos e formas de acesso 235
6.2 - Acesso a arquivos e suas operações 236
6.3 - Arquivos do tipo texto 238
6.4 - Arquivos do tipo binário 242
6.5 - Diretórios, caminhos e nomes de arquivos 247
6.6 - Gerenciamento com banco de dados 250

Capítulo 7 - Pacotes

7.1 - Gerenciador de pacotes 255
7.2 - Módulos, uma forma de pacote interno 263
7.3 - Criação manual de pacotes 268
7.4 - Definição automatizada de pacotes 276
7.5 - Submissão de pacotes no *Github* 284

Capítulo 8 - Exemplos de aplicação

8.1 - Numeração arábica e romana 289
8.2 - Valor por extenso 292
8.3 - Operações binárias: bit-wise, bit-shift e outras 294
8.4 - Dígito verificador 298
8.5 - Funções para classificação de dados 306
8.6 - Criptografia de Caesar 313
8.7 - Verbo de primeira conjugação 315
8.8 - Acesso direto em arquivo binário 316

Apêndices

A - SQLite 321
B - Git bash 327
C - Argumentos variados 329
D - Tratamento de exceções (algo a mais) 331
E - Formatação de milhares 339
F - Programação genérica 343
G - Arranjos com funções 347
H - Escopo de variáveis 351
I - Definição de constantes 355

Referência bibliográfica 357

PREFÁCIO

A linguagem Julia é diferente de outras linguagens de programação em diversos aspectos. Neste sentido, convido você a fazer uma viajem de aprendizado inicial, dinâmico e aplicado com este livro. Este texto, apesar de introdutório, é sem sombra de dúvidas o mais completo material publicado no Brasil em língua portuguesa até o momento, além de ser o primeiro livro nacional indexado internacionalmente junto ao ISBN (*International Standard Book Number*).

O estilo de apresentação de seu conteúdo é dinâmico e dirigido, principalmente quando relacionado ao uso interativo do ambiente de programação a partir da orientação de uso de diversas funcionalidades contextualizadas. Para seu estudo parte-se do pressuposto de que você possui bons conhecimentos de lógica de programação, além da devida destreza de uso de seu sistema operacional no tocante aos procedimentos de gerenciamento de arquivos e diretórios (pastas).

Por possuir um ambiente interativo de operação, Julia oferece a seu usuário um mecanismo rápido de acesso a seus recursos. Isso permite experimentar muitas de suas operações sem a necessidade de escrever longos programas. E é neste sentido que este livro segue sua apresentação: grande parte da abordagem deste texto segue o caminho interativo de uso da linguagem sem deixar de lado a parte programada.

Este livro não passou por revisão gramatical, estando em sua forma mais bruta, edição pré-alpha. Por isso, poderão ser encontrados erros de escrita, concordância ou outros que passaram desapercebidos, exceto os códigos de programas os quais foram exaustivamente testados. Auxílios, neste sentido e mesmo na estrutura dos programas, serão sempre bem-vindos, desde que sejam realizados sem nenhum interesse financeiro ou comercial e posicionados de forma respeitosa. Os colaboradores deste trabalho serão mencionados sempre nas próximas edições da obra na seção colaboradores com seus nomes completos e por extenso, dispostos em ordem alfabética após a seção sobre o autor.

Ao final deste estudo o leitor estará capacitado a entender e aplicar as técnicas básicas de programação no desenvolvimento de aplicações Julia. Este trabalho é dividido em oito capítulos mais apêndices e deverá ser estudado em sua totalidade.

No capítulo 1 faz-se a apresentação introdutória e inicial de uso da linguagem passando por sua: obtenção; instalação; configuração; uso de algumas operações matemáticas diversas com uso de operadores aritméticos; uso de constantes e funções; definição de variáveis; detecção de tipos de dados e instalação, uso e remoção inicial de pacotes. Caso não queira instalar a linguagem Julia em seu sistema, você poderá fazer uso da linguagem em modo online a partir do endereço **https://replit.com/languages/julia** (1.4.1) ou em **https://julialang.org/learning/tryjulia/** (1.6.3).

O capítulo 2 abrange o estilo de programação estruturada e modular. Neste capítulo você tem contato com uma possível forma de codificação de programas na linguagem. São apresentados os recursos tradicionais de programação estruturada (definição e utilização de laços *while* e *for*) e de programação modular (definição e utilização de sub-rotinas).

No capítulo 3 é mostrado o uso das coleções de dados administradas na linguagem. Desta forma você tem acesso a diversos detalhes sobre o uso e manipulação de dados baseados em arranjos, listas, tuplas, dicionários, conjuntos, estruturas e tratamento de cadeias de caracteres.

O capítulo 4 explora alguns recursos da linguagem que possibilitam escrever códigos quase funcionais, além de abordar suas restrições operacionais pelo fato da linguagem não ser modelada diretamente para este paradigma. Neste sentido você tem acesso há alguns recursos funcionais que podem ser usados. São apresentadas orientações sobre o comportamento funcional, simulação de comportamento de padrões, definições de funções anônimas (que podem ser usadas livremente em qualquer estilo de programação dentro do ecossistema Julia), uso de funções de ordem superior e funções de primeira classe.

No capítulo 5 é apresentado a aplicação de alguns recursos da linguagem considerados auxiliares, pois são esses recursos responsáveis por grande parte da potencialidade de trabalho. São apresentados recursos de: randomização; números racionais e complexos; manipulação de data e hora, conversão entre tipos de dados, operações estatísticas, uso de gráficos e operações com quadros de dados.

O capítulo 6 mostra a aplicação e uso dos recursos de persistência no armazenamento de dados na forma de arquivos. São apresentados os usos de arquivos do tipo texto e tipo binário com operações de acesso sequencial, além do uso integrado da linguagem com o *Gerenciador de Banco de Dados SQLite*.

No capítulo 7 descreve-se os passos básicos de uso dos recursos de gerenciamento de pacotes, além dos detalhes de desenvolvimento de módulos, pacotes internos e externos, bem como a publicação de pacotes na plataforma Github.

O capítulo 8 demonstra o uso e aplicação da linguagem sob o desenvolvimento de alguns projetos e/ou operação. Neste sentido são explorados alguns recursos apresentados na obra além de outros novos. Parte dos programas apresentados seguem o estilo do capítulo 2, mas também mostram um estilo conveniente de produzir programas em Julia.

Os apêndices complementam detalhes que podem ser usados na produção de seus programas ou auxiliar o estudo geral deste livro. Aqui você encontra orientação para instalação do SQLite e Git bash, além de certos recursos complementares aos demais capítulos.

Os arquivos de programas desenvolvidos neste livro estão disponibilizados para *download* junto a plataforma Github a partir do endereço: **https://github.com/J-AugustoManzano/LivroLinJulia**.

Ao final do estudo desta obra você terá em mãos o mesmo conhecimento que obteria em um curso básico da linguagem. Como sempre, um grande abraço e excelente aprendizado. Que este trabalho seja de grande proveito e o investimento, ora nele feito, seja válido e proveitoso no final do seu estudo.

Augusto Manzano

"O aprendizado é um tesouro que segue seu dono por todos os lugares." - Provérbio chinês

SOBRE O AUTOR

José Augusto Navarro Garcia Manzano é professor com mestrado possuindo formação em Análise e Desenvolvimento de Sistemas, Ciências Econômicas e Licenciatura em Matemática atuando na área de Tecnologia da Informação, Computação e Informática (desenvolvimento de software, ensino e treinamento) desde 1986. Participou do desenvolvimento de aplicações computacionais para áreas de telecomunicações e comércio, possuindo participações no mundo de código-aberto (*open source*). Como professor iniciou sua atividade docente em cursos livres, trabalhando posteriormente em empresas de treinamento e atuando desde então nos ensinos técnico e superior.

Atualmente é professor com dedicação exclusiva no IFSP (Instituto Federal de Educação, Ciência e Tecnologia de São Paulo, antiga Escola Técnica Federal). Em sua carreira desenvolveu competências e habilidades para ministrar componentes curriculares de Lógica de programação (Algoritmos), Estrutura de dados, Microinformática, Informática, Linguagens de programação estruturada, Linguagens de programação orientada a objetos, Engenharia de software, Sistemas de informação, Engenharia da informação, Arquitetura de computadores e Tecnologias Web. Possui conhecimento de uso de diversas linguagens de programação imperativas, descritivas e de marcação, tais como: BASIC (clássico e estruturado), COBOL, COMAL, Logo, Scratch, Assembly, Pascal, FORTRAN (77 e 90), C, C++, D, Java, F#, Modula-2, C#, Lua, HTML, XHTML, CSS, Javascript, VBA, Ada, Rust, Python, LISP, Haskell, Standard ML, OCaml, Miranda, Hope, Groovy, Julia e Elixir. Possui mais de uma centena de livros publicados, além de artigos no Brasil e exterior.

COLABORADORES

SEU NOME AQUI

CAPÍTULO 1

LINGUAGEM JULIA

O objetivo deste capítulo é apresentar conceitos introdutórios e básicos para a utilização da linguagem de programação para computadores Julia. São apresentados diversos detalhes como: histórico da linguagem; obtenção, instalação e configuração; operações matemáticas diversas com uso de operadores aritméticos, constantes e funções; definição de variáveis; detecção de tipos de dados e instalação, uso e remoção de pacotes.

1.1 - Breve histórico da linguagem

A linguagem de programação Julia[1] começou a ser desenvolvida em fevereiro de 2009 por quatro cientistas da computação do *Massachusetts Institute of Technology* (MIT): Alan Edelman, Jeff Bezanson, Stefan Karpinski e Viral B. Shah com foco no desenvolvimento de programas voltados a área científica, tento sua primeira versão publicada no ano de 2012 (MAGALHÃES, ÁVALOS & SANTOS, 2017, p. 1). A versão usada para o desenvolvimento deste livro é a de número **1.8.2** lançada em 29 de setembro de 2022, podendo ser executada nos sistemas operacionais Windows, Linux, FreeBSD e Mac OS.

No site oficial da linguagem (**https://julialang.org/**) encontra-se a descrição de algumas características de ações de Julia, sendo dinâmica como a linguagem Ruby, multiparadigma como a linguagem C++ e de código aberto como a linguagem Lua. Possui comportamento semelhante a linguagens interpretadas como Python, Elixir, Haskell, entre outras e alta velocidade de execução como as encontradas em linguagens como C, C++, Rust, Ruby entre outras. Os comandos escritos no ambiente interativo da linguagem são automaticamente compilados e então executados, como também ocorre com a linguagem Lua. O recurso de compilação em tempo real é conhecido como JIT (*Just-In-Time*), sendo uma forma de compilação dinâmica da linguagem.

Segundo Magalhães, Ávalos & Santos (2017, p. 3) a linguagem "apresenta diversas funções matemáticas e com alta precisão dos valores numéricos" usados e acrescenta que opera seu processamento com "recursos para computação paralela e distribuída, além da facilidade para integrar com funções em C sem necessidade de nenhuma API específica e em Python através do pacote PyCall". É uma linguagem moderna e de fácil aprendizagem.

A linguagem pode ser usada de dois modos distintos: um de forma programada a partir do uso de um editor de textos, como bloco de notas, vi (vi-ai) ou Juno; outra a partir de um *prompt* de comando específico conhecido pela sigla *REPL* (*Read–Eval–Print Loop*) usado em modo interativo. O ambiente *REPL* é útil para a obtenção rápida de respostas, sendo um ambiente bastante adequado para a aprendizagem de diversos recursos da linguagem.

O ambiente *REPL* se caracteriza por ser um local de trabalho de visual simplista e minimalista controlado por um laço (*Loop*) que o mantém aberto até que a função de encerramento "**`exit()`**" seja informada na linha de comando. Este ambiente faz a leitura (*Read*), avaliação da expressão escrita (*Eval*) e a apresentação de algum resultado (*Print*), que pode ser uma resposta ao que se pretende avaliar ou mensagem de erro se algo foi informado incorretamente.

[1] O nome Julia não possui nenhum motivo ou significado sendo apenas um nome despretensioso dado a linguagem.

Por ser Julia uma linguagem de programação de computadores com conotação científica, ela é mais voltada para uso de problemas relacionados ao paradigma declarativo funcional. No entanto, possui também suporte ao paradigma imperativo e de orientação a objetos, o que faz desta linguagem ser multiparadigma podendo ser usada na solução de problemas gerais, apesar de seu forte apelo especializado na computação numérica e científica, que é seu forte.

1.2 - Obtenção e instalação

Para fazer a obtenção da linguagem Julia acesse primeiramente o sítio oficial em **https://julialang.org/** como apresenta a figura 1.1 e selecione no menu superior a opção "**Download**" ou acione o botão verde com o mesmo nome

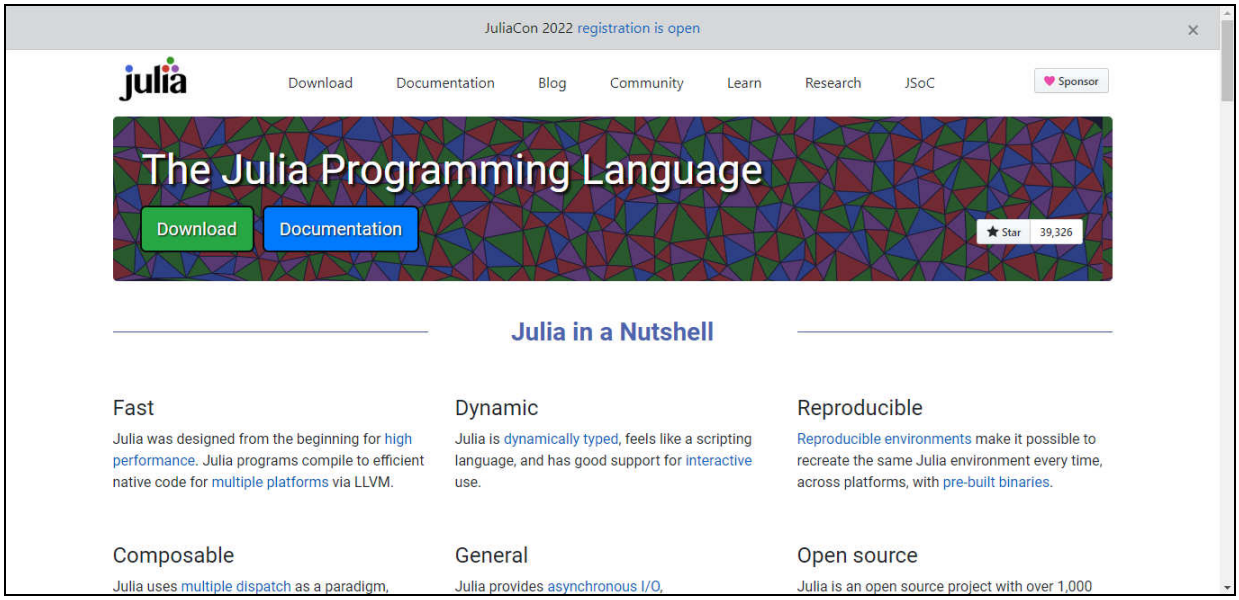

Figura 1.1 - Sítio oficial linguagem Julia.

Após selecionada uma das opções de *download* é apresenta a página "**Download Julia**" como mostra a figura 1.2, contendo as opções relacionadas ao sistema operacional em uso. Selecione seu sistema operacional e baixe o arquivo para seu sistema.

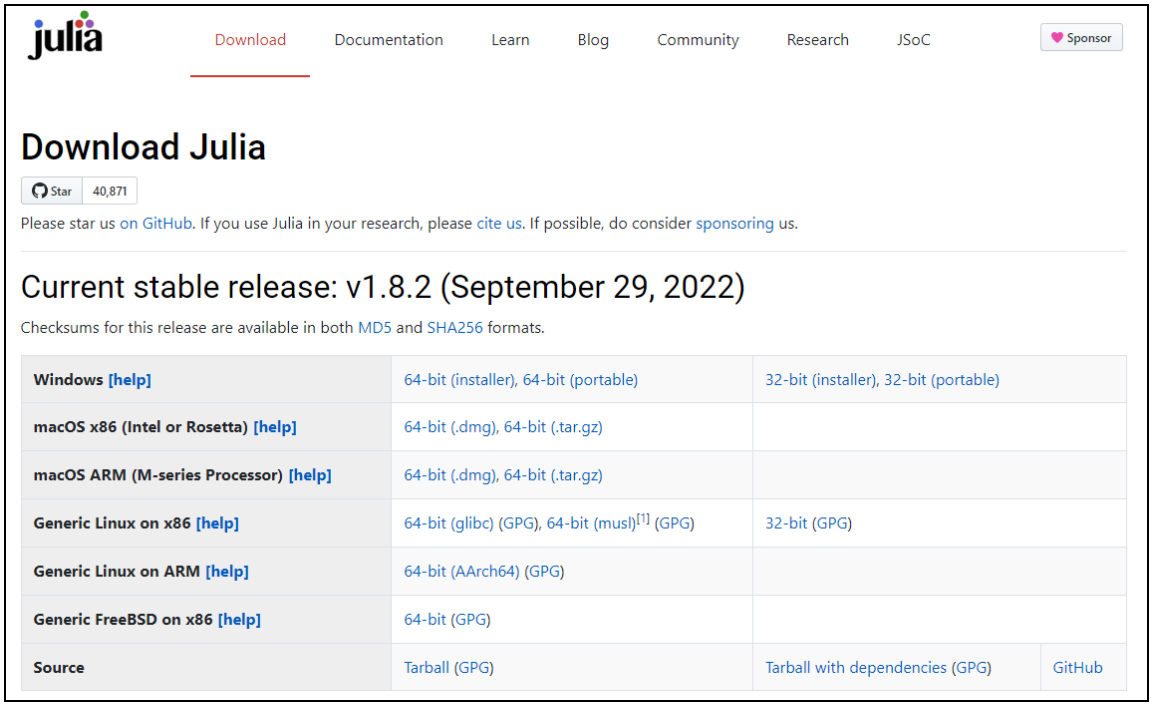

Figura 1.2 - Opções de download.

1.2.1 - Instalação: Windows

Considerando o sistema operacional Windows (*ação realizada com Windows 10*) selecione, preferencialmente, a opção "**64-bit (installer)**" para instalação em computador de 64 bits ou a opção "**32-bit (installer)**" para 32 bits, Caso queira instalar a linguagem em um *pendrive* para Windows selecione a opção "**64-bit (portable)**" ou "**32-bit (portable)**".

Levando em consideração um computador de 64 bits e a seleção da opção "**64-bit (installer)**" aguarde a cópia para seu sistema do programa de instalação "**julia-1.8.2-win64**". Vá até o local onde o arquivo foi copiado e execute o programa de instalação como administrador, confirme a execução do programa com o botão "**Executar**" quando apresentada a caixa de mensagem "**Controle de Conta do Usuário**". Será apresentada a tela de instalação inicial como mostra a figura 1.3.

Note que no campo "**Installation Directory**" é indicado "**C:\Users\Augusto\AppData\Local\Programs\Julia-1.8.2**" como local de gravação do programa. É pertinente para facilitar o acesso a linguagem fazer alteração deste caminho de instalação. Assim sendo, acione o botão "**Browse**" e selecione como local o diretório "**Arquivos de Programas**" (ou como indicado "**Program Files**") como mostra a figura 1.4 e acione o botão "**OK**".

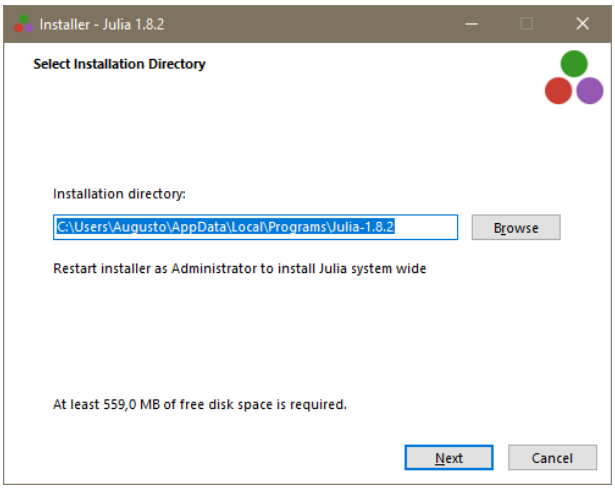

Figura 1.3 - Opções de download.

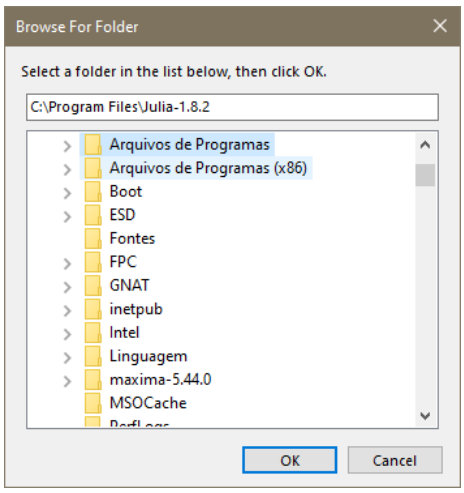

Figura 1.4 - Seleção de local apropriado para instalação.

Na sequência acione o botão "**Next**" por duas vezes e aguarde o processo de instalação ser executado até sua finalização, que poderá ser um pouco demorado. Ao termina da ação é apresentada a tela de finalização da instalação. Acione neste momento o botão "**Finish**".

Observe que na área de trabalho do sistema operacional Windows foi criado um ícone chamado **Julia 1.8.2**. Neste instante acione o referido ícone e será apresentado o ambiente *REPL (Read-Eval-Print-Loop)* da linguagem como mostra a figura 1.5.

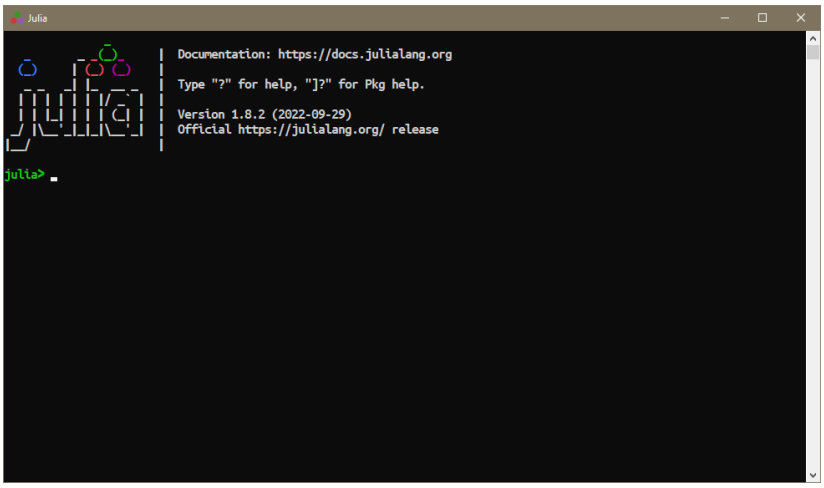

Figura 1.5 - Tela do modo REPL (Windows).

Para encerrar a execução do programa *REPL* acione o botão "**X**" ou execute no *prompt* de comando do ambiente a chamada da função "**`exit()`**".

Agora é necessário realizar outro teste de execução do ambiente *REPL* da linguagem Julia. Diferentemente do acesso via o ícone da área de trabalho abra a janela de linha de comando "*Prompt de Comando*" e na linha de comando apresenta excute o comando "**`julia`**" e acione a tecla "**`<Enter>`**" e observe a apresentação de uma mensagem de erro, como indicado junto a figura 1.6.

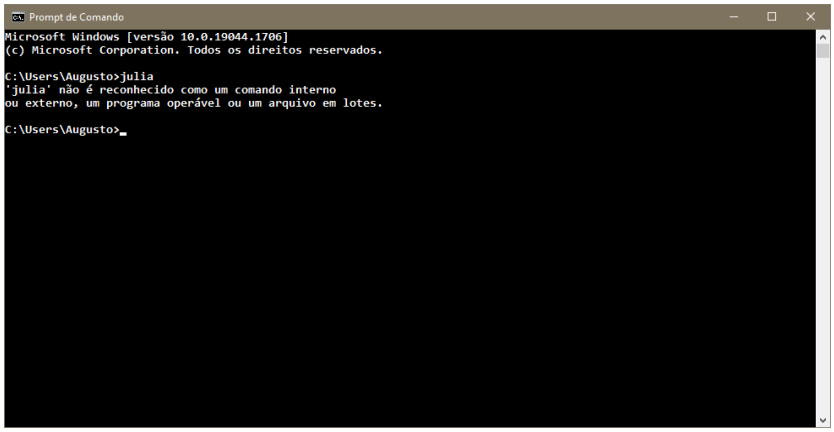

Figura 1.6 - Prompt de Comando com apresentação de erro de execução.

Observe que é possível executar o ambiente da linguagem de dentro da interface gráfica, mas não no modo da interface texto. Isto ocorre devido à necessidade de ter que fazer alguns ajustes no sistema, ou seja, é necessário configurar a variável de ambiente "PATH" do sistema. Para tanto, execute na linha de comando do *prompt* a instrução:

```
control sysdm.cpl
```

Aguarde o surgimento da caixa de diálogo "*Propriedades do Sistema*", selecione na sequência a guia "*Avançado*" de modo que se tenha apresentado o equivalente a imagem da figura 1.7.

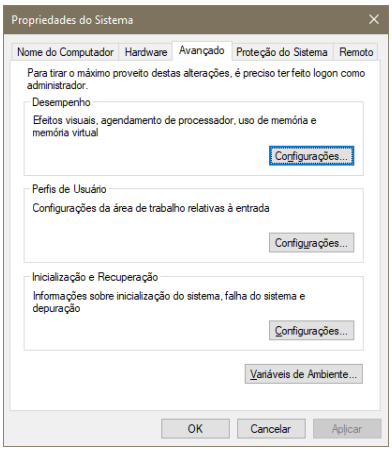

Figura 1.7 - Seleção da guia "Avançado" em "Propriedades do Sistema".

Acione em seguida o botão "**`Variáveis de Ambiente`**" para que seja apresenta a tela de configuração com mesmo nome indicado na figura 1.8. Veja que a tela de configuração possui duas áreas de seleção, sendo a área "*Variáveis de usuário para*" destinada a configurações específicas para o usuário ativo e a área "*Variáveis do sistema*" destinada a configurações gerais para todo o sistema, incluindo-se todos os usuários cadastrados nesse computador.

Veja que em ambas as janelas há a indicação de uma variável chamada "**`Path`**". Você pode escolher qualquer uma dessas duas opções. A opção de seleção para a área "*Variáveis do sistema*" é mais conveniente. Assim, escolha a variável "*Path*" dessa área.

Após selecionar a variável "*Path*" da área "*Variáveis do sistema*" acione o botão "**`Editar...`**" que apresentará a tela "*Editar a variável de ambiente*" como mostra a figura 1.9.

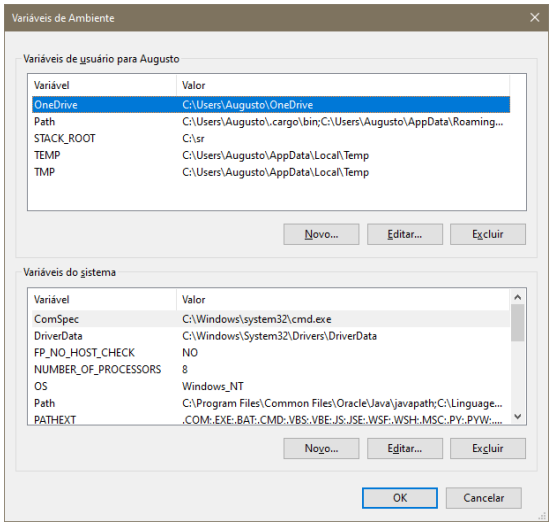

Figura 1.8 - Tela de configuração "Variáveis de Ambiente".

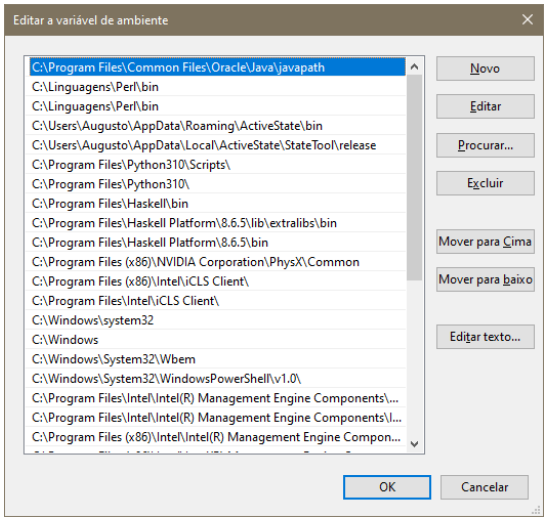

Figura 1.9 - Configuração da variável de ambiente, neste caso, Path.

Neste momento acione o botão "**Novo**" e informe no campo apresentado o local (diretório) em que a linguagem Julia foi instalada, ou seja, "**`C:\Program Files\Julia-1.8.2\bin`**". Na sequência acione o botão "**OK**" por três vezes.

Feche a janela "*Prompt de Comando*" e abra-a novamente para que a alteração definida seja por ela reconhecida. Na sequência, excute o comando:

```
julia
```

Acione a tecla "**`<Enter>`**" e veja se é apresentada a tela do ambiente de ação da linguagem como indicado na figura 1.10.

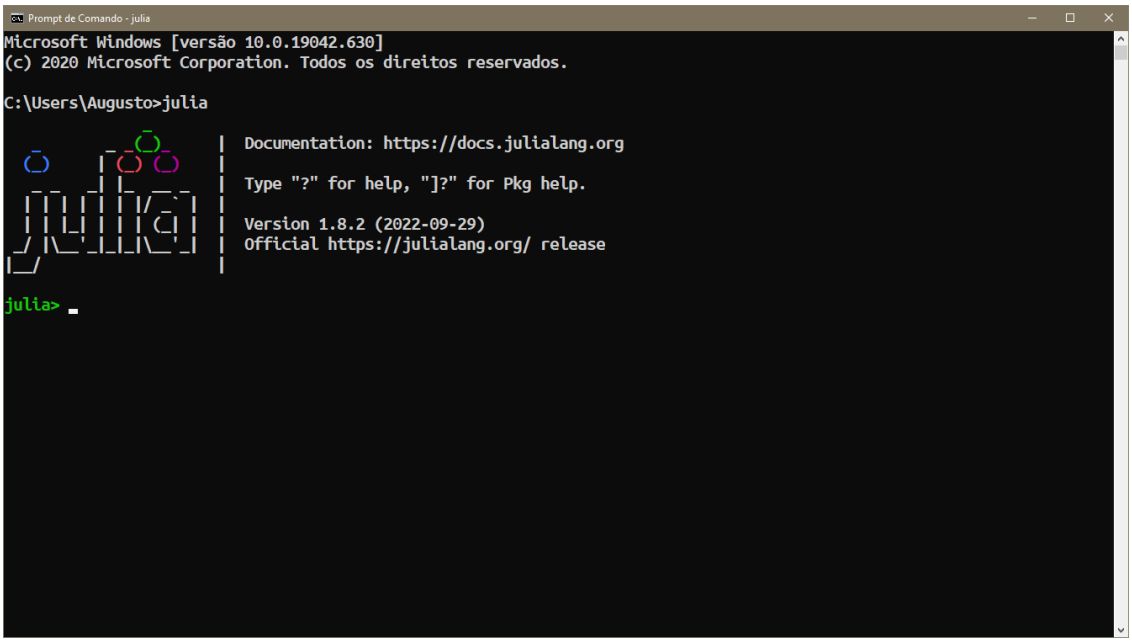

Figura 1.10 - Tela de ação do modo REPL executada a partir da linha de comando (Windows).

A partir dos passos anteriores seu ambiente de trabalho no sistema operacional Windows para a linguagem Julia está devidamente instalado e configurado para uso tanto interno no modo gráfico, a partir do ícone de acesso, como para uso no modo texto a partir da linha de comando do *prompt* do sistema.

A configuração de acesso a linha de comando é útil quando se deseja, principalmente, fazer uso da linguagem Julia a partir de um editor de texto comum ou voltado especificamente a programação.

1.2.2 - Instalação: Linux

Considerando o sistema operacional Linux (*ação realizada com Fedora Linux 33*) selecione, preferencialmente, a opção "**64-bit (glibc)**" para instalação em computador de 64-bit ou a opção "**32-bit (GPC)**" para 32-bit. Aguarde a cópia para seu sistema do programa de instalação "**julia-1.8.2-linux-x86_64.tar.gz**".

Abra o programa "**Terminal**" e entre no diretório "**Downloads**" (**cd Downloads**) e execute na linha de comando do *prompt* do sistema a instrução:

```
sudo tar -zxvf julia-1.8.2-linux-x86_64.tar.gz -C /opt
```

Aguarde a operação de descompactação e instalação da linguagem Julia dentro do diretório "**/opt**", local apropriado para a instalação de software externo ao sistema.

Encerrada esta etapa é necessário configurar a variável de ambiente "**PATH**". Para fazer isso permanentemente abra o arquivo "**~/.bashrc**" com um editor de sua preferência e acrescente na última linha a instrução:

```
export PATH=$PATH:/opt/julia-1.8.2/bin
```

Grave a alteração indicada e saia do programa de edição de textos. Em seguida execute as instruções:

```
source ~/.bashrc
clear
```

Na sequência, excute o comando:

```
julia
```

Acione a tecla "**<Enter>**" e veja se é apresentada a tela do ambiente *REPL* de operação da linguagem sendo apresentado como indicado na figura 1.11.

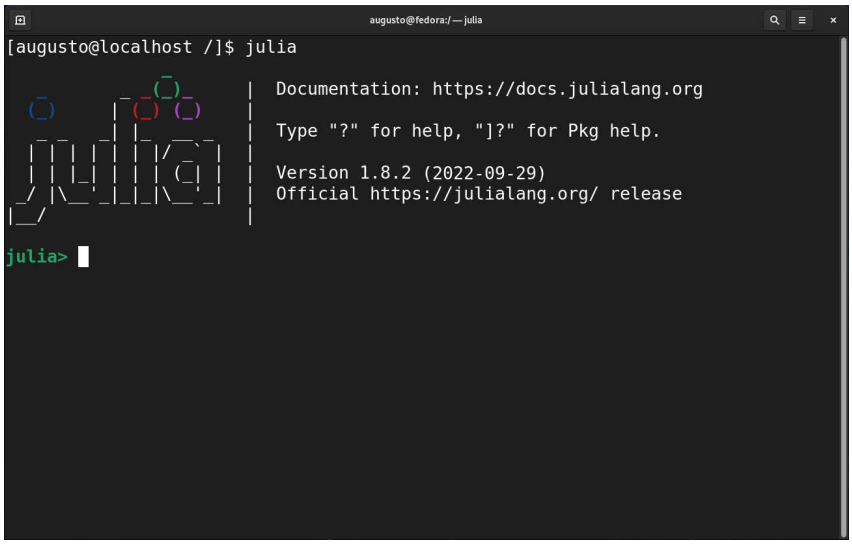

Figura 1.11 - Tela do modo REPL (Linux).

A partir dos passos anteriores seu ambiente de trabalho no sistema operacional Linux para a linguagem Julia está devidamente instalado e configurado para uso a partir da linha de comando.

Para encerrar a execução do programa *REPL* acione o botão "**X**" ou execute no *prompt* de comando do ambiente a chamada da função "**exit()**".

1.2.3 - Instalação: macOS

Considerando o sistema operacional macOS (*ação realizada com Catalina*) selecione, preferencialmente, a opção "**64-bit (.dmg)**" para instalação. Aguarde a cópia para seu sistema do programa de instalação "**julia-1.8.2-mac64.dmg**".

Assim que o programa estiver copiado, abra o programa "**Finder**" e a partir da pasta "**Downloads**" coloque em execução o programa "**julia-1.8.2-mac64.dmg**". Será apresentada a montagem de uma unidade que entra em execução apresentando a janela "**Julia-1.8.2**" como indicado na figura 1.12. Copie o ícone da linguagem para dentro da pasta "**Applications**", arrastando-o. Acione *eject* na unidade apresentada para "**Julia-1.8.2**", abra a janela "**Aplicativos**" no "**Finder**" e acione o ícone do programa "**Julia-1.8**" e será apresentado o ambiente *REPL* da linguagem como indicado na figura 1.13.

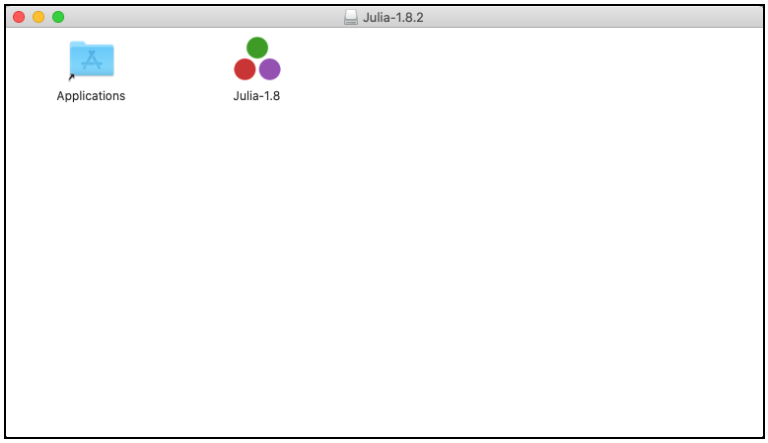

Figura 1.12 - Tela da janela Julia-1.8.2 (macOS).

Figura 1.13 - Tela do modo REPL (macOS).

Para encerrar a execução do programa *REPL* acione o botão "**X**" ou execute no *prompt* de comando do ambiente a chamada da função "**`exit()`**" e acione o botão "**X**" da janela.

Agora é necessário realizar outro teste de execução do ambiente *REPL* da linguagem Julia. Diferentemente do acesso via o ícone da pasta "**Aplicativos**", abra a janela de linha de comando "*Terminal*" e na linha de comando apresenta excute o comando "**`julia`**" e acione a tecla "**`<Enter>`**" e observe a apresentação de uma mensagem de erro, como indicado junto a figura 1.14.

Observe que é possível executar o ambiente da linguagem de dentro da interface gráfica, mas não no modo da interface texto. Isto ocorre devido à necessidade de ter que fazer alguns ajustes no sistema, ou seja, é necessário configurar a variável de ambiente "**PATH**" do sistema. Para tanto, execute na linha de comando do *prompt* a instrução:

```
sudo ln -s /Applications/Julia-1.8.app/Contents/Resources/julia/bin/julia /usr/local/bin/julia
```

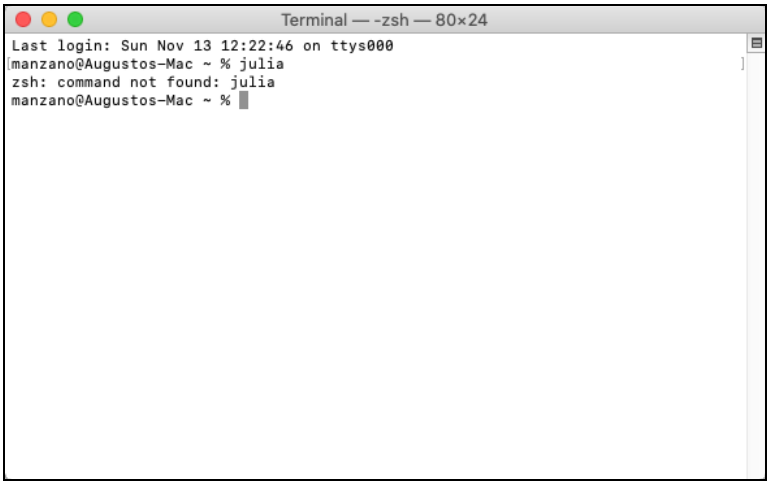

Figura 1.14 - Terminal com apresentação de erro de execução.

Feche a janela "*Terminal*" e abra-a novamente para que a alteração definida seja por ela reconhecida. Na sequência, excute o comando:

```
julia
```

Acione a tecla "**<Enter>**" e veja se é apresentada a tela do ambiente de ação da linguagem como indicado na figura 1.15.

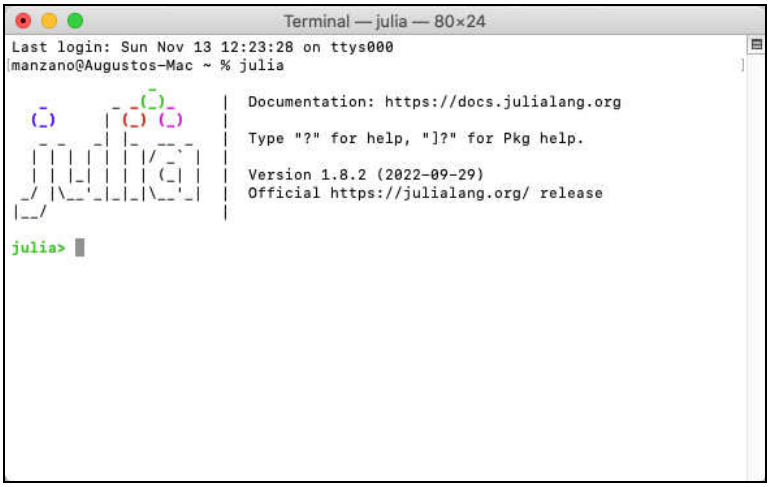

Figura 1.15 - Tela de ação do modo REPL executada a partir da linha de comando (macOS).

A partir dos passos anteriores seu ambiente de trabalho no sistema operacional macOS para a linguagem Julia está devidamente instalado e configurado para uso tanto interno no modo gráfico, a partir do ícone de acesso, como para uso no modo texto a partir da linha de comando.

A configuração para a linha de comando é útil quando se deseja fazer uso da linguagem Julia a partir de um editor de texto comum ou voltado especificamente a programação.

1.3 - Cálculos matemáticos

O forte de qualquer linguagem de programação, por mais simples que seja, é realizar operações lógicas e matemáticas a fim de explorar os computadores naquilo em que foram idealizados há fazer. É claro que existem diversas linguagens de programação voltadas ao atendimento de certas necessidades. A linguagem Julia é voltada para atender eminentemente operações matemáticas, e neste sentido possui um extenso conjunto de ferramentas operacionais como: constantes, operadores aritméticos e funções matemáticas.

1.3.1 - Constantes matemáticas

A ciência matemática para sua expressão se utiliza como parte de seu ferramental de operação o uso de diversas constantes matemáticas. Uma constante matemática é por sua natureza a expressão de uma quantidade numérica que possui sempre o mesmo valor, tendo como exemplos básicos os valores de π (constante de Arquimedes), e (número de Euler), φ (proporção áurea - número de ouro), entre outras.

No segmento de constantes a linguagem Julia permite o acesso há algumas constantes definidas internamente em seu ecossistema de operação, existindo funções de escopo global e funções de escopo externo. As funções de escopo global podem ser usadas diretamente no *prompt* do *REPL* e as funções de escopo externo devem ser usadas com a indicação do nome do pacote (biblioteca) a que pertencem, neste caso "**`Base.MathConstant`**".

Há duas formas de uso das constantes pertencentes a um pacote, sendo de forma direta ou indireta. Na forma indireta basta indicar o nome do pacote e mencionar a constante e na forma direta usa-se o comando "**`using`**" com a indicação do pacote. Veja na tabela 1.1 as funções de escopo global e externo presentes na linguagem Julia.

Constante	Resultado	Operação
`pi [π]`	`3.1415926535897...`	`Constante de Arquimedes (global).`
`Base.MathConstants.eulergamma [γ]`	`0.5772156649015...`	`Constante de Euler-Mascheroni (externo).`
`Base.MathConstants.golden [φ]`	`1.6180339887498...`	`Constante de golden ratio (externo).`
`Base.MathConstants.catalan`	`0.9159655941772...`	`Constante de Catalan (externo).`
`Base.MathConstants.e [e]`	`2.7182818284590...`	`Número de Euler (externo).`

Tabela 1.1 - Exemplos de constantes da linguagem Julia.

O uso das constantes externas torna-se trabalhoso se tiver que informar o nome do pacote "**`Base.MathConstant`**", sendo uma forma de mitigar este trabalho fazer uso no *prompt* do ambiente *REPL* da instrução:

```
using Base.MathConstants
```

A partir da colocação em uso do pacote "**`Base.MathConstant`**" ocorre o carregamento em memória das constantes do pacote. A partir deste momento o comportamento dessas constantes se assemelha a forma de uso das constantes globais. Para ver as constantes existentes na memória, neste momento, execute o código de programa seguinte:

```
for constantes in names(Base.MathConstants)
    simbolo = eval(constantes)
    if typeof(simbolo) <: Irrational
        println(constantes, "\t", simbolo)
    end
end
```

Não se preocupe com o que está escrito no código anterior. No decorrer do estudo apresentado neste livro isso se tornará claro e será devidamente explicado. Aguarde com um pouco de paciência.

Observe que no ambiente *REPL* as constantes podem ser usadas de forma extensa a partir de seus nomes de identificação ou por meio de seus símbolos matemáticos baseados em uma letra grega ou outro caractere que represente sua operação. Para conseguir fazer uso do símbolo, por exemplo, "π" ao invés de usar o termo "pi" para obter seu valor basta escrever no *prompt* do *REPL* os caracteres "**`\pi`**" (contra barra e pi), acionar em seguida as teclas "**`<Tab>`**" para apresentar o caractere e "**`<Enter>`**" para ver o resultado. Desta forma experimente escrever os seguintes rótulos:

```
\pi          acione "<Tab>" para obter o caractere "π" e em seguida acione "<Enter>";
\gamma       acione "<Tab>" para obter o caractere "γ" e em seguida acione "<Enter>";
\euler       acione "<Tab>" para obter o caractere "e" e em seguida acione "<Enter>".
```

Além da possibilidade de uso de constantes definidas no ecossistema Julia é possível na instância de uso do ambiente *REPL* definir constantes pessoais que são válidas enquanto o ambiente está carregado em memória. Ao sair do ambiente e retornar em momento a frente as constantes definidas não são disponibilizadas, devendo-se redefini-las novamente. Para se fazer a definição de constantes particulares deve-se usar o comando "**const**" a partir da seguinte sintaxe:

```
const <nome da constante> = <valor da constante>
```

Como exemplo, considere definir uma constante que retorne o valor da constante de Pitágoras, ou seja, que retorne o valor de "**√2**". Para tanto, escreva no *prompt* do *REPL* a seguinte instrução:

```
const Pythag = sqrt(2)
```

A partir dessa definição quando se quiser obter o valor da constante basta executar na linha de comando do prompt do *REPL* a chamada da constante "**Pythag**" para obter como resposta o valor "**1.4142135623730951**".

Um detalhe adicional importante é que a linguagem Julia é do tipo *case-sensitive*, ou seja, é uma linguagem que diferencia caracteres maiúsculos e minúsculos. Desta forma se for executa a chamada da constante de Pitágoras como "**pythag**" será retornada o erro "**ERROR: UndefVarError: pythag not defined**" informando que o termo "*pythag*" não está definido "*not defined*".

Além da geração dos símbolos de constantes com letras gregas há uma série de caracteres baseados em formato Unicode que podem ser apresentados dentro do ambiente e utilizados como nomes de constantes e variáveis (tema visto mais à frente) a partir do uso da tecla de contra barra "****" e com acionamento da tecla "**<Tab>**". Veja alguns exemplos:

```
\alpha        acione "<Tab>" para obter o caracter "α";
\gamma        acione "<Tab>" para obter o caracter "γ";
\sigma        acione "<Tab>" para obter o caracter "σ";
\Alpha        acione "<Tab>" para obter o caracter "A";
\Gamma        acione "<Tab>" para obter o caracter "Γ";
\Sigma        acione "<Tab>" para obter o caracter "Σ";
\delta        acione "<Tab>" para obter o caracter "δ";
\Delta        acione "<Tab>" para obter o caracter "Δ";
\^2           acione "<Tab>" para obter o caracter "²";
\^1           acione "<Tab>" para obter o caracter "¹";
\_1           acione "<Tab>" para obter o caracter "₁";
\_a           acione "<Tab>" para obter o caracter "ₐ";
\equiv        acione "<Tab>" para obter o caracter "≡";
\ge           acione "<Tab>" para obter o caracter " ≥";
\le           acione "<Tab>" para obter o caracter " ≤";
\ne           acione "<Tab>" para obter o caracter " ≠";
\in           acione "<Tab>" para obter o caracter "∈";
\notin        acione "<Tab>" para obter o caracter "∉";
\sqrt         acione "<Tab>" para obter o caracter "√";
\:apple:      acione "<Tab>" para obter o caracter "🍎";
\:banana:     acione "<Tab>" para obter o caracter "🍌";
\:pear:       acione "<Tab>" para obter o caracter "🍐";
\:camel:      acione "<Tab>" para obter o caracter "🐫";
\:dog:        acione "<Tab>" para obter o caracter "🐶";
\:cat:        acione "<Tab>" para obter o caracter "😸";
```

```
\:turtle:     acione "<Tab>" para obter o caractere "🐢";
\:car:        acione "<Tab>" para obter o caractere "🚗";
```

É importante ressaltar que no sistema operacional Windows devido a suas configurações alguns caracteres podem não ser devidamente apresentados.

Além das constantes matemáticas há outras constantes na linguagem que podem ser usadas, tais como:

```
julia> nothing  # especificação de nenhum valor

julia> VERSION  # indica a versão da linguagem em uso
v"1.7.3"

julia> DEPOT_PATH  # mostra os locais onde a linguagem está instalada no sistema
3-element Vector{String}:
 "C:\\Users\\Augusto\\.julia"
 "C:\\Program Files\\Julia-1.7.2\\local\\share\\julia"
 "C:\\Program Files\\Julia-1.7.2\\share\\julia"

julia> Sys.BINDIR  # mostra o local onde o executável Julia está armazenado
"C:\\Program Files\\Julia-1.7.3\\bin"
```

1.3.2 - Operadores aritméticos

Os operadores aritméticos são os mecanismos que permitem a definição de equações em programação, ou seja, são responsáveis pela execução do processamento matemático. Para o estabelecimento dessas operações além das constantes é necessário possuir um conjunto de ferramentas para a efetivação desses cálculos. Neste sentido, Julia apresenta um conjunto de operadores que atendem a diversas operações de cálculos matemáticos, como apresenta-se na tabela 1.2 os principais operadores.

Operador	Operação	Tipo	Operação	Prioridade
+	"+n" ou "n"	Unário	Manutenção de sinal.	-
-	-n	Unário	Inversão de sinal.	-
=	x = n	Binário	Atribuição do valor "n" em "x".	-
^	x ^ n	Binário	Exponenciação de x^n.	1
^ (y / n)	x ^ (y / n)	Binário	Radiciação de $\sqrt[n]{x^y}$.	1
/	x / n	Binário	Divisão de "x" por "n".	2
\	x \ n	Binário	Divisão inversa de "n" por "x".	2
//	x // n	Binário	Fração (racional) de "x" sobre "n".	2
%	x % n	Binário	Resto da divisão de "x" por "n".	2
*	x * n	Binário	Multiplicação de "x" por "n".	2
+	x + n	Binário	Adição de "x" por "n".	3
-	x - n	Binário	Subtração de "x" por "n".	3

Tabela 1.2 - Operadores aritméticos suportados pela linguagem Julia.

A tabela 1.2 mostra os operadores aritméticos disponibilizados e indica a ordem de precedência no cálculo de cada operação. Havendo a necessidade de fazer a mudança da ordem de precedência utilize os símbolos de parênteses.

Em **https://docs.julialang.org/en/v1/base/math/#math-ops** você tem acesso ao conjunto completo de operadores aritméticos disponibilizadas na linguagem Julia.

A seguir são apresentados alguns testes de uso dos operadores aritméticos indicados. Os exemplos mostram a operação indicada após o indicativo de *prompt* "**julia>**" em negrito em cor verde e seus respectivos resultados em itálico na cor azul. Informe no *prompt* apenas a operação desejada, ou seja, a parte do cálculo grafado de forma normal:

```
julia> +5  # definição de valor numérico positivo
5

julia> -5  # definição de valor numérico negativo
-5

julia> 1_000  # mesmo que 1000, mas com separação
1000

julia> 2 ^ 3  # cálculo da exponenciação de índice 3
8

julia> 25 ^ (1 / 3)  # cálculo da raiz cúbica de 25
2.924017738212866

julia> (5 ^ 2) ^ (1 / 2)  # raiz quadrada de 5 ao quadrado
5.0

julia> 9 / 2  # divisão com quociente real
4.5

julia> 7 \ 21  # divisão inversa de 21 por 7
3.0

julia> 2 // 3  # fração: dois terços
2//3

julia> 2 // 3 + 5 // 4  # soma de frações
23//12

julia> 2 // 3 + 5  # este tema é explorado no capítulo 5
17//3

julia> 9 % 2  # resto da divisão de 9 por 2
1

julia> 9.5 * 2.3  # multiplicação
21.849999999999998

julia> 2.5 + 3.5  # adição
6.0

julia> 7 - 2.5  # subtração
4.5
```

```
julia> 5 + 2 * 6
17

julia> (5 + 2) * 6  # mudança de prioridade de cálculo
42

julia> 5 + pi
8.141592653589793

julia> 5 - π
1.8584073464102069

julia> Base.MathConstants.golden / 3
0.5393446629166316

julia> +(1, 2, 3, 4, 5)  # forma alternativa para: 1 + 2 + 3 + 4 + 5
15
```

Observe que nos exemplos apresentados ocorreram a amostragem de resultados inteiros ou de ponto flutuante dependendo de seu retorno. A linguagem Julia identifica automaticamente se o valor a ser retornado deve ser inteiro ou de ponto flutuante.

Para a efetivação de cálculos aritméticos é possível, como demonstrado, fazer uso de valores numéricos entre si e com o uso das constantes disponibilizadas na linguagem.

1.3.3 - Funções matemáticas

Uma linguagem de programação é em essência uma calculadora. Desta forma, é comum encontrar a definição de constantes, operadores aritméticos e um conjunto de funções, destacando-se neste instante as funções matemáticas. Veja na tabela 1.3 a apresentação de algumas funções matemáticas mais populares.

Função	Operação
abs(x)	Retorna o valor sempre como positivo.
cbrt(x)	Retorna o valor da raiz cúbica de "x".
ceil(x)	Retorna o arredondamento para cima de "x".
cos(x)	Retorna o cosseno de um ângulo "x".
div(x, y)	Retorna o quociente inteiro de "x" dividido por "y".
exp(x)	Retorna o valor "*e*" elevado à "x".
factorial(x)	Retorna o fatorial do número "x".
floor(x)	Retorna o arredondamento para baixo de "x".
gcd(x, y, ...)	Retorna o maior divisor comum entre dois ou mais números.
hypot(x, y)	Retorna a hipotenusa dos catetos "x" e "y".
iseven(n)	Retorna verdadeiro se "n" for par.
isodd(n)	Retorna verdadeiro se "n" for ímpar.
isequal(x, y)	Retorna verdadeiro se "x" e "y" forem iguais.
lcm(x, y, ...)	Retorna o mínimo múltiplo comum entre dois ou mais números.
log(x)	Retorna o valor do logaritmo de "x" na base 2.
log10(x)	Retorna o valor do logaritmo de "x" na base 10.
max(x, y, ...)	Retorna o maior valor entre dois ou mais números.

Função	Operação
min(x, y, ...)	Retorna o menor valor entre dois ou mais números.
prod(faixa; [inic])	Retorna a multiplicação sucessiva da faixa de valores.
round(n)	Retorna o valor "n" arredondado matematicamente.
rationalize(x)	Retorna o valor de "x" na forma de fração.
sin(x)	Retorna o seno de um ângulo "x".
sqrt(x)	Retorna o valor da raiz quadrada de "x".
sum(faixa; [inic])	Retorna o somatório da faixa de valores.
tan(x)	Retorna a tangente de um ângulo "x".
trunc({tipo], x)	Retorna o inteiro de um número do tipo float sem arredonda-lo.

Tabela 1.3 - Funções matemáticas da linguagem Julia (algumas).

Em **https://docs.julialang.org/en/v1/base/math/#Mathematical-Functions** você tem acesso ao conjunto completo de funções disponibilizadas na linguagem Julia. A seguir são mostrados alguns testes de uso de algumas funções da linguagem. Há nesses testes, em algumas funções, o uso do operador "**|>**" conhecido como operador de fluxo ou operador de canalização que tem por finalidade passar o resultado de retorno de uma função como argumento de outra função na forma de uma expressão de transferência de ação.

```
julia> abs(-5)  # valor absoluto (módulo) de um número
5

julia> -5 |> abs
5

julia> cbrt(27)  # raiz cúbica
3.0

julia> ceil(5.1)  # arredondamento para o teto de um valor
6.0

julia> cos(1)  # cosseno: experimente "1 |> cos"
0.5403023058681398

julia> div(5, 2)  # divisão com quociente inteiro
2

julia> exp(2)  # exponencial do número "e" elevado a "2"
7.38905609893065

julia> factorial(5)  # fatorial de um número (de 0 até 20)
120

julia> factorial(BigInt(50))  # fatorial com BigInt() de 0 até aprox/e 100.000.000 (demorado)
30414093201713378043612608166064768844377641568960512000000000000

julia> floor(5.9)  # arredondamento para o piso de um valor
5.0
```

```
julia> gcd(1, 2, 4, 5)  # maior divisor comum
1

julia> hypot(2.5, 7)  # hipotenusa
7.433034373659253

julia> lcm(1, 2, 4, 5)  # mínimo múltiplo comum
20

julia> log(2)  # logaritmo na base "2"
0.6931471805599453

julia> log10(2)  # logaritmo na base "10"
0.3010299956639812

julia> max(1, 2, 4, 5)  # maior valor de uma lista de valores
5

julia> min(1, 2, 4, 5)  # menor valor de uma lista de valores
1

julia> prod(1:5)  # produto sucessivo de uma faixa de valores: cálculo de "1 x 2 x 3 x 4 x 5"
120

julia> prod(1:5; init=3)  # com multiplicador no argumento "init"
360

julia> sin(3)  # seno
0.1411200080598672

julia> sqrt(25)  # raiz quadrada
5.0

julia> sum(1:5)  # somatório de uma faixa de valores, neste caso de "1 a 5"
15

julia> sum(1:5; init=2)  # somatório com soma do valor de "init"
17

julia> iseven(4)  # indica se o valor numérico é par
true

julia> isodd(3)  # indica se o valor numérico é impar
true

julia> isequal(3, 4)  # indica se os valores são iguais
false
```

```
julia> tan(3)  # tangente
-0.1425465430742778

julia> abs(tan(3))  # ou "tan(3) |> abs", ou "3 |> tan |> abs", ou "abs(3 |> tan)"
0.1425465430742778

julia> (5.6 - pi) / abs(tan(Base.MathConstants.catalan))
1.8877004201279761

julia> trunc(2.5)  # extração do exponente de um número mantendo a mantissa em zero
2.0

julia> trunc(Int64, 2.5)  # extração da parte inteira de um número real
2

julia> trunc(Int64, 9 / 2)
4

julia> round(4.55)  # arredondamento matemático de valor
5.0

julia> round(4.50)  # arredondamento matemático de valor
4.0

julia> rationalize(9.3)  # mostra o valor decimal (real) como fração
93//10
```

Veja que nos exemplos apresentados não foram só utilizadas as funções para retornar resultados, foram usadas também funções com funções e valores numéricos com constantes e funções. Observe que não a limite para a definição de diversas combinações possíveis.

1.4 - Variáveis

As operações matemáticas apresentadas anteriormente resolvem uma parte do problema de cálculo computacional ao permitir que se use o ambiente da linguagem Julia como uma calculadora. No entanto, é importante a um programa de computador poder armazenar em memória valores a serem utilizados. O armazenamento de valores em memória se faz a partir do uso de *variáveis*.

A ideia de variável fundamenta-se na capacidade de ser um espaço de memória usado para armazenar certo valor por determinado espaço de tempo. É este espaço de memória um recurso utilizado para auxiliar a efetivação das ações de processamento, tanto matemático como lógico. A definição de variáveis em Julia se faz a partir da sintaxe:

```
<nome da variável> = <valor de inicialização>
```

O *nome da variável* nunca pode ser iniciado por um caractere numérico ou símbolo de interrogação, podendo-se usar caracteres maiúsculos e minúsculos acentuados, sendo preterido o uso de caracteres minúsculos, segundo as regras de codificação da linguagem. Quando houver a definição de nomes compostos de variáveis estes podem ser definidos arbitrariamente a partir das convenções: *snake case*[2], *camel case*[3] ou *pascal case*[4]. No entanto, as regras de boas práticas da linguagem ditam

[2] *Convenção "snake case" segue o estilo: data_de_nascimento.*

[3] *Convenção "camel case" segue o estilo: dataDeNascimento.*

que para nomes de variáveis e funções deve-se usar o estilo *snake case* e para definição de nomes de tipos e módulos usa-se o estilo *camel case*.

Entre a definição do nome e do valor associado a variável para sua inicialização em memória deve ser usado um espaço em branco antes e depois do símbolo de atribuição "**=**", apesar desta regra não ser eminentemente obrigatória. Neste sentido observe a sintaxe de definição de uma variável chamada "**a**" com valor "**2**" a partir da execução de "**a = 2**":

```
julia> a = 2
2
```

Lembre-se, como já comentado, que a linguagem Julia é do tipo *case-sensitive* e por esta razão variáveis definidas em letras maiúsculas e minúsculas não serão a mesma variável mesmo que tenham o mesmo nome. Por exemplo, uma variável definida com o nome "*XPTO*" será diferente de uma variável definida "*xpto*".

Um outro detalhe importante a ser considerado é que é possível definir variáveis baseadas em letras do alfabeto grego. Desta forma, veja a definição da variável chamada *beta* "**β**" com valor "**1**" a partir da execução de "**β = 1**":

```
julia> β = 1
1
```

Lembre-se de que para conseguir a letra beta escreve-se o código "**\beta**" e aciona-se em seguida a tecla "**<Tab>**" informando os demais caracteres.

A partir deste momento tem-se armazenado em memória os valores "**2**" em "**a**" e "**1**" em "**β**". Assim, é possível efetuar certo cálculo matemático. Por exemplo, execute a instrução "**a + β**" a seguir para que seja apresentado o resultado da soma entre as duas variáveis:

```
julia> a + β
3
```

A partir da definição de variáveis é possível realizar diversas operações matemáticas combinando-se variáveis, constantes e funções. Veja alguns exemplos:

```
julia> tan(3) / β
-0.1425465430742778

julia> (a + abs(tan(Base.MathConstants.catalan))) / a
1.6511645916367226

julia> a * 2
4

julia> a2
4
```

Entre os diversos exemplos fornecidos observe as duas últimas indicações onde é apresentado o resultado "**4**". Veja que na primeira ação é usada a operação "**a * 2**" e na segunda ação é usada a operação "**a2**". Perceba que Julia entende a operação "**a2**" como sendo a multiplicação da variável "**a**" com o valor numérico "**2**".

---

[4] *Convenção "pascal case" segue o estilo: DataDeNascimento.*

As variáveis em um programa podem ser usadas para a efetivação de dois tipos de operações: uma operação de ação e outra de controle. As variáveis de ação são usadas na definição de valores que apresentam alguma resposta de sua operação. Já as variáveis de controle são usadas para auxiliar o processamento de tomada de decisões e execução de laços (tema que será visto mais adiante nesta obra).

Além das variáveis definida o ambiente Julia possui uma variável interna chamada "**ans**". Essa variável tem por característica retornar sempre o último valor armazenado na memória. Por exemplo, execute a instrução seguinte:

```
julia> ans
4
```

Veja que foi retornado o último resultado operado na memória, ou seja, o valor "**4**", resultado da operação "**a2**".

Uma variável definida em memória mantém armazenado o último valor a ela associado. Veja a próxima sequência de instruções com a definição da variável "*XPTO*":

```
julia> xpto = 1
1

julia> xpto
1

julia> xpto = 3
3

julia> xpto
3
```

As variáveis definidas em memória podem ter seu conteúdo "apagado" a partir do uso de um recurso chamado "**nothing**". Veja a remoção do conteúdo da variável "*XPTO*" a partir da sintaxe:

```
julia> xpto = nothing
<não mostra nada>

julia> xpto
<não mostra nada>
```

O recurso "**nothing**" não apaga imediatamente o conteúdo de uma variável, mas marca-o para remoção deixando-o inacessível. A remoção ocorre quando o ambiente da linguagem executar automaticamente um recurso interno chamado *coletor de lixo* (*garbage collector*), mas caso queira executar manualmente o coletor de lixo basta no *prompt* do ambiente *REPL* executar a instrução: "**GC.gc()**".

Um detalhe importante: os conteúdos de uma variável podem ser alterados ou "apagados", mas uma vez definida a variável não pode ser removida da memória no modo interativo. No modo programado uma variável criada fica ativa enquanto o programa em uso permanece ativo na memória.

1.5 - Tipos de dados básicos (primitivos)

Julia é uma linguagem que processa os dados que manipula de forma dinâmica. Isto significa que não é necessário especificar para certo dado seu tipo explicitamente antes de seu uso. Assim que um dado é colocado em uso seu tipo é detectado automaticamente pela linguagem. Julia opera com uma estrutura hierárquica de tipos formada por subtipos e supertipos. Assim sendo, os tipos de dados são organizados como abstratos "**AbstractType**" e concretos "**ConcreteTypes**".

Um tipo de dado concreto é aquele que identifica isoladamente certo dado, como: caractere, cadeia, número de ponto flutuante, número inteiro, lógico, etc e não possui abaixo dele nenhuma instância de outro tipo. Já um tipo de dado abstrato possui instâncias de outros tipos. O tipo de dado abstrato tem como finalidade diferenciar possíveis dados concretos que sejam idênticos em diferentes instâncias de utilização. Desta forma, os tipos de dados concretos são subtipos naturais dos dados abstratos.

Considerando que a hierarquia de tipos seja semelhante a uma árvore, os dados abstratos são, em essência, os galhos que podem possuir como folhas dados concretos ou mesmo outros dados abstratos. A raiz dessa árvore é o supertipo denominado "**Any**" plantado em um solo chamando "**DataType**".

A detecção de um tipo de dado é obtida com o uso da função "**typeof()**" a partir da sintaxe:

```
typeof(<valor>)
```

Desta forma, para ver o tipo de dado que o supertipo "**Any**" possui, execute no *prompt* de comando do ambiente a instrução:

```
julia> typeof(Any)
DataType
```

O supertipo "**Any**" por sua vez, possui um conjunto de tipos que podem ser concretos ou abstratos fornecendo uma rede hierárquica de tipos. Caso ocorra, de determinado dado não possuir um tipo específico este será naturalmente do tipo "**Any**". Para o estudo desta obra são considerados os tipos abstratos "**AbstractChar**", "**AbstractString**" e "**Number**" do supertipo "**Any**".

Os tipos abstratos "**AbstractChar**" e "**AbstractString**" são recursos que estabelecem as bases para a manipulação de dados que operem respectivamente com elementos textuais definidos a partir de caracteres e cadeias. O termo caractere é aplicado quando se define isoladamente entre aspas simples um símbolo como letra, número ou pontuação e o termo cadeia é aplicado quando se define entre aspas inglesas um conjunto de caracteres. Tanto os caracteres como as cadeias têm por finalidade tratar dados que não são usados em ações de processamento matemático ou lógico. Desta forma, os dados dessas duas categorias podem ser alfabéticos quando usam letras ou símbolos de pontuação ou alfanuméricos quando fazem uso dos símbolos numéricos (desde que esses símbolos numéricos não tenham valor matemático).

O tipo abstrato "**AbstractChar**" possui o tipo concreto "**Char**".

O tipo abstrato "**AbstractString**" possui os tipos concretos "**String**", "**SubString**" e "**SubstitutionString**".

O tipo abstrato "**Number**" caracteriza-se por abranger um tipo concreto denominado "**Complex**" e um tipo abstrato chamado "**Real**", onde o tipo concreto "**Complex**" é destinado ao uso de números no formato complexo e o tipo abstrato "**Real**" que possui os tipos abstratos "**AbstractFloat**", "**AbstractIrrational**", "**Integer**" e "**Rational**".

O tipo abstrato "**AbstractFloat**" possui os tipos concretos "**BigFloat**", "**Float16**", "**Float32**" e "**Float64**".

O tipo abstrato "**AbstractIrrational**" possui o tipo concreto "**Irrational**".

O tipo abstrato "**Integer**" possui o tipo concreto "**Bool**" e os tipos abstratos "**Signed**" e "**Unsigned**".

O tipo abstrato "**Signed**" é formado pelos tipos concretos "**Int8**", "**Int16**", "**Int32**", "**Int64**", "**Int128**" e "**BigInt**".

O tipo abstrato "**Unsigned**" é formado pelos tipos concretos "**UInt8**", "**UInt16**", "**UInt32**", "**UInt64**" e "**UInt128**".

A verificação de pertencimento de um subtipo hierarquicamente superior é obtida com o uso da função "**supertypes()**" a partir da sintaxe:

```
supertype(<supertipo>)
```

Observe a seguir exemplos de apresentação do supertipo que determinado tipo possui:

```
julia> supertype(Number)
Any
```

```
julia> supertype(AbstractChar)
Any

julia> supertype(AbstractString)
Any
```

A verificação da existência de tipos concretos e abstratos relacionados a determinado tipo é realizada com o uso da função "**subtypes()**" a partir da sintaxe:

```
subtypse(<subtipo>)
```

Observe a seguir exemplos de indicação de tipos abstratos e concretos pertencentes ao tipo indicado:

```
julia> subtypes(AbstractChar)
1-element Vector{Any}:
 Char

julia> subtypes(Number)
2-element Vector{Any}:
 Complex
 Real

julia> subtypes(Real)
4-element Vector{Any}:
 AbstractFloat
 AbstractIrrational
 Integer
 Rational

julia> subtypes(Signed)
6-element Vector{Any}:
 BigInt
 Int128
 Int16
 Int32
 Int64
 Int8
```

Os tipos abstratos servem para organizar a estrutura de definição dos tipos concretos. No frigir dos ovos, o que importa diretamente são os dados concretos. Observe a seguir alguns exemplos de apresentação dos tipos de dados concretos relacionados e inferidos a partir de certos valores definidos:

```
julia> typeof('a')
Char

julia> typeof("abcde")
String

julia> typeof(1)
Int64
```

```
julia> typeof(123456789000000000000)
Int128

julia> typeof(123456789000000000000000000000000000000)
BigInt

julia> typeof(1.0)
Float64

julia> typeof(2//3)
Rational{Int64}

julia> typeof(true)
Bool

julia> typeof((Base.MathConstants).catalan)
Irrational{:catalan}
```

Veja que dentro do exposto não é necessário se preocupar com os tipos de dados em uso, uma vez que a linguagem os trata automaticamente. No entanto, conhecer os tipos de dados e sua abrangência de operação na memória permite ter noção efetiva dos valores numéricos em operação. Veja na tabela 1.4 os limites dos dados numéricos do tipo "**Integer**".

Os valores indicados nos tipos "**8**", "**16**", "**32**", "**64**" e "**128**" correspondem ao número de bits usados em memória para cada um dos dados definidos. Esses valores correspondem respectivamente a "**1**", "**2**", "**4**", "**8**" e "**16**" bytes de memória.

A detecção das faixas de valores para dados do tipo abstrato "**Integer**" podem ser operacionalizadas com o uso das funções "**typemin()**" e "**typemax()**" que respetivamente apresentam os valores mínimos e máximo da faixa de valores operacionalizados a partir do tipo de dado concreto inteiro indicado a partir da sintaxe:

```
typemin(<tipo concreto>)
typemax(<tipo concreto>)
```

Tipo	Abrangência
Int8	-128 a 127
Int16	-32768 a 32767
Int32	-2147483648 a 2147483647
Int64	-9223372036854775808 a 9223372036854775807
Int128	-170141183460469231731687303715884105728 a 170141183460469231731687303715884105727
UInt8	0 a 255
UInt16	0 a 65535
UInt32	0 a 4294967295
UInt64	0 a 18446744073709551615
UInt128	0 a 340282366920938463463374607431768211455
Bool	false ou true

Tabela 1.4 - Tipos de dados inteiros e sua abrangência em memória.

Observe a seguir exemplos de indicação de obtenção dos valores de limite de operação dos tipos concretos de dados inteiros:

```
julia> typemin(Int32)
-2147483648

julia> typemax(Int32)
2147483647
```

Em relação aos tipos de dados do tipo "**Real**" tem-se como faixa de abrangência em memória os tipos indicados junto a tabela 1.5, estabelecidos de acordo com a norma *IEEE-754*. As colunas com *sinal*, *expoente* e *mantissa* mostram a quantidade de bits usada para a representação numérica do valor de ponto flutuante.

Tipo	Abrangência	Sinal	Expoente	Mantissa
Float16	$\pm 6.10\times10^{-5}$ a $\pm 6.55\times10^{4}$	1	5	10
Float32	$\pm 1.18\times10^{-38}$ a $\pm 3.4\times10^{38}$	1	8	23
Float64	$\pm 2.23\times10^{-308}$ a $\pm 1.18\times10^{308}$	1	11	52

Tabela 1.5 - Tipos de dados reais e sua abrangência em memória.

Os tipos de dados "**BigInt**" e "**BigFloat**" existem na linguagem Julia para uso com valores que possuam precisão numérica arbitrária. Neste trabalho está temática não será tratada.

Além de verificar a hierarquia entre os tipos de dados é possível verificar se determinado tipo concreto ou abstrato pertence a certo tipo abstrato. Para tanto, pode-se fazer uso do operador "**<:**" que tem como resposta os valores "**true**" ou "**false**". Assim sendo, observe os exemplos seguintes:

```
julia> Float32 <: Real
true

julia> Float32 <: Integer
false
```

Apesar dos tipos de dados serem dinâmicos pode ocorrer o desejo ou necessidade de especificar para uma variável um determinado tipo de dados. Imagine que se queira definir uma variável chamada "**vlr**" com o valor "**1**" a ser armazenado como um dado do tipo "**Int8**", uma vez que qualquer valor inteiro é definido automaticamente como sendo do tipo "**Int64**". Neste caso, execute as seguintes instruções:

```
julia> vlr = Int8(1)
1

julia> typeof(vlr)
Int8
```

Veja que para alterar o tipo do dado foi usada a função "**Int8()**" junto a definição do valor na declaração da variável. Perceba que para cada tipo de dado concreto há uma função com mesmo nome para permitir o tratamento de conversão de um tipo a outro sobre determinado dado. No entanto, este recurso é disponibilizado para dados numéricos, uma vez que manipulação textual deve ser considerada a parte.

Rapidamente, para definir um valor numérico e convertê-lo para formato cadeia use a função "**string()**" ou a função "**repr()**", e para a ação inversa use a função "**parse()**". Observe a seguir um pequeno exemplo de conversão do valor inteiro "**11**" armazenado na variável "**xpto1**" para formato cadeia e a transformação da cadeia de valor "**321**" da variável "**xpto2**" em formato inteiro de 16 bits.

```
julia> xpto1 = repr(11)
"11"
```

```
julia> typeof(xpto1)
String

julia> xpto2 = parse(Int16, "321")
321

julia> typeof(xpto2)
Int16

julia> xpto3 = string(22)
"22"

julia> typeof(xpto3)
String

julia> xpto4 = parse(Float64, xpto3)
22.0

julia> typeof(xpto4)
Float64
```

Apesar de Julia conseguir automaticamente inferir o tipo de dado a partir de certo valor indicado é adequado indicar manualmente o uso de certo tipo, pois isto acelera a execução de um programa. Você verá em partes desta obra o uso de valores com a indicação de seus tipos de dados, e quando isso for feito será com a intenção de deixar o código mais rápido para sua compilação e posterior execução. A definição de tipos para uma variável será implementada no modo de uso programado da linguagem e não na forma interativa como visto no *prompt* REPL.

Informações detalhadas sobre os tipos de dados podem ser obtidas a partir da leitura pormenorizada do documento oficial da linguagem disponível em **https://docs.julialang.org/en/v1/manual/types/**.

1.6 - Pacotes externos

A linguagem Julia, assim como a linguagem Python, possui uma grande variedade de pacotes externos que podem ser instalados em seu ecossistema. São mais de 7.400 pacotes até maio de 2022. Esses pacotes são componentes que acrescentam recursos desenvolvidos por diversas pessoas, tornando-se uma grande biblioteca de funcionalidades.

Assim que a linguagem é instalada, nela são instalados os pacotes padrão, por exemplo, "**Base**" e outros podem ser instalados a qualquer momento a partir do repositório da linguagem. Além da instalação de novos pacotes é possível fazer a atualização dos pacotes instalados. Para instalar ou atualizar os pacotes existentes execute primeiramente a instrução:

```
julia> using Pkg
```

Desta forma o recurso "**Pkg**" é carregado na memória e a partir daí é possível atualizar e instalar pacotes. Antes de buscar novos pacotes é ideal atualizar os pacotes existentes. Para tanto, execute na linha de comando do ambiente de programação a instrução seguinte e aguarde sua conclusão. Isso pode demorar um pouco:

```
julia> Pkg.update()
```

Após a execução da instrução anterior um conjunto de informações são apresentados como indico a seguir:

```
  Installing known registries into `C:\Users\Augusto\.julia`
    Updating registry at `C:\Users\Augusto\.julia\registries\General.toml`
  No Changes to `C:\Users\Augusto\.julia\environments\v1.7\Project.toml`
  No Changes to `C:\Users\Augusto\.julia\environments\v1.7\Manifest.toml`
[ Info: We haven't cleaned this depot up for a bit, running Pkg.gc()...
      Active manifest files: 1 found
      Active artifact files: 0 found
      Active scratchspaces: 0 found
     Deleted no artifacts, repos, packages or scratchspaces
```

Assim que a atualização é realizada um conjunto de informações é apresentado. Pronto, a partir deste instante o ecossistema Julia encontra-se atualizado.

Agora para um teste de instalação considere instalar um pacote que permita realizar operações de criptografia de mensagens. Assim sendo, execute a instrução seguinte:

```
julia> Pkg.add("ClassicalCiphers")
```

Em seguida a execução da instrução anterior é apresentada uma série de mensagens indicado a ocorrência da instalação do pacote mencionado, podendo essa ocorrência ser diferente em seu computador, devido a pacotes que já tenham sido instalados:

```
| Resolving package versions...
| Installed ClassicalCiphers — v2.1.1
└ Updating `C:\Users\Augusto\.julia\environments\v1.7\Project.toml`
  [ecf26e93] + ClassicalCiphers v2.1.1
     Updating `C:\Users\Augusto\.julia\environments\v1.7\Manifest.toml`
  [ecf26e93] + ClassicalCiphers v2.1.1
  [56f22d72] + Artifacts
  [8f399da3] + Libdl
  [37e2e46d] + LinearAlgebra
  [9a3f8284] + Random
  [ea8e919c] + SHA
  [9e88b42a] + Serialization
  [2f01184e] + SparseArrays
  [10745b16] + Statistics
  [e66e0078] + CompilerSupportLibraries_jll
  [4536629a] + OpenBLAS_jll
  [8e850b90] + libblastrampoline_jll
Precompiling project...
  4 dependencies successfully precompiled in 2 seconds
```

A partir desse passo o pacote "**`ClassicalCiphers`**" encontra-se instalado, possuindo para uso os algoritmos: *Affine*, *Atbash*, *Caesar*, *Enigma (M3 Army)*, *Hill*, *Monoalphabetic substitution*, *Playfair*, *Portas*, *Rail Fence*, *Solitaire* e *Vigenère*. Para obter detalhes de cada algoritmo de criptografia do pacote "**`ClassicalCiphers`**" aconselha-se ler o documento do pacote que se encontra disponível em: **https://juliapackages.com/p/classicalciphers**.

Para saber os pacotes instalados execute a instrução:

```
julia> Pkg.status()
```

Para colocar o pacote em uso execute a instrução:

```
julia> using ClassicalCiphers
```

A título de teste, para cifrar a mensagem "**atacar base inimiga ao amanhecer**" a partir da palavra-chave "**bololo**" utilizando-se o algoritmo de Vigenère a seguir:

```
julia> encrypt_vigenere("atacar base inimiga ao amanhecer", "bololo")
BHLQLFCODSTBJATULOPOXOYVFQPF
```

Observe que a função "**encrypt_vigenere()**" retira da mensagem todos os espaços em branco e por padrão mostra o texto em caracteres maiúsculos (mensagens cifradas são por padrão apresentadas em maiúsculo). Para reverter, ou seja, para decifrar a mensagem execute a instrução:

```
julia> decrypt_vigenere("BHLQLFCODSTBJATULOPOXOYVFQPF", "bololo")
"atacarbaseinimigaaoamanhecer"
```

Ao ser revertida a cifra aplicada, o texto retorna sem espaços em branco em formato minúsculo (mensagens decifradas são por padrão apresentadas em minúsculo).

Veja que o uso de pacotes permite a você customizar a linguagem para suas necessidades e assim como pacotes são instalados também podem ser desinstalados. Para tanto, execute na linha de comando do ambiente a instrução:

```
julia> Pkg.rm("ClassicalCiphers")
```

No entanto, para se perceber a ação de remoção do pacote deve-se encerrar o ambiente e em seguida voltar com um novo acesso.

Dentro da temática de uso de pacotes há um pacote que possibilita visualizar a estrutura de árvore das definições dos tipos de dados na linguagem Julia para o tipo "**Number**". Este pacote chamasse "**AbstractTrees**". Para colocá-lo em uso, execute a instrução (TATE, 2020, p. 8):

```
julia> using AbstractTrees
| Package AbstractTrees not found, but a package named AbstractTrees is available from a
  registry.
 | Install package?
 |   (@v1.7) pkg> add AbstractTrees
 └ (y/n) [y]: y
   Updating registry at `C:\Users\Augusto\.julia\registries\General.toml`
  Resolving package versions...
  Installed AbstractTrees — v0.4.2
   Updating `C:\Users\Augusto\.julia\environments\v1.7\Project.toml`
 [1520ce14] + AbstractTrees v0.4.2
   Updating `C:\Users\Augusto\.julia\environments\v1.7\Manifest.toml`
 [1520ce14] + AbstractTrees v0.4.2
Precompiling project...
  1 dependency successfully precompiled in 3 seconds (119 already precompiled)
```

Na sequência execute a instrução apresenta, mas não se preocupe com os detalhes, apenas escreva a instrução:

```
julia> AbstractTrees.children(x::Type) = subtypes(x)
```

E depois execute a instrução:

```
julia> print_tree(Number)
Number
├─ Complex
└─ Real
   ├─ AbstractFloat
   │  ├─ BigFloat
   │  ├─ Float16
   │  ├─ Float32
   │  └─ Float64
   ├─ AbstractIrrational
   │  └─ Irrational
   ├─ Integer
   │  ├─ Bool
   │  ├─ Signed
   │  │  ├─ BigInt
   │  │  ├─ Int128
   │  │  ├─ Int16
   │  │  ├─ Int32
   │  │  ├─ Int64
   │  │  └─ Int8
   │  └─ Unsigned
   │     ├─ UInt128
   │     ├─ UInt16
   │     ├─ UInt32
   │     ├─ UInt64
   │     └─ UInt8
   └─ Rational
```

Os pacotes são recursos que dinamizam e aumentam a capacidade operacional da linguagem. Antes de resolver um problema é interessante verificar se já há algum pacote pronto para o problema. Não havendo, tem-se em mãos uma grande oportunidade de escrever e publicar um pacote quando se estiver mais craque na programação desta linguagem.

1.7 - Ambiente interativo

O ambiente interativo *REPL* além de executar as ações e comandos da linguagem possui uma série de recursos adicionais a serem conhecidos. Neste ambiente encontra-se integrado ao executável "**julia**" o *REPL* e um compilador. Este ambiente permite realizar a avaliação rápida de tudo que é escrito em sua linha de comando, fornece por meio das teclas de setas para cima e para baixo acesso ao histórico dos comandos utilizados no ambiente em todas as seções que foram abertas, mesmo após o uso da função de saída "**exit()**" que pode ser substituída pela combinação de teclas "**<Ctrl> + <D>**".

Além da combinação "**<Ctrl> + <D>**" há um extenso conjunto de combinação de teclas que dão suporte a três categorias de operação, sendo: controle do programa, movimento do cursor e edição, dos quais os mais comuns são indicados na tabela 1.6.

O ambiente *REPL* pode ser operado a partir de cinco *prompts* diferentes, a saber: *julia* (modo padrão), *help* (modo ajuda), *shell* (modo sistema), *package* (modo pacote) e *search* (modo pesquisa).

O modo padrão "**julia>**" é o modo mais comum e mais utilizado de todo o ecossistema Julia: é onde toda a “mágica” acontece, dispensando grandes comentários uma vez que foi o *prompt* mais usado até o presente momento.

O modo "**help**" pode ser ativado apenas quando o cursor está no início da linha do *prompt*. Este modo aparece após o uso da tecla "**<?>**". Assim que o modo estiver ativado como "**help?>**" o ambiente permitirá apresentar informação de auxílio para qualquer conteúdo inserido imediatamente após seu *prompt*. Para fazer um teste, entre por exemplo a palavra "**String**" e veja a apresentação da explicação deste termo. O modo ajuda é encerrado com o uso da tecla "**<Backspace>**" no início da linha de seu *prompt*.

Combinação	Descrição
Controle do programa	
<Ctrl> + <D>	Sai do ambiente (quando buffer estiver vazio).
<Ctrl> + <C>	Interrompe ou cancela um código de programa em execução.
<Ctrl> + <L>	Limpa a tela do ambiente.
<Ctrl> + <J>	Gera uma linha no prompt executando ação (o mesmo que a tecla <Enter>.
<Alt> + <Enter>	Alterna a tela em tamanho cheio ou padrão.
<Enter>	Gera uma linha no prompt executando ação.
<?>	Entra no modo de ajuda.
<;>	Entra no modo shell do prompt do sistema.
<Tab>	Sai do modo shell e volta ao prompt padrão.
Movimento do cursor	
<seta direita>	Move cursor para a direita um caractere na linha de texto.
<seta esquerda>	Move cursor para a esquerda um caractere na linha de texto.
<Ctrl> + <seta direita>	Move curso para o fim da palavra a direita.
<Ctrl> + <seta esquerda>	Move curso para o início da palavra a esquerda.
<Home>	Move cursor para início da linha.
<End>	Move cursor para o final da linha.
<seta para cima>	Move uma linha para cima no histórico de comandos.
<seta para baixo>	Move uma linha para baixo no histórico de comandos.
<Ctrl> + <F>	Faz o mesmo que a seta para a direita.
<Ctrl> + <B>	Faz o mesmo que a seta para a esquerda.
<Ctrl> + <A>	Faz o mesmo que a tecla <Home>.
<Ctrl> + <E>	Faz o mesmo que a tecla <End>.
<Ctrl> + <P>	Faz o mesmo que a tecla de seta para cima.
<Ctrl> + <N>	Faz o mesmo que a tecla de seta para baixo.
Edição	
<Backspace>	Remove da linha o caractere a esquerda.
<Delete> ou <Del>	Remove caractere sobre o cursor.
<Ctrl> + <_>	Desfaz as últimas alterações.
<Ctrl> + <H>	Faz o mesmo que a tecla <Backspace>.
<Ctrl> + <D>	Faz o mesmo que a tecla <Delete>.
<Ctrl> + <W>	Remove a palavra a esquerda do cursor.
<Ctrl> + <K>	Remove caracteres até o final da linha a partir e certa posição.
<Ctrl> + <Y>	Copia na posição o texto removido com <Ctrl> + <K>.
<Ctrl> + <T>	Alterna caracteres sobre o cursor.
<Ctrl> + <R>	Abre prompt especial de pesquisa para busca em sentido atrás.
<Ctrl> + <S>	Abre prompt especial de pesquisa para busca em sentido a frente.

Tabela 1.6 - Teclas de controle do ambiente REPL.

O modo "**`shell`**" pode ser ativado apenas quando o cursor está no início da linha do *prompt*. Este modo aparece após o uso da tecla "**`<;>`**". Assim que o modo estiver ativado como "**`shell>`**" é possível, por exemplo, executar comandos externos do sistema operacional, exceto no sistema Windows onde para que isso ocorra deve ser chamado um dos programas de *prompt* deste sistema: "*cmd*" ou "*poweshell*". Para sair do sistema e retornar ao *prompt* "**`julia>`**", execute no *prompt* do terminal em modo texto o comando "*exit*". O modo *shell* pode ser encerrado com o uso da tecla "**`<Backspace>`**" no início da linha de seu *prompt*.

O modo "**package**" pode ser ativado apenas quando o cursor está no início da linha do *prompt*. Este modo aparece após o uso da tecla "**<]>**", podendo ser encerrado com o uso das teclas "**<Ctrl> + <C>**" ou a tecla "**<Backspace>**" no início da linha de seu *prompt*. O modo de operação "**(@v1.7) pkg>**" possui um conjunto próprio de comandos e seu modo próprio de ajuda. Para detalhes operacionais deste modo consulte seu manual em: **https://julialang.github.io/Pkg.jl/v1/**.

O modo "**search**" pode ser ativado com o uso das teclas "**<Ctrl> + <R>**" / "**<Ctrl> + <S>**". A letra "**R**" indica ocorrência de busca para trás e o "**S**" indica a ocorrência de busca a frente. Este modo é operado com base nos registros do histórico das ações efetuadas em todo os demais modos de operação. Quando usada a combinação de teclas com a letra "**R**" será apresentado o *prompt* "**(reverse-i-search)`':**" e se usada a combinação de teclas com a letra "**S**" será apresentado o *prompt* " **(forward-i-search)`':**". Neste modo de operação na medida em que se escreve algo o texto informado é apresentado entre os símbolos "**`**" e "**'**" existentes no *prompt*. O resultado mais recente que corresponde à consulta será atualizado dinamicamente à direita dos dois pontos à medida que mais informações forem escritas. Para encontrar outros resultados use as combinações de teclas de consulta "**<Ctrl> + <R>**" / "**<Ctrl> + <S>**". Assim que localizar o que se deseja basta acionar a tecla "**<Enter>**" para que a linha escolhida seja apresentada no *prompt* "**julia>**", podendo ser então processada. Para sair do modo de pesquisa use das teclas "**<Ctrl> + <C>**".

Para obter mais detalhes sobre o *REPL* consulte **https://docs.julialang.org/en/v1/stdlib/REPL/**.

1.8 - Programa "Alo, mundo!" e a linha de comando

Agora que você possui uma visão inicial e geral de uso do ambiente interativo *REPL* Julia é hora de ver, de forma, simples o uso do modo programável da linguagem.

Para realizar um rápido teste considere escrever o famoso programa "**Alo, mundo!**". Para este feito é importante criar em seu diretório ou pasta de trabalho um arquivo de programa Julia contendo certo código de programa. Desta forma, abra o editor de texto padrão de seu sistema operacional e escreva a seguinte instrução:

```
println("Alo, mundo!")
```

Após escrever o código anterior, grave-o obrigatoriamente com a extensão "**.jl**". Como sugestão grave o programa com o nome "**alo.jl**". Veja que o código de programa usa um recurso chamado "**println()**" que efetua a apresentação de certo dado e movimenta o cursor após a escrita para a próxima linha.

Para ver o programa em execução não carregue para a memória o ambiente Julia. Faça o seguinte, abra a janela de linha de comando do modo texto de seu sistema operacional, posicione-se em seu local de gravação (pasta ou diretório do usuário) em seguida no *prompt* do sistema escreva a instrução (sem usar o indicativo "**$:>**"):

```
$:> julia alo.jl
```

Acione a tecla "**<Enter>**" e observe a apresentação da mensagem "**Alo, mundo!**" na linha seguinte a sua chamada. A figura 1.17 mostra o resultado apresentado junto ao *prompt* do sistema operacional Windows, sendo que este resultado não será muito diferente nos outros sistemas operacionais.

Veja então que a linguagem Julia possui dois modos de operação testados neste capítulo. O modo interativo onde comandos são informados dentro do ambiente *REPL* e o modo programado executado a partir de um código chamado na linha de comando. Perceba que aqui está sendo feita uma introdução ao acesso da linguagem Julia via linha de comando.

O programa executável "**julia**" possui uma série de recursos disponibilizados quando utilizado na linha de comando, pois certas opções de operação ficam disponíveis neste modo de ação. Veja exemplos de certas operações:

- Para apresentar a versão da linhagem execute a instrução:

  ```
  $:> julia -v
  ```

- Para ver o modo de ajuda da linha de comando execute a instrução:

  ```
  $:> julia –h
  ```

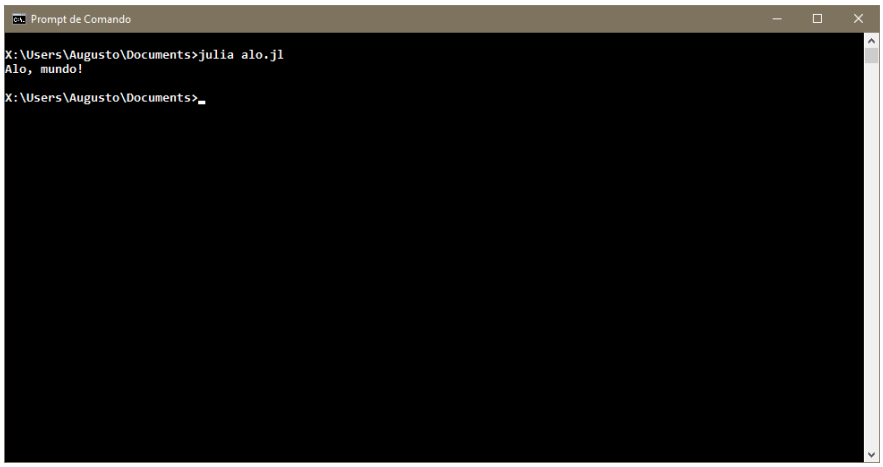

Figura 1.17 - Resultado da execução de um programa Julia.

- Para carregar o modo interativo em preto e branco execute a instrução:

```
$:> julia --color=no
```

As chaves de opções "**-v**" e "**-h**" podem ser substituídas respectivamente por "**-version**" e "**-help**".

Outro modo de executar um programa Julia é fazê-lo de dentro do ambiente interativo REPL. Neste caso, recomenda-se estar posicionado no local onde o programa esteja gravado e fazer a execução do ambiente seguindo a seguinte sintaxe:

```
julia> include("alo.jl")
```

A Figura 1.18 mostra o resultado da execução da instrução "**include("alo.jl")**".

Figura 1.18 - Execução indireta de um programa Julia.

A partir deste instante você tem em mãos o mínimo necessário para iniciar sua jornada na linguagem Julia. Observe que são necessários apenas a linguagem em si devidamente instalada e configurada, além de ter em mãos um editor de textos para produzir os códigos de programas.

1.9 - Palavras-chave

Além do uso de variáveis, constantes, funções e operadores, como os aritméticos, uma linguagem de programação possui um conjunto de comandos que visam ser a forma de comunicação para que sejam passadas ordens e instruções a um computador. Neste sentido, a linguagem Julia possui com os comandos as palavras-chave indicadas na tabela 1.7.

Os nomes dos comandos identificados na tabela 7.5 são palavras reservadas da linguagem. Nenhuma variável, constante, sub-rotina ou a definição de um tipo de dado poderá ser nomeada com qualquer uma dessas palavras.

Todo o conjunto de comunicação de ações usados em Julia, sejam: variáveis, constantes, funções ou operadores, além dos comandos (palavras-chave) são eminentemente expressões que quando em uso retornam algum tipo de resultado. Cada expressão caracteriza-se por ser a definição de uma linha de comunicação entre o ambiente da linguagem e seu usuário.

Palavras-chave (comandos)			
abstract	else	if	quote
baremodule	elseif	import	return
begin	end	in	struct
break	export	let	try
catch	finally	local	using
const	for	macro	where
continue	function	module	while
do	global	mutable	

Tabela 1.7 - Palavras-chave: linguagem Julia.

CAPÍTULO 2

PROGRAMAÇÃO ESTRUTURADA E MODULAR

Este capítulo mostra os detalhes gerais e preliminares a respeito dos paradigmas de programação estruturada (definição e utilização de laços while e for) e de programação modular (definição e utilização de sub-rotinas). Neste sentido, o capítulo apresenta: as ações de entrada e saída de dados; uso dos controles de fluxo; procedimentos; funções recursivas; linhas de comentários, detalhes sobre documentação e regras de escrita de código Julia.

2.1 - Entrada e saída de dados

Anteriormente foram apresentados alguns conceitos básicos e iniciais para que se possa fazer uso da linguagem Julia. No entanto, para que certas operações sejam bem-sucedidas é necessário controlar as ações de entrada e saída de dados. Para realizar a entrada de dados pode-se usar a função "**`readline`**", já para a saída usa-se os procedimentos "**`print`**" e/ou "**`println`**", que efetuam respectivamente a apresentação de certo conteúdo sem a movimentação do cursor para a linha seguinte e com a movimentação do cursor para a linha seguinte.

Para demonstrar os recursos de entrada e saída de dados, que podem ser operacionalizados dentro de um programa, será considerada uma situação em efetua-se a entrada de dois valores quaisquer representados pelas variáveis "*A*" e "*B*", efetuar a troca desses valores e apresentar a variável "*A*" com o valor da variável "*B*" e a variável "*B*" com o valor da variável "*A*". Grave o código do programa seguinte com o nome "**`c02ex01.jl`**".

```
1  # Entrada dos valores de qualquer tipo
2
3  print("Entre dado para variável <A>: ")
4  a = readline()
5
6  print("Entre dado numérico para variável <B>: ")
7  b = readline()
8
9  # Processamento da troca
10
11 x = a
12 a = b
13 b = x
14
15 # Saída com a troca efetivada
16
17 println("<A> = ", a)
18 println("<B> = ", b)
19
20 print("Tecle <Enter> para encerrar programa... ")
21 readline()
```

Observe que o programa "**c02ex01.jl**" caracteriza-se por ser um bloco de operações divididos em seguimentos determinados nas linhas "**1**", "**9**" e "**15**", que se caracterizam por serem linhas de comentários. Essas linhas não são executadas pelo ambiente Julia, servindo apenas de pontos de orientação do que o programa faz.

Nas linhas "**3**" e "**6**" estão sendo usados o procedimento "**print**" para a apresentação de uma mensagem ao usuário do programa, mantendo o cursor na mesma linha logo após a apresentação da mensagem.

Nas linhas "**4**" e "**5**" estão sendo usadas a função "**readline**" que efetua uma entrada de dados de forma indireta transferindo o conteúdo recebido a variável a sua esquerda.

Nas linhas "**11**", "**12**" e "**13**" encontra-se definido o bloco de processamento de troca de valores entre as variáveis "*A*" e "*B*" operacionalizadas com auxílio da variável "*X*".

Nas linhas "**17**" e "18" usa-se o procedimento "**print**" que apresenta a mensagem juntamente do valor de cada variável e salta uma linha em branco a frente.

Na linha "**20**" faz-se a apresentação de uma mensagem que orienta acionar "**<Enter>**" para encerar a execução do programa controlado na linha "**21**", onde a função de entrada "**readline**" é usada para fazer uma pausa e aguardar o acionamento da tecla "**<Enter>**". Atente que neste momento não se faz a atribuição da entrada na função a alguma variável.

O uso de "**print**" e "**println**" possibilita a aplicação de alguns efeitos de apresentação de dados, como o uso de códigos de *escape* com o caractere de contra barra "****", semelhante ao estilo encontrado na linguagem C como apresenta a tabela 2.1.

Escape	Ação
\\	Usa-se quando necessário apresentar o caractere: \.
\"	Usa-se quando necessário apresentar o caractere: ".
\'	Usa-se quando necessário apresentar o caractere: '.
\n	Usa-se quando necessário gerar uma nova linha na cadeia.
\r	Usa-se quando se deseja efeito <Enter> na cadeia a partir de certa posição.
\b	Usa-se quando se deseja fazer retorno de posição em certa cadeia.
\t	Usa-se quando se deseja gerar tabulação horizontal.
\a	Usa-se quando se deseja gerar um sinal de áudio de alerta.
\NNN	Usa-se quando se deseja apresentar valor numérico no formato octal.
\xNN	Usa-se quando se deseja apresentar valor numérico no formato hexadecimal.

Tabela 2.1 - Códigos de escape com contra barra.

Para realizar testes rápidos com o uso dos caracteres de *escape* abra o ambiente interativo Julia e execute as seguintes instruções:

```
julia> println("\\texto entre contra barras\\")
\texto entre contra barras\

julia> println("\"texto entre aspas inglesas\"")
"texto entre aspas inglesas"

julia> println("\'texto entre aspas simples\'")
'texto entre aspas simples'

julia> println("texto com quebra de linha \na partir daqui")
texto com quebra de linha
a partir daqui
```

```
julia> println("texto com efeito \r\"<Enter>\" a partir daqui")
"<Enter>" a partir daqui

julia> println("texto com retorno de caractere \ba partir daqui")
texto com retorno de caractere partir daqui

julia> println("texto com tabulação \ta partir daqui")
texto com tabulação     a partir daqui

julia> println("alerta: atenção\a")
alerta: atenção [toca alerta após apresentação da mensagem]

julia> Int('\012')
10

julia> Int('\x0A')
10
```

Observe que os testes realizados com os códigos de *escape* indicados na tabela 2.1 foram estabelecidos com o uso do procedimento "**`println`**", exceto os códigos de escape "**`\NNN`**" e "**`\xNN`**", em que as indicações da letra "**`N`**" representam valores numéricos e são podem ser usados apenas com a função "**`Int`**". Os códigos de *escape* "**`\NNN`**" e "**`\xNN`**" são específicos para tratamento numérico, enquanto os demais códigos de *escape* são específicos para tratamento de cadeias de caracteres.

O uso dos códigos de escape "**`\NNN`**" e "**`\xNN`**" abrem uma exceção de tratamento que requer um espaço de discussão a parte em relação a função "**`Int`**". A instrução "**`Int('\012')`**" efetua a conversão do valor numérico octal "**12**" em seu equivalente inteiro numérico decimal "**10**", assim como a instrução "**`Int('\x0A')`**" faz a conversão do valor numérico hexadecimal "**a**" em seu equivalente inteiro numérico decimal, neste caso, também "**10**". A função "**`Int`**" é útil para detectar, além da conversão de número octais/hexadecimais em decimal, o valor numérico do caractere na tabela Unicode. Veja alguns exemplos:

```
julia> Int('a')
97

julia> Int('A')
65

julia> Int('z')
122

julia> Int('π')
960
```

A operação inversa para detecção de um caractere a partir de seu valor inteiro de identificação é conseguida com o uso da função "**`Char`**". Observe alguns exemplos de uso:

```
julia> Char(122)
'z': ASCII/Unicode U+007A (category Ll: Letter, lowercase)

julia> Char(960)
'π': Unicode U+03C0 (category Ll: Letter, lowercase)
```

A saída da função "**Char**" apresenta o caractere indicando algumas características. O código "**122**" corresponde ao caractere "**z**" do tipo "**ASCII/Unicode**", em que na tabela *ASCII*[1] corresponde ao valor de identificação *112* e *U+007A* na tabela *Unicode*. Note que o caractere "**π**" possui apenas o código correspondente a tabela *Unicode* com código *U+03C0*, não existindo similar na tabela *ASCII*. Para apresentar apenas do caractere sem as indicações de características use os procedimentos "**print**" e "**println**". Veja os exemplos seguintes:

```
julia> print(Char(122))
z

julia> print(Char(960))
π
```

Há outro procedimento chamado "**printstyled**" que permite apresentar mensagens em cores e efeitos diferentes da forma padrão. Veja alguns exemplos de uso deste recurso no modo interativo da linguagem:

```
julia> printstyled("Mensagem\n"; color=:red)
Mensagem

julia> printstyled("Mensagem\n"; color=:red, underline=true)
Mensagem

julia> printstyled("Mensagem\n"; color=:red, reverse=true)
Mensagem
```

O procedimento "**printstyled**" não efetua salto de linha após a apresentação da mensagem. Por esta razão, se desejar este tipo de ação use o código de *escape* "**\n**" ao final da mensagem. A escolha de cores é realizada após a indicação do símbolo "**;**" assim que a mensagem é apresentada utilizando-se a palavra-chave "**color**" que pode ser definida a partir de uma constante com o nome da cor ou um número inteiro. A tabela 2.2 mostra a tabela de cores com as constantes disponíveis para uso.

Além das cores apresentadas na tabela 2.2 há ainda outras 240 cores, perfazendo um total de 256 cores.

Constante	Número	Cor
:black	0	Preto
:red	1	Vermelho escuro
:green	2	Verde escuro
:yellow	3	Amarelo escuro
:blue	4	Azul escuro
:magenta	5	Magenta escuro
:cyan	6	Ciano escuro
:white	7	Cinza claro
:light_black	8	Cinza escuro
:light_red	9	Vermelho claro
:light_green	10	Verde claro
:light_yellow	11	Amarelo claro
:light_blue	12	Azul claro
:light_magenta	13	Magenta claro
:light_cyan	14	Ciano claro
:white	15	Branco

Tabela 2.2 - Tabela de cores e códigos numéricos.

[1] American Standard Code for Information Interchange.

Observe o uso da cor de tom vermelho baseada no código numérico "**1**":

```
julia> printstyled("Mensagem\n"; color=1, underline=true)
Mensagem
```

Para visualizar o conjunto de cores considere o programa seguinte. Não se preocupe com instruções ainda não apresentadas e desconhecidas por você, apenas escreva e execute o código. Grave o código do programa seguinte com o nome "**c02ex02.jl**".

```
 1  # Apresentação de cores
 2
 3  println("Listagem de cores")
 4  println()
 5
 6  for cores = 0:255
 7      printstyled("$(cores) - Mensagem colorida\n"; color=cores)
 8  end
 9
10  print("Tecle <Enter> para encerrar programa... ")
11  readline()
```

Os detalhes indicados nas linhas "**5**" e "**11**" caracterizam por gerar na tela um efeito de salto de linha em branco melhorando a legibilidade do que é estrito.

Entre as linhas "**6**" e "**8**" encontra-se a estrutura de um laço, técnica que permite repetir um trecho de código um determinado número de vezes. Mais adiante neste capítulo será dado atenção a esta técnica.

Um detalhe que deve ser considerado é o uso do recurso "**$()**" que pode ser usado junto dos procedimentos "**print**" e "**println**". Este recurso possibilita definir internamente em uma cadeia o local onde certo conteúdo será apresentado. Veja este exemplo:

```
julia> a = 1
1

julia> b = 2
2

julia> println("Os valores $(a) e $(b) foram definidos externamente.")
Os valores 1 e 2 foram definidos externamente.
```

Há outro recurso para saída de dados que pode ser aplicado. Trata-se do procedimento "**show**" que assemelhasse ao procedimento "**println**". Para ver seu funcionamento no modo interativo informe as seguintes instruções:

```
julia> println("Mensagem")
Mensagem

julia> show("Mensagem")
"Mensagem"
```

Veja que a diferença entre os dois procedimentos é que a apresentação do procedimento "**show**" ocorre com o dado entre aspas inglesas. O procedimento "**show**" também aceita o uso do recurso "**$()**". Este procedimento quando escreve uma mensagem não movimenta o cursor para a próxima linha exigindo o uso do código de *escape* "**\n**".

A partir de tudo o que foi exposto até este momento considere um programa que efetue a leitura de dois valores numéricos e apresente os resultados da soma, subtração e multiplicação dos dois valores lidos. Grave o código do programa seguinte com o nome "**c02ex03.jl**".

```
1  printstyled("Cálculo de valores\n"; color=:light_yellow, reverse=true)
2  println()
3
4  printstyled("Entre o 1o. valor numérico: "; color=:light_cyan)
5  a = parse(Float64, readline())
6
7  printstyled("Entre o 2o. valor numérico: "; color=:light_cyan)
8  b = parse(Float64, readline())
9
10 r1 = a + b
11 r2 = a - b
12 r3 = a * b
13
14 println()
15 printstyled("Resultado soma ......: $(r1)\n"; color=:light_magenta)
16 printstyled("Resultado subtração .: $(r2)\n"; color=:light_magenta)
17 printstyled("Resultado divisão ...: $(r3)\n"; color=:light_magenta)
18
19 println()
20 print("Tecle <Enter> para encerrar programa... ")
21 readline()
```

O programa "**c02ex03.jl**" faz uso de recursos apresentados neste e no capítulo anterior. Atente principalmente para as linhas "**5**" e "**8**". Note o uso da função "**parse**" que converte a entrada realizada pela função "**readline**" em formato numérico do tipo "**Float64**". Esta conversão se faz necessária uma vez que a entrada de dados ocorre sempre com a função "**readline**" no formato cadeia (texto).

Note que é possível em Julia fazer a apresentação de dados vinculados a mensagens de duas formas diferentes: uma simples a partir do formato "**println("Mensagem ", dado)**" e outra com o uso do recurso "**$()**" a partir do formato "**println("Mensagem $(dado)")**".

2.2 - Tomada de decisão

Uma etapa importante na atividade de programação é realizar o direcionamento do fluxo de execução a partir da aplicação de regras condicionais nas ações de tomadas de decisão. A tomada de decisão pode ser simples, composta ou seletiva; pode ser operacionalizada a partir do uso de operadores relacionais e/ou operadores lógicos; além do controle do que pode ou não ser executado em um programa.

2.2.1 - Operadores relacionais

Os operadores relacionais são os mecanismos que permitem a definição de inequações em programação, ou seja, são responsáveis pela execução do processamento lógico a partir do estabelecimento de condições simples. Neste sentido, Julia apresenta um conjunto de operadores que atendem a diversas operações de cálculos lógicos, como apresenta-se na tabela 2.3 que apresenta os operadores principais encontrados na linguagem[2].

As condições estabelecidas com operadores relacionais resultam apenas um de dois resultados possíveis: "**true**" para a condição verdadeira e "**false**" para a condição falsa, podendo serem aplicadas em relações lógicas entre variáveis versus variáveis e entre variáveis versus constantes.

[2] *Há outros operadores que não estão sendo tratados: "===" e "!==".*

Operador	Operação	Operação
==	a == b	Igual a
>	a > b	Maior que
<	a < b	Menor que
>=	a >= b	Maior ou igual a
<=	a <= b	Menor ou igual a
!=	a != b	Não igual (diferente de)
:>	a <: b	É um subtipo de

Tabela 2.3 - Operadores relacionais suportados pela linguagem Julia.

Para confirmar a funcionalidade das relações lógicas observe os próximos exemplos com a indicação de relações estabelecidas entre variáveis versus variáveis e variáveis versus constantes:

```
julia> a = 1
1

julia> b = 2
2

julia> a == b
false

julia> a > b
false

julia> a < b
true

julia> a >= b
false

julia> a + 2 > b
true

julia> a + 1 > b
false

julia> a + 1 >= b
true

julia> a < b
true

julia> a != b
true

julia> a == 1
true
```

```
julia> a > 1
false

julia> a < 1
false

julia> a >= 1
true

julia> a <= 1
true

julia> a != 1
false
```

O operador de subtipo "**<:**" é um recurso que pode ser usado em duas circunstâncias: uma como operador relacional e, neste caso, retornará "**true**" ou "**false**" se na relação lógica "**a <: b**" obtém-se "**true**" se "*A*" é um subtipo de "*B*" e outra como um recurso de definição de subtipo de dado a partir de um tipo definido da linguagem. No primeiro capítulo ocorre o uso de um trecho de código com o uso da instrução "**if typeof(simbolo) <: Irrational**", onde verifica-se se o tipo do símbolo "**typeof(simbolo)**" é um subtipo do tipo de dado "**Irrational**".

2.2.2 - Operadores lógicos

Os operadores lógicos são os mecanismos complementares a definição de inequações em programação, ou seja, são responsáveis pela execução do processamento lógico a partir do estabelecimento de condições compostas, se estiver em uso os efeitos de conjunção e disjunção. Neste sentido, Julia apresenta um conjunto de operadores que atendem a diversas operações de cálculos lógicos, como apresenta-se na tabela 2.4.

Operador	Operação	Prioridade
!	Negação (not / não)	1
&&	Conjunção (and / e)	2
||	Disjunção (or / ou)	3

Tabela 2.4 - Operadores lógicos suportados pela linguagem Julia.

Do ponto de vista filosófico, conjunção é a relação lógica entre proposições (entende-se, na esfera da programação, o termo proposição como condição) que geram um resultado lógico verdadeiro quando todas as proposições forem verdadeiras. Observe na tabela 2.5 o uso do operador lógico de conjunção "**&&**" (**e**).

Do ponto de vista filosófico, lógica de disjunção inclusiva é a relação lógica entre proposições que geram resultado lógico verdadeiro quando pelo menos uma das proposições for verdadeira. Em seguida, observe a tabela verdade 2.6 para o operador de conjunção "**||**" (**ou**).

Condição 1	Condição 2	Resultado
verdadeira	verdadeira	verdadeiro
verdadeira	falsa	falso
falsa	verdadeira	falso
falsa	falsa	falso

Tabela 2.5 - Operador lógico de conjunção.

Condição 1	Condição 2	Resultado
verdadeira	verdadeira	verdadeiro
verdadeira	falsa	verdadeiro
falsa	verdadeira	verdadeiro
falsa	falsa	falso

Tabela 2.6 - Operador lógico de disjunção.

Do ponto de vista filosófico, negação é a rejeição ou a contradição do todo ou de parte desse todo. Pode ser a relação entre uma proposição "**p**" e sua negação "**não-p**". Se "**p**" for verdadeira, "**não-p**" é falsa, e se "**p**" for falsa, "**não-p**" é verdadeira. Em seguida observe a tabela verdade 2.7 para o operador de negação "**!**" (**não**).

Condição 2	Resultado
verdadeira	falso
falsa	verdadeiro

Tabela 2.7 - Operador lógico de negação.

As condições estabelecidas com operadores lógicos resultam apenas um de dois resultados possíveis: "**true**" para a condição verdadeira e "**false**" para a condição falsa, podendo serem aplicadas em relações lógicas entre mais de uma condição. Para confirmar a funcionalidade das relações lógicas observe os próximos exemplos com relações estabelecidas entre variáveis versus variáveis e variáveis versus constantes.

```
julia> a == 1 && b == 2
true

julia> a == 1 && b == 1
false

julia> a == 2 && b == 1
false

julia> a == 2 && b == 2
false

julia> a == 1 || b == 2
true

julia> a == 1 || b == 1
true

julia> a == 2 || b == 1
false

julia> a == 2 || b == 2
true

julia> !(a == 1)
false

julia> !(a == 1) || b == 2
true

julia> a == 1 && b == 2 || a > b
true

julia> a == 1 && b == 2 && a > b
false

julia> a == 1 || b > 2 && a > b
true

julia> (a == 1 || b > 2) && a > b
false
```

```
julia> a == 1 && typeof(a) <: Integer
true

julia> a == 1 && typeof(a) <: Float64
false

julia> a == 1 && typeof(a) <: Real
true

julia> 1 < 2 < 3  # Julia permite criar condições com mais de dois elementos por relação
true
```

A partir de uma visão básica de uso dos operadores lógicos, auxiliados pelo uso dos operadores relacionais toma-se eficiente conhecer o uso deste ferramental em ações de desvios condicionais e posteriormente na execução de laços para a repetição de blocos de ações operacionais determinado programa.

2.2.3 - Desvio condicional simples

A ação de desvio condicional com característica simples baseia-se na definição de certa condição que quando verdadeira permite a execução de certa ação e quando falsa desvia o fluxo da operação para fora do bloco de decisão. Para realizar ação de tomada de decisão com desvio condicional simples usa-se a instrução:

```
if condição
   [ação executada se a condição tiver resultado lógico verdadeiro]
end
[ação executada quando condição falsa ou após condição ser verdadeira]
```

Entre os comandos "**if**" e "**end**" ocorre a indicação das instruções subordinadas ao bloco adjacente. A indicação do componente "*condição*" deve ser substituído pela expressão lógica da condição a ser utilizada. Os trechos sinalizados entre colchetes são instruções a serem executadas pelo programa.

Como ilustração de tomada de decisão simples, considere programa que leia dois valores numéricos reais, processe a adição e mostre o resultado se este for maior que dez. Grave o programa seguinte com o nome "**c02ex04.jl**".

```
 1  print("Entre o 1o. valor numérico: ")
 2  a = parse(Float64, readline())
 3
 4  print("Entre o 2o. valor numérico: ")
 5  b = parse(Float64, readline())
 6
 7  r = a + b
 8
 9  if r > 10
10      println("Resultado: ", r)
11  end
12
13  println()
14  print("Tecle <Enter> para encerrar programa... ")
15  readline()
```

O programa "**c02ex04.jl**" ao ser executado somente apresentará a mensagem "**Resultado:**" com o valor de resposta da linha "**10**" somente se a condição da linha 9 for efetivamente verdadeira, caso a condição seja falsa o fluxo é desviado para a linha "**11**".

2.2.4 - Desvio condicional composto

A ação de desvio condicional com característica composta baseia-se na definição de certa condição que quando verdadeira permite a execução de certa ação e quando falsa faz a execução de uma outra ação prevista para esta situação. Para realizar ação de tomada de decisão com desvio condicional composto usa-se a instrução:

```
if condição
    [ação executada se a condição tiver resultado lógico verdadeiro]
else
    [ação executada se a condição tiver resultado lógico falso]
end
[ação executada após condição ser falsa ou ser verdadeira]
```

Como ilustração de tomada de decisão composta, considere programa que efetue a leitura de valor numérico inteiro entre 20 e 90. O programa deve apresentar a mensagem "*Valor está na faixa permitida*", caso o valor esteja na faixa, se não apresentar mensagem "*Valor fora da faixa permitida*". Grave o código do programa com o nome "**c02ex05.jl**".

```
 1  print("Entre valor numérico inteiro: ")
 2  n = parse(Int, readline())
 3
 4  if n >= 20 && n <= 90
 5      println("Valor está na faixa permitida")
 6  else
 7      println("Valor fora da faixa permitida")
 8  end
 9  println()
10  print("Tecle <Enter> para encerrar programa... ")
11  readline()
```

O programa "**c02ex05.jl**" ao ser executado apresentará a mensagem "*O número está na faixa de 20 a 90*" indicada na linha "**5**" caso a condição da linha "**4**" seja verdadeira e não sendo será apresentada a mensagem "*O número está fora da faixa de 20 a 90*" indicada na linha "**7**".

Um detalhe importante a ser considerado no código do programa "**c02ex05.jl**" é a definição do tipo "**Int**" como argumento da função "**parse**" na linha "**2**" para fazer a recepção de valores do tipo inteiro.

2.2.5 - Desvio condicional então-se

A linguagem Julia não possui um recurso que possibilite a execução da ação de desvio condicional seletivo como encontrado na linguagem C com o comando "**switch**". A ação que chega mais próxima a este tipo de operação é o uso do comando "**elseif**" dentro da estrutura de decisão comandos "**if**" e "**end**" como alternativa simplificada ao uso do comando "**else**". Para realizar ação de tomada de decisão com desvio condicional então se usa a instrução:

```
if condição1
    [ação executada se a condição1 tiver resultado lógico verdadeiro]
elseif condição2
    [ação executada se a condição2 tiver resultado lógico verdadeiro]
else
    [ação executada se a condição1 e condição2 forem falsas]
end
[ação executada após validação das condições]
```

Como ilustração de tomada de decisão então-se, considere programa que efetue a leitura de dois valores numéricos quaisquer "*A*" e "*B*" do tipo real de 16 bits e indique por mensagens: se "*A*" é maior que "*B*", se "*B*" é maior que "*A*" ou se são os dois valores iguais. Grave o código do programa seguinte com o nome "**c02ex06.jl**".

```
 1  print("Entre valor numérico para a variável <A>: ")
 2  a = parse(Float16, readline())
 3
 4  print("Entre valor numérico para a variável <B>: ")
 5  b = parse(Float16, readline())
 6
 7  if a > b
 8      println("O valor da variável <A> é maior que o da variável <B>.")
 9  elseif a < b
10      println("O valor da variável <B> é maior que o da variável <A>.")
11  else
12      println("Os valores das variáveis <A> e <B> são iguais.")
13  end
14
15  println()
16  print("Tecle <Enter> para encerrar programa... ")
17  readline()
```

Ao executar o programa faça-o pelo menos três vezes a fim de validar todas as ocorrências definidas. Observe detalhadamente o funcionamento dos desvios entre as linhas "**7**" e "**13**".

2.2.6 - Desvio com operador condicional

O operador condicional, também chamado por muitos como operador ternário, é formado pelos símbolos interrogação "**?**" e dois pontos "**:**", onde são definidas uma condição e as ações para o resultado lógico da condição quando verdadeiro ou quando falso (PRIYA & RANJEET, 2006, p. 58) a partir da seguinte instrução:

```
[condição] ? [ação para condição verdadeira] : [ação para condição falsa]
```

O operador condicional (ternário) "**?**" pode ser considerado como uma forma de definição de instrução "**if**" e "**else**" reduzida.

Para demonstrar o uso do operador condicional "**?**" considere um programa que efetue a leitura de dois valores numéricos inteiros e apresente como resultado o maior valor informado. Grave o código do programa seguinte com o nome "**c02ex07.jl**".

```
 1  print("Entre valor numérico para a variável <A>: ")
 2  a = parse(Int, readline())
 3
 4  print("Entre valor numérico para a variável <B>: ")
 5  b = parse(Int, readline())
 6
 7  r = a > b ? a : b
 8
 9  println("O maior valor informado é = $(r)")
10
11  println()
12  print("Tecle <Enter> para encerrar programa... ")
13  readline()
```

Ao executar o programa, experimente fornecer valores, como "*A*" com "**7**" e "*B*" com "**3**", ou "*A*" com "**2**" e "*B*" com "**8**". Experimente, também, fornecer valores iguais.

2.2.7 - Tratamento de exceções (introdução)

O tratamento de exceções visa controlar o fluxo dos programas no sentido de evitar diversos erros, em especial aqueles que ocorrem de forma inesperada e muitas vezes fora do escopo de controle do próprio programa. Este tipo de tratamento de erro permite capturar erros ocorridos durante a execução de um programa, erros esses que acabam ocasionando a interrupção do funcionamento adequado do programa em execução, e até a perda irreversível de dados que estejam armazenados temporariamente na memória principal. Para mais detalhes consulte o apêndice "**D**".

Existem erros cometidos por enganos, erros propositais, erros por partes ruins de um código escrito, entre outras tantas razões. Muitas vezes, erros podem ocorrer sem que o programador tenha consciência de que eles existem no código que ele escreveu. Normalmente esse efeito colateral acontece devido à falta de experiência do programador com a linguagem de programação em uso.

Seja qual for a razão, erros existem e geram um código daninho, que pode fracassar o mais belo projeto de desenvolvimento de software. Tentar eliminá-los ou neutralizá-los ao máximo é um dever a ser cumprido, no sentido de buscar e aumentar a qualidade do software entregue a um cliente.

Muitos dos erros ocorridos em um programa podem ser facilmente contornados com a aplicação de alguns cuidados e procedimentos essenciais, como, por exemplo, o uso de técnicas de desenvolvimento de algoritmos computacionais, desenvolvimento de diagramas de blocos ou diagramas de quadros, testes de mesa, testes de caixa-preta, testes de caixa branca, entre outros mecanismos existentes. Esse tipo de erro normalmente é lógico.

No entanto, existem diversos outros erros que podem ocorrer na execução de um programa, que fogem ao controle do programador. São erros muitas vezes imperceptíveis em um primeiro momento pelo desenvolvedor e que podem ocasionar efeitos colaterais indesejáveis, quando da execução do programa. Esse tipo de erro gera o que se chama de exceção.

Infelizmente, é impossível prever todas as situações de ocorrência de erros, mas é plenamente possível eliminar muitos dos riscos existentes ou que porventura venham a existir, com simples cuidados. Algumas ocorrências somente serão detectadas ao longo do tempo de existência do ciclo de vida do software em uso.

A exceção ocorre devido a algum tipo de erro gerado pelo programa, que inicialmente foge do conhecimento do desenvolvedor. Para o desenvolvedor tudo está aparentemente certo, sua lógica parece perfeita, mas o programa não aceita a implementação, e começa a retornar uma série de erros, que deixa a equipe de desenvolvimento de "cabelo em pé". Muitos desses erros podem ser tratados com exceção.

Para utilizar o tratamento de exceções em um programa escrito em linguagem de programação Julia, deve ser considerado o uso de blocos especiais de códigos, delimitados pelos comandos "**try**" (detectar) e "**catch**" (capturar). O comando "**try**" estabelece o bloco de código que tem por finalidade cercar a área do programa em que um erro de exceção pode ocorrer e o bloco delimitado pelo comando "**catch**" tem a finalidade de tratar a exceção ocorrida a partir da estrutura sintática:

```
try
    [bloco de código a ser tratado para detecção eventual de erro]
catch [variável_opcional]
    [informação do erro ocorrido]
end
```

O trabalho de tratamento de erros não é uma tarefa muito fácil, pois é complicado saber onde um erro pode ocorrer. É necessário desenvolver com o tempo a sensibilidade necessária para perceber esses detalhes. Para demonstrar um erro típico, considere um simples programa que tenha a finalidade de apresentar o resultado armazenado em memória da raiz quadrada de um valor numérico inteiro. O programa deve interceptar a tentativa de entrada de valor negativo apresentando uma mensagem informando esta ocorrência além de indicar o erro de exceção interno. Grave o código do programa seguinte com o nome "**c02ex08.jl**".

```
1  print("Informe o valor numérico inteiro: ")
2  n = parse(Int, readline())
3
4  try
5      r = sqrt(n)
6      println("Resultado raiz quadrada: ", r)
7  catch erro_detectado
8      println("Valor numérico fornecido inválido: motivo valor negativo")
9      println("Erro ocorrido: ", erro_detectado)
10 end
11
12 println()
13 print("Tecle <Enter> para encerrar programa... ")
14 readline()
```

Ao ser executado o programa, faça testes com entradas de valores positivos e negativos e perceba que dentro do comando "**try**" é tratada a exceção de detecção de valores negativos. Se um valor fornecido for negativo o erro ocorrido dentro do bloco "**try**" é desviado para o bloco "**catch**" que é tratado evitando a interrupção inesperada do programa.

2.3 - Laços para repetições de blocos

É comum que trechos de programas tenham que ser repetidos por algumas razões. Essas repetições podem ocorrer de duas formas básicas: a primeira de forma iterativa (quando se conhece o número de vezes que a quantidade de repetições deve ser realizada) e a segunda de forma interativa (quando não se sabe de antemão a quantidade de vezes que as repetições devem ocorrer). Para atender a este quesito a linguagem Julia possui dois comandos, sendo um condicional e outro incondicional.

2.3.1 - Laço condicional

O laço condicional existente em Julia caracteriza-se por ser do tipo pré-teste com fluxo verdadeiro, sendo representado pelo comando "**while**" a partir da seguinte estrutura sintática:

```
while condição
    [bloco de código a ser repetido enquanto condição for verdadeira]
end
```

O laço condicional "**while**" é uma estrutura de repetição que pode ser usada para a realização de repetições iterativas e interativas.

Como exemplo de uso de laço condicional iterativo crescente considere o seguinte programa que ter por finalidade efetuar a apresentação em tela dos valores de "**1**" a "**5**" de "**1**" em "**1**". Grave o código do programa seguinte com o nome "**c02ex09.jl**".

```
1  i = 1
2  while i <= 5
3      println(i)
4      global i = i + 1
5  end
6
7  println()
8  print("Tecle <Enter> para encerrar programa... ")
9  readline()
```

Observe que o código do programa "**c02ex09.jl**" apresenta um novo detalhe na linha 4 referente ao uso do comando "**global**". O comando "**global**" indica que a variável "/" declarada na linha "1" tem escopo de ação global também

entre as linhas de "**2**" até "**5**", ou seja, é reconhecida como sendo a mesma variável. Se o comando "**global**" não for usado na linha "**4**" a variável "*i*" terá escopo local entre as linhas "**2**" até "**5**", não tendo nenhuma relação com a declaração da linha "**1**", pois a definição na linha "**1**" e sua ação de incremento na linha "**4**" não são a mesma variável, caso seja omitido o comando "**global**" criando um efeito de ambiguidade na memória. Veja que ao ser executado o programa são apresentados os valores de "**1**" (definido na linha "**1**") a "**5**" (definido na condição da linha "**2**") de "**1**" em "**1**" (definido na linha "**4**").

A operação de incremento escrita como "**global i = i + 1**" pode ser definida utilizando-se um efeito de operadores de acumulação. Desta forma, esta linha pode ser codificada como "**global i += 1**". O operador de atualização "**+=**" efetua ação de soma cumulativa, podendo-se usar outras combinações de acumuladores como: "**-=**", "***=**", "**/=**", "**%=**" e "**^=**" respectivamente para as operações cumulativas de subtração, multiplicação, divisão, resto e exponenciação.

Como exemplo de uso de laço condicional iterativo decrescente considere o seguinte programa que ter por finalidade efetuar a apresentação em tela dos valores de "**5**" a "**1**" de "**1**" em "**1**". Grave o código do programa seguinte com o nome "**c02ex10.jl**".

```
1  i = 5
2  while i >= 1
3      println(i)
4      global i -= 1
5  end
6
7  println()
8  print("Tecle <Enter> para encerrar programa... ")
9  readline()
```

Veja que na linha 4 encontra-se o uso da operação "**global i -= 1**" que é uma maneira de realizar a operação similar de decremento "**global i = i - 1**" para diminuir a contagem de "**1**" em "**1**" a partir de "**5**".

Outro detalhe neste programa é a definição da condição "**i >= 1**" na linha "**2**". Observe que para usar laço crescente iterativo usa-se o operador relacional "**<=**" e quando decrescente usa-se o operador relacional "**>=**".

Na sequência veja exemplo de execução de laço condicional interativo de um programa que apresente valores crescentes a partir de "**1**" até que uma resposta não seja informada a pergunta de continuidade. Grave o código do programa seguinte com o nome "**c02ex11.jl**".

```
1  i = 1
2  resp = "S"
3  while resp == "S" || resp == "SIM"
4      println(i)
5      global i += 1
6      println("Deseja nova sequência? ")
7      print("<S> para SIM, qualquer letra para não: ")
8      global resp = uppercase(readline())
9  end
10
11 println()
12 print("Tecle <Enter> para encerrar programa... ")
13 readline()
```

Na linha 2 define-se a variável "*RESP*" iniciada com o valor "**S**" para validar a entrada no laço indicado na linha "**3**" e apresentar o valor da variável "*i*" atual. A linha "**8**" aguarda a entrada de um valor de continuidade ou não do laço, caracterizando a ação de laço interativo. Na linha "**3**" usa-se o operador lógico de disjunção para validar a entra da resposta de continuidade como "**S**" ou "**SIM**",

2.3.2 - Laço incondicional

O laço incondicional existente em Julia caracteriza-se por ser uma ferramenta puramente iterativa, sendo representado a partir de duas formas de expressão.

A primeira forma de uso do laço "**for**" é para efetuar a contagem iterativa de um valor inicial, até um valor final, o qual pode seguir uma evolução uma a um ou a evolução determinada a partir de certo incremento. A forma sintática de uso desta forma segue a seguinte sintaxe:

```
for <variável> =/in <valor inicial> : <valor de incremento> : <valor final>
    [bloco de código a ser repetido a quantidade de vezes definida]
end
```

A segunda forma de uso do laço "**for**" é para efetuar a iteração sobre um conjunto de dados, em que a variável implicada com o comando "**in**" é iterada uma a uma sobre cada elemento do conjunto. No entanto, esta forma também pode fazer iterações como a forma anterior. A forma sintática de uso desta forma segue a seguinte sintaxe:

```
for <variável> =/in <[conjunto de valores]>
    [bloco de código a ser repetido a quantidade de vezes definida no conjunto]
end
```

O laço "**for**" é usado para a realização de progressões, sendo que os símbolos "**=**" e "**in**" possuem o mesmo significado. No entanto, se utiliza o símbolo "**=**" em situações de contagem direta e o símbolo "**in**" em situações de iteração de coleção de valores na forma de conjuntos. Detalhes sobre o uso de conjuntos será apresentada mais adiante.

O programa a seguir apresenta os valores crescentes de "**1**" a "**5**" de "**1**" em "**1**" em tela. Grave o código do programa seguinte com o nome "**c02ex12.jl**".

```
1  for i = 1:5
2      println(i)
3  end
4
5  println()
6  print("Tecle <Enter> para encerrar programa... ")
7  readline()
```

A estrutura do laço definida na linha "**1**" pode também ser escrita como "**for i = 1:5**". Esta estrutura itera na variável "**i**" valores de "**1**" a "**5**" de "**1**" em "**1**". Veja que o valor de incremento, quando de um em um é omitido, mas a mesma linha poderia estar escrita como "**for i = 1:1:5**".

Veja no próximo exemplo um programa que a partir do laço "**for**" efetue a apresentação decrescente de valores de "**5**" a "**1**" de "**1**" em "**1**". Grave o código do programa seguinte com o nome "**c02ex13.jl**".

```
1  for i = 5:-1:1
2      println(i)
3  end
4
5  println()
6  print("Tecle <Enter> para encerrar programa... ")
7  readline()
```

Veja na linha "**1**" a definição do valor de decremento "**-1**", obrigatoriamente definido entre os valores "**5**" e "**1**".

Considere como mais um exemplo de iteração de contagem uma situação em que sejam apresentados os valores de "**1**" a "**5**" de "**2**" em "**2**". Grave o código do programa seguinte com o nome "**c02ex14.jl**".

```
1  for i = 1:2:5
2      println(i)
3  end
4
5  println()
6  print("Tecle <Enter> para encerrar programa... ")
7  readline()
```

Ao ser executado o programa são apresentados os valores "**1**", "**3**" e "**5**" a partir da definição da contagem na linha "**1**".

A título de ilustração veja o próximo exemplo que determina a iteração de uma variável sobre os elementos de um conjunto de dados na forma de lista. Grave o código do programa seguinte com o nome "**c02ex15.jl**".

```
1  for i in [1, 2, 5.5, 7.9, "eu", "Julia"]
2      println(i)
3  end
4
5  println()
6  print("Tecle <Enter> para encerrar programa... ")
7  readline()
```

Ao ser executado o programa observe a apresentação de cada elemento constante no conjunto de dados não importando seu tipo. Veja que na linha "**1**" a variável "*i*" percorre cada elemento do conjunto capturando-o para em seguida ser apresentado.

Apesar de ser possível o uso do operador "**=**" junto ao comando "**for**" como demonstrado este deve ser evitado e em seu lugar deve-se usar o operador "**in**" de acordo com as recomendações das boas práticas de programação em Julia apresentadas por sua comunidade (**https://github.com/invenia/BlueStyle**).

2.3.3 - Interrupção de laço

A execução de um laço pode ser interrompida durante sua evolução. Essa operação pode ser realizada com o comando "**break**" ou com o comando "**continue**". O comando "**break**" faz sua interrupção de forma abrupta encerrando compulsoriamente a execução do laço. Já o comando "**continue**" faz uma pausa momentânea na execução do laço em processamento.

Como teste operacional considere um programa que por meio de um laço iterativo que efetua a contagem de "**1**" a "**5**" de "**1**" em "**1**" interrompa de forma abrupta com o comando "**break**" o laço de forma abrupta quando o contador estiver com valor igual a "**3**". Grave o código do programa seguinte com o nome "**c02ex16.jl**".

```
1  i = 1
2  while i <= 5
3      println(i)
4      if i == 3
5          break
6      end
7      global i = i + 1
8  end
9
10 println()
11 print("Tecle <Enter> para encerrar programa... ")
12 readline()
```

Veja que assim que o contador chega a "**3**" o programa é encerrado sem que a contagem continue em andamento. Desta forma, são apresentados os valores "**1**", "**2**" e "**3**".

Agora veja um programa similar que efetua, neste caso, a interrupção de um laço "**for**". Grave o código do programa seguinte com o nome "**c02ex17.jl**".

```
1  for i in 1:5
2      println(i)
3      if i == 3
4          break
5      end
6  end
7
8  println()
9  print("Tecle <Enter> para encerrar programa... ")
10 readline()
```

Ao ser executado a saída do programa "**c02ex17.jl**" será idêntica ao programa "**c02ex16.jl**".

Para um teste de uso do comando "**continue**" considere um programa que apresente valores de "**1**" a "**5**", exceto o valor "**3**". Grave o código do programa seguinte com o nome "**c02ex18.jl**".

```
1  for i in 1:5
2      if i == 3
3          continue
4      end
5      println(i)
6  end
7
8  println()
9  print("Tecle <Enter> para encerrar programa... ")
10 readline()
```

Ao ser executado o programa todos os valores são apresentados exceto o valor "**3**". Veja que o comando "**continue**" efetua uma pausa temporária (momentânea) na operação e dá sequência ao restante da operação diferente do comando "**break**" que interrompe e encerra a operação. O comando "**continue**" pode ser usado tanto com laço "**for**" como com o laço "**while**".

2.3.4 - Desvio incondicional

A ação de desvio incondicional é muito criticada por diversos profissionais da área de desenvolvimento de software. No entanto, se uma linguagem de programação possui tal recurso, isso não é à toa, é pelo fato de existirem situações onde esse recurso é necessário e conveniente de uso.

Basicamente a linguagem Julia não possui comandos para esta operação, em seu lugar possui como recurso o chamado macros (existem um conjunto de macros em Julia). Um macro caracteriza-se por um nome de identificação que executa certa ação como um comando, sendo precedido do símbolo arroba "**@**".

Para o efeito de uso de desvio incondicional considere os macros "**@goto**" e "**@label**" que possuem como estrutura sintática a forma escrita:

```
@goto <nome>
@label <nome>
```

A macro "**@goto**" tem por finalidade desviar o fluxo de processamento de um programa para o ponto que se encontra sinalizado com a macro "**@label**". Esses macros para funcionarem necessitam estar definidas na mesma área de escopo de programa. Uma forma de garantir que elas operem dentro do mesmo escopo é delimitá-las entre os comandos "**begin**" e "**end**" a partir da sintaxe:

```
begin
    [definição de um bloco de escopo]
end
```

O problema com o uso de macros "**@goto**" e "**@label**" é o programador fazer uso de forma inadequada. Existem algumas situações em que se justifica seu uso. Aqui são apresentadas duas situações onde o uso deste efeito é adequado.

A primeira situação é fazer a simulação em Julia de um laço condicional do tipo pós-teste com fluxo falso semelhante ao comando "**repeat/until**" existente na linguagem Pascal. Para tanto, considere um programa que aceite a entrada apenas de valor numérico inteiro positivo impar que seja maior que zero e apresente como saída o resultado do quadrado desse valor sem armazená-lo em memória. Grave o código do programa seguinte com o nome "**c02ex19.jl**".

```
 1  println("Validação de entrada - simulação \"repeat\".")
 2  println()
 3
 4  begin
 5      @label repita
 6          print("Informe um número positivo inteiro ímpar: ")
 7          try
 8              global n = parse(Int, readline())
 9              if iseven(n) == false && n > 0
10                  @goto fim_repita
11              end
12              @goto repita
13          catch
14              @goto repita
15          end
16      @label fim_repita
17  end
18
19  println()
20  println("Quadrado da entrada: ", n ^ 2)
21
22  println()
23  print("Tecle <Enter> para encerrar programa... ")
24  readline()
```

A parte crucial do programa "**c02ex19.jl**" situa-se entre as linhas "**4**" e "**17**", as quais estabelecem uma área de escopo de operação para o programa delimitada pelos comandos "**begin**" e "**end**".

A linha "**5**" estabelece a simulação para repetição do laço a partir da instrução "**@label repita**", a qual é finalizada na linha "**16**" com a instrução "**@label fim_repita**".

A primeira ação de validação do programa é garantir que aceite a entrada apenas de valor positivo. Isso é realizado entre as linhas "**7**" e "**15**". Se um valor de ponto flutuante for informado o programa desconsidera as linhas de "**9**" a "**12**" e desvia o fluxo para a linha "**5**" solicitando a entrada novamente.

Caso seja informado um valor inteiro este é validado entre as linhas "**9**" e "**12**". Nesse trecho a linha "**9**" verifica se algum valor numérico incorreto é fornecido, ou seja, se o valor fornecido for par e maior que zero faz o desvio para a linha "**5**". A validação de valor par é realizada pela função "**iseven**" que retorna como resposta valores "**true**" ou "**false**".

Como segundo exemplo de uma segunda situação com uso dos macros "**@goto**" e "**@label**" é considerar um programa que mostre o terno pitagórico, lembrando que termo pitagórico é formado por três números naturais a, b e c, de modo que $a^2 + b^2 = c^2$. Grave o código do programa seguinte com o nome "**c02ex20.jl**".

```
1  println("Terno pitagórico.")
2  println()
3
4  begin
5      for c in 1:10
6          for b in 1:10
7              for a in 1:10
8                  if a ^ 2 + b ^ 2 == c ^ 2
9                      println("Termo pitagórico encontrado: $(a), $(b) e $(c).")
10                     @goto fim
11                 end
12             end
13         end
14     end
15     @label fim
16 end
17
18 println()
19 print("Tecle <Enter> para encerrar programa... ")
20 readline()
```

Veja que no programa "**c02ex20.jl**" é necessário fazer o encerramento uma única vez de três laços encadeados quando o termo pitagórico é encontrado. Se você substituir na linha "**10**" a instrução "**@goto fim**" por "**break**" verá uma ocorrência desastrosa.

2.3.5 - Formatações

Até este momento foram utilizadas a apresentação de dados da forma mais simples que a linguagem suporta. Os eventos de saída foram operacionalizados com as funções "**print/println**" e "**show**", mas é importante ressaltar que essas funções não oferecem uma forma de saída que seja formatada.

Para realizar formatações na apresentação de dados é necessário fazer uso do macro "**@printf**" que disponibiliza um conjunto de códigos de formatação que melhoram a apresentação visual de dados com a linguagem assemelhando-se a função "**printf**" encontrada na linguagem C. No entanto, para fazer uso desse recurso é necessário colocar em uso o pacote "**Printf**".

A estrutura sintática básica do macro "**@printf**" é obtida a partir da forma escrita:

```
@printf("%formato", argumentos, ...)
```

Onde, *argumentos* indica a relação dos dados a serem apresentados que devem ser especificados separados por vírgula e *formato* indicado após o caractere "**%**" o código de formatação relacionado ao tipo de dado a ser apresentado na posição em que o código é especificado entre aspas inglesas. Veja alguns exemplos interativos a seguir.

```
julia> using Printf

julia> @printf("Alo, %s", "Mundo!")
Alo, Mundo!

julia> @printf("%s, %s", "Alo", "Mundo!")
Alo, Mundo!

julia> @printf("Valor numérico inteiro com deslocamento de 5 posições: %5i\n", 987)
Valor numérico inteiro com deslocamento de 5 posições:   987
```

```
julia> @printf("Alo, %10s", "Mundo!")
Alo,     Mundo!

julia> @printf("Valor numérico decimal com duas casas: %.2f\n", 987.654321)
Valor numérico decimal com duas casas: 987.65

julia> @printf("Valor em notação científica: %e\n", 987.654321)
Valor em notação científica: 9.876543e+02

julia> @printf("Valor em notação científica com duas casas decimais: %.2e\n", 987.654321)
Valor em notação científica com duas casas decimais: 9.88e+02

julia> @printf("Valor numérico inteiro com zeros a frente: %05i\n", 987)
Valor numérico inteiro com zeros a frente: 00987

julia> @printf("Valor numérico decimal com formato 7.2 posições: %7.2f\n", 987.654321)
Valor numérico decimal com formato 5.2 posições:  987.65

julia> @printf("Valor numérico decimal ou científico: %g e %g\n", 9.87, 98700000.0)
Valor numérico decimal ou científico: 9.87 e 9.87e+07

julia> @printf("Os caracteres são: \'%c\', \'%c\' e \'%c\'\n", 'A', 'B', 'C')
Os caracteres são: 'A', 'B' e 'C'

julia> @printf("Número hexadecimal em ponto flutuante: %a\n", 0xff)
Número hexadecimal em ponto flutuante: 0x1.fep+7

julia> @printf("Número decimal como inteiro: %d\n", 987.654321)
Número decimal como inteiro: 988

julia> @printf("Número notação científica como inteiro: %d\n", 2e10)
Número notação científica como inteiro: 20000000000

julia> @printf("Base 10 em base 16 como ponteiro: %p\n", 10)
Base 10 em base 16 como ponteiro: 0x000000000000000a

julia> @printf("Número inteiro em octal: %o\n", 10)
Número inteiro em octal: 12

julia> @printf("Número inteiro em hexadecimal: %x\n", 10)
Número inteiro em hexadecimal: a
```

A partir de uma visão geral dos códigos de formatação para cada tipo de dados fica mais fácil demonstrar seu efeito em um contexto de programa. Desta forma, considere um programa que apresenta a tabuada de um número inteiro qualquer positivo situado na faixa numérica entre os valores de entrada "**1**" e "**10**". Grave o código do programa seguinte com o nome "**c02ex21.jl**".

```
 1  using Printf
 2
 3  println("TABUADA")
 4  println()
 5
 6  while true
 7      print("Informe número positivo: ")
 8      global n = parse(Int, readline())
 9      if n >= 1 && n <= 10
10          break
11      end
12  end
13
14  println()
15  i = 1
16  while i <= 10
17      @printf("%2i X %2i = %3i\n", n, i, n * i)
18      global i += 1
19  end
20
21  println()
22  print("Tecle <Enter> para encerrar programa... ")
23  readline()
```

Observe que na linha "**1**" é colocada como primeira ação a chamada do pacote "**Printf**" para que o programa tenha suporte a operação do macro "**@printf**" definida na linha 24.

Entre as linhas "**6**" e "**12**" está sendo realizado um laço que valida a entrada do usuário, permitindo a execução do processamento se um valor condizente com o esperado for informado. Se o valor fornecido estiver fora do esperado o laço permanece em continua execução. Esse trecho é uma forma alternativa de validação para evitar o uso do macro de desvio incondicional "**@goto**" em uso no programa "**c02ex19.jl**".

2.4 - Sub-rotinas

Sub-rotinas podem ser chamadas de subprogramas ou módulos. Daí vem o nome *programação modular* que permite o uso desta técnica para duas situações: primeira para organizar um programa em módulos e segundo para colocar em prática a ideia de reaproveitamento de código de maneira simplificada.

Dentro do contexto da programação modular são duas as formas de se expressar sub-rotinas: uma como procedimento e outra como função. Um procedimento é uma estrutura utilizada, normalmente, nas ações de entrada e saída, eventualmente em ações de processamento. No entanto, funções são usadas para a efetivação de processamento, pois sempre retornam um resultado em seu nome após sua execução.

A linguagem Julia possui em seu ecossistema apenas o recurso *function* não operando de forma explicita com o conceito de *procedure*. Neste caso, como quem não tem cão caça como gato: usa-se a definição de função sem considerar o retorno de valor para simular a definição de um procedimento.

Um detalhe importante sobre Julia é que a definição de sub-rotinas e seu uso seguem o estilo *buttom-up*, ou seja, define-se primeiro a sub-rotina antes de seu uso, o que é um funcionamento bastante lógico.

2.4.1 - Procedimento

Um procedimento é um bloco de programa identificado por um nome de referência, podendo ter ou não o uso de argumentos como passagem de parâmetro podendo ser chamado em qualquer parte do programa principal ou de um programa que faça uso da sub-rotina.

Este tipo de sub-rotina possui como comportamento retornar seu controle operacional para a primeira linha de instrução após sua chamada sem devolver nenhuma resposta de processamento a sua chamada. A estrutura sintática deste tipo de sub-rotina é representada com a sintaxe:

```
function <nome_procedimento>([lista de argumentos])
    [definição de um bloco de escopo]
end
```

A *lista de argumentos* é opcional e será usada quando a sub-rotina precisa receber algum conteúdo passado entre o trecho de programa que chama a sub-rotina e a própria sub-rotina.

Para demonstrar o uso de sub-rotinas considere um programa em estilo calculadora que simule as operações básicas de uma calculadora que opere com a entrada de dois valores numéricos após a escolha da operação a ser executada. O programa deve apresentar uma lista de opções (menu) com as operações matemáticas de adição, subtração, multiplicação e divisão, além de uma opção de saída do programa. Escolhida a opção desejada, deve ser solicitada a entrada de dois valores numéricos para que seja possível executar o processamento escolhido. Após a execução da operação, o programa deve apresentar o resultado. Após a execução de qualquer uma das operações de cálculo, o programa deve voltar para o menu de seleção. Grave o código do programa seguinte com o nome "**c02ex22.jl**".

```
 1  using Printf
 2
 3  function pausa()
 4      println()
 5      print("Tecle <Enter> para encerrar programa... ")
 6      readline()
 7      println()
 8  end
 9
10  function rotSoma()
11      println()
12      println("ROTINA: Adição")
13      println()
14      print("Entre o 1o. valor: ")
15      a = parse(Float64, readline())
16      print("Entre o 2o. valor: ")
17      b = parse(Float64, readline())
18      r = a + b
19      println()
20      @printf("Resultado = %.2f\n", r)
21      pausa()
22  end
23
24  function rotDiferença()
25      println()
26      println("ROTINA: Subtração")
27      println()
28      print("Entre o 1o. valor: ")
29      a = parse(Float64, readline())
30      print("Entre o 2o. valor: ")
31      b = parse(Float64, readline())
32      r = a - b
33      println()
34      @printf("Resultado = %.2f\n", r)
35      pausa()
36  end
37
38  function rotProduto()
39      println()
42      println("ROTINA: Multiplicação")
43      println()
```

```
42      print("Entre o 1o. valor: ")
43      a = parse(Float64, readline())
44      print("Entre o 2o. valor: ")
45      b = parse(Float64, readline())
46      r = a * b
47      println()
48      @printf("Resultado = %.2f\n", r)
49      pausa()
50  end
51
52  function rotQuociente()
53      println()
54      println("ROTINA: Divisão")
55      println()
56      print("Entre o 1o. valor: ")
57      a = parse(Float64, readline())
58      print("Entre o 2o. valor: ")
59      b = parse(Float64, readline())
60      if b == 0
61          println()
62          println("Resultado: ERRO")
63      else
64          r = a / b
65          println()
66          @printf("Resultado = %.2f\n", r)
67      end
66      pausa()
67  end
68
69  opção = 0
70  while opção != 5
71      println("PROGRAMA CALCULADORA - V# 1")
72      println()
73      println("[1] - Adição")
74      println("[2] - Subtração")
75      println("[3] - Multiplicação")
76      println("[4] - Divisão")
77      println("[5] - Fim de programa")
78      println()
79      print("Escolha uma opção: ")
80      global opção = parse(Int, readline())
81      if opção != 5
82        if opção == 1
83            rotSoma()
84        elseif opção == 2
85            rotDiferença()
86        elseif opção == 3
87            rotProduto()
88        elseif opção == 4
89            rotQuociente()
90        else
91            println("Opção inválida - Tente novamente")
92        end
93      end
94  end
```

Ao executar o programa "**c02ex22.jl**" procure realizar todos os testes possíveis: entre valores inválidos quando da apresentação do menu e valore válidos. Efetue testes em todos os cálculos para perceber o funcionamento de chamadas das sub-rotinas estabelecidas.

Veja que cada uma das sub-rotinas estabelecidas efetua ações de entrada, saída e o processamento de cada cálculo estabelecido. Uma sub-rotina que efetue entrada e saída não deve ser tecnicamente uma função, mas sim um procedimento, mesmo que esteja realizando algum processamento, como as sub-rotinas exemplificadas.

O fato da linguagem não ter uma palavra reservada para identificação de procedimentos não significa em absoluto que esteja operando apenas com função, simplesmente por estar em uso a palavra-chave "**function**". O que, de fato, determina o tipo de sub-rotina não é o nome de sua identificação, mas sim seu comportamento operacional.

2.4.2 - Escopo e visibilidade de variáveis

A linguagem Julia é projetada para ser operada a partir do uso de sub-rotinas, ou seja, é uma linguagem projetada para atender a *filosofia de desenvolvimento estruturado* de programas. Mas cuidado, não confunda *filosofia de desenvolvimento estruturado* com o conceito do sub paradigma de *programação estruturada*. A *filosofia de desenvolvimento estruturado* abrange o uso dos sub paradigmas de: programação estruturada e de programação orientada a objetos, os quais são partes de um paradigma maior chamado *paradigma imperativo*.

As linguagens de programação que operam seus recursos a partir da filosofia de desenvolvimento estruturado usam para suas ações o tratamento de escopo e visibilidade de variáveis em que seu estado de ação na memória de um computador, podendo ser *global* ou *local*.

Anteriormente ao programa "**c02ex22.jl**" as variáveis usadas nos códigos de programas exigiram o uso do comando "**global**" quando declaradas dentro das estruturas dos laços "**while**" e "**for**" ou dentro de blocos explícitos definidos com os comandos "**begin**" e "**end**". Isso se fez necessário para garantir que essas variáveis fossem "vistas" por todo o programa. Quando uma variável é declarada dentro de uma estrutura de laço ou bloco explícito é considerada automaticamente local e para poderem ser usadas nas ações de entrada e processamento matemático precisam ter seu escopo definido como global.

Veja que no programa "**c02ex22.jl**" as variáveis definidas dentro das sub-rotinas são locais e não usam a declaração do comando "**global**" por não estarem sendo usadas dentro de estruturas de laços e definição de blocos e serem "visíveis" apenas dentro do escopo onde estão definidas. No entanto, a variável "*OPÇÃO*" declarada na parte principal do programa necessita ser declarada com escopo global.

Uma variável global pode ser operacionalizada em qualquer parte do programa e uma variável local somente é operada na área em que se encontra definida. Esse nível de abrangência é o que define o escopo e a visibilidade da variável na memória de um computador.

A título de ilustração considere um programa que utiliza variáveis chamadas "*X*" com estado global e local de existência em memória. Grave o código do programa seguinte com o nome "**c02ex23.jl**".

```
 1  x = 99
 2  println("Resultado do \'x\' global ...: ", x)
 3
 4  function teste()
 5      x = 88
 6      println("Resultado do \'x\' local ....: ", x)
 7  end
 8
 9  teste()
10  println("Resultado do \'x\' global ...: ", x)
11
12  println()
13  print("Tecle <Enter> para encerrar programa... ")
14  readline()
```

Ao ser executado o programa são apresentadas três respostas como saída, semelhante a imagem da figura 2.1. Observe as apresentações referentes as linhas "**2**", "**6**" e "**10**" do programa.

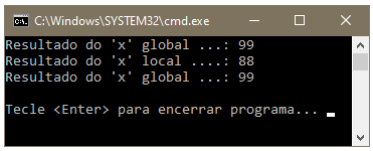

Figura 2.1 - Saída do teste de escopo e visibilidade de variáveis locais.

Veja que a linha "**2**" apresenta o valor "**99**" referente a definição da variável "*X*" na linha "**1**". Depois na linha "**6**" é apresentado o valor "**88**" referente a definição da variável "*X*" na linha "**5**". Por último, na linha "**10**" o programa volta a apresentar o valor "**99**" referente a definição da variável "*X*" na linha "**1**". Veja que as definições das variáveis "*X*" nas linhas "**1**" e "**5**" não fazem referência a mesma variável, pois se assim fosse os valores apresentados seriam "**99**", "**88**" e "**88**". Este efeito mostra que as duas variáveis "*X*" são locais, seus valores só são operacionalizados em suas respectivas áreas de escopo. A chamada da sub-rotina na linha "**9**" suspende e bloqueia a variável "*X*" na memória com o valor "**99**", cria uma outra variável "*X*" com valor "**88**" e quando encerrada a sub-rotina, esta destrói o acesso a variável "*X*" com o valor "**88**", restaurando a outra variável "*X*" com valor "**99**".

A título de ilustração considere um programa que utiliza uma variável chamada "*X*" com estado global de existência em memória. Grave o código do programa seguinte com o nome "**c02ex24.jl**".

```
 1  x = 99
 2  println("Resultado do \'x\' global ...: ", x)
 3
 4  function teste()
 5      global x = 88
 6      println("Resultado do \'x\' local ....: ", x)
 7  end
 8
 9  teste()
10  println("Resultado do \'x\' global ...: ", x)
11
12  println()
13  print("Tecle <Enter> para encerrar programa... ")
14  readline()
```

Ao ser executado o programa são apresentadas três respostas como saída, semelhante a imagem da figura 2.2. Observe as apresentações referentes as linhas "2", "6" e "10" do programa.

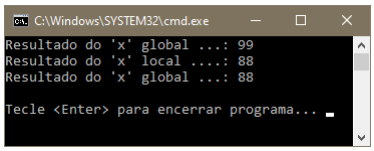

Figura 2.1 - Saída do teste de escopo e visibilidade de variável global.

Veja que o valor inicial de "**99**" da variável "*X*" definido na linha "**1**" é alterado na linha "**5**" para "**88**", sendo este valor mantido na linha "**10**" após a execução da sub-rotina devido ao fato de definição de escopo global junto a linha "**5**". A definição da linha "**5**" diz para o programa que a variável definida na linha "**1**" e a referência na linha "**5**" são relacionadas a mesma variável.

Veja que na linguagem Julia quando se deseja operar com escopo global é necessário mencioná-lo na estrutura de laço ou definição de bloco. Caso isso não seja realizado, o programa operará automaticamente com escopo global. Este efeito em Julia é diferente de outras linguagens onde o escopo de visibilidade das variáveis depende do local onde são definidas.

A primeira versão do programa calculadora foi definida com o uso de variáveis locais dentro das sub-rotinas de operação. A segunda versão, a seguir, demonstra o uso de todas as variáveis do programa como escopo global. A mudança do escopo local para o escopo global permitiu a aplicação do efeito de refinamento do programa simplificando as rotinas de operação ao se estabelecer rotinas especificas para a efetivação da entrada e saída de dados. Na primeira versão o programa está escrito com 94 linhas considerando linhas em branco, na segunda versão o programa possui 89 linhas considerando linhas em branco. Grave o código do programa seguinte com o nome "**c02ex25.jl**".

```
1  using Printf
2
3  function pausa()
4      println()
5      print("Tecle <Enter> para encerrar programa... ")
6      readline()
7      println()
8  end
9
10 function entrada()
11     println()
12     print("Entre o 1o. valor: ")
13     global a = parse(Float64, readline())
14     print("Entre o 2o. valor: ")
15     global b = parse(Float64, readline())
16 end
17
18 function saída()
19     println()
20     @printf("Resultado = %.2f\n", r)
21     pausa()
22 end
23
24 function rotSoma()
25     println()
26     println("ROTINA: Adição")
27     entrada()
28     global r = a + b
29     saída()
30 end
31
32 function rotDiferença()
33     println()
34     println("ROTINA: Subtração")
35     entrada()
36     global r = a - b
37     saída()
38 end
39
42 function rotProduto()
43     println()
44     println("ROTINA: Multiplicação")
45     entrada()
46     global r = a * b
47     saída()
48 end
49
50 function rotQuociente()
51     println()
52     println("ROTINA: Divisão")
53     entrada()
54     if b == 0
55         println()
56         println("Resultado: ERRO")
57         pausa()
58     else
59         global r = a / b
60         saída()
61     end
62 end
63
```

```
64  opção = 0
65  while opção != 5
66      println("PROGRAMA CALCULADORA - V# 2")
67      println()
68      println("[1] - Adição")
69      println("[2] - Subtração")
70      println("[3] - Multiplicação")
71      println("[4] - Divisão")
72      println("[5] - Fim de programa")
73      println()
74      print("Escolha uma opção: ")
75      global opção = parse(Int, readline())
76      if opção != 5
77        if opção == 1
78            rotSoma()
79        elseif opção == 2
80            rotDiferença()
81        elseif opção == 3
82            rotProduto()
83        elseif opção == 4
84            rotQuociente()
85        else
86            println("Opção inválida - Tente novamente")
87        end
88      end
89  end
```

Ao ser executado o programa não será possível perceber as mudanças realizadas, pois essas afetam a estrutura do código, mas não seu funcionamento e para mais detalhes consulte o apêndice deste livro.

2.4.3 - Passagem de parâmetro

Devido ao efeito de definição de escopo e visibilidade de variáveis, principalmente devido ao escopo local a fim de economizar e dinamizar o consumo de memória torna-se necessário usar a ideia de passagem de parâmetro para transportar valores entre sub-rotinas e suas chamadas. Em Julia a passagem de parâmetro ocorre somente por valor.

O recurso de passagem de parâmetro permite estabelecer uma linha de comunicação entre os conteúdos dos trechos de sub-rotinas os trechos de programa que chamam essas sub-rotinas, principalmente quando se usam variáveis com escopo local.

A passagem de parâmetro é efetivada quando se usam variáveis locais, e a variável local de uma sub-rotina necessita receber ou enviar conteúdo para a variável local de outra sub-rotina. A passagem de parâmetro ocorre entre o trecho de programa chamador e o trecho de programa chamado, podendo ser definida de forma simplista sem se preocupar com o tipo de dado passado ou com o estabelecimento do tipo de dado a ser operacionalizado.

Para demonstrar um programa que efetue o cálculo do fatorial de um valor qualquer com passagem de parâmetro simples em estilo procedimento considere o código de programa seguinte. Grave o código do programa seguinte com o nome "**c02ex26.jl**".

```
 1  println("PROGRAMA FATORIAL - V# 1")
 2  println()
 3
 4  function fatorial(n)
 5      fat = 1
 6      for i in 1:n
 7          fat *= i
 8      end
 9      println(fat)
10  end
11
```

```
12  print(("Entre valor inteiro: ")
13  n = parse(Int, readline())
14
15  fatorial(n)
16
17  println()
18  print("Tecle <Enter> para encerrar programa... ")
19  readline()
```

Ao ser executado o programa perceba que o valor fornecido é passado como argumento da sub-rotina efetuando internamente seu cálculo e apresentação de seu resultado.

Para demonstrar um programa que efetue o cálculo do fatorial de um valor qualquer com passagem de parâmetro com tipo de dado especificado em estilo procedimento considere o código de programa seguinte. Grave o código do programa seguinte com o nome "**c02ex27.jl**".

```
 1  println("PROGRAMA FATORIAL - V# 2")
 2  println()
 3
 4  function fatorial(n::Int64)
 5      fat = 1
 6      for i in 1:n
 7          fat *= i
 8      end
 9      println(fat)
10  end
11
12  print("Entre valor inteiro: ")
13  n = parse(Int, readline())
14
15  fatorial(n)
16
17  println()
18  print("Tecle <Enter> para encerrar programa... ")
19  readline()
```

Ao se executar a segunda versão do programa perceberá que seu funcionamento é idêntico a primeira versão. A diferença está na especificação do tipo de dado do argumento "*N*" definido na linha "**4**".

As sub-rotinas podem ser operacionalizadas a partir de uma lista de parâmetros incluindo-se parâmetros opcionais. Parâmetros opcionais são fornecidos com valores padrão. Quando esses valores não são indicados ao se chamar uma função, os valores padrão são usados.

Como ilustração de uso de listas de parâmetros, incluindo-se opcionais considere um programa que usa uma função de teste que recebe três parâmetros como seus argumentos de operação, sendo os dois primeiros obrigatórios e o terceiro opcional. O programa deve somar os dois primeiros parâmetros e multiplicar pelo terceiro parâmetro. Grave o código do programa seguinte com o nome "**c02ex28.jl**".

```
 1  println("PROGRAMA TESTE")
 2  println()
 3
 4  function teste(a, b, c = 1)
 5      println((a + b) * c)
 6  end
 7
 8  print("Entre valor <A>: ")
 9  a = parse(Int, readline())
10
18  print("Entre valor <B>: ")
19  b = parse(Int, readline())
```

```
20
21  println()
22
23  teste(a, b)
24  teste(a, b, 2)
25
26  println()
27  print("Tecle <Enter> para encerrar programa... ")
28  readline()
```

Ao ser executado o programa note que os resultados apresentados pelas linhas "**23**" e "**24**" mostram respectivamente um valor sem a definição do terceiro parâmetro e o resultado com a definição do terceiro parâmetro.

2.4.4 - Função

A partir de uma visão geral sobre uso e definição de sub-rotinas como procedimentos e uso de passagens de parâmetro tem-se em mãos um conjunto de elementos operacionais para uso das sub-rotinas como funções propriamente dito. Para a definição de uma função se usa a seguinte sintaxe:

```
function <nome_procedimento>([lista de argumentos])
    [definição de um bloco de escopo]
    return <valor de retorno da função>
end
```

Veja o detalhe de uso do comando "**return**" que tem por finalidade fazer a devolução de um valor de resposta a execução de sua operação. Na prática este comando é opcional, pois a função retornará a última expressão caso o comando seja omitido. No entanto, é ideal manter sua declaração de forma explícita, pois isso ajuda a diferenciar de uma função usada como simulação de procedimento.

Para demonstrar um programa que efetue o cálculo do fatorial de um valor qualquer a partir de uma função considere o código de programa seguinte. Grave o código do programa seguinte com o nome "**c02ex29.jl**".

```
 1  println("PROGRAMA FATORIAL - V# 3")
 2  println()
 3
 4  function fatorial(n::Int64)
 5      fat = 1
 6      for i in 1:n
 7          fat *= i
 8      end
 9      return fat
10  end
11
12  print("Entre valor inteiro: ")
13  n = parse(Int, readline())
14
15  println(fatorial(n))
16
17  println()
18  print("Tecle <Enter> para encerrar programa... ")
19  readline()
```

Ao ser executado o programa entre um valor para o cálculo do fatorial e veja o comportamento no uso da sub-rotina do tipo função.

O próximo exemplo de programa considera o programa calculadora em sua terceira versão utilizando-se uma sub-rotina para efetivação dos cálculos. Grave o código do programa seguinte com o nome "**c02ex30.jl**".

```
1  using Printf
2
3  function pausa()
4      println()
5      print("Tecle <Enter> para encerrar programa... ")
6      readline()
7      println()
8  end
9
10 function entrada()
11     println()
12     print("Entre o 1o. valor: ")
13     global a = parse(Float64, readline())
14     print("Entre o 2o. valor: ")
15     global b = parse(Float64, readline())
16 end
17
18 function saída()
19     println()
20     @printf("Resultado = %.2f\n", r)
21     pausa()
22 end
23
24 function cálcula(a, b, operador = "String")
25     if operador == "+" return a + b end
26     if operador == "-" return a - b end
27     if operador == "*" return a * b end
28     if operador == "/" return a / b end
29 end
30
31 function rotSoma()
32     println()
33     println("ROTINA: Adição")
34     entrada()
35     global r = cálcula(a, b, "+")
36     saída()
37 end
38
39 function rotDiferença()
42     println()
43     println("ROTINA: Subtração")
44     entrada()
45     global r = cálcula(a, b, "-")
46     saída()
47 end
48
49 function rotProduto()
50     println()
51     println("ROTINA: Multiplicação")
52     entrada()
53     global r = cálcula(a, b, "*")
54     saída()
55 end
56
57 function rotQuociente()
58     println()
59     println("ROTINA: Divisão")
60     entrada()
61     if b == 0
62         println()
63         println("Resultado: ERRO")
64         pausa()
65     else
```

```
66          global r = cálcula(a, b, "/")
67          saída()
68      end
69  end
70
71  opção = 0
72  while opção != 5
73      println("PROGRAMA CALCULADORA - V# 3")
74      println()
75      println("[1] - Adição")
76      println("[2] - Subtração")
77      println("[3] - Multiplicação")
78      println("[4] - Divisão")
79      println("[5] - Fim de programa")
80      println()
81      print("Escolha uma opção: ")
82      global opção = parse(Int, readline())
83      if opção != 5
84        if opção == 1
85            rotSoma()
86        elseif opção == 2
87            rotDiferença()
88        elseif opção == 3
89            rotProduto()
90        elseif opção == 4
91            rotQuociente()
92        else
93            println("Opção inválida - Tente novamente")
94        end
95    end
96  end
```

Nesta terceira versão atente para o trecho de linhas entre "**24**" e "**29**", no qual encontra-se definida a sub-rotina de função que efetua os cálculos de acordo com o operador aritmético associado a operação solicitada.

Devido a característica de Julia operar apenas com passagem de parâmetro por valor a linguagem permite que uma função retorne mais de um valor como saída de sua operação, ação conhecida como *operação de despacho múltiplo*. É oportuno esclarecer que este tipo de ocorrência contraria o que prega a ciência matemática sobre o tema *funções*, onde cada função possui como retorno um valor.

Para efetuar um teste considere no ambiente interativo a definição de uma função chamada "**quatroOperações**" que recebe dois argumentos e retorna o resultado das quatro operações aritméticas básicas. Assim sendo, escreva a seguinte função:

```
julia> function quatroOperações(x::Float64, y::Float64)
           adc = x + y
           sub = x - y
           mul = x * y
           dvs = x / y
           return adc, sub, mul, dvs
       end
quatroOperações (generic function with 1 method)

julia> x = quatroOperações(5.5, 2.35)
(7.85, 3.15, 12.925, 2.3404255319148937)
```

Veja que ao ser efetuada a chamada da função "**quatroOperações**", passado, dois argumentos numéricos ocorre a apresentação de quatro resultados retornados na forma de uma tupla (estrutura de dados com valores separados por vírgulas e delimitados entre parênteses).

A tupla é um conjunto de dados que pode ter seu primeiro e último valores retornados por meio de funções especificas. Veja os exemplos a seguir:

```
julia> first(x)
7.85

julia> last(x)
2.3404255319148937
```

Uma função que retorna mais de um valor pode ser usada para gerar múltiplas atribuições. Veja os exemplos de ação seguintes:

```
julia> a, b, c, d = quatroOperações(5.5, 2.35)
(7.85, 3.15, 12.925, 2.3404255319148937)

julia> a
7.85

julia> b
3.15

julia> c
12.925

julia> d
2.3404255319148937
```

Um exemplo mais prático com o uso de despacho múltiplo de resultados é a definição de uma sub-rotina de função que retorne os valores do cálculo de uma equação completa de segundo grau. Para tanto, no modo interativo escreva as instruções seguintes:

```
julia> function eq2grau(a, b, c)
          if a != 0 && b != 0 && c != 0
              try
                  Δ = sqrt(b ^ 2 - 4a * c)
                  if Δ == 0
                      return -b / 2a
                  else
                      x1 = (-b + Δ) / 2a
                      x2 = (-b - Δ) / 2a
                  end
                  return x1, x2
              catch
                  return println("Não existem raízes reais.")
              end
          else
              return println("Os valores devem ser diferentes de \"zero\".")
          end
      end
eq2grau (generic function with 1 method)
```

Atente para o uso da variável "Δ" junto a função "**eq2grau**". Lembre-se de que para conseguir usar variáveis deste tipo é necessário escrever a instrução "**\Delta<Tab>**" no modo interativo. Veja as respostas retornadas pela função.

```
julia> eq2grau(1, 2, -15)
(3.0, -5.0)

julia> eq2grau(-1, 2, -15)
Não existem raízes reais.

julia> eq2grau(0, 2, -15)
Os valores devem ser diferentes de "zero".

julia> eq2grau(1, 0, -15)
Os valores devem ser diferentes de "zero".

julia> eq2grau(0, 0, -15)
Os valores devem ser diferentes de "zero".

julia> eq2grau(0, 0, 0)
Os valores devem ser diferentes de "zero".

julia> eq2grau(2, -2, -15)
(3.2838821814150108, -2.2838821814150108)

julia> eq2grau(4, -4, 1)
0.5
```

Há alguns outros detalhes sobre o uso de sub-rotinas que não são aqui apresentados devido ao escopo deste trabalho. Após este estudo aconselha-se a pesquisar outras fontes sobre Julia. Caso seja necessário algum detalhe adicional este será apresentado em momento oportuno, mas se quiser dê uma olhada nos apêndices "**F**", "**G**" e "**H**" deste livro.

2.5 - Símbolos e Macros

Anteriormente foram usados recursos que se caracterizam por serem elementos operacionais fundamentados em macros (recursos inicializados sempre com o caractere "**@**"). Foram usados os macros: "**@goto**", "**@label**" e "**@printf**", e além dessas, há outros macros da linguagem, além da possibilidade de se fazer a criação de macros personalizadas. Veja alguns macros internas:

```
julia> x = @show 1 + 2 * 3  # mostra o desenvolvimento da equação e armazena resultado em "x"
1 + 2 * 3 = 7
7

julia> @show 5^2+3 5/2;  # executa equações separadas por espaço em branco
5 ^ 2 + 3 = 28
5 / 2 = 2.5

julia> @time factorial(10)  # mostra o tempo de execução do recurso
  0.000005 seconds
3628800
```

```
julia> @info "Programação Julia"  # mostra mensagem de informação
[ Info: Programação Julia

julia> function fatorial(n::Int64)  # definição interativa de função
           fat = 1
           for i in 1:n
               fat *= i
           end
           return fat
       end
atorial (generic function with 2 methods)

julia> @code_native fatorial(5)  # mostra Assembly do código da função
        .text
; ┌ @ REPL[44]:1 within `fatorial`
        pushq   %rbp
        movq    %rsp, %rbp
        xorl    %r8d, %r8d
; │ @ REPL[44]:3 within `fatorial`
; │┌ @ range.jl:5 within `Colon`
; │└┌ @ range.jl:354 within `UnitRange`
; │└└┌ @ range.jl:359 within `unitrange_last`
        testq   %rcx, %rcx
        cmovnsq %rcx, %r8
; │└└└
        jle     L54
; │ @ REPL[44]:4 within `fatorial`
        negq    %r8
        movl    $1, %ecx
        movl    $1, %eax
        nopl    (%rax)
; │┌ @ int.jl:88 within `*`
L32:
        imulq   %rcx, %rax
; │└
; │┌ @ range.jl:837 within `iterate`
; │└┌ @ promotion.jl:468 within `==`
        leaq    (%r8,%rcx), %rdx
        incq    %rdx
; │└└
        incq    %rcx
; │└┌ @ promotion.jl:468 within `==`
        cmpq    $1, %rdx
; │└└
        jne     L32
; │ @ REPL[44]:6 within `fatorial`
        popq    %rbp
        retq
; │ @ REPL[44] within `fatorial`
L54:
        movl    $1, %eax
; │ @ REPL[44]:6 within `fatorial`
        popq    %rbp
        retq
        nopl    (%rax)
; └
```

Veja que macros são recursos que em um primeiro momento se parecem com o uso de sub-rotinas de funções ou procedimentos. As funções/procedimentos são recursos que ao receberem valores por meio de passagens de parâmetros em seus argumentos efetuam seu processamento em tempo de execução podendo retornar ou não valores como resposta a sua operação. Já, macros são sub-rotinas especiais que pegam a passagem de parâmetro recebida em seus argumentos e as usa para gerar códigos de programas na linguagem. Este efeito caracteriza-se como elemento na linguagem Julia de metaprogramação (maneira pela qual um programa manipula a si mesmo).

Macros são componentes que possibilitam a criação, por exemplo, de estruturas de controle que a linguagem não possua em seu ecossistema. Isso possibilita a criação de recursos operacionais que a linguagem efetivamente não tenha. Se você conhece LISP já tem noção do uso de macros que operam ações, como: declaração "**`if`**", laço "**`for`**" e o estilo "**`switch-case`**". A propósito é da influência da linguagem LISP que a linguagem Julia tira seu sistema de macros. A definição de macros segue a seguinte sintaxe:

```
macro <nome_da_macro>(<lista de argumentos>)
    [<return> <:definição>]
end
```

A vantagem no uso de macros é que este tipo de sub-rotina é avaliado uma única vez pelo compilador da linguagem quando em uso diferentemente de uma sub-rotina de função/procedimento, podendo ser replicado quantas vezes for necessário dentro de um programa sem consumo no tempo de processamento.

Mas antes de definir um macro é importante ter noção do que é um *símbolo* e como defini-los. Um *símbolo* é a constituição de um valor imutável do tipo cadeia, vinculado a uma variável conhecida como *string interning* (cadeia internada) que tem em memória apenas uma cópia de seu nome, sendo definido a partir da sintaxe:

```
:<nome_do_símbolo>
```

O nome de identificação de um símbolo pode ser definido com uso de uma sequência de caracteres alfanuméricos ou a partir do uso de valores numéricos com operadores aritméticos ou a partir de qualquer outro conteúdo válido da linguagem, definidos entre parênteses. Veja dois exemplos de definição de símbolos:

```
julia> y1 = :xpto
:xpto

julia> y2 = :(2 + 4)
:(2 + 4)
```

As variáveis "*Y1*" e "*Y2*" possuem respectivamente associadas como valores os símbolos "*:xpto*" e "*:(2 + 4)*". O recurso de símbolos é importante por permitir que a linguagem Julia consiga ter pontos de ligação com diversos outros recursos, como por exemplo, o uso de bancos de dados externos, ou neste caso com o uso na definição de macros.

Para apresentar o valor associado ao símbolo de certa variável pode-se usar o procedimento "**`println`**" como indicados a seguir:

```
julia> println(y1)
xpto

julia> println(y2)
2 + 4
```

Veja que o conteúdo apresentado se assemelha a definição de cadeias de caracteres definidos junto a cada variável. Quando um símbolo possui a definição de uma expressão aritmética este pode ter seu conteúdo calculado a partir do uso da função "**`eval()`**" a partir da sintaxe:

```
julia> println(eval(y2))
6
```

A partir do conhecimento de como se faz a definição de símbolos é possível fazer o uso de macros. Veja como exemplo a definição do macro "**mensagem**" que apresenta o texto "*Estudo de linguagem Julia*". A macro pode ser definida no modo programado ou no modo interativo. Veja-a no modo interativo.

```
julia> macro mensagem()
           return :(println("Estudo de linguagem Julia"))
       end
@mensagem (macro with 1 method)

julia> @mensagem
Estudo de linguagem Julia
```

Observe que a macro "**mensagem**" faz o retorno de um símbolo que contém a apresentação da mensagem indicada. É importante salientar que o comando "**return**" é opcional e pode ser omitido. Desta forma também é válida a definição:

```
julia> macro mensagem()
           :(println("Estudo de linguagem Julia"))
       end
@mensagem (macro with 1 method)
```

A título de ilustração veja a seguir um macro que apresenta como resultado o quadrado de um valor numérico fornecido como argumento.

```
julia> macro calcQuadrado(n)
           return :($n ^ 2)
       end
@calcQuadrado (macro with 1 method)

julia> @calcQuadrado(4)
16
```

Veja que quando se faz uso de argumento como parâmetro do macro é obrigatório para uso do parâmetro fazer a aplicação do símbolo "**$**" (cifrão) antes da variável. Este artifício permite a macro realizar a interpolação do valor fornecido no argumento e utilizá-lo como substituto da variável definida dentro do macro.

Macros podem ser definidos com códigos mais complexos e neste caso pode-se omitir o uso do comando "**return**" e a definição de símbolos, mas usa-se em seu lugar os comandos "**quote**" e "**end**" para definir um escopo de trabalho para a ação. Antes de apresentar um código de macro com estrutura mais complexo veja três maneiras de se fazer a repetição de cadeias de caracteres em Julia:

```
julia> repeat("Julia ", 5)
"Julia Julia Julia Julia Julia "

julia> repeat("Julia\n", 5)
"Julia\nJulia\nJulia\nJulia\nJulia\n"

julia> "Julia " ^ 5
"Julia Julia Julia Julia Julia "
```

```
julia> "Julia\n" ^ 5
"Julia\nJulia\nJulia\nJulia\nJulia\n"

julia> ^("Julia ", 5)
"Julia Julia Julia Julia Julia "

julia> ^("Julia\n", 5)
"Julia\nJulia\nJulia\nJulia\nJulia\n"
```

Os exemplos apresentados escrevem a cadeia indicada certo número de vezes em uma única linha, mas não permitem escrever a cadeia em linhas separadas. Desta forma, considere escrever um macro que efetue operação similar, mas que permita escrever uma cadeia separada por linhas. Neste sentido atente para o código seguinte:

```
julia> macro repita(conteúdo, n)
           quote
               for i in 1 : $n
                   print($conteúdo)
               end
           end
       end
@repita (macro with 1 method)

julia> @repita("Julia ", 5)
Julia Julia Julia Julia Julia

julia> @repita("Julia\n", 5)
Julia
Julia
Julia
Julia
Julia
```

Observe que a macro "**repita**" cria uma área de escopo entre os comandos "**quote**" e "**end**" para a definição de uma área de operação usada pelo comando "**for**". Os comandos "**quote**" e "**end**" possuem similaridade com os comandos "**begin**" e "**end**" já apresentados e que não podem ser usados com macros. Os comandos "**quote**" e "**end**" permitem a definição de uma área de trabalho que aceita a criação de variáveis locais.

A chamada de um macro pode ser realizada de forma simplificada sem o uso de parênteses e vírgulas. Para tanto, execute a instrução:

```
julia> @repita "Julia\n" 5
Julia
Julia
Julia
Julia
Julia
```

Veja que com os exemplos indicados de macros possuem parte de seu comportamento similar ao comportamento de uma função. Podem ter retorno com o comando "**return**" se assemelhando a uma função ou não se assemelhando a um procedimento. As sub-rotinas de função/procedimento são adequadas para a definição de módulos internos a um programa, podendo também serem usadas de forma externa quando definidas para uso geral. Já as sub-rotinas de macros são mais adequadas para uso genérico e externo, não sendo muito práticas para definições específicas. No entanto, não use este conselho ao pé da letra, pois podem existir exceções.

Os exemplos de sub-rotinas de macros parecem-se com sub-rotinas de função/procedimento, mas possuem funcionamento diferente. O código definido dentro de um macro é copiado para o ponto do programa onde sua chamada é indicada, diferente de uma função que é executada quando chamada. A execução do macro "**repita**" na prática é substituído pela execução da instrução:

```
julia> for i in 1:5 print("Julia ") end
Julia Julia Julia Julia Julia
```

Macros são formas de simplificar algumas operações da linguagem, permitindo realizar a redefinição de operações ou criar soluções para problemas que a linguagem não atenda diretamente, e como a apresentada por Danisch (2019) com a proposta do macro "**assert**", indicada a seguir de forma adaptada:

```
julia> macro assert(expressão)
         :($expressão ? nothing : error("Falha na expressão: ", $(string(expressão))))
       end
@assert (macro with 1 method)
```

O macro "**assert**" está preparado para receber uma expressão da linguagem. Se a expressão estiver correta nada é apresentado, mas se tiver algo de errado uma mensagem de erro apontando a expressão é retornada. Para fazer uso adequado deste macro é necessário usar os comandos "**try**", "**catch**" e "**end**". Veja a seguir um exemplo que tenta validar a expressão "**1 == 2**":

```
julia> try
           @assert(1 == 2)
       catch erro
           println(erro)
       end
ErrorException("Falha na expressão: 1 == 2")
```

Ao ser executada a instrução anterior ocorre a captura da ocorrência de erro ao verificar se "**1 == 2**" que gera um retorno "**false**" pelo comando "**catch**" que faz a apresentação da mensagem "**Falha na expressão: 1 == 2**" atribuída a variável "*ERRO*".

2.6 - Recursividade

Após visão geral sobre o uso de sub-rotinas de funções/procedimentos e macros é importante focar a atenção ao uso do efeito de recursividade aplicado sobre sub-rotinas de função. O termo *recursividade* expressa a propriedade daquilo que se pode repetir um número indefinido de vezes. Na computação indica a repetição da mesma também conhecida como "equação de recorrência". O efeito de recursividade ou recorrência aplica-se naturalmente em sub-rotinas que possuem a capacidade natural de retornar valor como resposta a sua operação, neste caso, sub-rotinas do tipo função.

Graças à característica operacional de retorno de uma função é possível desenvolver sub-rotinas que chamam a si mesmas, ou seja, efetuando recursão (ODIFREDDI & COOPER, 2012). Uma função recursiva é uma estrutura simples que representa um algoritmo de cálculo complexo (MENESES, 2013, p. 216).

Uma função recursiva não pode chamar a si mesma indiscriminadamente, pois, se assim o fizer, entrará em um processo infinito de chamadas exaustivas (função circular). Portanto, é necessário que a função recursiva tenha obrigatoriamente a definição de uma condição de encerramento (chamada de *aterramento*), que é em si a solução mais simples e menor para o problema avaliado chamado de *caso base*.

As funções recursivas podem ser implementadas a partir de duas óticas: *recursividade simples* ou *comum* e *recursividade de cauda*.

2.6.1 - Simples

A *recursividade simples* é uma técnica que armazena o valor temporário de sua chamada na pilha de memória e aguarda a ocorrência das chamadas sucessivas e dos armazenamentos sucessivos dos demais valores até que ocorra a condição de aterramento. Assim que a condição de aterramento ocorre a função inicia as chamadas de retorno operacionalizando o valor armazenado da pilha com o valor calculado no retorno até chegar ao resultado esperado.

Como exemplo de demonstração de uso de recursividade simples considere desenvolver um programa de computador que efetue a leitura de um número inteiro e retorne o resultado do fatorial deste número a partir do uso de sub-rotina que opere sob a ótica da recursividade simples.

Fórmula fatorial: recursividade simples

$$n! = \begin{cases} 1 & \text{, se } n = 0 \\ n \cdot (n - 1)! & \text{, se } n > 0 \end{cases}$$

Com base na fórmula apresentada observe a seguir o código de programa que efetua a apresentação do cálculo de um fatorial com recursividade simples. Grave o código do programa seguinte com o nome "**c02ex31.jl**".

```
 1  function fatorial(n::Int64)
 2      if n == 0
 3          fat = 1
 4      else
 5          fat = n * fatorial(n - 1)
 6      end
 7    return fat
 8  end
 9
10  print("Entre valor inteiro: ")
11  limite = parse(Int, readline())
12
13  println(fatorial(limite))
14
15  println()
16  print("Tecle <Enter> para encerrar programa... ")
17  readline()
```

Ao observar o código do programa, nota-se na linha "**2**" a definição da condição "**n == 0**" que quando falsa executará a operação da linha "**5**" em que a variável "*FAT*" é atribuído pelo resultado da operação "**n * fatorial(n - 1)**". O parâmetro "*N*" determina o número de vezes que a operação deve ser efetuada. A indicação " **fatorial(n - 1)**" caracteriza-se por ser uma nova instância de chamada à função realizando em si mesma nova operação até que a condição da linha "**2**" se torne verdadeira. Cada instância de chamada da função ocorre de forma a empilhar cada chamada na pilha de memória para depois, no retorno, efetuar as multiplicações sucessivas do valor retornado com o valor armazenado na pilha, a fim de calcular a fatorial solicitada.

A recursividade, de forma geral, é considerada uma ação elegante ao facilitar a abstração e a modularidade no desenvolvimento de funções complexas. No entanto, em virtude do efeito de empilhamento a recursividade simples consome espaço de memória e deve ser pensada a partir de outra ótica, como o uso de recursão de cauda.

Para o cálculo da fatorial define-se na linha "**1**" o tipo de dado inteiro de 8 bits, uma vez que os valores que podem ser operados estão entre "**1**" e "**20**". Qualquer valor acima de "**20**" não apresenta o resultado do cálculo de forma correta.

2.6.2 - Cauda

A *recursividade em cauda* é uma técnica que garante efetivamente menor consumo de memória por não ser necessário guardar e empilhar a posição da chamada da função na pilha de memória, ou seja, nenhum processamento adicional

após o encerramento da função é realizado, sendo que resultado final é calculado e obtido no próprio argumento antes de seu retorno.

Como exemplo de demonstração de uso de recursividade de cauda considere um programa de computador que efetue a leitura de um número inteiro e retorne o resultado do fatorial deste número a partir do uso de sub-rotina que opere sob a ótica da recursividade de cauda.

Fórmula fatorial: recursividade de cauda

$$(n,\ p)!\ \underset{p\ =\ 1}{=} \begin{cases} p & ,\ \text{se } n = 0 \\ (n - 1,\ n \cdot p)!, & \text{se } n > 0 \end{cases}$$

Com base na fórmula apresentada observe a seguir o código de programa que efetua a apresentação do cálculo de um fatorial com recursividade de cauda. Grave o código do programa seguinte com o nome "**`c02ex32.jl`**".

```
 1  function fatorial(n::Int64, p::Integer=1)
 2      if n == 0
 3          fat = p
 4      else
 5          fat = fatorial(n - 1, n * p)
 6      end
 7    return fat
 8  end
 9
10  print("Entre valor inteiro: ")
11  limite = parse(Int, readline())
12
13  println(fatorial(limite))
14
15  println()
16  print("Tecle <Enter> para encerrar programa... ")
17  readline()
```

A função "**`fatorial(n = "Int8", p::Integer = 1)`**" usa dois parâmetros: o parâmetro obrigatório da variável "*N*" referente ao valor recebido do usuário que determina a quantidade de iterações a serem efetivadas para a obtenção do cálculo final esperado na forma de um inteiro de 8 bits e o segundo parâmetro opcional representado pela variável "*P*" do tipo inteira inicializada com o valor "**`1`**", o qual auxilia a obtenção do resultado do fatorial calculado e não deve em hipótese alguma ser fornecido manualmente. Quando uma função possui a definição de argumento opcional, seu parâmetro não deve ser passado como demonstra a chamada da função na linha "**`13`**".

2.7 - Terminal ANSI

O modo *Terminal ANSI* oferece controle padronizado das ações vinculadas ao uso do terminal de vídeo e teclado. Terminal é um equipamento que oferece ao usuário uma interface de acesso ao computador, como o teclado e o monitor de vídeo. Nos microcomputadores modernos o terminal se confunde com o teclado e o monitor de vídeo integrados ao gabinete do equipamento.

O uso do equipamento chamado *terminal* é comum em computadores de grande porte, em que se tem um console de acesso às funções do computador que esteja distante do terminal, diferentemente de um mini ou microcomputador cujas funções de entrada e saída em vídeo (terminal) e de processamento (CPU) são conectadas próximas umas das outras, quando não no mesmo console, como é o caso dos *notebooks*.

O modelo mais famoso de *terminal* para computadores de grande e médio porte foi o VT100, da empresa DEC. Esse terminal foi o primeiro a possuir um microprocessador e um dos primeiros a incorporar o controle por sequência de *escape ANSI*. Como a sequência de *escape* foi uma proposta de padronização do órgão ANSI (*American National Standards Institute* - Instituto Nacional Americano de Padrões) para os recursos de acesso a terminais da época, o

padrão do modelo VT100 acabou por prevalecer até os dias atuais em diversos sistemas operacionais, como ocorre com o UNIX (incluindo-se os muitos sabores Linux). Em relação aos sistemas operacionais da empresa Microsoft, o recurso está disponível, de forma indireta, nos antigos sistemas MS-DOS e em todas as versões do Windows de 32 *bits*, pelo menos até a versão 10. No entanto, como o uso do padrão 64 *bits* vem aumentando, possivelmente este recurso deixe de ser usado na medida em que a própria Microsoft deixar de produzir versões de seu sistema operacional para uma arquitetura de 32 *bits*, pois o recurso é inexistente nas versões de 64 *bits* do Windows.

A linguagem Julia possui internamente controle sobre as operações do modo *Terminal ANSI* não necessitando realizar nenhum tipo de configuração da linguagem ou mesmo do sistema operacional em uso. Existem diversas ações que podem ser executadas por um programa de computador no modo console de texto, para alterar parte do comportamento do teclado e do monitor de vídeo. Entre essas ações, é possível alterar a forma de exibição do monitor de vídeo, controlar o local onde o cursor será exibido e redefinir as teclas do teclado. Essas ações podem ser controladas por meio dos códigos de *escape*.

Para controlar o vídeo ou o teclado por meio dos recursos do modo *Terminal ANSI*, é necessário usar comandos específicos, precedidos do código de controle de *escape* que, na linguagem Julia, consiste no uso do código numérico octal "**`\033`**" ou pelo símbolo "**`\e`**". Todo valor octal, em linguagem de programação Julia, é iniciado com o símbolo contra barra "**`\`**" e o algarismo zero "**`0`**" à esquerda.

Os comandos (sequência de escape) são divididos em dois grupos de ação (vídeo e teclado). No grupo *vídeo*, é possível manipular vídeo, controlar e movimentar o cursor e os atributos de tela. No grupo *teclado*, é possível redefinir a ação de qualquer tecla.

A seguir, é apresentada apenas uma parte da tabela com os códigos de controle de vídeo. As funções do teclado não serão usadas, por isso são omitidas. Para maiores detalhes sobre códigos de controle de *escape*, consulte o sítio **ANSI CODES**, no endereço **http://bluesock.org/~willg/dev/ansi.html** (visitados em junho de 2022).

As tabelas 2.8 e 2.9 apresentam os códigos de controle ANSI para limpar vídeo, formatar texto e manipular cursor. A indicação **ESC** na coluna ***Código escape*** deve ser substituída pelo código numérico octal **\033** ou pelo código alfabético **\e**.

Código escape	Função	Descrição
ESC[2J	Limpa vídeo	Limpa vídeo e posiciona o cursor no topo esquerdo da tela.
ESC[K	Limpa linha	Limpa linha a partir da posição em que o cursor estiver parado.
ESC[4m	Liga sublinhado	Ativa modo de escrita em formato sublinhado.
ESC[24m	Desliga sublinhado	Desativa modo de escrita em formato sublinhado.
ESC[1m	Liga negrito	Ativa modo de escrita em formato negrito.
ESC[21m	Desliga negrito	Desativa modo de escrita em formato negrito.

Tabela 2.8 - Código ANSI para limpar vídeo e formatar texto.

Código escape	Função	Descrição
ESC[<n>A	Cursor p/ cima	Move cursor para cima o número de vezes indicado em "n".
ESC[<n>B	Cursor p/ baixo	Move cursor para baixo o número de vezes indicado em "n".
ESC[<n>C	Cursor p/ direita	Move cursor para direita o número de vezes indicado em "n".
ESC[<n>D	Cursor p/ esquerda	Move cursor para esquerda o número de vezes indicado em "n".
ESC[<l>;<c>H	Posiciona cursor	Move cursor para linha "l" e coluna "c" indicadas.
ESC[s	Salva cursor	Salva a posição em que o cursor se encontra na tela.
ESC[u	Restaura cursor	Restaura a posição do cursor anteriormente gravada.

Tabela 2.9 - Código ANSI para manipular cursor.

Os recursos do modo *Terminal ANSI* vão além dos códigos apresentados como, por exemplo, fazer a manipulação do modo de cores do vídeo onde é possível definir cores de fundo e de texto. Observe os detalhes indicados na tabela 2.10 para as cores de fundo e a tabela 2.11 para as cores de texto.

Código escape	Função	Descrição
ESC[40m	Cor preto	Define cor de fundo como preto.
ESC[41m	Cor vermelho	Define cor de fundo como vermelho.
ESC[42m	Cor verde	Define cor de fundo como verde.
ESC[43m	Cor marrom	Define cor de fundo como marrom.
ESC[44m	Cor azul	Define cor de fundo como azul.
ESC[45m	Cor magenta	Define cor de fundo como magenta.
ESC[46m	Cor ciano	Define cor de fundo como ciano.
ESC[47m	Cor cinza claro	Define cor de fundo como cinza claro.

Tabela 2.10 - Código ANSI para tratamento de cores de fundo de tela.

Código escape	Função	Descrição
ESC[30m	Cor preto	Define cor de texto como preto.
ESC[31m	Cor vermelho	Define cor de texto como vermelho.
ESC[32m	Cor verde	Define cor de texto como verde.
ESC[33m	Cor marrom	Define cor de texto como marrom.
ESC[34m	Cor azul	Define cor de texto como azul.
ESC[35m	Cor magenta	Define cor de texto como magenta.
ESC[36m	Cor ciano	Define cor de texto como ciano.
ESC[37m	Cor cinza claro	Define cor de texto como cinza claro.
ESC[1;30m	Cor cinza escuro	Define cor de texto como cinza escuro.
ESC[1;31m	Cor vermelho claro	Define cor de texto como vermelho claro.
ESC[1;32m	Cor verde claro	Define cor de texto como verde claro.
ESC[1;33m	Cor amartelo	Define cor de texto como amarelo.
ESC[1;34m	Cor azul claro	Define cor de texto como azul claro.
ESC[1;35m	Cor magenta claro	Define cor de texto como magenta claro.
ESC[1;36m	Cor ciano	Define cor de texto como ciano claro.
ESC[1;37m	Cor branco	Define cor de texto como branco.

Tabela 2.11 - Código ANSI para tratamento de cores de texto na tela.

Os recursos de operação do modo *Terminal ANSI* são mais adequados para uso com o modo de programação da linguagem, não sendo muito prático no modo interativo. Veja a seguir um programa exemplo que faz o tratamento de cores de fundo (segundo plano) e cores de texto (primeiro plano) por meio do modo *Terminal ANSI*. Grave o código do programa seguinte com o nome "**`c02ex33.jl`**".

```
1  # Constantes para cores de texto e fundo
2
3  const black    =  0  # Preto
4  const blue     =  1  # Azul
5  const green    =  2  # Verde
6  const cyan     =  3  # Ciano
7  const red      =  4  # Vermelho
8  const magenta  =  5  # Magenta
9  const brown    =  6  # Marrom
10 const lgray    =  7  # Cinza claro
11
12 # Constantes para cores de fundo
13
14 const dgray    =  8  # Cinza escuro
15 const lblue    =  9  # Azul claro
```

```
16  const lgreen   = 10  # Verde claro
17  const lcyan    = 11  # Ciano claro
18  const lred     = 12  # Vermelho claro
19  const lmagenta = 13  # Magenta claro
20  const yellow   = 14  # Amarelo
21  const white    = 15  # Branco
22
23  function normal()
24      print("\e[0m")
25  end
25
27  function clear()
28      print("\e[2J\e[0;0H")  # limpa tela e posiciona cursor no canto esquerdo superior
29  end
30
31  function background(cor = "Int8")
32      if cor ==  0 print("\e[40m") end  # Preto
33      if cor ==  1 print("\e[44m") end  # Azul
34      if cor ==  2 print("\e[42m") end  # Verde
35      if cor ==  3 print("\e[46m") end  # Ciano
36      if cor ==  4 print("\e[41m") end  # Vermelho
37      if cor ==  5 print("\e[45m") end  # Magenta
38      if cor ==  6 print("\e[43m") end  # Marrom
39      if cor ==  7 print("\e[47m") end  # Cinza claro
40  end
41
42  function text(cor = "Int8")
43      if cor ==  0 print("\e[30m")   end  # Preto
44      if cor ==  1 print("\e[34m")   end  # Azul
45      if cor ==  2 print("\e[32m")   end  # Verde
46      if cor ==  3 print("\e[36m")   end  # Ciano
47      if cor ==  4 print("\e[31m")   end  # Vermelho
48      if cor ==  5 print("\e[35m")   end  # Magenta
49      if cor ==  6 print("\e[33m")   end  # Marrom
50      if cor ==  7 print("\e[37m")   end  # Cinza claro
51      if cor ==  8 print("\e[1;30m") end  # Cinza escuro
52      if cor ==  9 print("\e[1;34m") end  # Azul claro
53      if cor == 10 print("\e[1;32m") end  # Verde claro
54      if cor == 11 print("\e[1;36m") end  # Ciano claro
55      if cor == 12 print("\e[1;31m") end  # Vermelho claro
56      if cor == 13 print("\e[1;35m") end  # Magenta claro
57      if cor == 14 print("\e[1;33m") end  # Amarelo
58      if cor == 15 print("\e[1;37m") end  # Branco
59  end
60
61  function color(fundo = "Int8", texto = "Int8")
62      if fundo < 0 && fundo > 7 && texto < 0 && texto > 15
63          fundo = 0
64          texto = 7
65      end
66      normal()
67      background(fundo)
68      text(texto)
69  end
70
71  clear()
72  println("Teste de padrão de Cores com Código Numérico\n\n")
73
74  color(1,14)
75  println("Fundo: Azul // Texto: Amarelo")
76  println()
77
78  color(1,7)
```

```
79  println("Fundo: Azul // Texto: Cinza claro")
80  println()
81
82  color(4,14)
83  println("Fundo: Vermelho // Texto: Amarelo")
84  println()
85
86  color(0,2)
87  println("Fundo: Preto // Texto: Verde")
88  println()
89
90  color(7,12)
91  println("Fundo: Cinza claro // Texto: Vermelho claro")
92  println()
93
94  color(brown,white)
95  println("Fundo: Marrom // Texto: Branco")
96  println()
97
98  color(0,7)  # Volta fundo e texto para padrão
99
100 println()
101 print("Tecle <Enter> para encerrar programa... ")
102 readline()
```

O programa "**c02ex33.jl**" além dos códigos de *escape* indicados das tabelas de 2.8 até 2.11 faz uso de um código a mais identificado como "**\e[0m**" que tem por finalidade voltar o modo de operação ao normal após o uso de algum outro código de *escape*.

Neste programa para operar com um tipo de dado inteiro mais compacto está em uso nas linhas "**31**", "**42**" e "**61**" o indicativo de tipo "**Int8**" delimitado entre aspas inglesas como substituto da definição de tipo "**::Int64**", por exemplo.

Outro detalhe é a definição da função "**clear**" entre as linhas "**27**" e "**29**". Veja que são executados simultaneamente dois códigos de operação, sendo a limpeza da tela e o posicionamento do cursor no lado esquerdo superior, assim que a limpeza é efetivada.

2.8 - Documentação interna de código

Tão importante quanto codificar programas é fazer a documentação interna de seu código. A documentação auxilia a entender o que certa parte do código faz, tendo por finalidade descrever o comportamento de trechos do código. Essa ação torna-se útil, principalmente, no momento de efetivação das manutenções do programa em momentos futuros.

A ação de documentar códigos em linguagem Julia pode ser definida a partir de comentários de linhas ou comentários de parágrafos. Para cada forma de descrição a uma simbologia específica.

Os comentários de linha (também chamados comentários embutidos) são definidos a partir do uso do "**#**" tralha (ou *hash* em inglês) os quais foram já utilizados em algumas operações deste livro. Este tipo de comentário deve ser usado quando se deseja descrever o que certa linha de código efetua quando esta ação não é diretamente óbvia. Veja o modelo desta definição de dois exemplos de descrição indicados como *bom* e *ruim*.

```
x = x + 1  # incremento de "x" em "x" com mais "i" (ruim)
x = x + 1  # atualização do contador de entradas do usuário (bom)
```

Os comentários de linha devem ser separados por pelo menos com dois espaços em branco da expressão e devem ter um único espaço após o símbolo "**#**". Esse tipo de comentário deve ser escrito dentro de um limite máximo de linhas

(normalmente 80). Se o comentário for maior que o limite máximo de linha estabelecido este deve ser escrito acima do conteúdo a que faz referência. Veja o modelo desta definição.

```
# atualização do contador de entradas do usuário
x = x + 1
```

É importante quando se tem comentários de linhas, mantê-los devidamente atualizados de acordo com as mudanças realizadas no código. Mantenha sempre atenção a este quesito e atente-se a produzir o código da melhor forma possível para atender as boas práticas de programação na linguagem Julia.

Os comentários de linha podem ser usados alternativamente como elementos de identificação de grupos de ações no programa. No entanto, esses comentários devem expressar uma ideia completa. Caso o comentário seja uma frase ou sentença é importante que a primeira palavra seja iniciada com caractere maiúsculo e neste caso a frase ou sentença devem ser finalizadas com um símbolo de ponto, exceto esteja em sendo utilizada alguma palavra-chave da linguagem que comece com caractere minúsculo. No caso de se estar definindo frases ou sentenças curtas o ponto final pode ser omitido. Veja o modelo desta definição.

```
# Grupo de sub-rotinas para o controle de acesso aos dados do programa.
```

Os comentários de linha podem, algumas vezes, serem definidos em blocos. Um bloco consiste na definição de um ou mais parágrafos definidos a partir de frases ou sentenças completas. Cada frase ou sentença deve ser finalizada com uso de ponto. Veja o modelo desta definição.

```
# Programa "CALCULADORA"
# Aplicação destinada a execução de ações de adição, subtração, multiplicação e divisão.
# As rotinas de operação do programa são controladas com o uso da técnica de sub-rotinas.
```

Para a definição de blocos de comentários deve-se usar dos símbolos de abertura e fechamento de blocos "**#=**" e "**=#**", sendo que dentro do bloco o texto deve ser escrito delimitado a partir da terceira posição. Veja o modelo desta definição.

```
#=
   Programa "CALCULADORA"
   Aplicação destinada a execução de ações de adição, subtração, multiplicação e divisão
   definidas separadamente no código.
   As rotinas de operação do programa são controladas com o uso da técnica de sub-rotinas.
=#
```

De preferência para uso dos símbolos "**#=**" e "**=#**" quando houver a necessidade de definir comentários de blocos. Deixe o uso do símbolo "**#**" para a definição de comentários de linhas.

2.9 - Regras básicas de estilo

Neste trabalho procura-se seguir ao máximo as regras de estilo de escrita de códigos na linguagem Julia propostas pelo cânone da linguagem. No entanto, é oportuno ressaltar que há certos momentos em que seguir certas regras sintáticas e de estilo tornam-se impossíveis ou podem descaracterizar a ideia de ação a ser descrita em um programa como ocorreu no código do programa "**`c02ex33.jl`**" junto as funções "**`background`**" e "**`text`**" ao se fazer a simulação do estilo de uma estrutura de decisão seletiva do tipo *caso seja*.

Os detalhes apresentados neste tópico e no anterior a este são fundamentados e publicados no documento "*Blue: a Style Guide for Julia*" disponibilizado no endereço **https://github.com/invenia/BlueStyle** e se concentram nos elementos usados até o presente momento.

É importante considerar que um manual de estilo não é tacitamente a definição de regras obrigatórias a serem cegamente seguidas, mas são elementos que visam melhorar a consistência de escrita de código na linguagem. No entanto, está definido no guia de estilo Julia "*know when to be inconsistent*" (saiba quando ser inconsistente), ou seja, use sempre o bom senso. Então, veja alguns detalhes a serem considerados:

Comparações condicionais

Nas operações de comparações condicionais atente para o fato de que os operadores relacionais (==, >, < >=, <= e !=) não levam em conta os tipos de dados das contates. Desta forma, uma comparação como "**1.0 == 1**" (decimal == inteiro) é válida. Evite usar ruídos visuais em comparações condicionais. Neste caso use o estilo "**1.0 == 1.0**" ou "**1 == 1**". Respeitando o tipo de dado de uma variável quando a relação de comparação for *variável versus variável* ou *variável versus constante*.

Números

Os números de ponto flutuante devem sempre incluir um zero à esquerda e/ou à direita do valor. Não utilize as formas "**.1**", "**2.**" ou "**3.f0**". De preferência para os formatos "**0.1**", "**2.0**" e "**3.0f0**".

Desvio com operador condicional (operador ternário)

Desvios definidos com o uso de operador ternário devem ser usados apenas nos casos em que as ações possam ser expressas em uma única linha. Não encadeie operadores ternários, isso torna o código confuso. Se houver a necessidade de usar encadeamentos considere se expressar com os comandos "**if**", "**elseif**", "**else**" e "**end**".

Endentações nas estruturas de tomada de decisão e laços

Na definição de blocos subordinados de operações em tomadas de decisões com os comandos "**if**", "**elseif**", "**else**" e "**end**" e nas definições de laços com os comandos "**while**" e "**end**"; "**for**" e "**end**" use quatro espaços em branco e inicie o bloco na quinta posição. Os espaços em branco devem ser produzidos com a tecla de barra de espaço e não com a tecla "**<Tab>**". Use essa regra em todos os casos de blocos subordinados como no uso dos comandos "**begin/quote**" e "**end**".

Definição de iteração

Na definição de certa iteração com o símbolo "**:**" não use em seu entorno espaços em branco. Desta forma, use o estilo "**início:fim**" ou "**início:incremento:fim**" e não a forma "**início:fim**" ou "**início:incremento:fim**".

Laços for (para)

Na definição do fluxo iterativo de ação do laço nunca use o símbolo "**=**", em seu lugar use sempre o símbolo "**in**". Assim sendo, use "**for <var> = início:fim**" como "**for <var> in início:fim**".

Definição de nomes para funções

Os nomes de funções devem descrever a ação ou propriedade de sua operação independentemente do tipo de argumento em uso, pois o tipo do argumento deve fornecer essa informação. Uma função para o cálculo do fatorial de um número inteiro qualquer deve se chamar "**fatorial(n)**" e não chama-la "**fatorial_de_n(n)**", bem como "**fatorial(n::Int8)**" ao invés de chamar "**fatorial_de_n_inteiro(n)**". Os nomes das funções geralmente devem ser limitados a uma ou duas palavras minúsculas separadas por sublinhados (estilo "*snake case*"). No entanto, alguns programas anteriores utilizaram-se nomes com estilo "*camel case*"

Argumentos para funções

Ao chamar uma função, sempre separe seus argumentos de palavra-chave de seus argumentos posicionais com um ponto e vírgula. Isso evita erros em casos de ambiguidade de nomes.

Espaço em branco

De todas as estruturas de estilo da linguagem as recomendações para uso de espaços em branco é a mais extensa e são em seguida apresentadas:

- Evite espaços em branco estranhos entre parênteses, colchetes ou chaves. Use o estilo "**nome(x, y)**" e não a forma "**nome (x, y)**";
- Evite espaços em branco estranhos antes de vírgulas ou ponto e vírgulas. Use o estilo "**@show(x, y); x, y = y, x**" e não a forma "**@show(x , y) ; x , y = y , x** ";
- Evite espaços em branco em intervalos definidos com o símbolo "**:**". Use colchetes para definir de forma clara expressões em ambos os lados, ou seja, use o estilo "**[1:5]**" e não as formas "**[1: 5]**", "**[1 :5]**" ou "**[1 : 5]**". Use parênteses na definição de valores de intervalos compostos, ou seja, use o estilo "**[(lower + offset):(upper + offset)]**" e nunca a forma "**[lower + offset:upper + offset]**" que é considerada como "**[lower + (offset:upper) + offset]**";
- Evite usar mais de um espaço ao redor de um operador de atribuição (ou outro) para alinhá-lo com outro ou mesmo não usar espaço nenhum. Use o estilo "**x = 1**" e não as formas "**x=1**", "**x= 1**", "**x =1**" ou "**x = 1**";
- Envolva a maioria dos operadores binários com um espaço em branco de cada lado, como os operadores de: atribuição "**=**", atualização "**=+**, **=-**, **=*** e **=/**" e comparação "**==**, **<**, **>**, **<=**, **>=** e **!=**". Há alguns operadores binários que podem ser excluídos desta recomendação, sendo os operadores: de intervalo "**:**"; racional "**//**"; exponenciação "**^**" e o operador de atribuição "**=**" exclusivo a definição de argumentos opcionais em funções. Use os estilos "**x = a + 1**", "**i =+ 1**" e "**x^2 > 5**" e não as formas "**x=a+1**", "**i =+1**" e "**x^2>5**";
- Evite usar espaços em branco entre operadores unários de manutenção e inversão de sinal. Use o estilo "**-1**" e não a forma "**- 1**".

Linhas em branco

Evite linhas vazias (em branco) desnecessárias e em excesso. Linhas em branco para separar sub-rotinas e comentários de linha, bem como blocos de códigos são aceitáveis.

Quantidade de caracteres por linha

A extensão da quantidade de caracteres recomendada por linha é de no máximo 92 caracteres, mas em média considere 80 caracteres. Qualquer forma escrita acima da nonagésima segunda posição deve ser escrita na linha seguinte.

Chamadas de função que não podem caber em uma única linha dentro do limite de linha devem ser divididas de forma que as linhas contendo os colchetes de abertura e fechamento sejam recuadas no mesmo nível, enquanto os parâmetros da função são recuados um nível a mais. Na maior parte dos casos, os argumentos e/ou palavras-chave devem ser colocados em linhas separadas.

Outros detalhes de estilo serão usados quando forem necessários, mas independentemente disto não deixe de consultar o documento original e oficial.

CAPÍTULO 3

COLEÇÕES DE DADOS

Este capítulo mostra detalhes sobre o uso e manipulação de coleções de dados na forma de estruturas contiguas, ordenadas e indexadas de elementos. Dentro deste aspecto, a linguagem Julia opera com alguns recursos baseados em arranjos (arrays): listas de dados delimitada entre colchetes; tuplas (tuples): lista de dados delimitada entre parênteses, dicionários (dictionaries): lista de pares de dados delimitados entre parênteses e conjuntos (sets): lista de valores únicos separados por vírgula delimitados entre colchetes. Os dicionários e conjuntos são tratados com funcionalidades especiais chamadas construtores. Acrescenta-se ainda o uso de estruturas e tratamento de cadeias de caracteres.

3.1 - Arranjos

Um arranjo, no universo da linguagem Julia, caracteriza-se por ser uma coleção de dados heterogêneos delimitados entre colchetes que quando separados por vírgulas ou por ponto-e-vírgula são estruturas do tipo "**`Vector`**" (estruturas mais flexíveis de apenas uma dimensão) e quando separados por espaços em branco com ponto-e-vírgula são estruturas do tipo "**`Matrix`**" (estruturas menos flexíveis de duas dimensões). Os elementos nos arranjos são distribuídos de acordo com o posicionamento ordinal, ou seja, o primeiro elemento está no índice "**`1`**", o segundo no índice "**`2`**", e assim por diante.

Observe a definição de um arranjo do tipo vetor, dado "**`Vector`**", formado por uma quantidade "**`n`**" de linhas com a indicação de uma única coluna de dados. Este modo de apresentação é conhecido tecnicamente pelo nome de *column major* (coluna principal).

```
julia> [1, 2, 3]  # vetor composto por elementos inteiros
5-element Vector{Int64}:
 1
 2
 3

julia> Float64[1, 2, 3]  # vetor composto por elementos decimais
3-element Vector{Float64}:
 1.0
 2.0
 3.0

julia> [1; 2; 3]  # vetor composto por elementos inteiros
5-element Vector{Int64}:
 1
 2
 3
```

Observe a definição de um arranjo do tipo matriz, dado "**Matrix**", formado por uma quantidade "**n**" de linhas com quantidade "**m**" de colunas de dados.

```
julia> [1 2 3 4 5]  # matriz composta por elementos inteiros com 1 linha e 5 colunas
1×5 Matrix{Int64}:
 1  2  3  4  5

julia> [1 2 3 4 5; 6 7 8 9 0]  # matriz composta por elementos inteiros com 2 linhas e 5 colunas
2×5 Matrix{Int64}:
 1  2  3  4  5
 6  7  8  9  0
```

3.1.1 - Arranjo com uma dimensão: vetor

Veja exemplos de manipulação de dados em estruturas do tipo arranjo de uma dimensão para a manipulação de dados na forma de vetores. A maior parte dos exemplos são fornecidos a partir de listas de valores numéricos inteiros, mas podem ser usados sobre qualquer tipo de dado. Veja alguns exemplos de vetores com dados lógicos, de ponto flutuante, de caracteres, de cadeia e de qualquer tipo quando possui mais de um tipo no mesmo arranjo.

```
julia> [true, false]
2-element Vector{Bool}:
 1
 0

julia> [1.0, 2.0, 3.5]
3-element Vector{Float64}:
 1.0
 2.0
 3.5

julia> ['a', 'b', 'c']
3-element Vector{Char}:
 'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)
 'b': ASCII/Unicode U+0062 (category Ll: Letter, lowercase)
 'c': ASCII/Unicode U+0063 (category Ll: Letter, lowercase)

julia> ["a", "b", "c"]
3-element Vector{String}:
 "a"
 "b"
 "c"

julia> [1, 2.5, 'a', "b"]
4-element Vector{Any}:
 1
 2.5
  'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)
  "b"
```

Arranjos do tipo vetor podem ser definidos a partir da definição de faixas de valores definidos a partir do uso das funções "**collect**" e "**range**" indicadas a seguir:

```
julia> collect(1:5)  # gera valores de "1" a "5"
5-element Vector{Int64}:
 1
 2
 3
 4
 5

julia> collect(1:2:5)  # gera valores de "1" a "5" de "2" em "2"
3-element Vector{Int64}:
 1
 3
 5

julia> range(start=1, stop=5)
1:5

julia> range(1, 5)
1:5

julia> range(start=1, step=2, stop=5)
1:2:5

julia> range(1, step=2, stop=5)
1:2:5

julia> range(1, 3, step=2)
1.0:2.0:3.0

julia> range(1, step=2, length=50)
1:2:99

julia> collect(range(start=1, stop=5))
5-element Vector{Int64}:
 1
 2
 3
 4
 5

julia> range(1, 2, 5)
1.0:0.25:2.0

julia> collect(range(1, 2, 5))
5-element Vector{Float64}:
 1.0
 1.25
 1.5
 1.75
 2.0
```

A partir de uma visão geral observe a seguir alguns exemplos de ações sobre o uso de vetor definido a partir da variável "*X1*".

```
julia> x1 = [1, 4, 3, 2, 5]  # criação de vetor de elementos inteiros chamado "x1"
5-element Vector{Int64}:
 1
 4
 3
 2
 5

julia> x1  # apresentação do conteúdo do vetor armazenado na variável "x1"
5-element Vector{Int64}:
 1
 4
 3
 2
 5

julia> eltype(x1)  # apresenta o tipo de dado do vetor
Int64
```

Uma ação que pode ser aplicada sobre vetores e matrizes, os quais podem ser chamados genericamente de estruturas, é a obtenção do seu tamanho, ou seja, a quantidade de elementos que a estrutura possui. Isso é conseguido com o uso da função "**length(<estrutura>)**".

```
julia> length(x1)
5
```

Os elementos de uma estrutura, seja vetor ou matriz, são posicionados dentro da estrutura de forma ordinal. Para acessar ou alterar o conteúdo de certa posição basta indicar o nome da estrutura mencionado entre colchetes o número da posição que se deseja ter acesso.

```
julia> first(x1)  # mostra o primeiro elemento do vetor indicado
1

julia> x1[3]
3

julia> x1[3] = 7
7

julia> x1[3]
7

julia> x1
5-element Vector{Int64}:
 1
 4
 7
 2
 5
```

```
julia> first(x1, 3)  # mostra os três primeiros elementos do vetor indicado
3-element Vector{Int64}:
 1
 4
 7

julia> last(x1)  # mostra o último elemento do vetor indicado
5

julia> last(x1, 3)  # mostra os três últimos elementos do vetor indicado
3-element Vector{Int64}:
 7
 2
 5

julia> 4 in x1  # verifica se o elemento indicado "4" existe no vetor "x1"
true
```

Os elementos existentes em uma estrutura vetor ou matriz podem ser copiados diretamente para outra estrutura a partir da utilização da função "**copy(<estrutura>)**". Veja a cópia dos elementos do vetor "*X1*" multiplicados por "**2**" para o vetor "*Y1*".

```
julia> y1 = copy(x1 * 2)
5-element Vector{Int64}:
  2
  8
 14
  4
 10
```

Quando se tem mais de uma estrutura é possível realizar algumas operações lógicas e aritméticas entre seus elementos. Veja a seguir algumas dessas operações, desde que tais operações sejam realizadas em estruturas da mesma categoria: vetor com vetor ou matriz com matriz. Essas operações entre vetores e matrizes não é permitida.

```
julia> x1 == y1  # verifica se dois vetores são ou não iguais
false

julia> x1 + y1  # soma os elementos um a um das estruturas em uso
5-element Vector{Int64}:
  3
 12
 21
  6
 15
```

Uma operação muito comum com estruturas de dados vetores ou matrizes é fazer a junção de duas ou mais estruturas de mesmo tipo de forma que se tenha uma única estrutura formada por todos os dados. A junção é conseguida com o uso do operador "**;**" (ponto e vírgula). Para executar esta ação usa-se à operação seguinte

```
julia> z1 = [x1; y1]
10-element Vector{Int64}:
  1
  4
  7
  2
  5
  2
  8
 14
  4
 10
```

É possível ainda concatenar vetores para formar matrizes, por exemplo, concatenar os elementos dos vetores "*X1*" e "*Y1*" como colunas com os elementos dos vetores "*Y1*" e "*X1*" com as linhas. Observe esta ocorrência.

```
julia> z2 = [x1 y1; y1 x1]
10×2 Matrix{Int64}:
  1   2
  4   8
  7  14
  2   4
  5  10
  2   1
  8   4
 14   7
  4   2
 10   5
```

Quando se opera a partir de estruturas de vetores ou matrizes é possível fazer algumas operações de fatiamento. O fatiamento é uma forma de se obter a apresentação de elementos a partir de alguns cortes indicados na manipulação de índices dentro dos colchetes. Neste tipo de operação se estiver em uso uma matriz esta será apresentada como se fosse um vetor. Veja alguns exemplos.

```
julia> x1[2:4]  # apresentar elementos da posição "2" até posição "4"
3-element Vector{Int64}:
 4
 7
 2

julia> x1[1:2:5]  # apresentar elementos da posição "1" até posição "5" de "2" em "2"
3-element Vector{Int64}:
 1
 7
 5

julia> x1[4:end]  # apresentar elementos da posição "4" até o final do vetor
2-element Vector{Int64}:
 2
 5
```

```
julia> x1[3:end-1]  # apresentar elementos da posição "3" até a penúltima posição
2-element Vector{Int64}:
 7
 2
```

Os elementos existentes dentro de um vetor exclusivamente podem ser acrescidos ou removidos. Já os elementos de uma matriz não podem realizar tais operações, pois matrizes são estruturas estáticas não possuindo elasticidade.

Veja como exemplo de manipulação de vetor ação de inclusão de elemento no final da estrutura de dados exclusivamente do tipo vetor com a função "**push!(<estrutura>, <conteúdo>)**" ou use com a mesma sintaxe a função "**append!(<estrutura>, <conteúdo>)**".

```
julia> push!(x1, 8)
6-element Vector{Int64}:
 1
 4
 7
 2
 5
 8
```

Caso deseje acrescentar elemento no início do vetor use a função "**pushfirst!(<estrutura>, <conteúdo>)**".

```
julia> pushfirst!(x1, 9)
7-element Vector{Int64}:
 9
 1
 4
 7
 2
 5
 8
```

Além de acrescentar elementos em uma estrutura do tipo vetor é possível realizar a remoção de elementos. Para remover o último elemento de um vetor use a função "**pop!(<estrutura>)**".

```
julia> pop!(x1)
8
```

Caso deseje remover elemento no final do vetor use a função "**popfirst!(<estrutura>)**".

```
julia> popfirst!(x1)
9
```

Para remover elemento de posição específica use a função "**deleteat!(<estrutura>, <posição>)**". Veja um exemplo para a remoção do elemento "**3**".

```
julia> deleteat!(x1, 3)
4-element Vector{Int64}:
 1
 4
 2
 5
```

Os elementos de vetor podem ser classificados de forma crescente e decrescente se forem numéricos e ascendente e descendente se forem alfanuméricos. Para realizar esta operação use uma das funções "**sort(<estrutura> [, <argumentos>])**" ou "**sort!(<estrutura> [, <argumentos>])**".

A função "**sort**" efetua a classificação de forma lógica, ou seja, a classificação apresentada é apenas visual mantendo os elementos do vetor na ordem original que se encontram. Já a função "**sort!** " efetua a classificação de forma física modificando a posição dos elementos dentro do vetor.

```
julia> sort(x1)  # classificação lógica crescente dos elementos
4-element Vector{Int64}:
 1
 2
 4
 5


julia> sort(x1; rev=true)  # classificação lógica decrescente dos elementos
4-element Vector{Int64}:
 5
 4
 2
 1

julia> sort!(x1)  # classificação física crescente dos elementos
4-element Vector{Int64}:
 1
 2
 4
 5
```

As funções de classificação de elementos podem operar além do argumento "**rev**" com outros argumentos, destacando-se o argumento "**alg**" onde é possível escolher o padrão dos algoritmos de classificação desejado entre as opções: "**QuickSort**" ou "**MergeSort**".

```
julia> sort!(x1; rev=true, alg=QuickSort)
4-element Vector{Int64}:
 5
 4
 2
 1

julia> sort(x1; alg=MergeSort)
4-element Vector{Int64}:
 1
 2
 4
 5
```

Uma estrutura de dados vetor pode possuir elementos repetidos. Mas, há vezes em que não se deseja ver esses elementos repetidos. Neste caso, usa-se a função "**unique(<estrutura>)**". Há uma versão desta função chamada "**unique!(<estrutura>)**" que retira fisicamente os elementos do vetor. Para um teste em estrutura de dados do tipo vetor acrescente primeiro um elemento que seja repetido e em seguida execute as operações indicadas.

```
julia> push!(x1, 5)  # acrescentar elemento repetido na estrutura de vetor
5-element Vector{Int64}:
 5
 4
 2
 1
 5

julia> unique(x1)  # remove logicamente elementos repetidos no vetor
4-element Vector{Int64}:
 5
 4
 2
 1

julia> unique!(x1)  # remove fisicamente elementos repetidos no vetor
4-element Vector{Int64}:
 5
 4
 2
 1
```

Os elementos de uma estrutura vetor podem ter a posição de seus elementos revertida, ou seja, apresentados de forma invertida. Para esta operação considere a função "**reverse(<estrutura>)**"

```
julia> reverse(x1)
4-element Vector{Int64}:
 1
 2
 4
 5

julia> x1[end:-1:1]  # mesmo efeito da função "reverse(x1)"
4-element Vector{Int64}:
 1
 2
 4
 5
```

Outra operação importante é permitir que se pesquise elementos dentro do vetor. Assim sendo, considere o uso da função "**in(<elemento>, <estrutura>)**" que informa verdadeiro ou falso caso encontre ou não encontre o elemento indicado.

```
julia> in(3, x1)
false
```

Assim como é possível saber se determinado elemento faz ou não parte de uma estrutura é possível também apresentar a posição de certo elemento na estrutura. Para tanto, use a função "**findall(<elemento>, <estrutura>)**".

```
julia> findall(vlr -> vlr == 5, x1)
1-element Vector{Int64}:
 1
```

A função "**findall**" efetua sua pesquisa a partir da expressão "**vlr -> vlr == 5**", onde o elemento a ser procurado "**5**" é verificado a partir em cada posição da estrutura. A função partir de "**vlr -> vlr**" percorre toda a estrutura da primeira à última posição e caso não encontre o que busca retorna o resultado "**Int64[]**", se estiver em uso valores numéricos inteiros.

Veja algumas outras funções que podem ser usadas na manipulação de dados:

```
julia> maximum(x1)  # mostra o elemento de maior valor no vetor
5

julia> minimum(x1)  # mostra o elemento de menor valor no vetor
1

julia> sum(x1)  # mostra o somatório dos elementos do vetor
12

julia> prod(x1)  # mostra o produto dos elementos do vetor
12
```

Essas próximas funções só podem operar estruturas do tipo vetor.

```
julia> empty!(y1)  # limpa (remove) os elementos de um vetor
Int64[]

julia> isempty(y1)  # indica se um vetor está vazio (true) ou não vazio (false)
true

julia> resize!(x1, 8)  # redefine o tamanho de um vetor
8-element Vector{Int64}:
 5
 4
 2
 1
 0
 0
 0
 0
```

Observe que no conjunto de funções apresentadas as funções que possuem em seu nome o símbolo "**!**" são funções que alteram fisicamente os elementos dentro da estrutura. As funções de mesmo nome sem este símbolo operam suas ações apenas de forma lógica.

Como comentado a estrutura de uma matriz é mais restritiva (e será abordada em mais detalhes no próximo tópico) que a estrutura de um vetor. Imagine que haja a necessidade de realizar certa alteração em elementos de uma matriz. Neste caso, uma alternativa viável é transformar a matriz em um vetor, operacionalizar a ação necessária e em seguida transformar em matriz o vetor novamente. Para fazer esta operação de transição entre estruturas é ideal usar em conjunto a função que efetiva a cópia de elementos "**copy**". Atente nos exemplos a seguir as indicações do arranjo em vermelho.

```
julia> x2 = copy(permutedims(x1))  # criar matriz 1D a partir de um vetor
1×8 Matrix{Int64}:
 5  4  2  1  0  0  0  0
```

```
julia> x2
1×8 Matrix{Int64}:
 5  4  2  1  0  0  0  0

julia> x3 = copy(vec(x2))  # criar vetor a partir de uma matriz 1D
8-element Vector{Int64}:
 5
 4
 2
 1
 0
 0
 0
 0

julia> x3
8-element Vector{Int64}:
 5
 4
 2
 1
 0
 0
 0
 0
```

Veja que a função "**permutedims**" foi usada para gerar um arranjo matriz de uma dimensão com uma linha e certo número de colunas a partir da indicação de um arranjo vetor. A ação inversa foi produzida com o uso da função "**vet**".

As operações com arranjos podem ser definidas sobre outros dados, como os dos tipos cadeia e caractere brevemente indicados. Veja alguns exemplos de tratamento de dados sobre cadeias.

```
julia> texto = "Computador"
"Computador"

julia> texto[1]
'C': ASCII/Unicode U+0043 (category Lu: Letter, uppercase)

julia> println(texto[1])
C

julia> [x for x in texto]
10-element Vector{Char}:
 'C': ASCII/Unicode U+0043 (category Lu: Letter, uppercase)
 'o': ASCII/Unicode U+006F (category Ll: Letter, lowercase)
 'm': ASCII/Unicode U+006D (category Ll: Letter, lowercase)
 'p': ASCII/Unicode U+0070 (category Ll: Letter, lowercase)
 'u': ASCII/Unicode U+0075 (category Ll: Letter, lowercase)
 't': ASCII/Unicode U+0074 (category Ll: Letter, lowercase)
 'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)
 'd': ASCII/Unicode U+0064 (category Ll: Letter, lowercase)
 'o': ASCII/Unicode U+006F (category Ll: Letter, lowercase)
 'r': ASCII/Unicode U+0072 (category Ll: Letter, lowercase)
```

```
julia> println([x for x in texto])
['C', 'o', 'm', 'p', 'u', 't', 'a', 'd', 'o', 'r']

julia> permutedims([x for x in texto])
×10 Matrix{Char}:
 'C'  'o'  'm'  'p'  'u'  't'  'a'  'd'  'o'  'r'

julia> length(texto)
10

julia> texto[1:3]
"Com"

julia> texto[4:7]
"puta"

julia> texto[8:end]
"dor"

julia> texto[1:2:10]
"Cmuao"

julia> println(unique(texto))
['C', 'o', 'm', 'p', 'u', 't', 'a', 'd', 'r']

julia> reverse(texto)
"rodatupmoC"

julia> in('C', texto)
true

julia> findall(vlr -> vlr == 'C', texto)
1-element Vector{Int64}:
 1

julia> maximum(texto)
'u': ASCII/Unicode U+0075 (category Ll: Letter, lowercase)

julia> isempty(texto)
false
```

As funções com uso do símbolo "**!**", a função "**sort**" e as funções que efetuam cálculos sobre os elementos de vetores não funcionam com dados cadeia/caractere.

Os elementos de um vetor (ou matriz) podem ser operacionalizados com o uso de operações aritméticas diversas, mas para isso certo cuidado deve ser tomado. Por exemplo, multiplicar por certo valor todos os elementos do arranjo "*X1*". Neste caso se usa um operador vetorial de multiplicação, veja o seguinte exemplo.

Operação vetorial caracteriza-se por ser uma ação de cálculo aritmético aplicada a cada elemento de um arranjo numérico de dados.

```
julia> x1 = x1 .* 3  # ou: x1 = x1 * 3
5-element Vector{Int64}:
  3
 12
 21
  6
 15
```

Note que para aplicar certa operação sobre os elementos de um arranjo (operações vetoriais) usa-se antes do operador aritmético desejado o símbolo "**.**". Essa forma de expressão deixa caracterizado quando a operação aritmética é aplicada sobre arranjos diferenciando-a das operações aritméticas convencionais, aplicada somente a valores. Para realizar operações vetoriais pode-se usar os operadores apontados na tabela 3.1.

Operador	Operação vetorial
.^	Exponenciação
,^ (y / n)	Radiciação
./	Divisão
.\	Divisão inversa
.//	Fração
.%	Resto da divisão
.*	Multiplicação
.+	Adição
.-	Subtração

Tabela 3.1 - Operadores aritméticos vetoriais.

Operações vetoriais no contexto lógico também podem ser aplicadas com o uso de operadores relacionais e/ou lógicos, ressaltando que este tipo de operação retorna como resposta bits de sinalização "**1**" e "**0**" e não resultados lógicos do tipo "*verdadeiro*" e "*falso*". Veja-os indicados na tabela 3.2 e em seguida alguns exemplos de uso.

Operador	Operação vetorial
.>	Relacional maior que
.<	Relacional menor que
.>=	Relacional maior ou igual a
.<=	Relacional menor ou igual a
.==	Relacional igual a
.!=	Relacional não igual a
.|	Lógico de disjunção
.&	Lógico de conjunção
.!	Lógico de negação

Tabela 3.2 - Operadores relacionais vetoriais.

```
julia> 3 .<= [1, 2, 3, 4, 5]  # 0 = false | 1 = true
5-element BitVector:
 0
 0
 1
 1
 1
```

```
julia> [1, 2, 3, 4, 5] .== 4
5-element BitVector:
 0
 0
 0
 1
 0

julia> 3 .!= [1, 2, 3, 4, 5]
5-element BitVector:
 1
 1
 0
 1
 1

julia> ([1, 2, 3, 4, 5] .>= 2) .& ([1, 2, 3, 4, 5] .<= 4)  # conjunção vetorial
5-element BitVector:
 0
 1
 1
 1
 0

julia> .!(([1, 2, 3, 4, 5] .< 0) .| ([1, 2, 3, 4, 5] .> 4))  # não disjunção vetorial
5-element BitVector:
 1
 1
 1
 1
 0
```

Outro recurso disponibilizado na linguagem é a possibilidade de se realizar somatório cumulativo e produto cumulativo dos elementos de um vetor. Para tanto, execute a instrução:

```
julia> cumsum(x1)
5-element Vector{Int64}:
  3
 15
 36
 42
 57

julia> cumprod(x1)
5-element Vector{Int64}:
     3
    36
   756
  4536
 68040
```

Veja que a cada soma cumulativa ou produto cumulativo ocorre sobre a operação sobre a posição do próximo elemento, elemento a elemento. Observe que a cada operação é registrado na posição o resultado do valor anterior somado ou multiplicado com o valor atual mantendo na posição atual o resultado calculado.

Além das funcionalidades expostas para o tratamento de operações de gerenciamento de dados dentro dos arranjos, sejam vetores ou matrizes, há a possibilidade de operá-los de acordo com o que é exposto no estudo da teoria de conjuntos a partir de três operações típicas: união, Intersecção e diferença entre conjuntos. Veja essas operações a partir da definição de duas estruturas "*A*" e "*B*" na forma de vetores.

```
julia> a = [1, 2, 3, 4]
4-element Vector{Int64}:
 1
 2
 3
 4

julia> b = [3, 4, 5, 6]
4-element Vector{Int64}:
 1
 3
 3
 4
```

Veja na sequência os resultados das ações de união, Intersecção e diferença.

```
julia> union(a, b)
6-element Vector{Int64}:
 1
 2
 3
 4
 5
 6
julia> intersect(a, b)
2-element Vector{Int64}:
 3
 4

julia> setdiff(a, b)
2-element Vector{Int64}:
 1
 2
```

Se estiver em uso arranjos na forma de matrizes a apresentação dos resultados das operações de união, Intersecção e diferença serão iguais as respostas retornadas para os arranjos do tipo vetor.

Já que está sendo abordando o tratamento de dados de acordo com a teoria de conjuntos torna-se interessante falar sobre *compreensões*. É sabido que um conjunto pode ser descrito de duas formas: por extensão e por compreensão, observe:

```
A = {2, 4, 6, 8}              -> descrição de conjunto por extensão
A = {x / x é par e 1 < x < 9} -> descrição de conjunto por compreensão
```

A linguagem Julia permite o uso do efeito de compreensão para a definição de arranjos do tipo vetor. Veja alguns exemplos:

```
julia> [x for x in 1:9 if iseven(x)]  # valores pares de 1 até 9
4-element Vector{Int64}:
 2
 4
 6
 8

julia> [x for x in x1 if iseven(x)]
6-element Vector{Int64}:
 4
 2
 0
 0
 0
 0

julia> [x for x in 1:9 if iseven(x) && x > 2 && x < 8]
2-element Vector{Int64}:
 4
 6

julia> [x ^ 2 for x in x1]
8-element Vector{Int64}:
 25
 16
  4
  1
  0
  0
  0
  0

julia> [isodd(x) for x in x1]  # indica "1" para valor ímpar existente
8-element Vector{Bool}:
 1
 0
 0
 1
 0
 0
 0
 0

julia> [iseven(x) for x in 1:9]
9-element Vector{Bool}:
 0
 1
 0
 1
 0
 1
 0
 1
 0
```

```
julia> Any[iseven(x) for x in 1:9]
9-element Vector{Any}:
 false
  true
 false
  true
 false
  true
 false
  true
 false

julia> [for x in 1:9 if iseven(x)]
9-element Vector{Any}:
 false
  true
 false
  true
 false
  true
 false
  true
 false

julia> sum([x^2 for x in 1:5])  # somatório dos quadrados de 1 até 5
55

julia> [x + y for x in 1:5 for y in 3:4]
10-element Vector{Int64}:
 4
 5
 5
 6
 6
 7
 7
 8
 8
 9

julia> Float64[x * y for x in 2:3 for y in 3:4]
4-element Vector{Float64}:
  6.0
  8.0
  9.0
 12.0
```

É possível fazer uso do efeito similar a compreensão junto ao uso da função "**collect**":

```
julia> collect(x * y for x in 2:3 for y in 3:4)
4-element Vector{Int64}:
  6
  8
  9
 12
```

Além das funcionalidades de tratamento das operações sobre a teoria de conjuntos há mais algumas muito importantes e usadas e para realizar ações de redução - uso da função "**reduce**" para encontrar um valor cumulativo com base nos elementos de um arranjo, mapeamento - uso da função "**map**" para mapear todos os elementos de um arranjo e filtragem - uso da função "**filter**" para retirar do arranjo elemento baseado em certa condição.

```
julia> reduce(+, [1, 2, 3])
6

julia> reduce(+, x1)
12

julia> reduce(*, [1, 2, 3])
6

julia> reduce(/, [1, 2, 3])
0.16666666666666666

julia> reduce(-, x1)
-2

julia> map(x -> x * 2, [1, 2, 3])
3-element Vector{Int64}:
 2
 4
 6

julia> map(x -> x ^ 2, [1, 2, 3])
3-element Vector{Int64}:
 1
 4
 9

julia> map(x -> x * 2, x1)
8-element Vector{Int64}:
 10
  8
  4
  2
  0
  0
  0
  0

julia> filter(isodd, x1)
2-element Vector{Int64}:
 5
 1
```

```
julia> filter(iseven, x1)
6-element Vector{Int64}:
 4
 2
 0
 0
 0
 0
```

As operações de redução de arranjos podem além da função "**reduce**" serem operacionalizadas com as funções de dobra "**foldr**" - dobra a direita e "**foldl**" - dobra a esquerda.

```
julia> foldr(+, [1, 2, 3])
6

julia> foldr(*, [1, 2, 3])
6

julia> foldr(*, [1, 2, 3]; init = 2)
12

julia> foldr(+, x3)
12

julia> foldr(+, x3;  init = 1)
13

julia> foldl(+, [1, 2, 3])
6

julia> foldl(+, x3)
12
```

As funcionalidades de tratamento de redução, filtragem, dobras e mapeamento podem ser usadas sem problemas com os arranjos do tipo vetor e do tipo matriz.

Os arranjos de vetores podem ser inicializados de cinco maneiras básicas antes de serem usados. É possível inicializar vetores com valores zeros, uns ou indefinidos. Para esta operação considere as seguintes formas:

```
<variável> = Vector{tipo de dado}(undef, <quantidade>)
<variável> = zeros(<quantidade>)
<variável> = ones(<tipo de dado>, <quantidade>)
<variável> = trues(<quantidade>)
<variável> = falses(<quantidade>)
```

O indicativo *tipo de dado* se refere ao tipo de dado em uso e *quantidade* se refere a *quantidade* de elementos previamente definidos no vetor.

```
julia> a1 = Vector{Float64}(undef, 5)
5-element Vector{Float64}:
 0.0
 1.39362725e-315
 1.393191525e-315
 0.0
 8.95936824e-316

julia> a2 = zeros(5)
5-element Vector{Float64}:
 0.0
 0.0
 0.0
 0.0
 0.0

julia> a3 = ones(Int64, 5)
5-element Vector{Int64}:
 1
 1
 1
 1
 1

julia> a4 = trues(3)  # cria vetor com valores verdadeiros
3-element BitVector:
 1
 1
 1

julia> a5 = falses(3)  # cria vetor com valores falsos
3-element BitVector:
 0
 0
 0
```

Caso deseje inicializar um vetor com valores diferentes aos de definição (zeros, uns e indefinidos) é necessário considerar o uso da função "**`fill!`** " como a seguir:

```
julia> fill!(a1, 10)  # preencher com valor "10" o vetor "a1"
5-element Vector{Float64}:
 10.0
 10.0
 10.0
 10.0
 10.0

julia> a1
5-element Vector{Float64}:
 10.0
 10.0
 10.0
 10.0
 10.0
```

São muitas as operações que podem ser aplicadas sobre estruturas de dados do tipo arranjo. Para encerrar a apresentação de manipulação de vetores considere o próximo exemplo:

```
julia> aaa = [1, "abelha", 1.5, true]
4-element Vector{Any}:
    1
     "abelha"
    1.5
 true
```

Foi demonstrado anteriormente como inserir e remover elementos de um arranjo e dentro desta temática há a possibilidade de optar-se em manter apenas alguns elementos de um arranjo. Observe a próxima instrução:

```
julia> keepat!(aaa, 2)  # manter em "aaa" apenas o segundo elemento
1-element Vector{Any}:
 "abelha"

julia> aaa
1-element Vector{Any}:
 "abelha"

julia> nomes = ["José", "Maria", "Silvia", "Silvio", "João", "Sandra"]
6-element Vector{String}:
"José"
 "Maria"
 "Silvia"
 "Silvio"
 "João"
 "Sandra"

julia> nomes
6-element Vector{String}:
 "José"
 "Maria"
 "Silvia"
 "Silvio"
 "João"
 "Sandra"

julia> keepat!(nomes, [2, 5])  # manter em "nomes" o segundo e quinto elementos
2-element Vector{String}:
 "Maria"
 "João"
```

A partir da definição de uma variável do tipo arranjo de vetor é possível efetuar as operações anteriormente apresentadas para a manipulação de dados dentro da estrutura.

3.1.2 - Arranjo com duas dimensões: matriz

Veja exemplos de manipulação de dados em estruturas do tipo arranjo de duas dimensões para a manipulação de dados na forma de matrizes. Observe a seguir alguns exemplos de ações sobre o uso de matriz definida a partir da variável "*X4*" e outros exemplos.

```
julia> x4 = [1 2 3; 4 5 6; 7 8 9]  # matriz com 3 linha e 3 colunas
3×3 Matrix{Int64}:
 1  2  3
 4  5  6
 7  8  9

julia> [1 2 3;
        4 5 6;
        7 8 9]
3×3 Matrix{Int64}:
 1  2  3
 4  5  6
 7  8  9

julia> reverse(x4)
3×3 Matrix{Int64}:
 9  8  7
 6  5  4
 3  2  1

julia> eltype(x4)
Int64

julia> ndims(x4)  # apresenta a quantidade de dimensões de um arranjo
2

julia> size(x4)  # apresenta tupla com o tamanho em linhas e colunas de "x4"
(3, 3)

julia> size(x1)  # apresenta tupla com o tamanho em linhas e colunas de "x1"
(5,)

julia> x5 = [1 2 3; 4 5 6]
2×3 Matrix{Int64}:
 1  2  3
 4  5  6

julia> size(x4, 2)  # apresenta o tamanho em linhas a partir da indicação da coluna
3

julia> size(x5)
(2, 3)

julia> linhas, colunas = size(x5)
(2, 3)

julia> linhas
2

julia> colunas
3
```

```
julia> size(x5, 1)  # para obter o número de linhas
2

julia> size(x5, 2)  # para obter o número de colunas
3

julia> [[1 2 3]; [4 5 6]; [7 8 9]]  # mesmo efeito de [1 2 3; 4 5 6; 7 8 9]
3×3 Matrix{Int64}:
 1  2  3
 4  5  6
 7  8  9

julia> [[[1 2 3]]; [[4 5 6]]; [[7 8 9]]]  # vetor de matriz
3-element Vector{Matrix{Int64}}:
 [1 2 3]
 [4 5 6]
 [7 8 9]

julia> [[[1 2 3]], [[4 5 6]], [[7 8 9]]]  # vetor de vetor de matriz
3-element Vector{Vector{Matrix{Int64}}}:
 [[1 2 3]]
 [[4 5 6]]
 [[7 8 9]]

julia> x4[2, 1]  # recupera elemento da linha "2" coluna "1"
4

julia> in(8, x4)  # informa se o elemento é parte da matriz
true

julia> x4[:, 1]  # apresentar elementos da primeira coluna
3-element Vector{Int64}:
 1
 4
 7

julia> x4[:, 3] # apresentar elementos da terceira coluna
3-element Vector{Int64}:
 3
 6
 9

julia> x4[2, :]  # apresentar elementos da segunda linha
3-element Vector{Int64}:
 4
 5
 6

julia> x4[1:2, 2:3]  # mostra elementos da primeira/segunda linhas até terceira/quarta colunas
2×2 Matrix{Int64}:
 2  3
 5  6
```

```
julia> view(x4, 1:2, 2:3)  # resultado idêntico a x4[1:2, 2:3]
2×2 view(::Matrix{Int64}, 1:2, 2:3) with eltype Int64:
 2  3
 5  6

julia> sum(x4)
45

julia> length(x4)
9

julia> x4[2:4]  # apresentar elementos da posição "2" até posição "4" na forma de vetor
3-element Vector{Int64}:
 4
 7
 2

julia> [1 1 1; 2 2 2; 3 3 3]
3×3 Matrix{Int64}:
 1  1  1
 2  2  2
 3  3  3

julia> unique([1 1 1; 2 2 2; 3 3 3])
3-element Vector{Int64}:
 1
 2
 3

julia> permutedims(vec(x4))  # transforma matriz 2d em matriz 1D
1×9 Matrix{Int64}:
 1  4  7  2  5  8  3  6  9

julia> [1 2 3]  # matriz com uma linha e três colunas
1×3 Matrix{Int64}:
 1  2  3

julia> [1 2 3]'  # matriz com três linhas e uma coluna
3×1 adjoint(::Matrix{Int64}) with eltype Int64:
 1
 2
 3
```

As funções com o símbolo "**!**" e "**sort**" não funcionam com matrizes. Lembre-se de que matrizes são menos flexíveis e que caso necessite fazer alguma operação não permitida em uma matriz é necessário convertê-la em vetor, operar o vetor e transformar o vetor em matriz. Para fazer essa operação com matrizes de duas dimensões é necessário o uso das funções "**vec**" já conhecida e a função "**reshape**" que fará a conversão de um vetor em matriz de duas dimensões, ressaltando que a "**permutedims**" foi usada para converter um vetor em matriz de uma dimensão.

```
julia> x6 = vec(x4) # conversão da matriz 2D de 3x3 no vetor "x6"
9-element Vector{Int64}:
 1
 4
 7
 2
 5
 8
 3
 6
 9

julia> x7 = reshape(x6, (3,3))  # conversão do vetor em matriz 2D de 3x3
3×3 Matrix{Int64}:
 1  2  3
 4  5  6
 7  8  9

julia> x7[1, 1] = 10  # altera o valor da linha "1" coluna "1"
10

julia> x7
3×3 Matrix{Int64}:
 10  2  3
  4  5  6
  7  8  9

julia> x7[2] = 20  # altera o valor da linha "2" coluna "1"
20

julia> x7
3×3 Matrix{Int64}:
 10  2  3
 20  5  6
  7  8  9
```

A função "**reshape**" faz uso de dois argumentos, sendo o primeiro argumento a indicação do vetor a ser convertido e o segundo argumento a indicação da dimensão a ser definida dentro de uma tupla. Veja esses próximos exemplos:

```
julia> reshape(1:9, (3,3))
3×3 reshape(::UnitRange{Int64}, 3, 3) with eltype Int64:
 1  4  7
 2  5  8
 3  6  9

julia> reshape(1:8, (2,4))
2×4 reshape(::UnitRange{Int64}, 2, 4) with eltype Int64:
 1  3  5  7
 2  4  6  8
```

Há um detalhe no uso da função "**reshape**" a ser considerado. O segundo argumento deve ser formado a partir de uma tupla que representa a quantidade de elementos do primeiro argumento. Os dois conteúdos indicados na tupla

devem ser valores que multiplicados totalizam exatamente o valor dos elementos no vetor. Eventualmente um dos valores da tupla pode ser omitido e representado pelo símbolo de "**:**".

```
julia> reshape(1:9, (3,:))
3×3 reshape(::UnitRange{Int64}, 3, 3) with eltype Int64:
 1  4  7
 2  5  8
 3  6  9

julia> reshape(1:8, (2,:))
2×4 reshape(::UnitRange{Int64}, 2, 4) with eltype Int64:
 1  3  5  7
 2  4  6  8

julia> reshape(1:9, (:,3))
3×3 Matrix{Int64}:
 1  2  3
 4  5  6
 7  8  9

julia> reshape(1:8, (:,4))
2×4 reshape(::UnitRange{Int64}, 2, 4) with eltype Int64:
 1  3  5  7
 2  4  6  8

julia> reshape(1:8, (:,2))
4×2 reshape(::UnitRange{Int64}, 4, 2) with eltype Int64:
 1  5
 2  6
 3  7
 4  8
```

Apesar da emparente desvantagem que matrizes possuem sobre vetores essas estruturas são bastante poderosas. Veja a seguir dois exemplos de operação de multiplicação entre matrizes:

```
julia> x8 = [0 4 -3; -5 -3 2]
2×3 Matrix{Int64}:
  0   4  -3
 -5  -3   2

julia> x9 = [1 2; 4 3; 5 6]
3×2 Matrix{Int64}:
 1  2
 4  3
 5  6

julia> x8 * x9  # multiplicação de matrizes
2×2 Matrix{Int64}:
  1  -6
 -7  -7
```

```
julia> x10 = [1 -2 3; 4 -5 6]
2×3 Matrix{Int64}:
 1  -2  3
 4  -5  6

julia> x11 = [3, 2, 1]
1×3 Matrix{Int64}:
 3  2  1

julia> x10 * x11  # multiplicação de matriz com vetor
2-element Vector{Int64}:
 2
 8
```

Os arranjos podem ser inicializados de cinco maneiras básicas antes de serem usados. É possível inicializar vetores com valores zeros, uns ou indefinidos. Para esta operação considere as seguintes formas:

```
<variável> = Matrix{tipo de dado}(undef, <quantidade de linhas>, <quantidade de colunas>)
<variável> = Array{tipo de dado}(undef, <quantidade de linhas>, <quantidade de colunas>)
<variável> = zeros(<tipo de dado>, <quantidade de linhas>, <quantidade de colunas>)
<variável> = ones(<quantidade de linhas>, <quantidade de colunas>)
<variável> = reshape(<faixa de valores>, <quantidade de linhas>, <quantidade de colunas>)
```

O indicativo *tipo de dado* se refere ao tipo de dado em uso e *quantidade* se refere a *quantidade* de elementos previamente definidos no vetor.

```
julia> b1 = Matrix{Int64}(undef, 5, 3)
5×3 Matrix{Int64}:
 0  0  0
 0  0  0
 0  0  0
 0  0  0
 0  0  0

julia> b2 = zeros(Float64, 5, 4)
5×4 Matrix{Float64}:
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0

julia> b3 = ones(5, 6)
5×6 Matrix{Float64}:
 1.0  1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0  1.0
```

```
julia> b4 = Array{Int64}(undef, 4, 5)
4×5 Matrix{Int64}:
 0  0  0  0  0
 0  0  0  0  0
 0  0  0  0  0
 0  0  0  0  0

julia> b5 = reshape(1:12, 4, 3)
4×3 reshape(::UnitRange{Int64}, 4, 3) with eltype Int64:
 1  5   9
 2  6  10
 3  7  11
 4  8  12

julia> reshape(1:12, 2, :)  # "12" elementos em "2" linhas
2×6 reshape(::UnitRange{Int64}, 2, 6) with eltype Int64:
 1  3  5  7   9  11
 2  4  6  8  10  12
```

Veja que a função "**reshape**" pode ser usada de duas maneiras diferente para duas aplicações distintas.

A partir de uma visão geral sobre vetores e matrizes observe a definição seguinte de um vetor chamado "*X*" e uma matriz de uma linha com cinco coluna chamada "*Y*" cada uma com elementos numéricos de "**1**" a "**5**".

```
julia> x = [1, 2, 3, 4, 5]
5-element Vector{Int64}:
 1
 2
 3
 4
 5

julia> y = [1 2 3 4 5]
1×5 Matrix{Int64}:
 1  2  3  4  5
```

A partir dessas duas definições veja o resultado a execução da instrução seguinte:

```
julia> x * y
5×5 Matrix{Int64}:
 1   2   3   4   5
 2   4   6   8  10
 3   6   9  12  15
 4   8  12  16  20
 5  10  15  20  25
```

Observe que a linguagem efetuou a multiplicação dos elementos de cada coluna com os elementos de cada linha de forma direta.

Como apresentado a inicialização de um arranjo, seja vetor ou matriz, pode ser feito com a definição de valores zeros, uns ou a partir da definição explicita de um conjunto de valores. No entanto, há ainda uma outra forma de inicialização a partir do uso do construtor "**Array**" como demonstrado a seguir:

```
julia> Array{Int32}(undef, 4, 5)  # cria matriz de 4 linhas/5 colunas com valores inteiros zero
4×5 Matrix{Int32}:
 0  0  0  0  0
 0  0  0  0  0
 0  0  0  0  0
 0  0  0  0  0

julia> Array{Float64}(undef, 4)  # cria vetor de 4 linhas com valores de ponto flutuante
4-element Vector{Float64}:
   1.11850968e-315
   2.7515e-320
 NaN
   0.0
```

Veja que o construtor "**Array**" pode gerar os arranjos contento valores considerados lixo de memória. Entre as chaves do construtor é usado a definição do tipo de dado a ser usado no arranho. Dentro dos parênteses o primeiro argumento chamado "**undef**" é usado para indicar que o arranjo a ser definido deve ser feito com valores indefinidos.

3.2 - Tuplas

Uma tupla caracteriza-se por ser uma coleção ordenada de dados heterogêneos delimitados entre parênteses e separados por vírgulas na forma de um conjunto de dados de uma dimensão, assemelhando-se a um arranjo. Os arranjos são estruturas mutáveis e as tuplas são imutáveis, ou seja, seu uso é mais restritivo que o uso dos arranjos, pois não permite que elementos sejam acrescidos ou removidos de seu conjunto.

Veja algumas operações que podem ser realizadas com o uso de tuplas, lembrando que algumas delas podem já ser conhecidas e outras podem ser aplicadas com arranjos:

```
julia> t1 = (1, 2, 3, 4, 5, 'A', 'B', 'C')
(1, 2, 3, 4, 5, 'A', 'B', 'C')

julia> t1
(1, 2, 3, 4, 5, 'A', 'B', 'C')

julia> t1[6:end]
('A', 'B', 'C')

julia> t1[begin:5]
(1, 2, 3, 4, 5)

julia> t1[4:7]
(4, 5, 'A', 'B')

julia> t1[7]
'B': ASCII/Unicode U+0042 (category Lu: Letter, uppercase)

julia> t2 = ((1, 2), (3, 4))  # uma tupla de tuplas
((1, 2), (3, 4))

julia> t2[1]
(1, 2)
```

```
julia> t2[2][1]  # mostra da segunda tupla seu primeiro elemento
3

julia> t1[1] = 5  # erro, pois tupla é imutável, não aceita redefinição de valor
ERROR: MethodError: no method matching setindex!(::Tuple{Int64, Int64, Int64, Int64, Int64, Char, Char, Char}, ::Int64, ::Int64)
Stacktrace:
 [1] top-level scope
   @ REPL[42]:1

julia> eltype(t2)  # mostra o tipo de dado de uma tupla de tupla com elementos inteiros
Tuple{Int64, Int64}

julia> eltype(t1)  # mostra o tipo de dado para uma tupla heterogênea
Any

julia> length(t1)  # tamamnho de uma tupla
8

julia> unique(t1)  # mostra a tupla como um arranjo
8-element Vector{Any}:
 1
 2
 3
 4
 5
  'A': ASCII/Unicode U+0041 (category Lu: Letter, uppercase)
  'B': ASCII/Unicode U+0042 (category Lu: Letter, uppercase)
  'C': ASCII/Unicode U+0043 (category Lu: Letter, uppercase)

julia> reverse(t1)
('C', 'B', 'A', 5, 4, 3, 2, 1)

julia> in(3, t1)  # verifica se certo conteúdo existe na tupla
true

julia> findall(vlr -> vlr == 'A', t1)  # mostra a posição de certo elemento
1-element Vector{Int64}:
 6
```

Como comentado estruturas de dados do tipo tupla são imutáveis e isso as tornam mais restritivas, pois muitas operações permitidas em arranjos não funcionam com tuplas. Mas isso pode ser resolvido realizando-se a conversão dos dados de uma tupla em arranjo. Para tanto, veja as instruções para a criação de uma tupla e a tentativa de efetuar a classificação de seus elementos:

```
julia> t3 = (3, 2, 1)
(3, 2, 1)
```

```
julia> sort(t3)  # tentativa de classificação de uma tupla
ERROR: MethodError: no method matching sort(::Tuple{Int64, Int64, Int64})
Closest candidates are:
  sort(::AbstractUnitRange) at C:\Program Files\Julia-1.7.2\share\julia\base\range.jl:1304
  sort(::AbstractRange) at C:\Program Files\Julia-1.7.2\share\julia\base\range.jl:1307
  sort(::SparseArrays.SparseVector{Tv, Ti}; kws...) where {Tv, Ti} at C:\Program Files\Julia-
1.7.2\share\julia\stdlib\v1.7\SparseArrays\src\sparsevector.jl:1968
  ...
Stacktrace:
 [1] top-level scope
   @ REPL[90]:1
```

Uma forma de classificar os dados de uma tupla é transformá-la em arranjo, classificar os dados do arranjo e convertê-los novamente em tupla. Veja então a sequência seguinte de instruções:

```
julia> a1 = [x[1] for x in t3]  # converte uma tuple em um arranjo
3-element Vector{Int64}:
 3
 2
 1

julia> a2 = sort(a1)
3-element Vector{Int64}:
 1
 2
 3

julia> t4 = (a2...,)  # converte um arranjo em uma tupla
(1, 2, 3)
```

A conversão de uma tuple em arranjo também pode ser produzida com a instrução:

```
julia> a2 = [t4...]
3-element Vector{Int64}:
 1
 2
 3
```

A partir dessas operações de conversão entre estruturas elimina-se qualquer restrição aparente que estruturas de dados do tipo tupla possuam.

Dentro da temática de uso de tuplas há uma técnica chamada *tupla nomeada*. Uma tupla nomeada tem a capacidade de representar seus elementos a partir de itens de índices rotulados. Desta forma, é possível acessar os componentes individuais por rótulo ou por índice. Veja alguns exemplos:

```
julia> dados = (linguagem = "Julia", lançamento = "2012", tipo = "multiparadigma")
(linguagem = "Julia", lançamento = "2012", tipo = "multiparadigma")

julia> dados.linguagem
"Julia"

julia> dados.lançamento
"2012"
```

```
julia> dados.tipo
"multiparadigma"

julia> dados[1]
"Julia"

julia> dados[2]
"2012"

julia> dados[3]
"multiparadigma"
```

3.3 - Dicionários

No último exemplo sobre tuplas foi apresentado o efeito de tupla nomeada que mostrou a associação entre variáveis como índices e dados como seus valores. Uma tupla nomeada se parece muito com um dicionário, pois este tipo de estrutura de dados opera com associação de dados. Um dicionário se configura como sendo um conjunto de elementos separados por vírgulas e delimitado entre parênteses, onde cada elemento é formado por um índice, um sinal de apontamento e a definição de um valor que ficará associado ao índice definido. Tanto os valores como os índices podem ser definidos de forma numérica, alfabética e alfanumérica a partir do uso do construtor "**Dict**". Veja exemplos de definição de dicionários:

```
julia> d1 = Dict(1 => "Julia", 2 => "Python", 3 => "COBOL", 4 => "Pascal")
Dict{Int64, String} with 4 entries:
  4 => "Pascal"
  2 => "Python"
  3 => "COBOL"
  1 => "Julia"

julia> d2 = Dict("D" => "Julia", "C" => "Python", "B" => "COBOL", "A" => "Pascal")
Dict{String, String} with 4 entries:
  "B" => "COBOL"
  "A" => "Pascal"
  "C" => "Python"
  "D" => "Julia"

julia> d3 = Dict(x => x ^ 2 for x in 1:10)  # dicionário a partir de compreensão
Dict{Int64, Int64} with 10 entries:
  5  => 25
  4  => 16
  6  => 36
  7  => 49
  2  => 4
  10 => 100
  9  => 81
  8  => 64
  3  => 9
  1  => 1
```

Os dicionários são estruturas que possuem chaves exclusivas, as quais após sua definição não podem ser alteradas. O que se pode fazer é alterar o conteúdo apontado por uma chave, mas não a chave. Veja algumas operações que se podem realizar em dicionários a partir das chaves:

```
julia> haskey(d1, 1)  # verificar se o dicionário "d1" possui o índice "1"
true

julia> haskey(d1, 5)  # verificar se o dicionário "d1" possui o índice "5"
false

julia> d1[5] = "C++"  # acrescentar dado no dicionário "d1" com índice "5"
"C++"

julia> d1
Dict{Int64, String} with 5 entries:
  5 => "C++"
  4 => "Pascal"
  2 => "Python"
  3 => "COBOL"
  1 => "Julia"

julia> in((4 => "Pascal"), d1)  # verifica se dicionário possui chave e valor informados
true

julia> in((5 => "C"), d1)
false

julia> delete!(d1, 2)  # remove o elemento de índice "2"
Dict{Int64, String} with 4 entries:
  5 => "C++"
  4 => "Pascal"
  3 => "COBOL"
  1 => "Julia"

julia> keys(d1)  # mostra as chaves existentes no dicionário "d1"
KeySet for a Dict{Int64, String} with 4 entries. Keys:
  5
  4
  3
  1

julia> values(d1)  # mostra os valores existentes no dicionário "d1"
ValueIterator for a Dict{Int64, String} with 4 entries. Values:
  "C++"
  "Pascal"
  "COBOL"
  "Julia"

julia> for x in d1
           println(x)
       end  # mostra conteúdo do dicionário "d1" por compreensão
5 => "C++"
4 => "Pascal"
3 => "COBOL"
1 => "Julia"

julia> findmin(d1)  # mostra o elemento do dicionário com índice mais baixo
("C++", 5)
```

```
julia> findmax(d1)  # mostra o elemento do dicionário com índice mais alto
("Pascal", 4)
```

Quando se possui dois dicionários diferentes com a mesma estrutura é possível combiná-los formando um outro dicionário, desde que os índices sejam únicos. Observe na sequência as instruções indicadas:

```
julia> d4 = Dict("1" => "maça", "2" => "uva", "3" => "laranja", "4" => "abacate")
Dict{String, String} with 4 entries:
  "4" => "abacate"
  "1" => "maça"
  "2" => "uva"
  "3" => "laranja"

julia> d5 = Dict("5" => "abacaxi", "6" => "tomate", "7" => "ameixa")
Dict{String, String} with 3 entries:
  "5" => "abacaxi"
  "6" => "tomate"
  "7" => "ameixa"

julia> d6 = merge(d4, d5)  # junção de dois dicionário em "d6"
Dict{String, String} with 7 entries:
  "4" => "abacate"
  "1" => "maça"
  "5" => "abacaxi"
  "2" => "uva"
  "6" => "tomate"
  "7" => "ameixa"
  "3" => "laranja"

julia> for (chave,  vlr) in d6  # iteração sobre dicionário
          println("Chave $chave igual a: $vlr")
       end
Chave 4 igual a: abacate
Chave 1 igual a: maça
Chave 5 igual a: abacaxi
Chave 2 igual a: uva
Chave 6 igual a: tomate
Chave 7 igual a: ameixa
Chave 3 igual a: laranja

julia> using Printf

julia> d7 = Dict([i => @sprintf("%d", i) for i in 1:5])  # dicionário por compreensão
Dict{Int64, String} with 5 entries:
  5 => "5"
  4 => "4"
  2 => "2"
  3 => "3"
  1 => "1"
```

No exemplo anterior com a criação de um dicionário a partir de uma lista gerada por compreensão, tem-se o uso da sub-rotina de macro "**@sprintf**" do pacote "**Printf**" que tem por finalidade apresentar os elementos formatados com salto de linha após a escrita.

Dicionários são estruturas de dados muito flexíveis aceitando o acesso e gerenciamento de dados com tanta mobilidade como os arranjos. No entanto, certas operações necessitam serem realizadas de forma um pouco diferente, como por exemplo, apresentar os dados de um dicionário classificados a partir de sua chave:

```
julia> for chave in sort(collect(keys(d3)))
           println("$chave => $(d3[chave])")
       end
1 => 1
2 => 4
3 => 9
4 => 16
5 => 25
6 => 36
7 => 49
8 => 64
9 => 81
10 => 100
```

Assim como tuplas podem ser convertidas em arranjos e vice-versa o mesmo pode ser feito com dicionários e arranjos. Veja os próximos exemplos:

```
julia> a3 = [d1...]  # converte um dicionário em um arranjo
4-element Vector{Pair{Int64, String}}:
 5 => "C++"
 4 => "Pascal"
 3 => "COBOL"
 1 => "Julia"

julia> d8 = Dict(a3...,)  # converte um arranjo em dicionário
Dict{Int64, String} with 4 entries:
  5 => "C++"
  4 => "Pascal"
  3 => "COBOL"
  1 => "Julia"
```

Os dicionários quanto criados não seguem nenhuma ordem específica, os dados ficam dispostos arbitrariamente dentro da estrutura, mas há a possibilidade de se criar dicionários classificados, mas para isso é necessário o uso do tipo de dados "**SortedDict**" do pacote "**DataStructures**". Para tanto, execute as instruções seguintes:

```
julia> import DataStructures
| Package DataStructures not found, but a package named DataStructures is available from a
  registry.
 | Install package?
 |   (@v1.7) pkg> add DataStructures
 └ (y/n) [y]: y
   Updating registry at `C:\Users\Augusto\.julia\registries\General.toml`
  Resolving package versions...
   Updating `C:\Users\Augusto\.julia\environments\v1.7\Project.toml`
 [864edb3b] + DataStructures v0.18.13
 No Changes to `C:\Users\Augusto\.julia\environments\v1.7\Manifest.toml`
```

```
julia> d5 = DataStructures.SortedDict(d4)
DataStructures.SortedDict{Int64, String, Base.Order.ForwardOrdering} with 4 entries:
  1 => "Julia"
  3 => "COBOL"
  4 => "Pascal"
  5 => "C++"

julia> d5
DataStructures.SortedDict{Int64, String, Base.Order.ForwardOrdering} with 4 entries:
  1 => "Julia"
  3 => "COBOL"
  4 => "Pascal"
  5 => "C++"

julia> d6 = DataStructures.SortedDict("C" => "gamma", "B" => "beta", "A" => "alpha")
DataStructures.SortedDict{String, String, Base.Order.ForwardOrdering} with 3 entries:
  "A" => "alpha"
  "B" => "beta"
  "C" => "gamma"

julia> d6["B"]
"beta"
```

3.4 - Conjuntos

Há ainda a estrutura de dados conjunto. Um conjunto é uma coleção mutável de dados que não permite a existência de valores repetidos, ou seja, é uma estrutura que opera com valores únicos sem que haja uma ordem de classificação predefinida. Estruturalmente um conjunto se assemelha, em parte, ao visual usado pelos arranjos de uma dimensão.

Assim como os dicionários são criados com o construtor "`Dict`", os conjuntos são criados com o construtor "`Set`", podendo este recurso ser definido a partir de quatro variações:

```
julia> c1 = Set()  # cria conjunto para qualquer tipo de dado
Set{Any}()

julia> c2 = Set{Float64}()  # cria conjunto para um tipo específico de dado
Set{Float64}()

julia> c3 = Set{Int32}([1, 2, 3, 4])  # cria conjunto preenchido com dados iniciais
Set{Int32} with 4 elements:
  4
  2
  3
  1

julia> c4 = Set([1, 'A', "B", 4.0])  # cria conjunto com dados de qualquer tipo
Set{Any} with 4 elements:
  "B"
  4.0
  'A'
  1
```

Muitas das funções apresentadas anteriormente podem ser usadas com a estrutura de dados de conjuntos. Veja alguns exemplos de manipulação de dados dento de conjuntos:

```
julia> push!(c3, 5)  # acrescenta valor "5" no conjunto "c3"
Set{Int32} with 5 elements:
  5
  4
  2
  3
  1

julia> push!(c3, 7)
Set{Int32} with 6 elements:
  5
  4
  7
  2
  3
  1

julia> push!(c3, 9)
Set{Int32} with 7 elements:
  5
  4
  7
  2
  9
  3
  1

julia> pop!(c3, 2)   # remove do conjunto "c3" o elemento "2"
Set{Int32} with 6 elements:
  5
  4
  7
  9
  3
  1

julia> in(7, c3)  # verifica se o elemento "7" pertence ao conjunto "c3"
true

julia> in(8, c3)
false

julia> for i in c3
           println(i)
       end  # apresentar por iteração os elementos do conjunto "c3"
5
4
7
9
3
1
```

Veja que esta estrutura de dado se parece com a estrutura de dado de arranjo, e neste momento deve estar pensando então para que ela serve. Pois bem, esta estrutura serve para permitir realizar operações sobre arquivos como: união, intersecção e diferença. Para testar essas operações crie os dois próximos conjuntos:

```
julia> A = Set{Int32}([1, 2, 3, 4])
Set{Int32} with 4 elements:
  4
  2
  3
  1

julia> B = Set{Int32}([3, 4, 5, 6])
Set{Int32} with 4 elements:
  5
  4
  6
  3
```

Agora, execute as próximas operações aplicadas sobre conjuntos, matematicamente falando:

```
julia> union(A, B)  # união dos conjuntos "A" e "B"
Set{Int32} with 6 elements:
  5
  4
  6
  2
  3
  1

julia> intersect(A, B)  # intersecção dos conjuntos "A" e "B"
Set{Int32} with 2 elements:
  4
  3

julia> setdiff(A, B)  # diferença entre os conjuntos "A" e "B"
Set{Int32} with 2 elements:
  2
  1

julia> symdiff(A, B)  # diferença simétrica entre os conjuntos "A" e "B"
Set{Int32} with 4 elements:
  5
  6
  2
  1
```

As funções de união, intersecção, diferença e diferença simétrica (oposto da intersecção) podem ser aplicadas sobre dicionários, além de possuírem versões de atualização física "**`union!`**", "**`intersect!`**", "**`setdiff!`**" e "**`symdiff!`**", que efetuam alteração no primeiro argumento.

Assim como dicionários são convertidos em arranjos e vice-versa o mesmo pode ser feito com conjuntos e arranjos. Veja os próximos exemplos:

```
julia> c5 = [c4...]  # converte um conjunto em um arranjo
4-element Vector{Any}:
  "B"
 4.0
  'A': ASCII/Unicode U+0041 (category Lu: Letter, uppercase)
 1

julia> c6 = Set(c5)  # converte um arranjo em conjunto
Set{Any} with 4 elements:
  "B"
  4.0
  'A'
  1
```

Os conjuntos, assim como os dicionários quanto criados não seguem nenhuma ordem específica. No entanto, há a possibilidade de se criar conjuntos classificados a partir do tipo de dados "**SortedSet**" do pacote "**DataStructures**". Para tanto, estando com o pacote devidamente carregado em memória execute a operação seguinte:

```
julia> c7 = DataStructures.SortedSet(union(A, B))
DataStructures.SortedSet{Int32, Base.Order.ForwardOrdering} with 6 elements:
  1
  2
  3
  4
  5
  6

julia> c7
DataStructures.SortedSet{Int32, Base.Order.ForwardOrdering} with 6 elements:
  1
  2
  3
  4
  5
  6
```

3.5 - Estruturas (Classes)

Estruturas são definições que permitem estabelecer um tipo de dado derivado que opere a parir de um conjunto de campos pré-formatados a fim de atender a qualquer necessidade que os demais recursos de estruturas de dados não atendam. As estruturas são componentes de operação que permitem aproximar a linguagem Julia do paradigma de programação orientada a objetos, mesmo esta linguagem não tendo, por padrão, esta característica.

3.5.1 - Ser ou não ser orientada a objetos, eis a questão

Há profissionais da computação que aceitam Julia como uma linguagem orientada a objetos (OO) e há outros que não as aceita totalmente. Os profissionais que aceitam Julia como orientada a objetos justificam esta crença ao considerarem que tudo que é definido em Julia é, naturalmente, um objeto (Christianson, 2020). Dentro desta ótica, a definição de uma variável "*X*" atribuída ao valor numérico "*1*" deve ser considerada um objeto.

```
julia> x = 1
1
```

No entanto, partindo-se da ótica da orientação a objetos, cabe considerar que um objeto é em essência uma variável especial que além de armazenar dados pode possuir associada um conjunto de funcionalidades que determinam seu método de funcionamento a partir da definição de sua instância fundamentada em uma classe. Christianson (2020) reconhece que ao mesmo tempo que defende a orientação a objetos de Julia indica que a linguagem não possui alguns recursos tradicionais encontrados no paradigma da orientação a objetos, apontando que Julia consegue operar com os conceitos de: composição, encapsulamento e polimorfismo, além de operar com imutabilidade de campos dentro de estruturas (que serão consideradas classes).

Cabe levar em consideração um dito popular que diz que "*se parece um pato, nada como um pato e grasna como um pato, provavelmente é um pato*". Atente para o termo "provavelmente" e levando isso em consideração, mesmo que Julia possua componentes operacionais encontrados na orientação a objeto não é em tese uma linguagem orientada a objeto. Mas, há outro detalhe a ser considerado, mesmo que uma linguagem não seja orientada a objetos isto não significa que não seja possível programar dentro desta filosofia (pelo menos em parte). Isso pode ser feito, por exemplo, nas linguagens Lua e C com uso de ponteiros. Assim pode-se considerar que "*se parece um objeto, pode ser definido como um objeto e comporta-se semelhantemente como um objeto, então provavelmente seja um objeto*".

De forma geral, Julia não é propriamente uma linguagem orientada a objeto, mas desse paradigma possui alguns recursos que permitem a partir de certas considerações conseguir escrever programas baseados no estilo que fundamenta o modelo de POO (*Programação Orientada a Objeto*).

Outro material que poderá auxiliar esta forma de programação é o artigo "*Type-Dispatch Design: Post Object-Oriented Programming for Julia*" publicado por Christopher Rackauckas no serviço de blog "*Stochastic Lifestyle*" disponível no endereço **https://www.stochasticlifestyle.com/type-dispatch-design-post-object-oriented-programming-julia/** e usado como base deste tópico.

3.5.2 - Classe e objeto (simulados)

Por não ser diretamente Julia uma linguagem OO não há um comando para a definição de classe. O mais próximo disso é a definição de um tipo derivado de dado chamado estrutura, sendo definido a partir do comando "**`struct`**" que possibilita a criação de uma estrutura de dados heterogênea e imutável a partir da sintaxe:

```
struct <nome>
    <campo1>::<tipo>
    <campo2>::<tipo>
    (...)
    <campoN>::<tipo>
end

<variável> = <nome>(campo1, campo2, ..., campoN)
```

Uma estrutura de caracteriza por ser um tipo de dado derivado que pode ter um nome definido contendo um conjunto de campos de dados imutáveis que representam os atributos da estrutura, mas não possui nenhuma sub-rotina como membro para estabelecer os métodos de ação sobre os atributos dos campos existentes. A indicação do tipo para cada campo é opcional, mas deve ser considerado para otimizar as operações do compilador Julia em tempo de execução.

A indicação da variável definida a partir da estrutura caracteriza-se por ser a ocorrência de uma instância da variável na memória a partir da definição da estrutura.

A título de ilustração considere a definição de uma estrutura chamada "**`Aluno`**" contendo os campos "**`nome`**", "**`turma`**" e "**`média`**" que armazenarão atributos de identificação.

```
julia> struct Aluno
           nome::String
           turma::Char
           média::Float16
       end
```

```
julia> estudante1 = Aluno("Augusto Manzano", 'A', 10.0)
Aluno("Augusto Manzano", 'A', Float16(10.0))

julia> estudante1
Aluno("Augusto Manzano", 'A', Float16(10.0))

julia> estudante1.nome
"Augusto Manzano"

julia> estudante1.turma
'A': ASCII/Unicode U+0041 (category Lu: Letter, uppercase)

julia> estudante1.média
Float16(10.0)
```

A partir da definição da estrutura simulando uma classe chamada "**Aluno**" é definida a instância da variável "**Estudante**" como se fosse um objeto. Observe que para fazer uso do recurso basta indicar o nome do objeto seguido de um ponto e o nome do campo.

Para ver os nomes dos campos de uma estrutura use a função "**fieldnames**":

```
julia> fieldnames(Aluno)
(:nome, :turma, :média)
```

Para ver o tipo de dado de um campo de uma estrutura use a função "**fieldtype**":

```
julia> fieldtype(Aluno, :nome)
String
```

Por ser a estrutura um recurso que mantém os campos imutáveis, estes não podem ser alterados de forma direta como estruturas ou registros em outras linguagens. Desta forma, a tentativa de mudança do atributo de um campo gera um erro de execução. Observe a tentativa de mudança de turma.

```
julia> estudante1.turma = 'B'
ERROR: setfield!: immutable struct of type Aluno cannot be changed
Stacktrace:
 [1] setproperty!(x::Aluno, f::Symbol, v::Char)
   @ Base .\Base.jl:43
 [2] top-level scope
   @ REPL[19]:1
```

Observe que é apresentada mensagem informando que a estrutura é imutável e por esta razão não é possível realizar a alteração. Neste sentido a estrutura da linguagem Julia se parece com a definição de uma classe na orientação a objetos, pois uma classe é uma estrutura que possui seus campos-membro definidos como privados em sua forma padrão.

Mudanças são aceitas se uma nova instância da variável "**Estudante1**" for definida utilizando-se o método de construção da estrutura passando todo o conjunto de dados a partir da execução da instrução:

```
julia> estudante1 = Aluno("Augusto Manzano", 'B', 8.0)
Aluno("Augusto Manzano", 'B', Float16(8.0))

julia> estudante2 = Aluno("Sandra de Toledo", 'B', 9.9)
Aluno("Augusto Manzano", 'B', Float16(8.0))

julia> estudante1
Aluno("Augusto Manzano", 'B', Float16(8.0))
```

```
julia> estudante2
Aluno("Sandra de Toledo", 'B', Float16(9.9))
```

Quando uma nova instância de uma variável já existente na memória é redefinida ocorre nova criação desta variável, sendo que, os dados anteriores são removidos e substituídos pelo novo conjunto de dados imutáveis. No entanto, esse comportamento não é desejável em se tratando do desenvolvimento de uma aplicação que deverá receber alterações constantes. Neste caso, o que pode ser feito é fazer uso do pacote "**Setfield**" que permite acesso a campos de estruturas imutáveis. Para tanto, execute a instrução e confirme com "**Y**" a instalação do pacote caso este não esteja instalado:

```
julia> using Setfield
│ Package Setfield not found, but a package named Setfield is available from a registry.
│ Install package?
│   (@v1.7) pkg> add Setfield
└ (y/n) [y]: y
   Updating registry at `C:\Users\Augusto\.julia\registries\General.toml`
  Resolving package versions...
  Installed ConstructionBase ─ v1.4.0
  Installed Setfield ──────── v1.0.0
   Updating `C:\Users\Augusto\.julia\environments\v1.7\Project.toml`
 [efcf1570] + Setfield v1.0.0
   Updating `C:\Users\Augusto\.julia\environments\v1.7\Manifest.toml`
  [187b0558] + ConstructionBase v1.4.0
  [efcf1570] + Setfield v1.0.0
  [9fa8497b] + Future
Precompiling project...
  2 dependencies successfully precompiled in 5 seconds (117 already precompiled)
```

A partir do carregamento do pacote "**Setfield**" passa a ser possível fazer acesso aos campos de uma estrutura a partir do recurso de macro "**@set**" que opera como se fosse uma sub-rotina membro da estrutura para efetuar a alteração do atributo de certo campo da estrutura. Assim sendo, para alterar o conteúdo (atributo) de um campo da variável, ou se quiser dizer: objeto, "**estudante1**" execute a instrução:

```
julia> estudante1 = @set(estudante1.nome = "José Augusto Manzano")
Aluno("José Augusto Manzano", 'B', Float16(8.0))
```

Veja que a instrução anterior faz a alteração do conteúdo do campo "**nome**" do "**estudante1**" de "*Augusto Manzano*" para "*José Augusto Manzano*". O recurso macro "**@set**" funciona como se fosse uma sub-rotina membro que atua como se fosse um método do tipo "**setter**" que permite escrever atributos em campos membros que sejam privados ou protegidos.

Considerando, então que em Julia todos os valores manipulados são objetos é pertinente salientar que as sub-rotinas que deveriam ser membros de uma estrutura não são empacotadas com os objetos quando este são instanciados pela estrutura, pois as estruturas não aceitam a definição interna de sub-rotinas que controlem os atributos dos campos membros. Todo este trabalho deve ser realizado de forma externa ao objeto

Dentro do exposto a definição de uma estrutura é o que se tem mais próximo de uma classe na orientação a objetos e a variável definida a partir da estrutura é o que se tem mais próximo de uma instanciação de classe. Estruturas podem ser em Julia declaradas de forma concreta e abstrata.

As estruturas concretas são definidas a partir de um nome de identificação. Aceitam a definição de uma lista de campos-membros e a definição de um construtor, podendo serem instanciadas e usadas para armazenar e manipular informações.

As estruturas abstratas possuem apenas um nome de identificação. Não aceitam a definição de campos-membro e não possuem construtores. A finalidade deste tipo de estrutura é permitir a definição de relacionamento entre estruturas

concretas assemelhando-se a definição de classes concretas, sendo úteis na definição mais próximo possível do que é a herança ou composição.

3.5.3 - Encapsulamento

Como constatado a definição de estrutura em Julia ocorre, por padrão, de maneira imutável (com todos os campos-membro protegidos), ou seja, seu uso isolado e direto para edição não é possível. Essa ocorrência caracteriza-se por definir um nível de visibilidade de seus membros como protegidos, ou seja, nível de encapsulamento protegido.

Em linguagens orientadas a objetos ocorrem três níveis de encapsulamento: público, privado e protegido. Os elementos públicos e protegidos podem ser transferidos por herança ou composição, exceto elementos privados que pertencem tão somente a classe onde estão definidos. Como Julia não é orientado a objetos seu nível de encapsulamento pode ser considerado protegido em estruturas imutáveis ou público em estruturas mutáveis.

No tópico anterior a definição da estrutura (classe) "**Aluno**" apresenta seus campos-membro definidos, por padrão interno da linguagem, como imutáveis.

Os dados imutáveis de uma estrutura parecem-se, do ponto de vista da orientação a objetos, com o estado de encapsulamento protegido de uma classe. No entanto, é possível, em Julia, definir estruturas mutáveis de dados onde os componentes da estrutura podem ser considerados públicos. Para este tipo de definição considere a seguinte sintaxe:

```
mutable struct <nome>
    <campo1>::<tipo>
    <campo2>::<tipo>
    (...)
    <campoN>::<tipo>
end

<variável> = <nome>(campo1, campo2, ..., campoN)
```

A título de ilustração considere a definição de uma estrutura chamada "**Dados**" contendo os campos "**nome**", "**altura**" e "**idade**" que armazenarão atributos de identificação.

```
julia> mutable struct Dados
           nome::String
           altura::Float16
           idade::Int8
       end

julia> pessoa = Dados("Augusto Manzano", 1.80, 57)
Dados("Augusto Manzano", Float16(1.8), 57)

Julia> pessoa
Dados("Augusto Manzano", Float16(1.8), 57)
```

A partir da criação do objeto "**pessoa**" e sua inicialização é possível efetivar mudanças diretas nos dados. Veja a alteração no campo "**nome**".

```
Julia> pessoa.nome = "José Augusto Manzano"
"José Augusto Manzano"

Julia> pessoa
Dados("José Augusto Manzano", Float16(1.8), 57)
```

Veja que uma estrutura mutável aparentemente é mais atraente para uso que uma estrutura imutável. No entanto, seu gerenciamento em memória é muito mais lento, tanto que o uso de estruturas mutáveis é desaconselhado. Assim use,

sempre estruturas imutáveis que para alterarem seus dados necessitarão da definição de sub-rotinas especiais que sejam membros vinculados aos campos de uma estrutura.

3.5.4 - Método construtor

As estruturas em Julia permitem apenas a definição de uma única sub-rotina membro que atue como método construtor da estrutura em uso. Os métodos construtores são executados automaticamente quando certo evento previamente previsto venha a ocorrer ou quando se deseja inicializar a estrutura com a definição de valores padrão para o código do programa. A definição de métodos destrutores em Julia não existe.

Para verificar este tipo de ocorrência considere a definição de uma estrutura chamada "**Faixa**", de proposta apenas didática, que gera a apresentação de uma lista de valores numéricos inteiros definidos a partir de um valor inicial, até certo valor final com determinado incremento de variação indicado para a realização de um fluxo de contagem.

```
julia> struct Faixa
           ini::Int128  # valor inicial da contagem
           fim::Int128  # valor final da contagem
           inc::Int128  # valor de incremente da contagem entre início e fim
           Faixa(ini, fim, inc) = if ini > fim
                                      println("Entrada inconsistente")
                                  else
                                      new(ini, fim, inc)
                                      collect(range(start=ini, step=inc, stop=fim))
                                  end
       end

julia> Faixa(5, 1, 2)
Entrada inconsistente

julia> Faixa(1, 15, 2)
8-element Vector{Int64}:
  1
  3
  5
  7
  9
 11
 13
 15
```

Observe que ao serem fornecidos valores inconsistentes ocorre a apresentação da mensagem de erro e se os valores forem consistentes ocorre a apresentação da lista solicitada.

Veja na definição da estrutura a indicação de uma funcionalidade com o nome da estrutura, ou seja, "**Faixa**". Isso caracteriza a existência de um recurso construtor que é executado automaticamente após a definição dos valores indicados nos argumentos. Neste caso se o valor de início da contagem for maior que o valor do final da contagem ocorre a apresentação da mensagem de erro, caso contrário ocorre a geração da lista de valores solicitada.

Note que antes de proceder a apresentação da lista de valores pretendida usa-se uma função especial chamada "**new**" que tem por finalidade criar na memória, ou seja, instanciar um objeto do tipo indicado na estrutura. A função "**new**" é especifica somente para uso dentro de construtores. A função "**new**" pode ser usada a partir de uma atribuição direta a uma variável.

A definição de construtores dentro de uma estrutura pode ser feita a partir do uso direto do comando "**function**". Veja a estrutura "**Faixa**" recodificada com o uso do comando "**function**":

```
julia> struct Faixa
           ini::Int128
           fim::Int128
           inc::Int128
           function Faixa(ini, fim, inc)
               if ini > fim
                   println("Entrada inconsistente")
               else
                   new(ini, fim, inc)
                   collect(range(start=ini, step=inc, stop=fim))
               end
           end
       end

julia> Faixa(1, 15, 2)
8-element Vector{Int64}:
  1
  3
  5
  7
  9
 11
 13
 15

julia> Faixa(5, 1, 2)
Entrada inconsistente
```

De acordo com a documentação da linguagem é uma boa prática de programação evitar o uso de métodos construtores internos: apenas devem ser usados métodos que recebem os argumentos explicitamente e que imponham a verificação e transformação de erros essenciais. Métodos construtores de conveniência adicionais, fornecendo valores padrão ou transformações auxiliares, devem ser fornecidos como sub-rotinas externas a estrutura.

3.5.5 - Composição com tipo abstrato e polimorfismo

Em Julia, as estruturas concretas podem “ser compostas” a partir de estruturas abstratas. Não é possível implementar "composição" entre estruturas concretas, sendo que Julia não opera heranças propriamente dito. Em vez do conceito de herança usa-se um conceito muito próximo ao efeito de composição existente na programação orientada a objetos.

Para demonstrar o efeito de composição considere a definição de uma estrutura abstrata chamada "**Pessoa**" e a partir dessa estrutura a definição de duas estruturas concretas chamadas "**Estudante**" e "**Trabalhador**", as quais possuem os campos "**nome**" e "**ocupação**":

```
julia> abstract type
           Pessoa
       end

julia> struct Estudante <: Pessoa
           nome::String
           ocupação::String
       end

julia> struct Trabalhador <: Pessoa
           nome::String
           ocupação::String
       end
```

Observe que a definição das estruturas concretas "**Estudante**" e "**Trabalhador**" são aplicadas com o uso do operador de subtipo "**:>**" que estabelece que as estruturas concretas são um subtipo da estrutura abstrata. Neste momento, cria-se a composição das classes abstratas com a classe concreta.

Na sequência para cada estrutura concreta deve ser definida duas funções do tipo "**getter**" que devem apresentar os dados do objeto passado como argumento na definição de sua interface de comportamento.

```
julia> function mostraDados(pessoa::Estudante)
           println("Eu $(pessoa.nome), $(pessoa.ocupação).")
       end
mostraDados (generic function with 1 method)

julia> function mostraDados(pessoa::Trabalhador)
           println("Eu $(pessoa.nome), $(pessoa.ocupação).")
       end
mostraDados (generic function with 1 method)
```

Veja que neste instante está sendo aplicado o conceito de polimorfismo *ad-hoc* de sobrecarga entre as funções. Note a definição de duas funções com o mesmo nome as quais ficam na memória sobrecarregadas. A mudança da forma de comportamento depende do tipo de dado passado para o argumento "**pessoa**" que pode ser do tipo "**Estudante**" ou do tipo "**Trabalhador**".

Veja na sequência a definição de dois objetos (variáveis) referentes a indicação de duas pessoas distintas com os dados individuais de identificação desses indivíduos:

```
julia> pessoa1 = Estudante("Maria Alice", "não tenho profissão definida")
Estudante("Maria Alice", "não tenho profissão definida")

julia> pessoa2 = Trabalhador("Pedro José", "sou profissional na área que atuo")
Trabalhador("Pedro José", "sou profissional na área que atuo")
```

A partir das definições das estruturas (classes), da definição das sub-rotinas de acesso e apresentação dos atributos de cada campo das estruturas e da definição dos objetos "**pessoa1**" e "**pessoa2**", basta fazer a chamada das sub-rotinas para ver os conteúdos apresentados:

```
julia> mostraDados(pessoa1)
Eu Maria Alice, não tenho profissão definida.

julia> mostraDados(pessoa2)
Eu Pedro José, sou profissional na área que atuo.
```

As funções declaradas em Julia, como as funções "**mostraDados**", são por padrão polimórficas e aceitarão passagem de parâmetro para qualquer tipo previsto, ocorrendo pane em sua execução somente quando um tipo de dado para o qual as interfaces apropriadas não foram implementadas.

3.5.6 - Generalização como polimorfismo

Dentro da temática de definição de estruturas Julia permite o uso do efeito de generalização de tipos de dados, sendo este em essência uma maneira de aplicar o efeito de polimorfismo *universal* paramétrico. Uma estrutura do tipo genérica deve aceitar para operação dados de qualquer tipo ou mesmo de alguns tipos específicos. .

Para demonstrar o efeito de generalização considere a definição de uma estrutura simples que aceite a entrada de dois campos de tipos genéricos. Para tanto, execute a seguinte instrução:

```
julia> struct MinhaClasse1{T}
           campo1::T
           campo2::T
       end
```

Veja que a estrutura "**MinhaClasse1**" é definida com a indicação de tipo genérico entre chaves sinalizado com a letra "**T**". O uso da letra "**T**" é, meramente, um hábito na programação de computadores para referir-se a um tipo de dados, podendo ser usado qualquer outro rótulo de identificação.

A estrutura "**MinhaClasse1**" da forma que está definida pode operar com dois dados de mesmo tipo. Se forem fornecidos dois dados com tipos diferentes ocorrerá um erro de identificação de tipo. Veja essas ocorrências:

```
julia> MinhaClasse1(1, 2.0)
ERROR: MethodError: no method matching MinhaClasse1(::Int64, ::Float64)
Closest candidates are:
  MinhaClasse1(::T, ::T) where T at REPL[1]:2
Stacktrace:
 [1] top-level scope
   @ REPL[4]:1

julia> MinhaClasse1(1, 2)
MinhaClasse1{Int64}(1, 2)
```

Note que a segunda operação não aceitou a entrada de um valor inteiro e outro valor decimal. Isso ocorreu por estar em uso o mesmo código de tipo, neste caso, "**T**" para duas definições de entrada. Para ser aceito dois tipos de dados diferentes é necessário a definição de dois rótulos distintos de tipos genéricos. Veja como:

```
julia> struct MinhaClasse2{T1, T2}
           campo1::T1
           campo2::T2
       end
```

A partir dessa nova maneira de definição é possível ter uma estrutura que reconheça tipos de dados diferentes para cada campo. Note que será necessário para cada campo ter a definição de um rótulo de identificação de tipo de dados. Veja alguns exemplos sem ocorrência de erros:

```
julia> MinhaClasse2(1, 2)
MinhaClasse2{Int64, Int64}(1, 2)

julia> MinhaClasse2(1, 2.0)
MinhaClasse2{Int64, Float64}(1, 2.0)

julia> MinhaClasse2("Teste", 2.0)
MinhaClasse2{String, Float64}("Teste", 2.0)

julia> MinhaClasse2("Teste", true)
MinhaClasse2{String, Bool}("Teste", true)
```

A estrutura "**MinhaClasse2**" pode apresentar um problema em relação a aceitação de valores do tipo cadeia caso os campos da estrutura sejam usados para auxiliar ações de processamento matemático ou lógico. Neste caso, precisa-se usar uma outra estratégia de identificação de valores mais amplos, mas que não sejam tão amplos como ocorreu com a estrutura "**MinhaClasse2**". Veja o próximo exemplo:

```
julia> struct MinhaClasse3{T1 <: Real, T2 <: Real}
           campo1::T1
           campo2::T2
       end
```

Observe que neste exemplo está sendo indicado que o rótulo de identificação de tipo de dado, neste caso, genérico "**T1**" e "**T2**" estão sendo definidos com subtipo do tipo abstrato de dados "**Real**". Desta forma, os tipos de dados numéricos inteiros, decimais e os valores lógicos podem ser aceitos normalmente, pois todos são hierarquicamente vinculados ao tipo de dado "**Real**". Veja alguns exemplos com êxito e ´último com falha.

```
julia> MinhaClasse3(1, 2)
MinhaClasse2{Int64, Int64}(1, 2)

julia> MinhaClasse3(1, 2.0)
MinhaClasse3{Int64, Float64}(1, 2.0)

julia> MinhaClasse3(1.0, 2.0)
MinhaClasse3{Float64, Float64}(1.0, 2.0)

julia> MinhaClasse3(1.0, true)
MinhaClasse3{Float64, Bool}(1.0, true)

julia> MinhaClasse3("Teste", true)
ERROR: MethodError: no method matching MinhaClasse3(::String, ::Bool)
Closest candidates are:
  MinhaClasse3(::T1, ::T2) where {T1<:Real, T2<:Real} at REPL[10]:2
Stacktrace:
 [1] top-level scope
   @ REPL[15]:1
```

Veja que a partir da estrutura "**MinhaClasse3**" tem-se a definição de uma classe de dados genéricos preparada para operar com valores numéricos e lógicos, não aceitando a entrada de dados do tipo cadeia como aconteceu com a estrutura "**MinhaClasse2**".

Usar subtipo de dados é necessário junto a declaração dos campos-membro. No entanto, isso não se aplica na definição das sub-rotinas membro como métodos de operação.

Um exemplo apresentado no artigo que fundamenta este tópico é a geração de um tipo de dado para a apresentação de uma lista. Uma lista pode ser considerada como sendo uma coleção de valores heterogêneos contendo em sua estrutura uma cabeça (primeiro elemento da lista), uma cauda (contendo todos os elementos, exceto o primeiro) e a definição de um valor invisível de finalização chamado "**Nil**". Veja a definição sintática da estrutura para gerenciamento de uma lista.

```
julia> struct Nil
           # estrutura vazia: nil = nada
       end

julia> struct Lista{T}
           cabeça::T
           cauda::Union{Lista{T}, Nil}
       end
```

Veja que na estrutura "**Lista**" está sendo definido para o campo "**cauda**" o tipo de dado "**Union**", que tem por finalidade unir todas as instâncias de qualquer um de seus tipos de argumento criando um nó de acesso ao estilo lista ligada. Desta forma, será possível definir uma lista de conteúdos até que haja na lista um último valor "**Nil**".

Agora é preciso informar a sub-rotina que permitirá a criação de uma lista em memória a partir da definição de um arranjo. Para tanto, escreva o seguinte código:

```
julia> fazLista(arranjo::AbstractArray{T}) where T = foldr(Lista{T}, arranjo, init=Nil())
fazLista (generic function with 1 method)
```

O argumento da sub-rotina de função "**fazLista**" que representa um arranjo de valores de tipo genérico "**T**" para um arranjo abstrato é associado ao uso da função "**foldr**" a partir do uso do comando "**where**" que tem por finalidade criar um tipo de dados que seja união iterada de outros tipos, sobre todos os valores de alguma variável, neste caso representada pelo arranjo informado como argumento, transformando qualquer elemento do arranjo em um valor no corpo da função.

Para o funcionamento adequado da função "**fazLista**" é necessário definir os protocolos operacionais de iteração para a função. Esses protocolos adicionam métodos de ação a partir do estilo *biblioteca.função*: "**Base.iterate**" para o tipo de dado arranjo. Assim sendo defina as seguintes regras:

```
julia> Base.iterate(l::Lista) = iterate(l, l)

julia> Base.iterate(::Lista, l::Lista) = l.cabeça, l.cauda

julia> Base.iterate(::Lista, ::Nil) = nothing
```

A biblioteca "**Base**" é o módulo base de operação da linguagem contento as funcionalidades essenciais de uso. Todos os módulos de operação usados em Julia possuem implicitamente a biblioteca "**Base**", sendo isso necessário na grande maioria das operações. A funcionalidade "**iterate**" é um recurso que permite avançar o ponteiro de iteração em uma sequência de valores para obter o próximo elemento, gerando uma tupla como resultado.

A partir da definição das interfaces de funcionamento que estabelecem as regras operacionais basta criar a lista em memória. Para tanto, execute:

```
julia> lista = fazLista(1:5)
Lista{Int64}(1, Lista{Int64}(2, Lista{Int64}(3, Lista{Int64}(4, Lista{Int64}(5, Nil()))))))
```

A partir deste instante a lista está moldada em memória e para vela de uma forma mais humana, execute as seguintes instruções:

```
julia> for x in lista
           println(x)
       end
1
2
3
4
5
```

Agora basta criar uma sub-rotina ao estilo procedimento que gere e apresente a lista ao estilo das funcionalidades da linguagem (apenas uma proposta despretensiosa). Observe as instruções seguintes:

```
julia> function geraLista(lista)
           lista = fazLista(lista)
           println(lista)
           for x in lista
               println(x)
           end
       end
geraLista (generic function with 1 method)
```

```
julia> geraLista(1:5)
Lista{Int64}(1, Lista{Int64}(2, Lista{Int64}(3, Lista{Int64}(4, Lista{Int64}(5, Nil()))))
1
2
3
4
5
```

3.6 - Cadeias de caracteres

Uma cadeia de caracteres é formada por símbolos alfabéticos maiúsculos, símbolos alfabéticos minúsculos, símbolos numérico arábico-hindus e também símbolos especiais de pontuação que são delimitados entre aspas inglesas. Os caracteres isolados são chamados caracteres e quando usados dessa forma são delimitados entre aspas simples.

Os caracteres alfabéticos, numéricos e especiais usados em um computador correspondem a um total de 128 símbolos se considerado apenas os caracteres do idioma Inglês. Neste sentido, esse conjunto de 128 caracteres fazem parte da tabela chamada *ASCII*. No entanto a linguagem Julia opera com um conjunto maior de caractere abrangendo a tabela *Unicode* que estende a tabela *ASCII* com diversos caractere de idiomas diferentes do Inglês e outros caracteres não encontrados na tabela *ASCII*.

Parte das operações aqui tratadas foram apresentadas a partir do uso das funções "**Int**", "**Char**" e "**typeof**", além de outras ações anteriormente apresentadas. Observe a seguir a diferença na apresentação de caracteres e cadeias definidas entre aspas simples e aspas inglesas a partir do mesmo símbolo:

```
julia> c1 = 'A'
'A': ASCII/Unicode U+0041 (category Lu: Letter, uppercase)

julia> c2 = "A"
"A"

julia> typeof(c1)
Char

julia> typeof(c2)
String
```

Observe a diferença de informações obtidas com a função "**typeof**" em relação as variáveis "*C1*" e "*C2*" respectivamente como *Char* e *String*.

Dados caractere ou cadeia podem ser aglutinados como uma nova cadeia. Este efeito recebe o nome de *concatenação*, podendo ser realizada na linguagem Julia por meio da função "**string**". Observe os exemplos seguintes de concatenação:

```
julia> t1 = "Alo, "
"Alo, "

julia> t2 = "mundo!"
"mundo!"

julia> string(t1, t2)
"Alo, mundo!"

julia> string("A", "B", "C")  # concatenação de cadeias
"ABC"
```

```
julia> string('1', '2', 'X')  # concatenação de caracteres
"12X"

julia> string("ABC", '?')  # concatenação de cadeia com caractere
"ABC?"
```

Alternativamente concatenações podem ser efetivadas com o uso do caractere "*****" (asterisco). Note alguns exemplos:

```
julia> t1 * t2
"Alo, mundo!"

julia> *(t1, t2)
"Alo, mundo!"
julia> *("A", "B", "C")
"ABC"

julia> *('1', '2', 'X')
"12X"

julia> *("ABC", '?')
"ABC?"
```

A função "**string**" possui mais alguns recursos como por exemplo poder ser usada para converter um valor numérico na forma de cadeia. Atente para os seguintes exemplos:

```
julia> string(10)
"10"
julia> string(5.2)
"5.2"
```

A função "**string**" também pode ser usada para efetuar a conversão de base de um valor numérico inteiro. Observe os exemplos seguintes que convertem o valor numérico "**10**" no formato hexadecimal e octal:

```
julia> string(10, base = 16, pad = 4)
"000a"

julia> string(10, base = 8)
"12"
```

Para o tratamento de cadeias há um conjunto de ocorrências a partir de um conjunto de funções especializadas para este tipo de tarefa. Para a realização dos próximos testes defina a seguinte variável e observe os testes indicados:

```
julia> texto1 = " Estudo de linguagem Julia "
" Estudo de linguagem Julia "
```

Quantificar a quantidade de caracteres de uma cadeia:

```
julia> length(texto1)
27
```

Remoção de espaços em branco nas extremidades da cadeia:

```
julia> strip(texto1)  # remove espaços das extremidades
"Estudo de linguagem Julia"

julia> lstrip(texto1)  # remove espaço do lado esquerdo
"Estudo de linguagem Julia "

julia> rstrip(texto1)  # remove espaço do lado direito
" Estudo de linguagem Julia"
```

Inversão do texto de uma cadeia:

```
julia> reverse(texto1)
" ailuJ megaugnil ed odutsE "
```

Preenchimento de caracteres nas extremidades da cadeia:

```
julia> lpad(texto1, length(texto1) + 4, "Meu ")
"Meu  Estudo de linguagem Julia "

julia> rpad(texto1, length(texto1) + 7, " - 2022")
" Estudo de linguagem Julia  - 2022"
```

As funções "**lpad**" e "**rpad**" efetuam o preenchimento de conteúdo em uma cadeia com base no tamanho da cadeia a ser concatenada com mais a quantidade dos caracteres a serem inseridos na concatenação.

Troca de conteúdo de uma cadeia:

```
julia> replace(texto1, "Julia" => "Python")
" Estudo de linguagem Python "

julia> replace(strip(texto1), "Julia" => "")  # remover "Julia" da cadeia "texto1"
"Estudo de linguagem "
```

Apresentação de conteúdo em formato maiúsculo e minúsculo:

```
julia> uppercase(texto1)
" ESTUDO DE LINGUAGEM JULIA "

julia> lowercase(texto1)
" estudo de linguagem julia "
```

Decomposição da cadeia em palavras e/ou caracteres:

```
julia> split(texto1)
4-element Vector{SubString{String}}:
 "Estudo"
 "de"
 "linguagem"
 "Julia"
```

```
julia> split(texto1, "de")
2-element Vector{SubString{String}}:
 " Estudo "
 " linguagem Julia "

julia> split(texto1, "")
27-element Vector{SubString{String}}:
 " "
 "E"
 "s"
 "t"
 "u"
 "d"
 "o"
 " "
 "d"
 "e"
 " "
 "l"
 "i"
 ⋮
 "u"
 "a"
 "g"
 "e"
 "m"
 " "
 "J"
 "u"
 "l"
 "i"
 "a"
 " "
```

Veja que o ambiente Julia suprime/compacta parte da apresentação de uma lista grande de dados, utilizando-se o símbolo "**:**", para mostrar o conjunto de dados na tela. Caso queira, ver todos os dados na tela sem a ocorrência de compactação execute a seguinte instrução:

```
julia> show(stdout, "text/plain", split(texto1, ""))
```

O indicativo "**stdout**" direciona a apresentação do conteúdo para o periférico de saída padrão, neste caso, monitor de vídeo e diz para que a apresentação ocorra em modo normal de forma simples, ou seja, "**text/plain**".

A decomposição de uma cadeia em caracteres pode ser produzida com o uso da função "**collect**". Veja um exemplo com o uso da cadeia "*Julia*":

```
julia> collect("Julia")
5-element Vector{Char}:
 'J': ASCII/Unicode U+004A (category Lu: Letter, uppercase)
 'u': ASCII/Unicode U+0075 (category Ll: Letter, lowercase)
 'l': ASCII/Unicode U+006C (category Ll: Letter, lowercase)
 'i': ASCII/Unicode U+0069 (category Ll: Letter, lowercase)
 'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)
```

Repetição de trechos de cadeia certa quantidade de vezes:

```
julia> repeat("ABC ", 5)
"ABC ABC ABC ABC ABC "
```

Verificar se a cadeia possui determinado termo:

```
julia> contains(texto1, "Julia")
true

julia> contains(texto1, "Python")
false
```

Armazenamento de cadeia de caracteres configurada a partir de caracteres grafados indiretamente em valores hexadecimais positivos (forma alternativa de representação de valores binários) ASCII e apresentação:

```
julia> palavra = b"Julia"  # ou alternativamente "UInt8[0x4a, 0x75, 0x6c, 0x69, 0x61]"
5-element Base.CodeUnits{UInt8, String}:
 0x4a
 0x75
 0x6c
 0x69
 0x61

julia> String(palavra)  # ou alternativamente "join(Array{Char}(palavra))"
"Julia"

julia> palavra[1]
0x4a

julia> Char(palavra[1])
'J': ASCII/Unicode U+004A (category Lu: Letter, uppercase)

julia> letras = [palavra...]  # converte Base.CodeUnits em Vector
5-element Vector{UInt8}:
 0x4a
 0x75
 0x6c
 0x69
 0x61

julia> String(letras)
"Julia"
```

Os valores hexadecimais 4a, 75, 6c, 69 e 61 refere-se aos valores que referenciam a formação da palavra "*Julia*". O código "**0x**" refere-se a definição de um valor hexadecimal. Na definição da variável "*PALAVRA*" o uso do sinalizador de código "**b**" antes da definição de uma cadeia faz a conversão de cada caractere em seu código hexadecimal equivalente. O uso dos tipos "**String**" e "**Char**" permitem converter um código hexadecimal em seu formato ASCII nominal.

A utilização de cadeias de caracteres pode também fazer uso da função "**join**" usada para unir um arranjo de cadeia de caracteres em uma única cadeia, ou seja, realizar uma forma de concatenação mais aprimorada que o uso do operador de concatenação "*****". Veja mais alguns exemplos de uso dessa função:

```
julia> join(["Linguagem ", "Julia"])
"Linguagem Julia"

julia> join(["Linguagem", "programação"], " de ")
"Linguagem de programação"

julia> join(["Aprender", "a", "programar"], "-", "*")
"Aprender-a*programar"

julia> join(["Aprender", "a", "programar", "em", "Julia"], "-", "*")
"Aprender-a-programar-em*Julia"
```

Veja esses outros exemplos de manipulação de textos e caracteres que mostram as posições de certo conteúdo ou do trecho de conteúdo dentro de uma cadeia.

```
julia> findall(isequal('a'), "Linguagem de programação Julia")
4-element Vector{Int64}:
  6
 19
 21
 32

julia> findall("J", "Julia")
1-element Vector{UnitRange{Int64}}:
 1:1

julia> findall("lia", "Julia")
1-element Vector{UnitRange{Int64}}:
 3:5

julia> findall("o", "computador")
2-element Vector{UnitRange{Int64}}:
 2:2
 9:9

julia> findfirst(isequal('a'), "Linguagem de programação Julia")
6

julia> findlast(isequal('a'), "Linguagem de programação Julia")
32
```

Um detalhe importante na manipulação de cadeias e caracteres é conhecer os recursos de conversão de certo caractere em seu código equivalente ASCII e a partir do código ASCII obter o caractere correspondente. Veja os próximos exemplos:

```
julia> Int('A')  # obtém o código ASCII do caractere indicado
65

julia> Char(65)  # obtém o caractere do código ASCII informado
'A': ASCII/Unicode U+0041 (category Lu: Letter, uppercase)
```

Nas operações de tratamento de cadeias de caracteres pode-se fazer uso do tipo de caixa alta (*upper case*) ou caixa baixa (*lower case*) para verificar se determinado valor existe no arranjo atenuando um pouco a rigidez das operações lógicas. Observe as próximas instruções:

```
julia> carros = ["Uno", "Fusca", "Chevette", "Opala", "Rural"]
5-element Vector{String}:
 "Uno"
 "Fusca"
 "Chevette"
 "Opala"
 "Rural"

julia> "Opala" in carros
true

julia> "opala" in carros
false
```

Observe o resultado apresentado das duas últimas operações com uso do operador "**in**" que pode ser operacionalizado com dados numéricos. Veja que o arranjo "*CARROS*" possui em sua lista um automóvel chamado "**Opala**". No entanto a primeira das últimas operações retomou verdadeiro, mas a segunda retornou falso e isso se dá devido a diferença na composição da cadeia com o uso de caracteres em caixa alta e caixa baixa. Para atenuar este efeito, tonando a busca mais flexível sem perder sua eficácia, execute as próximas instruções:

```
julia> "opala" in lowercase.(carros)
true

julia> lowercase.("Opala") in lowercase.(carros)
true

julia> "OPALA" in uppercase.(carros)
true

julia> uppercase.("Opala") in uppercase.(carros)
true
```

Já é sabido que quando há a necessidade de aplicar uma operação sobre todos os elementos de um arranjo usa-se o conceito de operação vetorial a partir do uso do símbolo "**.**" como ponto de ligação da operação e o vetor. Foi comentado que este tipo de operador pode ser usado tanto com operadores aritméticos como com operadores lógicos, e agora fica claro que seu uso também se aplica a indução de certa função sobre a coleção de elementos de determinado arranjo.

Cabe dentro da discussão sobre manipulação de cadeias algumas considerações finais para este estudo. Quando se deseje definir apenas caracteres alfanuméricos usa-se na delimitação aspas simples, caso seja a definição de cadeia (palavras ou frases) se usa aspas inglesas. Se na sequência de caracteres deseja-se usar algum caractere especial que seja permitido usa-se o símbolo de contra barra "****".

Execute a instrução para definição de aspas dentro de uma cadeia.

```
julia> println("O automóvel \"Rural Willys\" foi lançado no Brasil em \'1958\'")
O automóvel "Rural Willys" foi lançado no Brasil em '1958'
```

Fazer uso do caractere contra barra pode, dependendo da situação, ser trabalhoso e cansativo. No entanto, há uma forma de atenuar um pouco esse possível desgaste, basta fazer uso de aspas inglesas triplas. Veja o próximo exemplo:

```
julia> println("""O automóvel "Rural Willys" foi lançado no Brasil em '1958'""")
O automóvel "Rural Willys" foi lançado no Brasil em '1958'
```

Caso deseje usar o símbolo de contra barra, mas não está com paciência para defini-lo nas posições necessárias poderá usar um artifício onde basta escrever o que deseja e deixar o ambiente Julia mostrar onde esses elementos devem ser colocados. Veja o seguinte:

```
julia> """O automóvel "Rural Willys" foi lançado no Brasil em '1958'"""
"O automóvel \"Rural Willys\" foi lançado no Brasil em '1958'"
```

Dentro das operações de pesquisa realizadas sobre cadeias de caracteres pode-se além das funções "**findall**", "**finfirst**" e "**findlast**" usar a função "**occursin**" que verifica se certa parte de uma cadeia faz parte de toda a cadeia. Observe suas operações:

```
julia> occursin("Julia", "Linguagem de programação Julia")
true

julia> occursin("Pascal", "Linguagem de programação Julia")
false

julia> occursin("L", "Linguagem de programação Julia")
true

julia> occursin("p", "Linguagem de programação Julia")
true

julia> occursin("X", "Linguagem de programação Julia")
false
```

ANOTAÇÕES

CAPÍTULO 4

PROGRAMAÇÃO (QUASE) FUNCIONAL

Este capítulo mostra os detalhes gerais da filosofia de programação funcional dentro do universo da linguagem Julia e suas restrições operacionais pelo fato da linguagem não ser modelada diretamente para este paradigma. Neste sentido você tem acesso há alguns recursos funcionais que podem ser usados em Julia. São apresentadas orientações sobre o comportamento funcional, simulação de comportamento de padrões, definições de funções anônimas (que podem ser usadas livremente em qualquer estilo de programação em Julia), uso de funções de ordem superior e funções de primeira classe.

4.1 - Ser ou não ser funcional, eis a questão

A linguagem Julia não é, em essência, uma linguagem funcional. No entanto, possui alguns recursos encontrados neste paradigma que permitem seguir em um caminho ao estilo funcional sem grandes problemas, lembrando que o mesmo ocorre em relação a Julia e o paradigma da programação orientada a objetos.

Dentro do universo da programação funcional Julia atente aos seguintes requisitos:

- Opera com funções de alta ordem ou funções de ordem superior;
- Opera com funções de primeira classe;
- Aceita o uso de recursividade;
- Aceita a definição de funções anônimas;
- É possível, com certo esforço, evitar a ocorrência de efeitos colaterais em variáveis.

Dentro do universo da programação funcional Julia não atente aos seguintes requisitos:

- Não faz composição e aplicação parcial de funções;
- Não opera com o uso e definição de variáveis imutáveis;
- Não opera com casamento de padrões.

Desta forma, para se trabalhar ao estilo da programação funcional com a linguagem Julia é necessário deixar de lado alguns pré-conceitos operacionais, fechar os olhos e codificar ao estilo funcional ao máximo possível, mesmo quebrando as regras do estilo de escrita de código. Parte da programação funcional não é a linguagem em si, mas o comportamento mental do programador, ou seja, a programação funcional não é somente feita com a ferramenta linguagem, mas também com o comportamento mental.

4.2 - Comportamento funcional

Um dos principais pontos apregoados pela programação funcional é evitar a geração de efeitos colaterais em variáveis. Por isso que linguagens deste paradigma operam com variáveis imutáveis e desaprovam o uso de variáveis de estado. O que não é praticado pela linguagem Julia.

As variáveis imutáveis são objetos que possuem como comportamento a capacidade de receber um só valor e os mantém inalterados ao longo do processamento do programa. Havendo a necessidade da redefinição de certo valor sobre a variável outra variável com mesmo nome é criada em memória e usada para armazenar o "novo" valor, preservando, desta forma, todos os valores anteriormente gerados.

Julia só opera com varáveis de estado. Isso faz com que uma variável com valor definido possa ser livremente alterada o que é um problema dentro da ótica da programação funcional, mas com alguns cuidados é possível chegar próximo ao estilo funcional ao produzir funções que sejam puras, ou seja, funções determinísticas que não aterem nenhum valor fora de sua abrangência e que retornem sempre o mesmo valor se usado o mesmo argumento como parâmetro Neste sentido qualquer variável a mais necessária para a função deve ser mantida de forma local dentro da área de escopo da função. Veja a seguir a definição de função para o cálculo do fatorial de um número qualquer utilizando a definição convencional e pura respetivamente.

```
julia> function fatorial1(n::Int64, p::Int64=1)
           if n == 0
               fat = p
           else
               fat = fatorial1(n - 1, n * p)
           end
           return fat
       end
geraLista (generic function with 1 method)

julia> fatorial1(5)
120

julia> function fatorial2(n::Int64, p::Int64=1)
           if n == 0
               return p
           else
               return fatorial2(n - 1, n * p)
           end
       end
geraLista (generic function with 1 method)

julia> fatorial2(5)
120
```

Observe que a função "**fatorial2**" tem uma proposta de pureza maior, sendo que eliminou o uso da variável interna "*FAT*" existente na função "**fatorial1**" estabelecendo o retorno da resposta da operação de forma direta. Funções puras devem ter o mínimo possível de definições internas de variáveis além das variáveis passadas em seus argumentos.

Veja que em ambas as funções está sendo usado o recurso de recursividade. Se seu desejo é seguir uma programação em estilo funcional, jamais use comandos de laços (pelo menos dentro das funções). Quando usar recursividade opte primeiramente pelo método cauda, deixe a recursividade simples apenas para situações em que a recursividade de cauda não é necessária ou que não se aplique.

4.3 - Simulação de casamento de padrões

Uma das principais essências de operação de uma linguagem de programação funcional é sem sombra de dúvidas o recurso do casamento de padrões. Infelizmente Julia não possui essa capacidade e não possui também nem sequer o uso de tomadas de decisão por seleção ao estilo *caso seja* que permite de forma natural implementar a ideia de casamento de padrões. No entanto, com um pouco de esforço imaginativo é possível definir uma estrutura de decisão simples e sequencial que se pareça, precariamente, com a definição de casamento de padrões, mesmo que passando por cima de algumas regras de escrita do código de escrita e documentação "*Blue: a Style Guide for Julia*", simulando de certa forma o efeito de casamento de padrões.

A título de ilustração considere a definição de uma função chamada "**valorx**" que ao receber um valor numérico inteiro qualquer retorne: "**0**" se for fornecido "**0**"; "**2**" se for fornecido "**1**" ou a multiplicação do valor recebido por dois se o valor estiver entre "**2**" e "**8**", caso contrário deve retornar o valor fornecido dividido por "**5**".

```
julia> function valorx(n::Int64)
           if n == 0 return 0 end
           if n == 1 return 2 end
           if n >= 2 && n <= 8 return n * 2 else return n / 5 end
       end
geraLista (generic function with 1 method)

julia> valorx(0)
0

julia> valorx(1)
2

julia> valorx(8)
16

julia> valorx(9)
1.8

julia> valorx(-2)
-0.4
```

No exemplo apresentado a simulação do casamento de padrões junto a função "**valorx**" estão definidas nas primeiras duas linhas de código, junto as condições sequenciais estabelecidas para validar os retornos dos valores "**0**" e "**2**" se fornecidos respetivamente os valores "**0**" e "**1**". A terceira linha caracteriza-se apenas pela definição da operação a ser efetivamente realizada caso as ações do casamento de padrões (mesmo que simulado) não ocorram. Este estilo de operação é conhecido como *avaliação preguiçosa*, ou seja, qualquer ocorrência de decisão do programa que ocorra desvia o programa para ação, executando-a e encerrando a operação em seguida sem que as demais linhas de código sejam avaliadas.

O efeito de casamento de padrões combinado com o efeito de recursividade fornece a base de trabalho usada amplamente na programação funcional. Veja na sequência a definição de uma função com recursividade de cauda que apresenta o termo da sequência de Fibonacci a partir do fornecimento de certa posição da sequência.

```
julia> function fibonacci(n::Int64, anterior::Int64=0, atual::Int64=1)
           if n == 0 return anterior         end
           if n == 1 return atual            end
           if n == 2 return anterior + atual end
           if n >= 2 return fibonacci(n - 1, atual, anterior + atual) end
       end
fibonacci (generic function with 3 methods)
```

Para a simulação da definição do casamento de padrões é interessante manter o alinhamento dos comandos "**end**" como indicados nas três primeiras linhas de código da função "**fibonacci**". veja na sequência o uso da função.

```
julia> fibonacci(0)
0

julia> fibonacci(1)
1

julia> fibonacci(2)
1

julia> fibonacci(3)
2

julia> fibonacci(10)
55
```

No entanto, nem tudo são flores neste tipo de simulação, pois dependendo de como segue-se o fluxo de programação ocorrerão certos problemas. Para se ter ideia, considere uma função que retorne o máximo divisor comum. Veja que no código apresentado usa-se o estilo de solução típica aplicado na programação funcional. Apesar da linguagem aceitar a função e executá-la não consegue operar com valores de média a grande proporção. Apenas podem ser usados valores de pequena monta.

```
julia> function mdc(n::Int64, m::Int64)
           if n == 0 return m end
           if n >= 1 return mdc(n % m, m) end
       end
fibonacci (generic function with 3 methods)


julia> mdc(8, 2)
2

julia> mdc(1024, 12)
ERROR: StackOverflowError:
Stacktrace:
 [1] mdc(n::Int64, m::Int64) (repeats 79984 times)
   @ Main .\REPL[48]:3
```

Veja que a tentativa de cálculo do MDC dos valores 1024 e 12 que resulta 4 retorna um erro de estouro de pilha. Internamente a linguagem possui a função "**gcd**" que efetua o cálculo adequadamente utilizando-se de outra estratégia que não a funcional.

```
julia> gcd(1024, 12)
4
```

Isso mostra que para fazer uso do estilo funcional em Julia é necessário levar em consideração diversos detalhes operacionais.

4.4 - Funções anônimas (lambda)

Uma função anônima ou função lambda é uma sub-rotina sem nome de identificação, não armazenada em memória que deve ser usada imediatamente a sua definição. O uso de funções anônimas não é uma capacidade operacional exclusiva da programação funcional, sendo encontrada em outros paradigmas. Para um exemplo simples considere a definição tradicional de uma função anônima que apresente o sucessor de um valor numérico do conjunto dos números reais.

```
julia> function (n::Real)
           return n + 1
       end
#1 (generic function with 1 method)
```

Observe que a função definida não possui nome de identificação. Logo, é uma função anônima e para colocá-la em uso execute a instrução seguinte:

```
julia> ans(1)
2
```

Veja que o recurso "**ans**" caracteriza-se por ser uma variável interna que traz da memória o último valor calculado e que esteja definido junto ao *prompt* do modo interativo REPL. Desta forma, é possível fazer uso da função anônima apenas uma vez, tanto que se a instrução anterior for repetida ocorrerá a apresentação de um erro.

```
julia> ans(1)
ERROR: MethodError: objects of type Markdown.MD are not callable
Stacktrace:
 [1] top-level scope
   @ REPL[7]:1
```

Além do estilo tradicional de definição de função, como apresentado, há uma forma de se definir funções anônimas ao estilo da programação funcional. Veja a seguinte definição para apresentar o valor sucessor de um número do conjunto de reais fornecido e o uso da função anônima declarada em apenas uma linha de código.

```
julia> x::Real -> x + 1
#3 (generic function with 1 method)

julia> ans(1)
2
```

Note que esta segunda forma de definição de funções anônimas é bem m ais conveniente e menos verbosa que a primeira forma.

Veja um outro exemplo de função anônima que opera dois argumentos. A função a seguir tem por finalidade apresentar a soma de dois valores numéricos fornecidos.

```
julia> (x::Real, y::Real) -> x + y
#4 (generic function with 1 method)

julia> ans(1, 2)
3
```

Veja que para indicar o uso de mais de um argumento é necessário especificar os argumentos a serem usados dentro de parênteses.

Veja em seguida uma aplicação de mapeamento sobre uma lista de valores inteiros de "**1**" a "**5**" ao estilo vetor com o uso de uma função anônima para geração dos quadrados dos valores da lista.

```
julia> map(x -> x ^ 2, [1, 2, 3, 4, 5])
5-element Vector{Int64}:
  1
  4
  9
 16
 25
```

A definição de funções anônimas é considerada, como aponta Moura (2019, p. 46), a menor linguagem de programação do mundo.

4.5 - Funções de primeira classe e ordem superior

Uma função é considerada de primeira classe quando pode ser usada como se fosse uma variável. Desta forma, uma função definida como variável pode ser utilizada como argumento de outra função.

Veja um exemplo simples de uso da função "**+**" para processar a adição entre dois valores numéricos. É importante ressaltar que qualquer operador, seja, aritmético, relacional ou lógico são por sua natureza funções:

```
julia> 2 + 3
5
```

Agora veja a definição de uma variável chamada "*ADIÇÃO*" definida a partir da função "**+**" e sua aplicação com valores numéricos. Observe que a variável tem que ser usada como uma sub-rotina de função convencional. Este é o efeito de definição de função de primeira classe.

```
julia> adição = +
+ (generic function with 208 methods)

julia> adição(2, 3)
5

julia> adição(2.5, 3.5)
6.0

julia> adição(2.5, 7)
9.5
```

Como exemplo de uso de função de primeira classe considere realizar a redução de uma lista de valores apresentando a multiplicação sucessiva de todos os seus elementos a partir da definição da função como variável "**mult**". Neste caso, considere uma lista com valores impares entre "**1**" a "**9**".

```
julia> mult = *
* (generic function with 364 methods)

julia> reduce(mult, [1, 3, 5, 7, 9])
945
```

Além da definição de funções de primeira classe Julia aceita o uso de funções de ordem superior. As funções de ordem superior se caracterizam por serem funções que são usadas como argumentos de outras funções. Para o uso de função de ordem superior considere apresentar todos os valores impares de uma lista entre "**0**" a "**9**".

```
julia> isodd(5)
true

julia> filter(isodd, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
5-element Vector{Int64}:
 1
 3
 5
 7
 9
```

Veja que a filtragem de valores impares ocorre com o uso da função interna "**isodd**" sendo passada como o primeiro argumento da função "**filter**".

Funções passadas como parâmetro de outras funções permitem operar sob a ótica do reaproveitamento de códigos. Isso possibilita a criação de códigos menores e simplificados para definir a composição de funções mais complexas, buscando-se diminuir ou neutralizar efeitos colaterais.

Dentro do contexto de definição de função de primeira classe pode-se definir em Julia funções semelhantes ao uso de funções currificadas da programação funcional. Veja o código da função "**adição2**" definida a seguir:

```
julia> function adição2(x)
            return function (y)
                        return x + y
                    end
       end
adição2 (generic function with 1 method)

julia> adição2(2)(3)
5
```

Observe que a função "**adição2**" para ser usada necessita da definição de dois argumentos definidos de forma isolada, cada qual com seu conjunto de parênteses. Isto se faz necessário a partir da definição interna de uma função anônima que recebe indiretamente o argumento "*Y*" e efetua a operação da soma com o argumento direto "*X*" recebido da função principal "**adição2**" retornando o resultado da operação a função principal. Uma função currificada pode ser utilizada dentro de outras funções, veja o exemplo:

```
julia> adicione10 = adição2(10)
#5 (generic function with 1 method)

julia> adicione10(5)
15
```

Operações currificadas podem ser definidas indiretamente como a definida para a variável "*ADICIONE10*" que passa a ser usada como função por ter sido atribuída a uma função atendendo o efeito de função de primeira classe e ordem superior.

A currificação caracteriza-se por ser uma técnica de definição de uma função que pode receber múltiplos parâmetros como argumentos de modo que esta função possa ser chamada como uma sequência de funções com argumentos de parâmetros únicos, sendo que este termo vem do matemático *Haskell Curry* (HIMITE, 2021). Isso significa converter uma função "**`f(a, b, c, ...)`**" em uma função como "**`f(a)(b)(c) ...`**".

CAPÍTULO 5

OPERAÇÕES AUXILIARES

Este capítulo apresenta a aplicação de alguns recursos da linguagem considerados auxiliares, pois são esses recursos responsáveis por grande parte da potencialidade de trabalho. São apresentados recursos de: randomização; números racionais e complexos; manipulação de data e hora, conversão entre tipos de dados, operações estatísticas, uso de gráficos e operações com quadros de dados.

5.1 - Randomização

Uma das aplicações mais dinâmicas em uma linguagem de programação é a capacidade de serem gerados por computador números aleatórios, mesmo que sejam número pseudoaleatórios. Neste sentido, Julia faz uso, por padrão, do algoritmo de geração de números pseudoaleatórios "**xoshiro256++**" criado no ano de 2019 por David Blackman e Sebastiano Vigna (**https://prng.di.unimi.it/xoshiro256plusplus.c**). O algoritmo utilizado na linguagem Julia pode operar tipos de dados de ponto flutuante, lógicos e inteiros a partir do uso da função "**rand**". Veja alguns exemplos:

```
julia> rand(Int, 4)  # gera vetor com 4 números inteiros de 64 bits
5-element Vector{Int64}:
 -9199331587784555184
  8068051363704836938
  -132895951160774136
 -1248462424427066039

julia> rand(Int8, 4)  # gera vetor com 4 números inteiros de 8 bits
5-element Vector{Int8}:
 -80
   1
 -35
 -80

julia> rand(UInt128, 4)  # gera vetor com 4 números inteiros positivos de 128 bits
5-element Vector{UInt128}:
 0x80bd96825a35218d0582cbf9573a61f5
 0x28331cd2bcb02aecc7f618da5f69fc8c
0xae68ab5730c82717153a00f0524d8072
 0x9f917722e8d9e47ab056ddbce178593e
```

Veja que no primeiro argumento da função "**rand**" define-se o tipo de dado e no segundo argumento a quantidade de valores gerados. Os tipos de dados usados podem ser: *Float16*, *Float32*, *Float64*, *BigFloat*, *Bool*, *Int8*, *UInt8*, *Int16*, *UInt16*, *Int32*, *UInt32*, *Int64*, *UInt64*, *Int128*, *UInt128* e *BigInt*.

```
julia> rand(4)  # gera vetor com 4 números de ponto flutuante de 64 bits
4-element Vector{Float64}:
 0.052824168263486904
 0.5958427442571373
 0.4127736573775882
 0.6676677371615384

julia> rand(Float32, (4, 3))  # gera matriz de 4 linhas e 5 colunas
4×3 Matrix{Float32}:
 0.959228   0.572275  0.128483
 0.444496   0.605728  0.85007
 0.103593   0.96625   0.425495
 0.0242174  0.124178  0.47105

julia> rand((0, 9))  # gera aleatoriamente o número 0 ou o número 9
0

julia> rand((0, 9))
9

julia> rand(0:9)  # gera número inteiro entre 0 e 9
8

julia> rand(-100:100)
-92

julia> rand(-10:-1)
-5

julia> rand(-10:-1) |> abs  # ou "abs(rand(-100:-1))"
3

julia> rand(10:20, 3)  # gera 3 números randômicos entre "10" e "20"
3-element Vector{Int64}:
 15
 10
 16

julia> rand(0.0:9.0)  # gera número de ponto flutuante entre 0 e 9
3.0

julia> rand()  # gera número de ponto flutuante entre 0 e 1
0.9313340484131136

julia> rand()
0.3287670950296768

julia> rand(Int64)  # gera número aleatório inteiro na faixa do tipo Int64
-1871583471660216700

julia> rand(UInt32)  # gera número aleatório inteiro positivo na faixa do tipo Int32
0xb7e58590

julia> 10rand()  # gera número aleatório multiplicando-o por 10
6.295354541977336
```

```
julia> 10 * rand()  # gera número aleatório multiplicando-o por 10
7.595324615370313

julia> 100rand()  # gera número aleatório multiplicando-o por 100
57.42808565160511

julia> sqrt(abs(rand()))  # raiz quadrada de número absoluto aleatório
0.8704726940397436

julia> rand() |> abs |> sqrt  # idêntica a operação anterior
0.7138541004095091

julia> sqrt(abs(rand())) / pi
0.3046506202029396

julia> (rand() |> abs |> sqrt) / pi  # idêntica a operação anterior
0.24048510237665355
```

Dentro da temática de randomização há outras funcionalidades de operação que dependem do uso de um pacote adicional chamado "**Random**". São funcionalidades que permitem o uso de ações mais dinâmicas, pois as ações apresentadas anteriormente fazem parte do pacote "**Base**" que é carregado automaticamente quando da execução do ambiente da linguagem Julia. Para tanto, execute a seguinte instrução:

```
julia> using Random
```

A partir do uso do pacote "**Random**" tornam-se disponíveis outras funcionalidades de apoio ao uso da randomização de valores.

O recurso padrão de randomização da linguagem Julia não permite a definição inicial de um valor semente. Um valor semente é a definição de um valor que inicializa a geração de valores aleatórios a partir deste valor. Desta forma, é possível especificar sequências de sorteios que se repetem quando o pacote "**Random**" está em uso. Veja os exemplos a seguir:

```
julia> Random.seed!(123)
TaskLocalRNG()

julia> rand(5)
5-element Vector{Float64}:
 0.521213795535383
 0.5868067574533484
 0.8908786980927811
 0.19090669902576285
 0.5256623915420473
```

Agora repita as duas ações anteriores e observe os resultados apresentados:

```
julia> Random.seed!(123)
TaskLocalRNG()

julia> rand(5)
5-element Vector{Float64}:
 0.521213795535383
 0.5868067574533484
 0.8908786980927811
 0.19090669902576285
 0.5256623915420473
```

Observe que ao se definir o valor de semente (*seed*) para a randomização obtém-se imediatamente a sua definição a mesma sequência de valores. Se uma nova semente não for especificada a próxima sequência de valores apresentada não se repetirá:

```
julia> rand(5)
5-element Vector{Float64}:
 0.3905882754313441
 0.044818005017491114
 0.933353287277165
 0.5805599818745412
 0.32723787925628356
```

O uso e definição de sementes para randomização é opcional, caso não seja especificado, Julia usa como semente um valor padrão gerado pelo próprio sistema.

Um valor de semente pode ser gerado a partir do algoritmo de *Mersenne-Twister* disponível no pacote "**Random**". O algoritmo *Mersenne-Twister* é um pseudogerador de número randômicos, baseado em números primos de *Mersenne*. Veja exemplos de uso da geração de semente baseadas nos primos de *Mersenne* e especificação de semente manual:

```
julia> MersenneTwister()
MersenneTwister(0x26b4954c5a7496219a44e5d2c647bfbb)

julia> MersenneTwister()
MersenneTwister(0xe36ebe557104a82f775e9d364800f403)

julia> MersenneTwister(123)  # especificação de uma semente (opcional)
MersenneTwister(123)
```

Para fazer uso do valor de semente gerado pelo algoritmo de *Mersenne-Twister* basta armazenar seu valor em uma variável de apoio e aplicar o uso dessa variável junto as funcionalidades do pacote "**Random**". Veja:

```
julia> gna = MersenneTwister()
MersenneTwister(0xc99db92699411f77c0b73d07546cfadf)

julia> gna
MersenneTwister(0xc99db92699411f77c0b73d07546cfadf, (0, 1002, 0, 5))
```

Veja na sequência alguns exemplos de uso do valor de semente *Mersenne-Twister*. Observe a criação de um arranjo com posições zero e seu preenchimento com valores aleatórios.

```
julia> arranjo = zeros(5)  # criar arranjo com posições zeradas
5-element Vector{Float64}:
 0.0
 0.0
 0.0
 0.0
 0.0

julia> rand!(gna, arranjo)  # preenchimento do arranjo com valores randômicos
5-element Vector{Float64}:
 0.7175754270405075
 0.8549472872086257
 0.28830171176441244
 0.23400653367645474
 0.6879376440093674
```

```
julia> arranjo
5-element Vector{Float64}:
 0.7175754270405075
 0.8549472872086257
 0.28830171176441244
 0.23400653367645474
 0.6879376440093674
```

Além da geração de valores randômicos numéricos é possível gerar sequências de caracteres alfabéticos ou alfanuméricos. Para tanto, considere o uso da função "**randstring**". Veja alguns exemplos de uso deste recurso:

```
julia> randstring()
"O2vZKWp2"

julia> randstring()
"SM6iZfIH"

julia> randstring("JANGM")
"MMAAJJAN"

julia> randstring("JANGM")
"GAJNAJAA"

julia> randstring("ACGT", 35)  # sequência aleatória de DNA
"TCGTGGTCCAGGCTGTGTATTTTGCAGGTAAAAGG"

julia> randstring(gna, 'A':'Z', 10)
"DMDCTDCFPD"

julia> randstring(MersenneTwister(321), 'A':'Z', 10)
"COMNUFYTLX"

julia> Random.seed!(123); randstring()
"WatBWO2v"
```

Para maiores detalhes e outras funcionalidades sobre randomização consulte a documentação oficial da linguagem.

5.2 - Data e hora

Um recurso necessário em programação é a manipulação de datas e horas. Neste sentido a linguagem Julia possui o pacote "**Dates**" que não faz parte direta do ambiente e de sua biblioteca de pacotes. Assim sendo, é necessário adicionar o pacote para uso, então execute a seguinte instrução:

```
julia> import Pkg; Pkg.add("Dates")
    Updating registry at `C:\Users\Augusto\.julia\registries\General.toml`
   Resolving package versions...
    Updating `C:\Users\Augusto\.julia\environments\v1.7\Project.toml`
  [ade2ca70] + Dates
  No Changes to `C:\Users\Augusto\.julia\environments\v1.7\Manifest.toml`

julia> using Dates
```

Observe que o ambiente interativo REPL da linguagem Jula aceita por vez uma linha de instrução antes do acionamento da tecla "**<Enter>**". Quando houver o desejo de executar mais de uma instrução use "**;**" para separar as instruções.

A partir deste ponto é possível realizar diversas operações com o uso de data e hora. Observe na próxima sequência de exemplos formas de fazer a extração ou definição de data e hora pessoal ou do sistema:

```
julia> meu_nascimento = Dates.Date(1965, 04, 26)
1965-04-26

julia> hor_nascimento = Dates.Time(17, 35, 00)
17:35:00

julia> Dates.year(meu_nascimento)  # ano de uma data
1965

julia> Dates.month(meu_nascimento)  # mês de uma data
4

julia> Dates.day(meu_nascimento)  # dia do mês de uma data
26

julia> Dates.dayofmonth(meu_nascimento)  # dia do mês de uma data
26

julia> Dates.dayofweek(meu_nascimento)  # dia da semana
1

julia> Dates.dayofweekofmonth(meu_nascimento)  # dia da semana no mês (4 = 4a. seg. no mês)
4

julia> Dates.daysofweekinmonth(meu_nascimento)  # número total de dias da semana no mês
4

julia> Dates.dayname(meu_nascimento)  # nome do dia da semana
"Monday"

julia> Dates.dayabbr(meu_nascimento)  # nome do dia da semana abreviado
"Mon"

julia> Dates.monthname(meu_nascimento)  # nome do mês
"April"

julia> Dates.monthabbr(meu_nascimento)  # nome do mês abreviado
"Apr"

julia> Dates.daysinmonth(meu_nascimento)  # quantidade de dias no mês
30

julia> Dates.dayofyear(meu_nascimento)  # quantidade de dias no ano a partir de 01/01
116

julia> Dates.daysinyear(meu_nascimento)  # quantidade total de dias do ano | 366 se ano bissexto
365

julia> Dates.yearmonthday(meu_nascimento)  # data como tupla com ano, mês e dia
(1965, 4, 26)
```

```
julia> Dates.yearmonth(meu_nascimento)  # ano e mês de uma data
(1965, 4)

julia> Dates.isleapyear(meu_nascimento)  # indica se o ano da data é bissexto
false

julia> Dates.week(meu_nascimento)  # ordinal da semana dentro do ano
17

julia> Dates.dayofweekofmonth(meu_nascimento)  # valor ordinal do mês no ano
4

julia> Dates.quarterofyear(meu_nascimento)  # quadrimestre do ano em que a data está
2

julia> Dates.dayofquarter(meu_nascimento)  # quantidade de dias em certo quadrimestre
26

julia> Dates.hour(hor_nascimento)  # hora
17

julia> Dates.minute(hor_nascimento)  # minutos
35

julia> Dates.second(hor_nascimento)  # segundos
0

julia> Dates.millisecond(hor_nascimento)  # milissegundos
0

julia> Dates.now()
2022-07-12T17:36:01.946

julia> pega_tempo = Dates.Time(Dates.now())  # pega HH:MM:SS.ms
17:19:13.345

julia> pega_data = Dates.Date(Dates.now())  # pega AAAA-MM-DD
2022-07-12

julia> Dates.today()  # data do dia
2022-07-12

julia> Dates.now(Dates.UTC)  # data no formato UTC
2022-07-12T20:26:56.083

julia> Dates.Date(2022)  # construtor de data a partir da indicação de ano
2022-01-01

julia> Dates.Date(2022, 12)  # construtor de data a partir da indicação de ano e mês
2022-12-01

julia> Dates.Date(2022, 12, 25)  # construtor de data a partir de data completa
2022-12-25
```

```
julia> Dates.Time(17)  # construtor de hora a partir da indicação de hora cheia
17:00:00

julia> Dates.Time(17, 55)  # construtor de hora a partir da indicação de hora e minuto
17:55:00

julia> Dates.Time(17, 55, 35)  # construtor de hora a partir da indicação de hora completa
17:55:35

julia> Dates.DateTime(2022, 12, 25, 17, 55, 35)  # definição de data e hora
2022-12-25T17:55:35
```

A partir de uma visão sobre definições de data e hora e obtenção de diversas informações sobre as definições de data e hora do sistema é importante conhecer algumas operações possíveis de serem estabelecidas sobre data e hora.

Uma das operações mais comuns entre datas é realizar operações aritméticas sobre faixas de datas. Veja a seguir alguns exemplos e considerações.

Considere por exemplo saber a idade a partir do ano de seu nascimento de uma pessoa ou ano de inauguração de uma empresa, por exemplo. Veja a instrução seguinte que apresenta a idade de uma pessoa:

```
julia> Dates.today() - meu_nascimento
20896 days
```

Veja que o resultado apresentado é a idade em quantidade de dias. O valor em dias é ideal para aplicações comerciais para saber, por exemplo, a quantidade de dias em atraso de um pagamento, mas pode não ser útil para outras aplicações. Para converter a quantidade de dias em anos há duas possibilidades: calculando-se manualmente ou realizando o uso de um pacote adicional chamado "**DayCounts**". Assim sendo, deixando por conta do pacote adicional execute as seguintes instruções:

```
julia> using DayCounts
 │ Package DayCounts not found, but a package named DayCounts is available from a registry.
 │ Install package?
 │   (@v1.7) pkg> add DayCounts
 └ (y/n) [y]: y
   Resolving package versions...
   Installed DayCounts — v0.1.0
    Updating `C:\Users\Augusto\.julia\environments\v1.7\Project.toml`
  [44e31299] + DayCounts v0.1.0
    Updating `C:\Users\Augusto\.julia\environments\v1.7\Manifest.toml`
  [44e31299] + DayCounts v0.1.0
Precompiling project...
  1 dependency successfully precompiled in 3 seconds (120 already precompiled)

julia> yearfrac(meu_nascimento, Dates.today(), DayCounts.ActualActualISDA())
57.21369863013699

julia> trunc(Int, yearfrac(meu_nascimento, Dates.today(), DayCounts.ActualActualISDA()))
57
```

A função "**yearfrac**" retorna seu resultado como sendo um valor de ponto flutuante (número fracionário) de anos entre duas datas, daí para ver este resultado como um número inteiro usa-se como apoio a função "**trunc**" com a definição de seu primeiro argumento como "**Int**". Veja que a função "**yearfrac**" faz uso de três argumentos, sendo o primeiro a data inicial, o segundo a data final e o terceiro a indicação do algoritmo *ISDA* usado para a contagem dos anos entre as datas (**https://docs.juliahub.com/DayCounts/Yny90/0.1.0/api/**).

Observe que há então uma maneira de obter-se a quantidade de dias e uma maneira de obter-se a quantidade de anos entre duas datas. Falta indicar a quantidade de meses entre duas datas. Para tanto, observe a seguinte instrução:

```
julia> length(meu_nascimento:Dates.Month(1):Dates.today()) - 1
686
```

Atente para a utilização da função "**length**" que tem por capacidade extrair o tamanho de um arranjo. Veja que para esta situação define-se um arranjo entre os limites da data menor em relação a data maior, especificando entre as datas o incremento mês a mês com a função "**Dates.Month**". Note que é possível operar nos cálculos entre datas com funções matemáticas.

Agora pensando um pouco mais especificamente veja como obter o número de semanas entre duas datas a partir da instrução:

```
julia> Dates.canonicalize(Dates.CompoundPeriod(Dates.today() - meu_nascimento))
2985 weeks, 1 day
```

Para gerar a quantidade de semanas usa-se primeiro o construtor "**Dates.CompoundPeriod**" que a partir da indicação de um número fixo de certo período de tempo (como a quantidade de dias entre duas datas) retorna como seu resultado um período composto deste tempo. O valor retornado pelo construtor "**Dates.CompoundPeriod**" é usado como argumento da função "**Dates.canonicalize**" que por sua vez reduz o período composto de tempo em sua forma canônica, como: semanas, dias, horas e minutos. Veja alguns exemplos desse recurso:

```
julia> Dates.canonicalize(Dates.CompoundPeriod(Dates.Hour(11), Dates.Hour(1)))
12 hours

julia> Dates.canonicalize(Dates.CompoundPeriod(Dates.Hour(11), Dates.Hour(2)))
13 hours

julia> Dates.canonicalize(Dates.CompoundPeriod(Dates.Hour(11), Dates.Hour(13)))
1 day

julia> Dates.canonicalize(Dates.CompoundPeriod(Dates.Hour(11), Dates.Hour(24)))
1 day, 11 hours

julia> Dates.canonicalize(Dates.CompoundPeriod(Dates.Hour(12) + Dates.Hour(24)))
1 day, 12 hours

julia> Dates.canonicalize(Dates.CompoundPeriod(Dates.Hour(-2), Dates.Minute(3)))
-1 hour, -57 minutes

julia> Dates.canonicalize(Dates.CompoundPeriod(Dates.Hour(-1), Dates.Minute(2)))
-58 minutes

julia> Dates.canonicalize(Dates.CompoundPeriod(Dates.Month(2), Dates.Week(-3)))
2 months, -3 weeks

julia> Dates.canonicalize(Dates.CompoundPeriod(Dates.Minute(80000)))
7 weeks, 6 days, 13 hours, 20 minutes

julia> Dates.canonicalize(Dates.CompoundPeriod(Dates.Minute(80987)))
8 weeks, 5 hours, 47 minutes
```

Para as ações apresentadas foram usados como apoio alguns construtores de tratamento de hora, minuto, mês e semana. Atente para o fato de que todo construtor em Julia é iniciado com letra maiúscula.

Dias, meses e anos podem ser adicionados e subtraídos de certa data a partir do uso e acesso aos construtores de data. Para este tipo de ação considere os exemplos:

```
julia> meu_nascimento + Dates.Day(20)  # adiciona 20 dias na data
1965-05-16

julia> meu_nascimento + Dates.Month(2)  # adiciona 2 meses na data
1965-06-26

julia> meu_nascimento + Dates.Year(57)  # adiciona 57 anos na data
2022-04-26

julia> meu_nascimento + Dates.Year(10) + Dates.Month(5) + Dates.Day(20)
1975-10-16

julia> meu_nascimento - Dates.Day(20)  # subtraí 20 dias na data
1965-04-06

julia> meu_nascimento - Dates.Month(2)  # subtraí 2 meses na data
1965-02-26

julia> meu_nascimento - Dates.Year(57)  # subtraí 57 anos na data
1908-04-26
```

Além dos cálculos com datas é possível realizar cálculos com margens de tempo, baseados em relógio a partir do pacote "**Dates**". Veja a apresentação da diferença de tempo entre dois horários especificados.

```
julia> tempo = Dates.Time(17, 55, 35) - Dates.Time(12, 58, 40)
17815000000000 nanoseconds
```

Observe que o retorno de diferença de tempo é apresentado em nano segundos e para converter este tempo em horas, minutos, segundos e milésimos de segundo atente para os próximos exemplos:

```
julia> Dates.CompoundPeriod(Dates.Nanosecond(tempo))  # nano segundos em nano segundos
17815000000000 nanoseconds

julia> Dates.CompoundPeriod(Dates.Microsecond(tempo))  # nano segundos em micro segundos
17815000000 microseconds

julia> Dates.CompoundPeriod(Dates.Millisecond(tempo))  # nano segundos em milésimos de segundos
17815000 milliseconds

julia> Dates.CompoundPeriod(Dates.Second(tempo))  # nano segundos em segundos
17815 seconds

julia> round(Dates.Second(tempo), Dates.Minute)  # segundos em minutos
297 minutes

julia> round(Dates.Second(tempo), Dates.Hour)  # segundos em horas
5 hours
```

Horas, minutos e segundos podem ser adicionados e subtraídos de certo horário a partir do uso e acesso aos construtores de hora. Para este tipo de ação considere os exemplos:

```
julia> hor_nascimento + Dates.Hour(2)  # adiciona 2 horas da hora
19:35:00

julia> hor_nascimento + Dates.Minute(10)  # adiciona 10 minutos da hora
17:45:00

julia> hor_nascimento + Dates.Second(57)  # adiciona 57 segundos da hora
17:35:57

julia> hor_nascimento + Dates.Hour(10) + Dates.Minute(5) + Dates.Second(20)
03:40:20

julia> hor_nascimento - Dates.Hour(20)  # subtraí 20 horas da hora
21:35:00

julia> hor_nascimento - Dates.Minute(2)  # subtraí 2 minutos da hora
17:33:00

julia> hor_nascimento - Dates.Second(57)  # subtraí 57 segundos da hora
17:34:03
```

Agora como exemplo contextualizado, considere apresentar a duração do tempo entre a definição de dois horários específicos.

```
julia> hi = Dates.Time(12, 32, 45)  # horário inicial da viagem
12:32:45

julia> hf = Dates.Time(17, 25, 11)  # horário final da viagem
17:25:11
```

Observe o uso do campo "**instant**" disponível junto as variáveis de tempo "*HI*" (hora inicial) e "*HF*" (hora final). Este campo permite capturar da hora especificada a quantidade de nano segundos existente sobre este valor e a partir desse valor realizar operações aritméticas.

Como outro exemplo considere a entrada de um automóvel em um estacionamento e apresente o tempo de permanência na saída para um mesmo dia. Observe as instruções seguintes:

```
julia> t1 = Dates.DateTime(2022, 10, 25, 13, 24, 34)
2022-10-25T13:24:34

julia> t2 = Dates.DateTime(2022, 10, 27, 19, 55, 09)
2022-10-25T19:55:09

julia> Dates.canonicalize(Dates.CompoundPeriod(Dates.DateTime(t2) - Dates.DateTime(t1)))
6 hours, 30 minutes, 35 seconds
```

Considere agora a entrada do automóvel em um determinado dia e hora e a saída em outro dia diferente do dia da entrada. Vejas as seguintes instruções:

```
julia> e1 = Dates.DateTime(2022, 10, 23, 12, 55, 23)
2022-10-23T12:55:23

julia> e2 = Dates.DateTime(2022, 10, 26, 15, 43, 12)
2022-10-26T15:43:12
```

```
julia> Dates.canonicalize(Dates.CompoundPeriod(Dates.DateTime(e2) - Dates.DateTime(e1)))
3 days, 2 hours, 47 minutes, 49 seconds
```

Além das operações de manipulação de data e hora tem-se ainda recursos para formatação desses dados com a finalidade de indicar formas diferentes de duas apresentações, principalmente em tela (ecrã) a partir da utilização dos construtores "**DateFormat**", "**Date**" e "**DateTime**". A formatação ocorre a partir da definição de uma série de códigos indicados na tabela 5.1.

Código	Referência	Formato	Exemplo de ocorrência
y	Dígito do ano.	"yyyy" ou "yy"	"YYYY" = 2022 ou "YY" = 22
m	Dígito do mês.	"m"	4 ou 04
u	Nome abreviado do mês.	"u"	Apr
U	Nome completo do mês.	"U"	April
e	Nome abreviado do dia da semana.	"e"	Mon
E	Nome completo do dia da semana.	"E"	Monday
d	Dia apresentado na forma de número.	"d"	9 ou 09
H	Dígito da hora.	"HH"	00
M	Dígito do mês.	"MM"	00
S	Dígito do segundo.	"S"	00
s	Dígito do milissegundo.	"s"	.000

Tabela 5.1 - Códigos de formatação para data e hora.

Para realizar testes rápidos com o uso dos códigos de formatação de data e hora a partir dos construtores "**DateTime**", "**Date**" e "**DateFormat**" execute as seguintes instruções:

```
julia> Dates.today()  # mostra data do sistema
2022-07-12

julia> Dates.now()  # mostra data e hora do sistema no formato UTC/ISO-8601
2022-07-12T17:45:55.211

julia> Dates.Date(Dates.now())  # mostra data do sistema - similar: "Dates.today()"
2022-07-12

julia> Dates.Time(Dates.now())  # mostra hora do sistema
17:45:55.211

julia> formato = Dates.DateFormat("dd/mm/yyyy")  # definer formato brasileiro
dateformat"dd/mm/yyyy"

julia> Dates.format(Dates.today(), formato)  # mostra data do sistema no formato "DD/MM/AAAA"
"12/07/2022"

julia> Dates.DateTime("19650426 173500", "yyyymmdd HHMMSS")
1965-04-26T17:35:00

julia> Dates.DateTime("26/04/1965 17:35", "dd/mm/yyyy HH:MM")
1965-04-26T17:35:00
```

```
julia> Dates.Date("Mon, 26 Apr 1965", "e, d u y")
1965-04-26

julia> Dates.DateTime("Mon, 26 Apr 1965 17:35:00", "e, d u y H:M:S")
1965-04-26T17:35:00

julia> Dates.format(Dates.now(), "e, dd u yyyy HH:MM:SS")
"Tue, 12 Jul 2022 17:46:21"

julia> Dates.format(Dates.now(), "E, dd U yyyy HH:MM:SS")
"Tuesday, 12 July 2022 17:46:21"

julia> dataUTC = Dates.DateTime("Thu, 14 Jul 2022 13:5:9", "e, d u y H:M:S")
2022-07-14T13:05:09

julia> Dates.format(dataUTC, "e: dd u yy, HH.MM.SS")  # com zeros nas unidades
"Thu: 14 Jul 22, 13.05.09"

julia> Dates.format(dataUTC, "e | dd/mm/yy - HHhMM")
"Thu | 14/07/22 - 13h05"

julia> Dates.format(dataUTC, "E: dd U yy, H.M.S")  # sem zeros nas unidades
"Thursday: 14 July 22, 13.5.9"
```

O pacote "**Dates**" é um recurso que possui muitas funcionalidades referentes ao tratamento de tempo. Destaca-se nesta entoada um recurso que permite operar com a frequência de períodos brevemente apresentado anteriormente. Neste sentido, cabe aqui algumas outras considerações a este respeito, pois os períodos são uma definição humana para marcação de agrupamento de tempo como: dias, meses, bimestres, trimestres, anos, etc.

Veja mais algumas operações aritméticas que podem ser realizadas com alguns construtores de tempo, ressaltando que a operação de multiplicação não é permitida:

```
julia> Dates.Year(1)
1 year

julia> Dates.Year(5)
5 years

julia> Dates.Year(10)
10 years

julia> Dates.Month(1)
1 month

julia> Dates.Month(4)
4 months

julia> Dates.Day(1)
1 day

julia> Dates.Day(30)
30 days

julia> Dates.Year(5) + Dates.Year(2)
7 years
```

```
julia> Dates.Year(5) - Dates.Year(2)
3 years

julia> Dates.Year(5) / Dates.Year(2)
2.5

julia> div(Dates.Year(5), Dates.Year(2))
2

julia> Dates.Year(5) % Dates.Year(2)
1 year

julia> Dates.Year(8) / 4
2 years
```

Todos os recursos de manipulação de tempo são possíveis de uso com intervalos estabelecidos a partir de arranjos. Veja alguns exemplos:

```
julia> periodo = Dates.Date(2022, 1, 1):Dates.Day(1):Dates.Date(2022, 2, 28)
Date("2022-01-01"):Day(1):Date("2022-02-28")

julia> bimestre = collect(periodo)
59-element Vector{Date}:
 2022-01-01
 2022-01-02
 2022-01-03
 2022-01-04
 2022-01-05
 2022-01-06
 2022-01-07
 2022-01-08
 2022-01-09
 2022-01-10
 2022-01-11
 2022-01-12
 2022-01-13
 ⋮
 2022-02-17
 2022-02-18
 2022-02-19
 2022-02-20
 2022-02-21
 2022-02-22
 2022-02-23
 2022-02-24
 2022-02-25
 2022-02-26
 2022-02-27
 2022-02-28
```

Observe que foi gerado um intervalo diário entre os meses de janeiro e fevereiro de 2022. Veja que a listagem completa não é apresentada (por ser muito grande) dos dias entre o período estabelecido.

Considerando que se queira obter do arranjo criado as datas apenas dos dias úteis, ou seja, desejando-se excluir so dias de sábado e domingo. Observe em seguida a obtenção da listagem de dias úteis a partir da ação de filtragem do arranjo:

```
julia> filter(x -> Dates.dayname(x) != "Saturday" && Dates.dayname(x) != "Sunday", periodo)
41-element Vector{Date}:
 2022-01-03
 2022-01-04
 2022-01-05
 2022-01-06
 2022-01-07
 2022-01-10
 2022-01-11
 2022-01-12
 2022-01-13
 2022-01-14
 2022-01-17
 2022-01-18
 2022-01-19
 ⋮
 2022-02-11
 2022-02-14
 2022-02-15
 2022-02-16
 2022-02-17
 2022-02-18
 2022-02-21
 2022-02-22
 2022-02-23
 2022-02-24
 2022-02-25
 2022-02-28
```

Veja a apresentação da listagem de todos os finais de semana entre o primeiro bimestre de 2022.

```
julia> filter(x -> Dates.dayname(x) == "Saturday" || Dates.dayname(x) == "Sunday", periodo)
18-element Vector{Date}:
 2022-01-01
 2022-01-02
 2022-01-08
 2022-01-09
 2022-01-15
 2022-01-16
 2022-01-22
 2022-01-23
 2022-01-29
 2022-01-30
 2022-02-05
 2022-02-06
 2022-02-12
 2022-02-13
 2022-02-19
 2022-02-20
 2022-02-26
 2022-02-27
```

A definição de períodos de tempo sobre datas permite encontrar a recorrência de certa data em um certo intervalo de tempo. Considere apresentar uma lista contendo todas as sextas-feiras com dia 13 entre as datas de 01/01/2022 até 31/12/2027.

```
julia> quinquênio  = Dates.Date(2022, 1, 1):Dates.Day(1):Dates.Date(2027, 12, 31)
Date("2022-01-01"):Day(1):Date("2027-12-31")

julia> filter(x -> Dates.dayname(x) == "Friday" && Dates.day(x) == 13, quinquênio)
10-element Vector{Date}:
 2022-05-13
 2023-01-13
 2023-10-13
 2024-09-13
 2024-12-13
 2025-06-13
 2026-02-13
 2026-03-13
 2026-11-13
 2027-08-13
```

Na listagem apresenta vê-se que no período de cinco anos os anos que possuem o menor número de sextas-feiras 13 são 2022, 2025 e 2027, já o ano que possui mais sextas-feiras 13 é 2026.

5.3 - Números racionais e complexos

As linguagens de programação são ferramentas que facilitam o acesso a mais comum das tarefas executada por um computador: realizar operações aritméticas dentro do universo do processamento matemática. Para atender a este quesito é comum encontrar, como visto, o uso de valores numéricos pertencentes aos conjuntos de números inteiros e números reais na forma de valores numéricos de ponto flutuante.

Operar com números inteiros e de ponto flutuante é uma ação comum a qualquer linguagem de programação, mas há algumas linguagens como Julia que operam valores racionais e complexos. Os números racionais em Julia são representados a partir da proporção de valores inteiros na forma de frações e os números complexos são compostos por uma parte real e outra parte imaginária, de modo que a parte imaginária corresponde à raiz quadrada de um número negativo.

Quanto aos números racionais estes já foram anteriormente apresentados de forma despretensiosa quando do uso do operador aritmético "**//**". Veja alguns exemplos:

```
julia> 2 // 3  # definição do valor: dois terços
2//3

julia> 1 // 2  # definição do valor: meio
1//2
```

Um detalhe anteriormente não comentado é a possibilidade de se fazer a extração do numerador e denominador de um valor numérico racional respectivamente a partir das funções "**numerator**" e "**denominator**" demonstrada a seguir:

```
julia> r1 = 5 // 3
5//3

julia> numerator(r1)
5

julia> denominator(r1)
3
```

Valores racionais podem ser usados livremente em operações aritméticas diversas. Veja alguns exemplos:

```
julia> 1//7 + 2//7 + 3//7  # soma de frações
6//7

julia> (1 + 2 + 3) // 7
6//7

julia> 1//4 + 2//5
13//20

julia> 1//3 + 2//5 + 3//9
16//15

julia> 3//7 - 2//7  # subtração de frações
1//7

julia> 8//7 - 2//7 - 3//7
3//7

julia> 8//7 - 2//6 - 3//5
22//105

julia> 8//7 * 2//7  # multiplicação de frações
16//49

julia> 1//2 * 2//3 * 3//4
1//4

julia> 1//2 / 2//3  # divisão de frações
3//4

julia> 1//2 / 2//2 / 3//2
1//3

julia> 1//2 / 5//3 / 10//7
21//100

julia> 1//3 + 3//5 * 4//9 - 5//7 / 11//19  # múltiplas operações
-244//385

julia> (1//3 + 3//5) * 4//9 - 5//7 / 11//19
-8513//10395

julia> 9//7 % 2//6  # resto de divisão entre frações
2//7

julia> 3//4 ^ 2//5  # exponenciação entre frações
3//80
```

```
julia> sqrt(3//4)
0.8660254037844386

julia> typeof(3//4)
Rational{Int64}
```

Observe que as definições de valores racionais são apresentadas na forma racional. Desejando ver os valores na forma de ponto flutuante é indicado o uso da função "**float**" demonstrada a seguir:

```
julia> float(5 // 3)
1.6666666666666667

julia> float(1//2 / 5//3 / 10//7)  # 21//100 = 0.21
0.21

julia> 1//2 == 1/2
true

julia> float(3//4 ^ 2//5)
0.0375
```

Quanto aos números complexos, este são em Julia valores definidos a partir de uma parte real e uma parte imaginária representada pela constante global "**im**" que indica a raiz quadrada do valor "**-1**", sendo indicado pelo formato "**9+9im**", em que o primeiro "**9**" representa a parte real e o segundo "**9**" representa a parte imaginária. Veja a indicação de alguns valores imaginários e a identificação de suas partes:

```
julia> 2 + 34im
2 + 34im

julia> 2 + (-34im)
2 - 34im

julia> -5 + 20im
-5 + 20im

julia> real(2 + 34im)  # extraí a parte real de um número complexo
2

julia> imag(2 + 34im)  # extraí a parte imaginária de um número complexo
34
```

É definido que não existe resultado para o cálculo de uma raiz quadrada de número negativos. No entanto, se considerar que o número negativo seja complexo esta operação é possível. Veja em seguida os resultados apresentados para a extração da raiz quadrada de "**25**" e atentativa de extração da raiz quadrada de "**-25**":

```
julia> sqrt(25)
5.0
```

```
julia> sqrt(-25)
ERROR: DomainError with -25.0:
sqrt will only return a complex result if called with a complex argument. Try sqrt(Complex(x)).
Stacktrace:
 [1] throw_complex_domainerror(f::Symbol, x::Float64)
   @ Base.Math .\math.jl:33
 [2] sqrt
   @ .\math.jl:567 [inlined]
 [3] sqrt(x::Int64)
   @ Base.Math .\math.jl:1221
 [4] top-level scope
   @ REPL[54]:1
```

Veja que a tentativa de executar o cálculo "**sqrt(-25)**" gerou a apresentação de um erro, mas também gerou a apresentação de uma mensagem informando que a função retornará um resultado complexo se for chamada com um argumento complexo e orienta para tentar fazer uso da forma "**sqrt(Complex(x))**". Veja o próximo exemplo:

```
julia> sqrt(Complex(-25))
0.0 + 5.0im
```

Observe que neste caso o valor da raiz quadrada de um número negativo é indicado na parte imaginária do seu resultado e desejando apresentar apenas este valor basta executar a seguinte instrução:

```
julia> imag(sqrt(Complex(-25)))
5.0
```

O uso de construtor "**Complex**" pode ser usado para indicar valores de ponto flutuante na forma imaginária. Veja alguns exemplos:

```
julia> Complex(-22)
-22 + 0im

julia> Complex(15)
15 + 0im

julia> a = 7.5
7.5

julia> b = 3.3
7.5

julia> Complex(a, b)  # definição de número complexo
7.5 + 3.3im
```

A partir de uma visão geral sobre a definição e alguns tratamentos de números complexos observe a indicação de algumas operações aritméticas com esses números:

```
julia> 1 + 2im + 2 + 3im  # adição de números imaginários
3 + 5im
```

```
julia> (1 + 2im) + (2 + 3im)  # parênteses deixa a indicação da operação mais clara
3 + 5im

julia> (1 + 2im) + (-2 + 3im)
-1 + 5im

julia> (1 - 2im) + (-2 + 3im)
-1 + 1im

julia> (1 - 2im) + (2 + 3im)
3 + 1im

julia> (1 - 2im) + (2 - 3im)
3 - 5im

julia> (1 + 2im) - (2 + 3im)  # subtração de números imaginários
-1 - 1im

julia> (1 + 2im) - (-2 + 3im)
3 - 1im

julia> (1 - 2im) - (-2 + 3im)
3 - 5im

julia> (1 - 2im) - (2 + 3im)
-1 - 5im

julia> (1 - 2im) - (2 - 3im)
-1 + 1im

julia> (1 + 2im) * (2 + 3im)  # multiplicação de números imaginários
-4 + 7im

julia> (1 + 2im) * (-2 + 3im)
-8 - 1im

julia> (1 - 2im) * (-2 + 3im)
4 + 7im

julia> (1 - 2im) * (2 + 3im)
8 - 1im

julia> (1 - 2im) * (2 - 3im)
-4 - 7im

julia> (1 + 2im) / (2 + 3im)  # divisão de números imaginários
0.6153846153846154 + 0.07692307692307691im
```

```
julia> (1 + 2im) / (-2 + 3im)
0.30769230769230776 - 0.5384615384615384im

julia> (1 - 2im) / (-2 + 3im)
-0.6153846153846154 + 0.07692307692307691im

julia> (1 - 2im) / (2 + 3im)
-0.30769230769230776 - 0.5384615384615384im

julia> (1 - 2im) / (2 - 3im)
0.6153846153846154 - 0.07692307692307691im

julia> (1 - 2im) ^ (2 - 3im)  # exponenciação de números imaginários
-0.01513267242272266 + 0.179867483913335im

julia> (1 - 2im) ^ 2.0
-3.0 - 4.0im

julia> 3(1 - 2im) ^ 2.0  # # múltiplas operações
-9.0 - 12.0im

julia> 3 * (1 - 2im) ^ 2.0
-9.0 - 12.0im

julia> (1 - 2im) * (2 + 3im) + (5 + 25im) / 2.0 * 2//3
9.666666666666666 + 7.333333333333332im

julia> sqrt(3 - 2im)  # raiz quadrada de número imaginário
1.8173540210239707 - 0.5502505227003375im

julia> sqrt(3 + 2im)
1.8173540210239707 + 0.5502505227003375im

julia> sqrt(25im)
3.5355339059327378 + 3.5355339059327373im
```

Entre as operações previstas para números imaginários há algumas funcionalidades auxiliares úteis para a realização de números imaginários, das quais já foram apresentas as funções "**real**" e "**imag**". Veja agora as funções de manipulação "**conj**" que efetua o conjugado de um número complexo e "**abs**" que pode ser usada para apresentar o absoluto de um número complexo.

```
julia> conj(1 - 2im)
1 + 2im

julia> conj(7 + 2im)
7 - 2im

julia> abs(7 + 2im)
7.280109889280518
```

```
julia> abs(7 - 2im)
7.280109889280518

julia> abs(-7 - 2im)
7.280109889280518

julia> abs(-7 + 2im)
7.280109889280518
```

5.4 - Conversões e bases numéricas

Em alguns momentos anteriores foi usada brevemente a função "**parse**" para o tratamento e conversão numérica da ação de entrada de dados por meio da função "**readline**". A função "**readline**" efetua a leitura de uma sequência de caracteres, não importando se são números, como uma sequência alfanumérica. Daí, p motivo de usar a função de conversão "**parse**".

A função "**parse**" analisa o conteúdo de uma cadeia como sendo um número e permite transformar os dados da sequência em uma forma numérica estabelecida, podendo ser gerado números inteiros, de ponto flutuante e complexos. O efeito inverso é conseguido com o uso da função "**string**".

Se o número a ser convertido for inteiro poderá ser indicado uma base numérica diferente da decimal para apresentar o número decimal a partir da base (de "5" até "62") indicada, se a sequência alfanumérica tiver definido um ponto a conversão considerará um número de ponto flutuante e se for indicado na sequência de caracteres um número no formato complexo ao estilo "**R±Iim**" também será convertido. Caso a cedia de caracteres não contenha um número será retornado um erro.

```
julia> parse(Int, "123")  # converte cadeia numérica em número
123

julia> string(123, base=10)  # converte número em cadeia numérica
"123"

julia> parse(Int8, "123")
123

julia> parse(Int32, "123")
123

julia> parse(Int64, "123")
123

julia> parse(Int, "12340", base=5)  # número "12340" base "5" = "970" base "10"
970

julia> string(970, base=5)
"12340"

julia> parse(Int, "12345670", base=8)
2739128
```

```
julia> string(2739128, base=8)
"12345670"

julia> parse(Int, "D431", base=16)
54321

julia> string(54321, base=16)
"d431"

julia> parse(Int, "G9", base=62)
1001

julia> string(1001, base=62)
"G9"

julia> parse(Float64, "1.3e-4")
0.00013

julia> parse(Complex{Float64}, "1.3e+4 + 2.7im")
13000.0 + 2.7im
```

Há outras conversões que podem ser estabelecidas com o uso da função "**convert**" que opera com dois argumentos: primeiro argumento a definição de um tipo e o segundo argumento a definição do valor.

Se o primeiro argumento for a definição de um tipo de dado "**Int**", o segundo argumento será aceito se este puder ser representado pelo tipo inteiro. Veja os dois exemplos seguintes, cujo primeiro é aceito e o segundo gera erro.

```
julia> convert(Int, 1.0)
1

julia> convert(Int, 1.1)
ERROR: InexactError: Int64(1.1)
Stacktrace:
 [1] Int64
   @ .\float.jl:812 [inlined]
 [2] convert(#unused#::Type{Int64}, x::Float64)
   @ Base .\number.jl:7
 [3] top-level scope
   @ REPL[70]:1
```

Observe que o primeiro exemplo é bem-sucedido, mas o segundo não por ser o valor "**1.1**" um dado não compatível com o tipo "**Int**" ou suas variantes.

Se o primeiro argumento for um tipo de dado "**AbstractFloat**" ou tipo "**Rational**" o valor do segundo argumento será retornado com um valor o mais próximo de sua real precisão.

```
julia> r2 = float(5 // 3)
1.6666666666666667

julia> r2
1.6666666666666667
```

```
julia> convert(Rational{Int8}, r2)
5//3

julia> convert(Rational{Int16}, r2)
5//3

julia> convert(Rational{Int32}, r2)
5//3

julia> convert(Rational{Int64}, r2)
7505999378950827//4503599627370496

julia> float(7505999378950827 // 4503599627370496)
1.6666666666666667
```

A função "**convert**" com a definição do tipo de dado racional "**Rational**" permite obter a partir de um valor numérico de ponto flutuante sua versão em formato fracionário.

A conversão inversa de valor inteiro em ponto flutuante com a função "**convert**" não apresenta problemas.

```
julia> convert(Float64, 1)
1.0
```

Veja exemplo de um valor inteiro de 64 bits convertido em formato hexadecimal e convertido de hexadecimal para sua versão em ponto flutuante:

```
julia> valor1 = 10
10

julia> typeof(valor1)
Int64

julia> valor2 = convert(UInt8, valor1)
0x0a

julia> typeof(valor2)
UInt8

julia> valor3 = convert(AbstractFloat, valor1)
10.0

julia> valor4 = convert(AbstractFloat, valor2)
10.0

julia> typeof(valor3)
Float64

julia> typeof(valor4)
Float64
```

Dentro da temática de conversão há a função "**oftype**" que tem por finalidade passar o tipo de dado do primeiro argumento para o segundo argumento.

```
julia> oftype(0.0, 3)  # transfere o tipo de ponto flutuante para inteiro
3.0

julia> oftype(0, 2.0)
2

julia> oftype(9, 8.0)
8

julia> oftype(9, 8.5)
ERROR: InexactError: Int64(8.5)
Stacktrace:
 [1] Int64
   @ .\float.jl:812 [inlined]
 [2] convert
   @ .\number.jl:7 [inlined]
 [3] oftype(x::Int64, y::Float64)
   @ Base .\essentials.jl:396
 [4] top-level scope
   @ REPL[89]:1
```

A conversão com a função "**oftype**" tem a mesma característica existente na função "**convert**", ou seja, o segundo argumento só é aceito se este puder ser representado pelo tipo definido no primeiro argumento. A solução básica para evitar este erro é fazer uso da função "**trunc**" ou da função "**floor**" para capar a mantissa do número, considerando-se apenas sua parte inteira quando definido seu primeiro argumento com o tipo "**Int**".

```
julia> trunc(Int, 1.1)
1

julia> trunc(1.1)
1.0

julia> floor(1.1)
1.0

julia> floor(1.1, digits=0)
1.0

julia> oftype(0, trunc(1.1))
1

julia> oftype(0, floor(1.1))
1

julia> convert(Int, trunc(1.1))
1
```

```
julia> convert(Int, floor(1.1))
1
```

A conversão de dados de ponto flutuante em inteiro com perda de mantissa também pode ser definida a partir das instruções:

```
julia> Int(round(1.1))
1

julia> Int(floor(1.1))
1
```

Outro tema vinculado a conversão de dados e a promoção de um dado de um tipo para outro tipo de maior abrangência com uso da função "**promote**". Esta função avalia os tipos de dados passados em seus argumentos e seleciona para todos os dados o tipo que tenha maior abrangência. Observe os próximos exemplos de promoção:

```
julia> promote(1, 2.0)  # promoção do valor "1" inteiro em ponto flutuante
(1.0, 2.0)

julia> promote(1.0, 2)  # promoção do valor "2" inteiro em ponto flutuante
(1.0, 2.0)

julia> promote(1, 2)  # mantém tipos por serem os dados de mesma categoria
(1, 2)

julia> promote(1, im)
(1 + 0im, 0 + 1im)

julia> promote(1, 2//3)
(1//1, 2//3)

julia> promote(1, 2.0, 3//4, im)
1.0 + 0.0im, 2.0 + 0.0im, 0.75 + 0.0im, 0.0 + 1.0im)

julia> promote(1, 2.0, 3//4)
(1.0, 2.0, 0.75)

julia> promote(2.0, 3//4)
(2.0, 0.75)
```

5.5 - Estatística

A *estatística* é um ramo de estudos da Ciência Matemática que coleta, análise, interpreta, organiza e apresenta dados numéricos variáveis relacionados a qualquer fenômeno sem estabelecer conclusões sobre fatos (ANDRADE, 2009).

A linguagem Julia possui o pacote padrão "**Statistics**" que contém funcionalidades básicas para operações estatísticas, além de diversos pacotes externos. Para saber as operações deste pacote é necessário colocá-lo em uso. Assim sendo, execute a instrução seguinte:

```
julia> using Statistics
```

Veja a seguir as operações estáticas executadas pelo pacote "**Statistics**":

Desvio padrão: medida que indica o grau de dispersão de um conjunto de dados, mostra o quanto um conjunto é uniforme na medida em que o valor do desvio padrão se aproxima de zero. Veja um exemplo para o cálculo do desvio padrão das alturas de seis pessoas.

```
julia> std([1.85, 1.93, 1.98, 1.88, 2.04, 1.81])  # sem conhecer a média
0.0854985379991962

julia> stdm([1.85, 1.93, 1.98, 1.88, 2.04, 1.81], 1.915)  # conhecendo a média
0.0854985379991962
```

A função "**std**" efetua o cálculo do desvio padrão e calcula internamente a média da série avaliada. Já a função "**stdm**" para fazer o cálculo do desvio padrão necessita que o valor da média (segundo argumento) seja informado.

Variância: medida de dispersão que mostra o quão cada valor do conjunto está próximo ou distante do valor da média, ou seja, quanto menor o valor da variância mais próximos os valores estão da média; quanto maior o valor da variância mais distante os valores estão da média. Veja um exemplo para o cálculo da variância das alturas de seis pessoas.

```
julia> var([1.85, 1.93, 1.98, 1.88, 2.04, 1.81])  # sem conhecer a média
0.007309999999999997

julia> varm([1.85, 1.93, 1.98, 1.88, 2.04, 1.81], 1.915)  # conhecendo a média
0.007309999999999997
```

A função "**var**" efetua o cálculo da variância e calcula internamente a média da série avaliada. Já a função "**varm**" para fazer o cálculo da variância necessita que o valor da média (segundo argumento) seja informado.

Correlação de Pearson: teste que mede a relação estatística entre duas sequências de valores numéricos: vetores ou matrizes (indicando a intensidade e a direção de relações lineares) e estabelece um coeficiente entre "**-1**" (correlação de direção negativa) e "**+1**" (correlação de direção positiva). O sinal indica a direção da correlação e o valor obtido indica a força da correlação como mostra a tabela 5.2 (MUKAKA, 2012, p. 69-71).

Correlação	Interpretação
+0.9 até +1.0 (−0.9 até −1.0)	Correlação muito forte.
+0.7 até +0.9 (−0.7 até −0.9)	Correlação forte.
+0.5 até +0.7 (−0.5 até −0.7)	Correlação moderada.
+0.3 até +0.5 (−0.3 até −0.5)	Correlação fraca.
0.0 até +0.3 (0.0 até −0.3)	Correlação desprezível.

Tabela 5.1 - Códigos de formatação para data e hora.

Se a associação não for linear entre os elementos o coeficiente não será indicado adequadamente. Veja um exemplo para o cálculo de correlação de Pearson entre o tempo de estudo de seis estudantes e as notas alcançadas.

```
julia> horas = [12.5, 10.3, 9.0, 7.5, 5.1, 3.0]
6-element Vector{Float64}:
 12.5
 10.3
  9.0
  7.5
  5.1
  3.0
```

```
julia> notas = [9.5, 4.3, 2.6, 5.5, 6.7, 2.5]
6-element Vector{Float64}:
 9.5
 4.3
 2.6
 5.5
 6.7
 2.5

julia> cor(horas, notas)  # calcula a correlação de Pearson entre os vetores "horas" e "notas"
0.5109205990838005

julia> cor(horas)  # devolve valor "1"
1.0

julia> cor(notas)  # devolve valor "1"
1.0
```

Covariância: medida que determina o quanto dois grupos de valores numéricos se relacionam, podendo ser positivo ou negativo. Se o resultado for positivo isto indica que há forte relação entre os números e se negativo que há fraca relação entre os números. Veja um exemplo para o cálculo de covariância entre as alturas e pesos de seis pessoas.

```
julia> altura = [1.85, 1.93, 1.98, 1.88, 2.04, 1.81]
6-element Vector{Float64}:
 1.85
 1.93
 1.98
 1.88
 2.04
 1.81

julia> peso = [85, 100, 90, 75, 80, 65]
6-element Vector{Int64}:
  85
 100
  90
  75
  80
  65

julia> cov(altura, peso)
0.47499999999999976
```

Média: medida que o valor médio entre os valores numéricos de um conjunto de dados: vetores ou matrizes. Veja um exemplo para o cálculo da média das alturas de seis pessoas.

```
julia> mean(altura)
1.915

julia> mean([1.85, 1.93, 1.98, 1.88, 2.04, 1.81])
1.915
```

Observe outros exemplos de obtenção de média.

```
julia> mean(1:10)  # média entre os valores de "1" até "10"
5.5

julia> mean(1:2:10)  # média entre os valores de "1" até "10" de "2" em "2"
5.0

julia> mean(√, [1, 2, 3, 4])  # média das raízes quadradas do vetor
1.5365660924854931


julia> mean([√1, √2, √3, √4])  # média das raízes quadradas do vetor
1.5365660924854931

julia> mean(√, [1 2 3 4; 5 6 7 8])  # média das raízes quadradas da matriz
2.038250065754465

julia> mean([1 2 3 4; 5 6 7 8])  # média da matriz
4.5

julia> mean([1 2 3 4; 5 6 7 8]; dims=1)  # média entre os elementos de cada posição
1×4 Matrix{Float64}:
 3.0  4.0  5.0  6.0

julia> mean([1 2 3 4; 5 6 7 8]; dims=2)  # média da 1a. linha e média da 2a. linha
2×1 Matrix{Float64}:
 2.5
 6.5
```

Mediana: é a indicação do número central (centralidade) de um conjunto (vetores ou matrizes) de valores numéricos. Veja um exemplo para apresentar a mediana das alturas de seis pessoas.

```
julia> median([1.85, 1.93, 1.98, 1.88, 2.04, 1.81])
1.9049999999999998

julia> median([1 2 3 4; 5 6 7 8])
4.5
```

Observe outros exemplos de obtenção de mediana.

```
julia> median([1 2 3 4; 5 6 7 8]; dims=1)  # mediana entre os elementos de cada posição
1×4 Matrix{Float64}:
 3.0  4.0  5.0  6.0

julia> median([1 2 3 4; 5 6 7 8]; dims=2)  # mediana da 1a. linha e média da 2a. linha
2×1 Matrix{Float64}:
 2.5
 6.5
```

Ponto médio: é o valor numérico resultante dos valores de limite inferior (menor valor) e superior (maior valor) existentes em uma sequência numérica somados e divididos por "**2**". Veja um exemplo para apresentar o ponto médio das alturas de seis pessoas.

```
julia> middle(altura)
1.925
```

```
julia> middle([1.85, 1.93, 1.98, 1.88, 2.04, 1.81])  # (1.81 + 2.04) / 2
1.925
```

Além das funcionalidades apresentadas o pacote "**Statistics**" possui algumas outras operações. Para mais detalhes consulte o endereço: **https://docs.julialang.org/en/v1/stdlib/Statistics/**.

5.6 - Gráficos com arranjos

Para realizar a apresentação de gráficos a linguagem Julia possui grande diversidade de pacotes, destacando-se especialmente o pacote "**Plots**" que é um dos mais conhecidos e usados. Para fazer a instalação do pacote gráfico execute a seguinte instrução:

```
julia> using Plots
```

Se for a primeira vez de sua execução será informado a possibilidade de instalação. Neste sentido, confirme "**y**" quando solicitado e aguarde o processo de instalação do pacote ser concluído.

A partir da instalação é possível gerar gráficos que serão apresentados em uma janela adicional. Para realizar um primeiro teste execute as seguintes instruções que criam as séries de valores a serem usadas:

```
julia> x1 = 1:10  # abscissas
1:10

julia> y1 = [0.70, 0.17, 0.65, 0.33, 0.75, 0.91, 0.47, 0.56, 0.94, 0.15]  # ordenadas
10-element Vector{Float64}:
 0.7
 0.17
 0.65
 0.33
 0.75
 0.91
 0.47
 0.56
 0.94
 0.15

julia> Plots.plot(x1, y1)  # pode ser executado "plot(x, y)"
```

O processamento para a geração de certo gráfico pode ser inicialmente pouco demorado. Aguarde a conclusão das operações internas.

Quando concluído será apresentada uma tela com a imagem de um gráfico de linhas semelhante ao apresentado junto a imagem da figura 5.1.

Pode ser que dependendo do sistema operacional em uso seja solicitada autorização de acesso a geração do gráfico. Se isto ocorrer autorize a ação para que se tenha acesso a geração do gráfico.

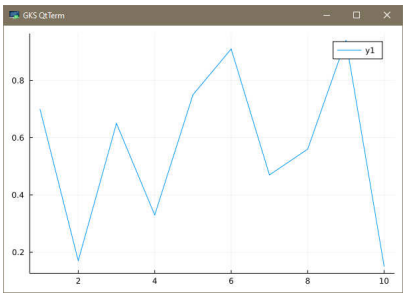

Figura 5.1 - Gráfico apresentado na linguagem Julia.

Além de apresentar um gráfico é possível estabelecer critérios de formatação a partir da definição de certos atributos de configuração para a imagem gerada, como: indicação de cores da linha e do fundo; tipos de apresentação da linha; definição de títulos para o gráfico e para as legendas; nível de rotação, entre outras possibilidades. Para ver um pouco deste efeito execute a seguinte instrução:

```
julia> Plots.plot(x1, y1,
                 title="Meu gráfico exemplo",
                 bg_inside=:orange,
                 linecolor=:blue,
                 line=:dashdot,
                 label="Meus valores",
                 xlabel="Abscissas",
                 ylabel="Ordenadas"
       )
```

Observe junto a figura 5.2 o efeito dos detalhes de atributos definidos e verifique junto a imagem cada uma das ocorrências.

Figura 5.2 - Gráfico com mudança de atributos.

Veja que alguns atributos são atribuídos a partir da definição de uma cadeia com textos que são plotados dentro do gráfico. Há outros que são atribuídos a partir do indicativo de símbolos como são a indicação dos nomes das cores e o tipo de linha apresentada.

O conjunto de atributos disponíveis para as definições dos elementos gráficos é extenso e para conhece-los sugere-se visitar os endereços:

- Atributos de séries - **https://docs.juliaplots.org/stable/generated/attributes_series/**:
- Atributos de plotagem - **https://docs.juliaplots.org/stable/generated/attributes_plot/**:
- Atributos de subplotagem - **https://docs.juliaplots.org/stable/generated/attributes_subplot/**:
- Atributos de eixos - **https://docs.juliaplots.org/stable/generated/attributes_axis/**.

Os exemplos seguintes de criação de gráficos usam algum atributo customizado para dar ideia geral dessa funcionalidade.

Um detalhe importante na definição de gráficos é que a imagem do gráfico pode ser montada aos poucos. Desta forma, o gráfico anterior pode conter mais uma linha traçada para indicar mais valores para as ordenadas. Assim sendo, considere a execução das próximas instruções:

```
julia> z1 = [1.25, 1.17, 0.88, 0.98, 1.75, 1.31, 0.99, 1.82, 0.94, 0.99]  # ordenadas
10-element Vector{Float64}:
 1.25
 1.17
 0.88
 0.98
 1.75
 1.31
 0.99
 1.82
 0.94
 0.99
```

```
julia> Plots.plot!(z1, label="Valores extras")
```

Observe que para conseguir o efeito de acréscimo de uma nova série de ordenadas é necessário fazer uso da função "**plot!**". Lembre-se de que toda função que tenha o caractere "**!**" é uma função que efetua alteração sobre o objeto indicado. Neste caso, o gráfico já está memorizado e a função insere a nova ordenada junto ao gráfico existente. A figura 5.3 mostra o gráfico com a nova informação inserida.

Veja que a nova inserção não afetou os estilos dos atributos definidos anteriormente. Perceba que o atributo "**label**" com o novo título é aplicado tão somente a segunda ordenada de valores, não afetando, em absoluto, os demais estilos anteriormente definidos.

O uso da função "**plot**" sem o símbolo "**!**" faz com que um novo gráfico seja gerado na memória. Então tenha tenção e cuidado entre os usos das funções "**plot**" e "**plot!**". Há outra maneira de se definir gráficos com múltiplos valores. Para ver este efeito siga as instruções seguintes e observe a imagem da figura 5.4 que mostra as linhas sobrepostas:

```
julia> x2 = 1:10
1:10

julia> y2 = [0.70 0.17; 0.65 0.33; 0.75 0.91; 0.47 0.56; 0.94 0.15;
             1.25 1.17; 0.88 0.98; 1.75 1.31; 0.99 1.82; 0.94 0.99]  # matriz 2x10
10×2 Matrix{Float64}:
 0.7   0.17
 0.65  0.33
 0.75  0.91
 0.47  0.56
 0.94  0.15
 1.25  1.17
 0.88  0.98
 1.75  1.31
 0.99  1.82
 0.94  0.99

julia> Plots.plot(x2, y2)
```

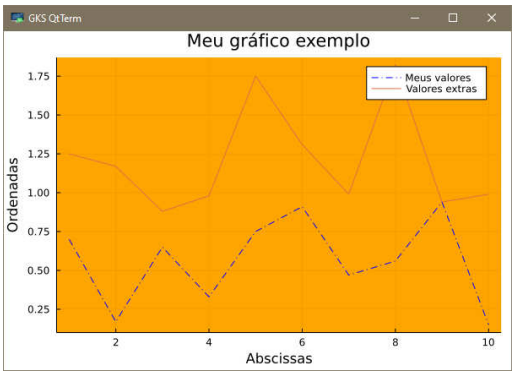

Figura 5.3 - Gráfico com nova ordenada.

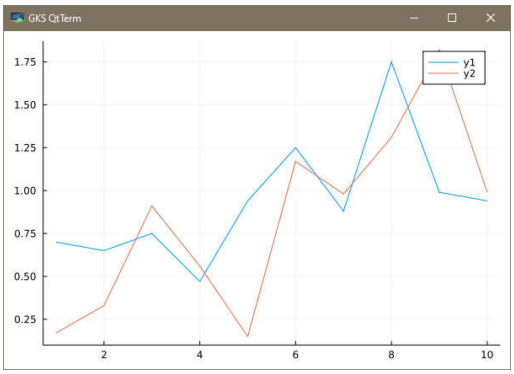

Figura 5.4 - Gráfico com dois valores nas ordenadas.

Agora veja o gráfico da figura 5.4 apresentado com algumas definições de atributos, como mostra a figura 5.5. Para o efeito execute a instrução:

```
julia> Plots.plot(x2, y2,
                  title="Meu segundo gráfico exemplo",
                  line=:dashdotdot,
                  label=["Série da linha 1" "Série da linha 2"]
       )
```

Note que para a figura 5.5 a definição do conteúdo das legendas das linhas é definida a partir do atributo "**label**" atribuído a um vetor contento os respectivos textos a serem considerados.

O visual das linhas em um gráfico pode ser apresentado com sua espessura mais acentuada. Para tanto, basta fazer uso do atributo "**linewidth**" ou "**lw**" que permite aumentar a largura da linha. Para tanto, execute a seguinte instrução e acompanhe a imagem na figura 5.6.

```
julia> Plots.plot(x2, y2, linewidth=4)
```

Figura 5.5 - Gráfico com atributos dos valores nas ordenadas.

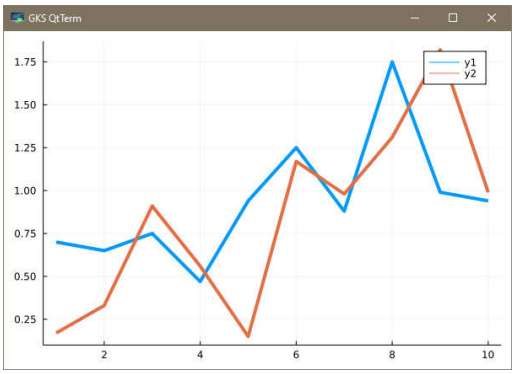

Figura 5.6 - Gráfico com linhas grafadas de forma forte.

O gráfico padrão gerado é o do tipo linhas como apresentado anteriormente, mas é possível fazer uso de outros padrões, como: barra, histograma, circular (pizza), superfície, ponto, gravetos entre outras possibilidades. Veja os resultados de algumas instruções de geração de gráficos entre as figuras 5.7 e 5.12.

```
julia> x3 = 1:10
1:10

julia> y3 = sin.(x3)
10-element Vector{Float64}:
  0.8414709848078965
  0.9092974268256817
  0.1411200080598672
 -0.7568024953079282
 -0.9589242746631385
 -0.27941549819892586
  0.6569865987187891
  0.9893582466233818
  0.4121184852417566
 -0.5440211108893698

julia> Plots.plot(x3, y3, seriestype=:bar, legend=false)  # figura 5.7: barras

julia> Plots.plot(x3, y3, seriestype=:scatter, legend=false)  # figura 5.8: pontos

julia> Plots.plot(x3, y3, seriestype=:sticks, legend=false)  # figura 5.9: gravetos

julia> Plots.plot(x3, y3, seriestype=:hline, legend=false)  # figura 5.10: linhas horizontais

julia> Plots.plot(x3, y3, seriestype=:vline, legend=false)  # figura 5.11: linhas verticais

julia> Plots.plot(x3, y3, seriestype=:shape, legend=false)  # figura 5.12: forma
```

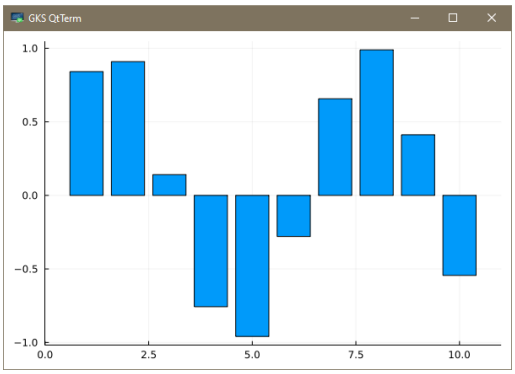

Figura 5.7 - Gráfico de barras.

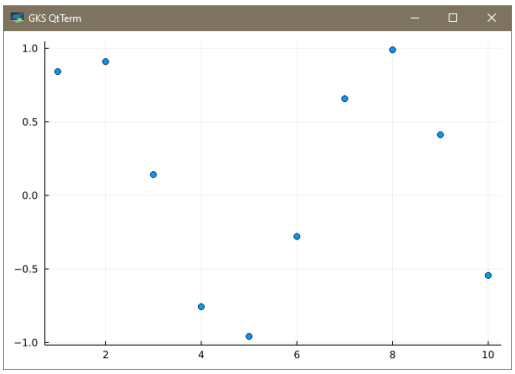

Figura 5.8 - Gráfico de pontos.

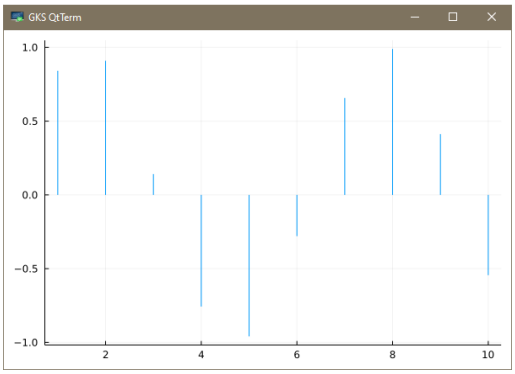

Figura 5.9 - Gráfico de gravetos.

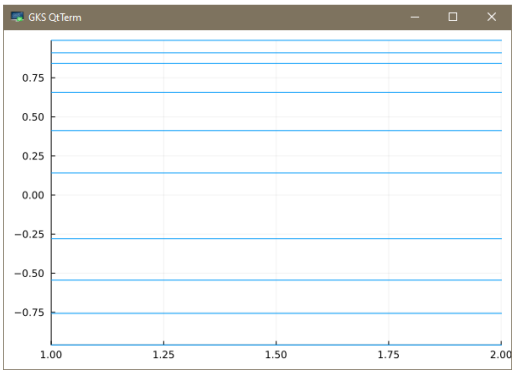

Figura 5.10 - Gráfico de linhas horizontais.

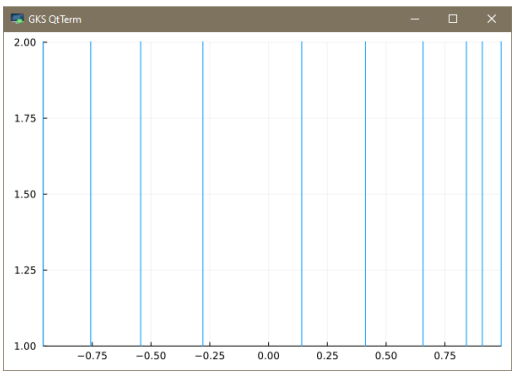

Figura 5.11 - Gráfico de linhas verticais.

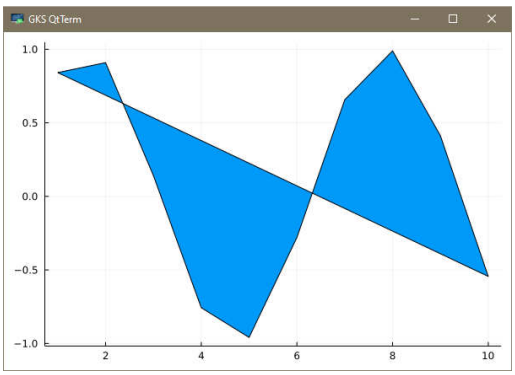

Figura 5.12 - Gráfico de forma.

Para a criação dos gráficos indicados entre as figuras 5.7 e 5.12 foi utilizado o atributo "**`legend`**" que com a definição do valor "**`false`**" omiti a apresentação da legenda da área plotada do gráfico. Já o atributo "**`seriestype`**" é usado para especificar o tipo de gráfico a ser apresentado a partir das séries de valores.

Os gráficos podem ser definidos com a combinação de tipos de série conjugados. Veja a próxima instrução que possibilita a definição de um gráfico com série de tipos combinadas de pontos com linhas indicado na figura 5.13:

```
julia> Plots.plot(x3, y3,
                  seriestype=[:scatter :line],
                  linecolor=[:black :red],
                  markercolor=[:black :red],
                  legend=false
       )
```

Para o efeito de combinação de séries é definido para o atributo "**seriestype**" um arranjo do tipo matriz contendo o tipo de série desejado. O mesmo se faz para os atributos "**linecolor**" e "**markercolor**" equacionando-se as cores a serem usadas em consonância com as séries definidas. Atente para a ordem desses elementos.

O atributo "**linecolor**" define a cor para as linhas e para os contornos dos pontos, já o atributo "**markercolor**" define as cores internas dos pontos no tipo de série "**:scatter**". Para ver as opções de cores suportadas acesse o endereço: **https://github.com/JuliaGraphics/Colors.jl/blob/master/src/names_data.jl**.

As cores usadas podem ser definidas a partir da indicação de seus nomes ou de seus códigos numéricos RGB (Red-Green-Blue). Experimente executar a instrução com a definição de cor em modo numérico no atributo "**linecolor**" e veja a apresentação do gráfico de forma idêntica a figura 5.13.

```
julia> Plots.plot(x3, y3,
                  seriestype=[:scatter :line],
                  linecolor=[RGB(0/255, 0/255, 0/255) RGB(255/255, 0/255, 0/255)],
                  markercolor=[:black :red],
                  legend=false
       )
```

A função "**RGB**" é um recurso que permite especificar um espectro de cor no padrão *vermelho-verde-azul*. Os valores das cores individuais variam de "**0**" (preto) até "**1**" (saturado), daí a necessidade de fazer a definição de cada valor de cor dividido pelo valor "**255**".

Os pontos apresentados em um gráfico do tipo "**scatter**" caracterizam-se por serem pequenos círculos, mas é possível criar pontos com outras imagens. Observe a instrução seguinte aplicando o uso do atributo "**marker**" e a imagem apresentada na figura 5.14:

```
julia> Plots.plot(x3, y3,
                  seriestype=[:scatter :line],
                  linecolor=[:red :red],
                  marker=:diamond,
                  legend=false
       )
```

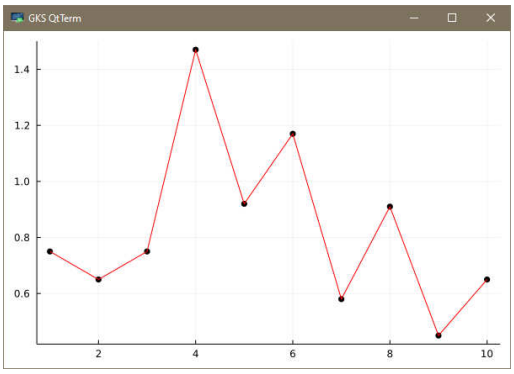

Figura 5.13 - Gráfico com tipos de série combinadas.

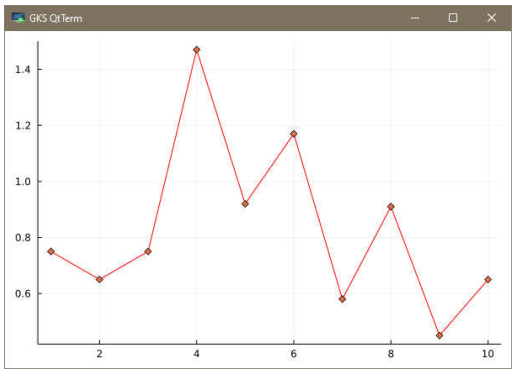

Figura 5.14 - Gráfico com pontos definidos como diamante.

Para o atributo "**marker**" são permitidas as opções: "auto", "circle", "cross", "diamond", "dtriangle", "hline", "heptagon", "hexagon", "ltriangle", "none", "octagon", "pentagon", "rect", "rtriangle", "star4", "star5", "star6", "star7", "star8", "utriangle", "vline", "xcross", "+" e "x".

Um efeito curioso do pacote "**Plots**" é permitir a apresentação de múltiplos gráficos em uma só imagem como mostra a figura 5.15. Para este efeito execute as seguintes instruções:

```
julia> x4 = 1:0.1:10
1.0:0.1:10.0
```

```
julia> y4_1 = sin.(x4)
91-element Vector{Float64}:
  0.8414709848078965
  0.8912073600614354
  ⋮
 -0.4575358937753214
 -0.5440211108893698

julia> y4_2 = cos.(x4)
91-element Vector{Float64}:
  0.5403023058681398
  0.4535961214255773
  ⋮
 -0.8891911526253609
 -0.8390715290764524

julia> y4_3 = -sin.(x4)
91-element Vector{Float64}:
 -0.8414709848078965
 -0.8912073600614354
  ⋮
  0.4575358937753214
  0.5440211108893698

julia> Plots.plot(x4, [y4_1 y4_2 y4_3], layout=(3, 1), legend=false)
```

Para gerar mais de um gráfico na mesma imagem usa-se o atributo "**layout**" que aceita a definição de uma tupla contendo no primeiro argumento a quantidade de gráficos a serem gerados e no segundo argumento a definição do número de colunas a ser usado para a apresentação dos gráficos. Experimente a seguinte instrução e veja a ocorrência indicada na figura 5.16.

```
julia> Plots.plot(x4, [y4_1 y4_2 y4_3], layout=(3, 2), legend=false)
```

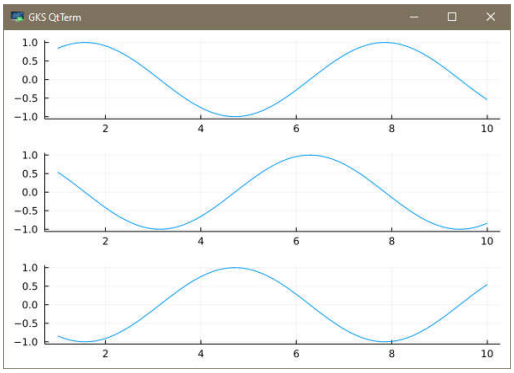

Figura 5.15 - Múltiplos gráficos.

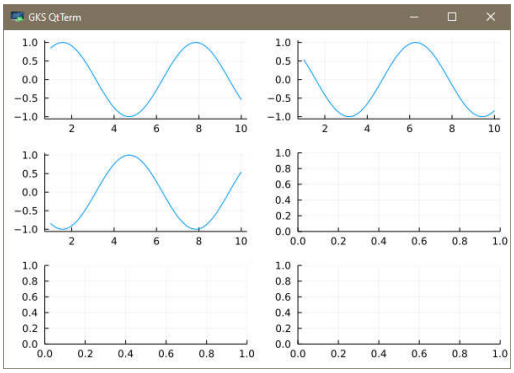

Figura 5.16 - Múltiplos gráficos em duas colunas.

Os gráficos gerados com o pacote "**Plots**" podem ser gravados como arquivos de imagem. Veja as instruções para conseguir esta operação:

```
julia> minha_imagem = Plots.plot(w1, [w2 w3 w4], layout=(3, 2), legend=false)

julia> savefig(minha_imagem, "MeuArquivo.png")
```

Um gráfico muito usado é o de tipo setores (conhecido como gráfico circular ou gráfico de pizza). Este gráfico para ser gerado com o pacote "**Plots**" deve seguir as seguintes instruções, observe depois a imagem da figura 5.17:

```
julia> x5 = ["Arroz", "Feijão", "Farinha", "Macarrão", "Milho"]
5-element Vector{String}:
 "Arroz"
 "Feijão"
 "Farinha"
 "Macarrão"
 "Milho"

julia> y5 = [100, 120, 110, 150, 200]
5-element Vector{Int64}:
 100
 120
 110
 150
 200

julia> Plots.plot(x5, y5,
                 seriestype=:pie,
                 title="Projeção de vendas"
      )
```

Figura 5.17 - Gráfico de setores.

A título de simplificação os gráficos podem ser gerados com funções específicas. Por exemplo, para gerar um gráfico de setores, como o da figura 5.17, pode-se fazer uso da função "**pie**" no lugar da função "**plot**". Observe a próxima instrução e note a imagem apresentada.

```
julia> Plots.pie(x5, y5, title="Projeção de vendas")
```

A vantagem de uso da função com o nome do tipo do gráfico em relação a função "**plot**" é a simplificação em não precisar fazer uso do atributo "**seriestype**". Desta forma, podem ser usadas funções com os nomes "**bar**", "**line**", "**scatter**" ou outro nome que corresponda ao gráfico a ser gerado.

Para os gráficos apresentados, de duas dimensões, basta apenas indicar a definição das coordenadas dos eixos das abcissas e ordenadas, não importando a quantidade de valores para o eixo das ordenadas. Mas há gráficos que para serem gerados necessitam de um conjunto maior de valores para definir as coordenadas: abcissas (eixo "x"), ordenadas (eixo "y") e cotas (eixo "z"), como gráficos tridimensionais. Veja as próximas instruções e observe a figura 5.18:

```
julia> x6 = 1:10  # eixo "x": abcissa
1:10

julia> y6 = 1:10  # eixo "y": ordenada
1:10
```

```
julia> z6 = x .* y  # eixo "z": cota | ".*": operação vetorial
10-element Vector{Float64}:
 0.7
 0.34
 1.9500000000000002
 1.32
 3.75
 5.46
 3.29
 4.48
 8.459999999999999
 1.5

julia> Plots.plot(x6, y6, z6,
                 seriestype=:line,
                 background_color_outside=:cyan,  # define a cor de todo o fundo
                 linecolor=:black,
                 markercolor=:black,
                 linewidth=4,
                 legend=false,
                 xlabel="Eixo X",
                 ylabel="Eixo Y",
                 zlabel="Eixo Z"
       )
```

O ponto de vista de um gráfico tridimensional é chamado em Julia de *câmera*. O ponto de posicionamento da câmera pode ser alterado pelo atributo "**camera**" ou "**cam**" que opera a partir de dois argumentos de posição, sendo o primeiro o ângulo do azimute (eixo z) e o segundo o ângulo de elevação (eixo y). Veja a próxima instrução e a imagem da figura 5.19:

```
julia> Plots.plot(x6, y6, z6,
                 seriestype=:line,
                 background_color_outside=:cyan,
                 linecolor=:black,
                 markercolor=:black,
                 linewidth=4,
                 legend=false,
                 camera=(10, 70),  # define o ângulo de visualização do gráfico
                 xlabel="Eixo X",
                 ylabel="Eixo Y",
                 zlabel="Eixo Z"
       )
```

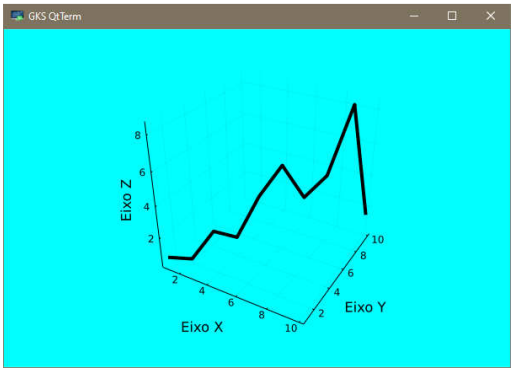

Figura 5.18 - Gráfico tridimensional.

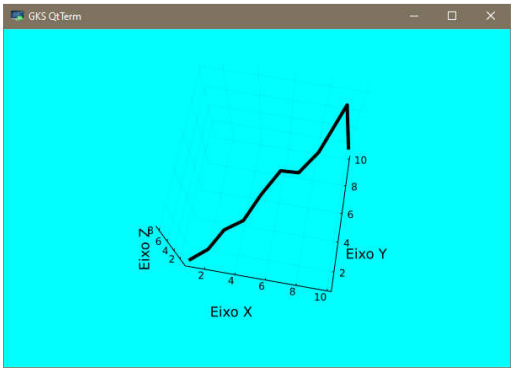

Figura 5.19 - Gráfico com mudança do ponto de vista.

5.7 - Gráficos com funções

Além da criação de gráficos baseados em arranjos de vetores e matrizes Julia possibilita a geração de gráficos a partir do uso de funções matemáticas, tanto nomeadas como anônimas definidas junto a função "**plot**". É possível operar com gráficos gerados a partir de funções polinomiais. Observe as instruções de geração de gráficos de funções polinomiais de primeiro até o sexto grau indicados entre as figuras 5.20 a 5.25, definidos a partir de funções anônimas.

```
julia> Plots.plot(x -> 2x + 1, legend=false)  # primeiro grau

julia> Plots.plot(x -> 2x^2 + 2x + 1, legend=false)  # segundo grau

julia> Plots.plot(x -> 2x^3 + 2x^2 + 2x + 1, legend=false)  # terceiro grau

julia> Plots.plot(x -> 2x^4 + 2x^3 + 2x^2 + 2x + 1, legend=false)  # quarto grau

julia> Plots.plot(x -> 2x^5 + 2x^4 + 2x^3 + 2x^2 + 2x + 1, legend=false)  # quinto grau

julia> Plots.plot(x -> 2x^6 + 2x^5 + 2x^4 + 2x^3 + 2x^2 + 2x + 1, legend=false)  # sexto grau
```

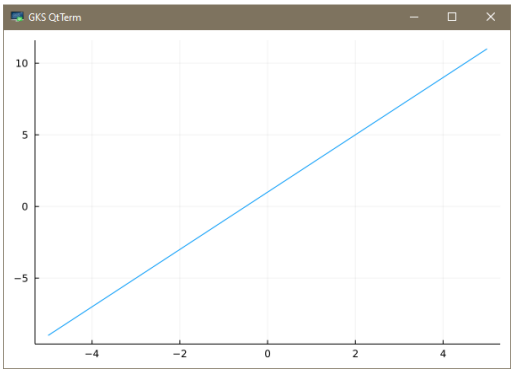

Figura 5.20 - Gráfico de função de primeiro grau.

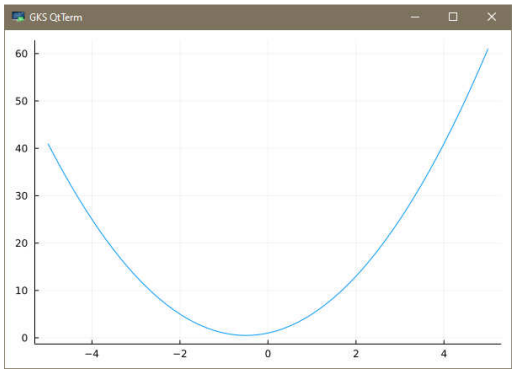

Figura 5.21 - Gráfico de função de segundo grau.

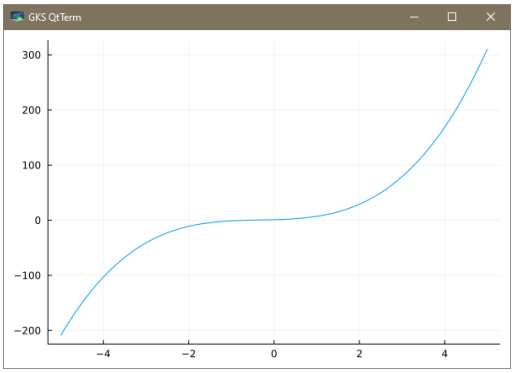

Figura 5.22 - Gráfico de função de terceiro grau.

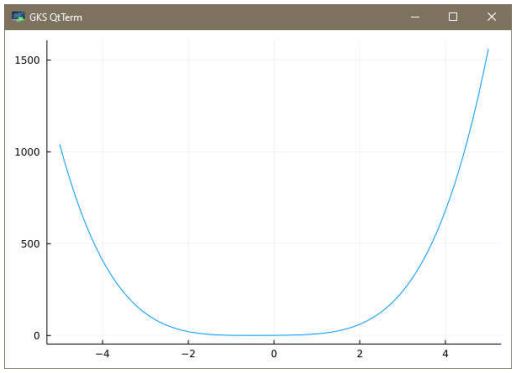

Figura 5.23 - Gráfico de função de quarto grau.

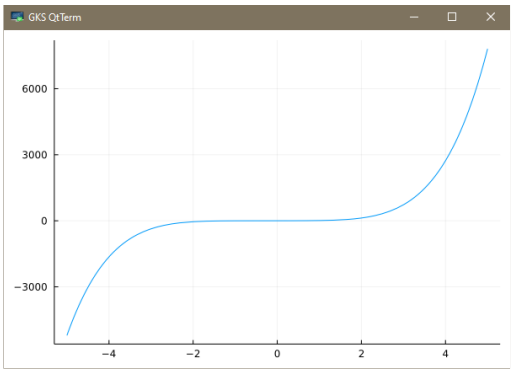

Figura 5.24 - Gráfico de função de quinto grau.

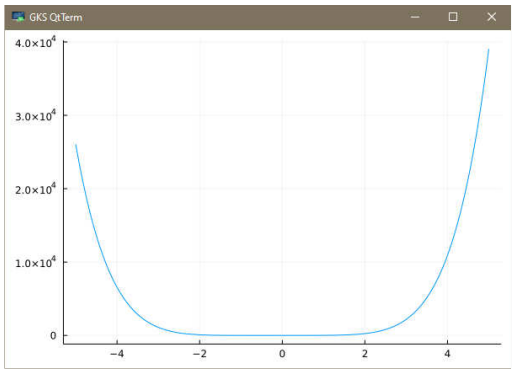

Figura 5.25 - Gráfico de função de sexto grau.

Gráficos gerados a partir de funções podem fazer uso em sua composição de funções matemáticas diversas. Veja alguns exemplos com aplicação de funções anônimas com uso de razões trigonométricas com resultados indicados entre as figuras 5.26 a 5.31.

```
julia> Plots.plot(x -> cos(x), legend=false)

julia> Plots.plot(x -> sin(x), legend=false)

julia> Plots.plot(x -> tan(x), legend=false)

julia> Plots.plot(x -> acos(x), legend=false)

julia> Plots.plot(x -> asin(x), legend=false)

julia> Plots.plot(x -> atan(x), legend=false)
```

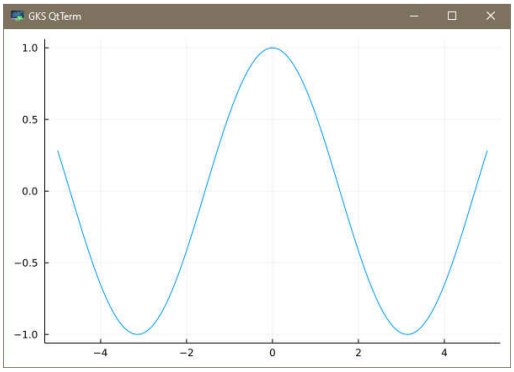

Figura 5.26 - Gráfico do cosseno.

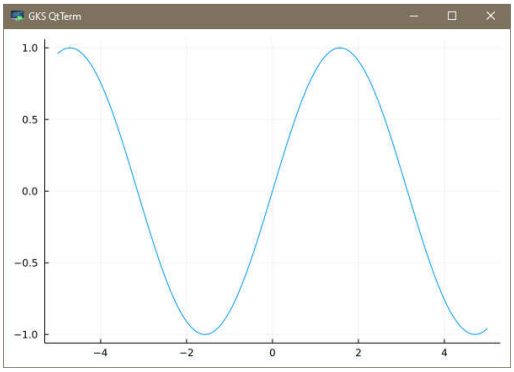

Figura 5.27 - Gráfico do seno.

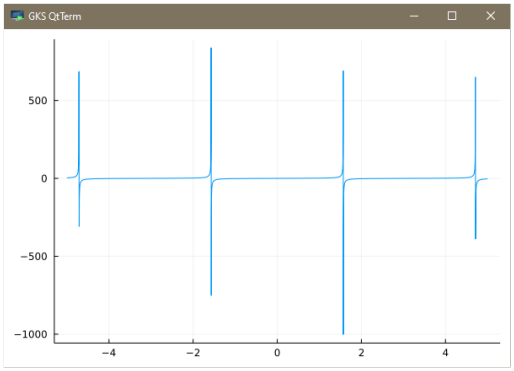

Figura 5.28 - Gráfico da tangente.

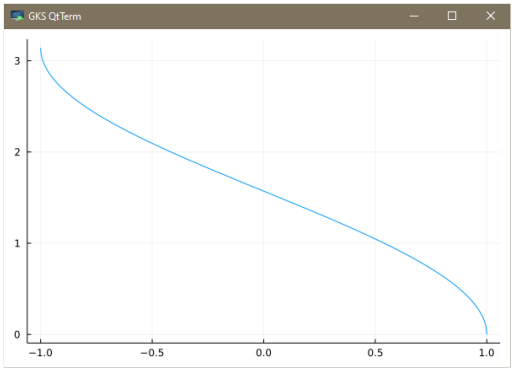

Figura 5.29 - Gráfico do arco cosseno.

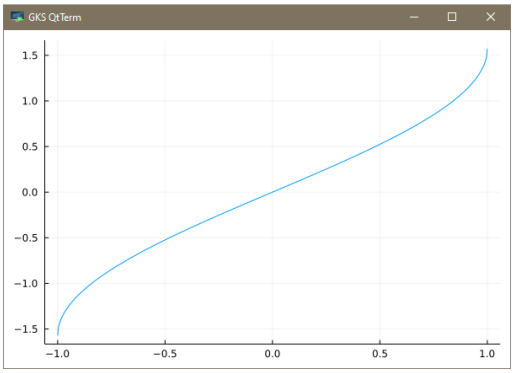

Figura 5.28 - Gráfico do arco seno.

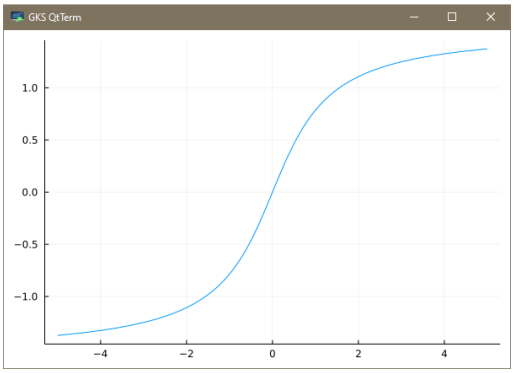

Figura 5.29 - Gráfico do arco tangente.

Um detalhe a ser considerado no uso de funções é a diferença de escrita quando são aplicadas sobre arranjos, onde é definido o uso de um "`.`" indicando que a função deve ser aplicada ponto a ponto sobre cada um dos valores do arranjo.

Todos os gráficos apresentados neste tópico foram gerados a partir dos padrões internos do pacote "**Plots**". No entanto, esses parâmetros automatizados podem ser alterados a partir de ajustes aplicados sobre os atributos de formatação e configuração, como visto no tópico anterior.

Veja como exemplo a apresentação do gráfico da função de primeiro grau "**f(x) = -3x**" junto a figura 5.30 criado a partir de seus padrões automatizados, exceto os atributos de espessura, cor de linha e omissão de legenda manualmente especificados:

```
julia> Plots.plot(x -> -3x, legend=false, linewidth=3, linecolor=:magenta)
```

Observe a definição dos valores de escala junto as linhas dos eixos. Note que esses valores foram gerados automaticamente. No entanto, essa escala pode ser alterada a partir da manipulação dos atributos "**xticks**" e "**yticks**". Veja a seguinte instrução e seu resultado junto a figura 5.31:

```
julia> Plots.plot(x -> -3x,
                  legend=false,
                  linewidth=3,
                  linecolor=:magenta,
                  xticks=-5:5,
                  yticks=-16:2:16
       )
```

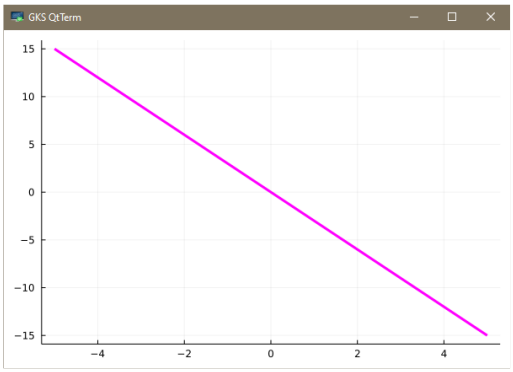

Figura 5.30 - Gráfico de função de primeiro grau.

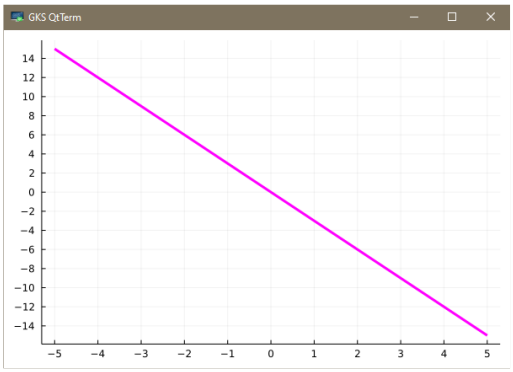

Figura 5.31 - Gráfico com atributos de eixos alterados.

Compare os valores apresentados nos eixos "**x**" e "**y**" das figuras 5.30 e 5.32. Observe o efeito de alteração dos atributos dos eixos "**xticks**" e "**yticks**".

A imagem do gráfico é plotada em um plano cartesiano marcado com uma grade. A imagem da grade pode ser removida do gráfico bastando fazer uso do atributo "**grid**". A figura 5.33 mostra o resultado da omissão da grade a partir da instrução:

```
julia> Plots.plot(x -> -3x,
                  legend=false,
                  linewidth=3,
                  linecolor=:magenta,
                  xticks=-5:5,
                  yticks=-16:2:16,
                  grid=false
       )
```

Já é conhecida a função "**plot!**", a qual efetua alterações no gráfico previamente gerado. Neste sentido, fica simples fazer a alteração de gráficos em memória, sem a necessidade de construir uma linha de código com a função "**plot**" que seja extensa. Assim sendo, considere a forma mais simples de apresentação de um gráfico a partir da execução da instrução que gera imagem idêntica a figura 5.34:

```
julia> Plots.plot(x -> -3x)
```

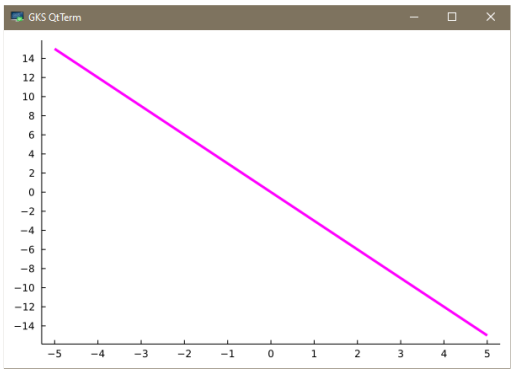

Figura 5.33 - Gráfico com atributos de grade omitido.

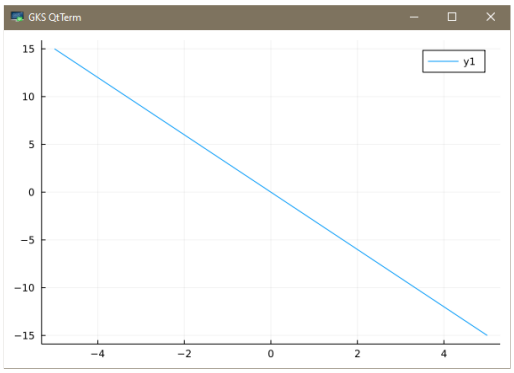

Figura 5.34 - Gráfico com atributos padrão.

A partir da apresentação básica de um gráfico, como mostra a figura 5.34 fica simples implementar alterações com a definição de linhas de códigos mais simples. Isso é útil, inclusive no trabalho em modo programado. Acompanhe as instruções de mudança de alguns atributos relacionados a legendas e eixos que podem ser definidos separadamente, mas atenção, nem todo atributo de configuração pode ser declarado de forma separada, alguns obrigatoriamente são declarados dentro da função "**`plot`**".

Atributos de efeitos sobre legendas

```
julia> plot!(legend=:bottomleft)  # canto inferior esquerdo

julia> plot!(legend=:bottomright)  # canto inferior direito

julia> plot!(legend=:topleft)  # canto superior esquerdo

julia> plot!(legend=:topright)  # canto superior direito (padrão)

julia> plot!(legend=:inside)  # centro do gráfico

julia> plot!(legend=:right)  # a direita centrada verticalmente

julia> plot!(legend=:left)  # a esquerda centrada verticalmente

julia> plot!(legend=:top)  # superior centrado horizontalmente

julia> plot!(legend=:bottom)  # inferior centrado horizontalmente

julia> plot!(legend=:outerbottomright)  # inferior à direita, fora do gráfico

julia> plot!(legend=:outerbottomleft)  # inferior à esquerda, fora do gráfico

julia> plot!(legend=:outertopright)  # superior à direita, fora do gráfico

julia> plot!(legend=:outertopleft)  # superior à esquerda, fora do gráfico
```

Atributos de efeitos sobre eixos

```
julia> plot!(xticks=vlr_infeiror:incremento:vlr_superior)  # escala do eixo "x"

julia> plot!(yticks=vlr_infeiror:incremento:vlr_superior)  # escala do eixo "y"

julia> plot!(zticks=vlr_infeiror:incremento:vlr_superior)  # escala do eixo "z"

julia> plot!(xlims=(vlr_infeiror, vlr_superior))  # limites do eixo "x"
```

```
julia> plot!(ylims=(vlr_infeiror, vlr_superior))  # limites do eixo "y"

julia> plot!(zlims=(vlr_infeiror, vlr_superior))  # limites do eixo "z"

julia> plot!(xlabel="título x")  # título do eixo "x"

julia> plot!(ylabel="título y")  # título do eixo "y"

julia> plot!(zlabel="título z")  # título do eixo "z"

julia> plot!(xtickfont=font(tamanho, "Fonte"))  # configura fonte do eixo "x"

julia> plot!(ytickfont=font(tamanho, "Fonte"))  # configura fonte do eixo "y"

julia> plot!(ztickfont=font(tamanho, "Fonte"))  # configura fonte do eixo "z"

julia> plot!(axis=true)  # habilita apresentação dos eixos

julia> plot!(axis=false)  # desabilita apresentação dos eixos

julia> plot!(xaxis=true)  # habilita apresentação do eixo "x"

julia> plot!(xaxis=false)  # desabilita apresentação do eixo "y"

julia> plot!(yaxis=true)  # habilita apresentação do eixo "y"

julia> plot!(yaxis=false)  # desabilita apresentação do eixo "y"

julia> plot!(zaxis=true)  # habilita apresentação do eixo "z"

julia> plot!(zaxis=false)  # desabilita apresentação do eixo "z"
```

Outros atributos

```
julia> plot!(size=(largura, altura))  # dimensiona tamanho do gráfico em pontos

julia> plot!(legendfont=font(tamanho, "Fonte"))  # altera fonte e tamanho da legenda
```

Agora veja as próximas instruções e seus efeitos no gráfico indicado na figura 5.35.

```
julia> Plots.plot(x -> 3x, label="Função: f(x) = 3x")

julia> Plots.plot!(legend=:outerbottomleft)

julia> Plots.plot!(size=(900, 400))

julia> Plots.plot!(xlims=(0, 10))

julia> Plots.plot!(ylims=(-5, 10))

julia> Plots.plot!(xticks=0:2:10)

julia> Plots.plot!(yticks=-5:4:10)

julia> Plots.plot!(linecolor=:magenta)

julia> Plots.plot!(xtickfont=font(10, "Corier"))

julia> Plots.plot!(ytickfont=font(10, "Corier"))

julia> Plots.plot!(legendfont=font(10, "Corier"))
```

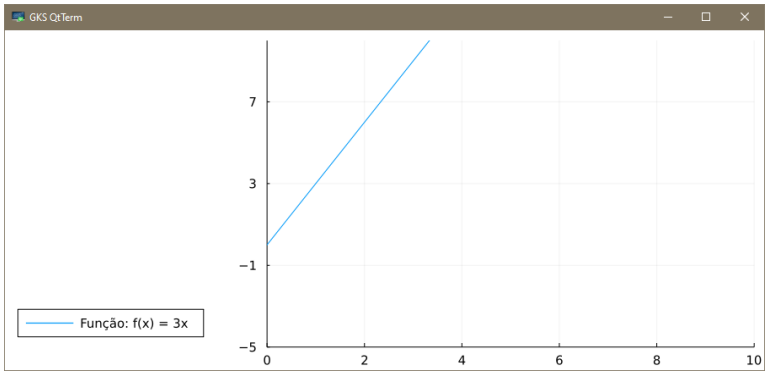

Figura 5.35 - Gráfico criado por partes 1.

Como comentado, há atributos que não podem ser usados isoladamente, destacando-se entre os atributos: "**legend**", "**linewidth**" e "**linecolor**". Para usá-los sempre os defina primeiro dentro da função "**plot**" e depois use a função "**plot!**", para configurar os demais atributos. Assim sendo, veja o próximo exemplo e seu efeito junto a figura 5.36.

```
julia> Plots.plot(x -> tan(x), label="tan(x)", linewidth=3, linecolor=:red)

julia> Plots.plot!(xlims=(-10, 10))

julia> Plots.plot!(xticks=-10:5:10)

julia> Plots.plot!(ylims=(-30, 30))

julia> Plots.plot!(yticks=-30:10:30)
```

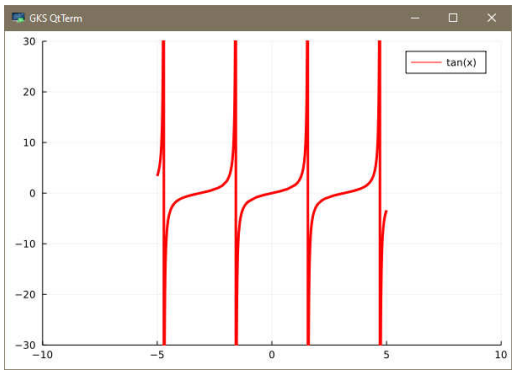

Figura 5.36 - Gráfico criado por partes 2.

Visto o uso de função anônima considere o uso de funções nomeadas ao estilo "**f(x)**" a partir de uma tonalidade mais matemático. Desta forma, observe a função a seguir:

```
julia> f(x) = cos(-x) * exp(x)
f (generic function with 1 method)

julia> f(-5)
0.0019113007712959706

julia> f(0)
1.0

julia> f(5)
42.09920106253839
```

Após a definição da função "**f(x)**", será feita a definição do domínio da função no eixo "**x**" variando a faixa de "**-5**" a "**5**", o qual criará na memória um arranjo do tipo vetor com onze elementos. Veja a seguir instrução para esta definição:

```
julia> x = range(-5, 5)
-5,5
```

Para a criação do eixo "**y**" será aplicada a função "**f(x)**" sobre cada ponto do vetor "**x**" a fim de criar um arranjo do tipo vetor contendo os valores calculados. Desta forma, execute a instrução:

```
julia> y = f.(x)
11-element Vector{Float64}:
   0.0019113007712959706
  -0.01197190052166259
  -0.04928882411191869
  -0.05631934999212789
   0.19876611034641298
   1.0
   1.4686939399158851
  -3.074932320639359
 -19.884530844146987
 -35.68773248011913
  42.09920106253839
```

A partir da criação dos vetores de valores "**x**" e "**y**" execute a próxima instrução para a apresentação do gráfico indicado na figura 5.37

```
julia> Plots.plot(x, y, color=:blue, legend=false, linewidth=3)
```

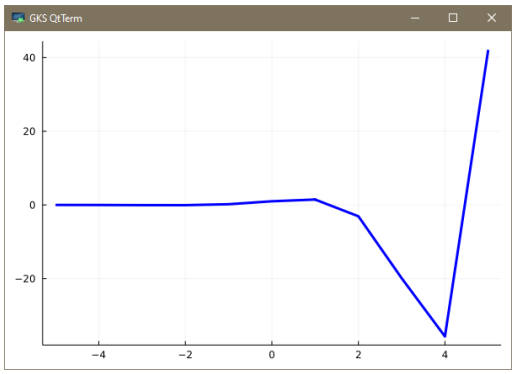

Figura 5.36 - Gráfico da função f(x) = exp(x) * sin(x) + cos(x) * exp(x).

Ao olhar o gráfico da figura 5.36 nota-se que há algo de errado, pois é esperada a apresentação de uma curva (parábola). Na verdade, não é um erro, mas sim a quantidade pequena de pontos a serem plotados. Para resolver esse efeito é necessário na definição do domínio do eixo "**x**" especificar o tamanho de pontos desejados para a curva, sendo um bom tamanho um valor na casa de "**100**", além de redefinir os valores do eixo "**y**". Assim sendo, execute a seguinte instrução:

```
julia> x = range(-5, 5, length=100)
-5.0:0.10101010101010101:5.0

julia> y = f.(x)
100-element Vector{Float64}:
   0.0019113007712959706
    ⋮
  42.09920106253839
```

A partir das novas definições dos valores para os eixos "**x**" e "**y**" execute a instrução de criação de gráfico seguinte e veja seu resultado na figura 5.37

```
julia> Plots.plot(x, y, color=:blue, legend=false, linewidth=3)
```

Com o objetivo de deixar o eixo "**x**" redefinido com uma faixa de valores diferenciada experimente usar outras funcionalidades separadas da função "**plot!**", que efetuam alterações diretas. Observe as instruções seguintes e seu efeito junto a imagem da figura 5.38.

```
julia> xlims!(-6, 6)

julia> yticks!([-30, 0, 40])
```

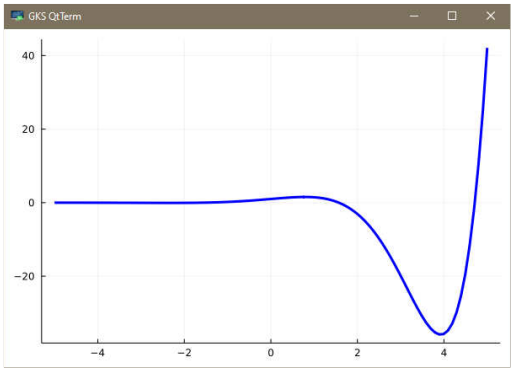

Figura 5.37 - Gráfico com ajuste dos pontos da curva.

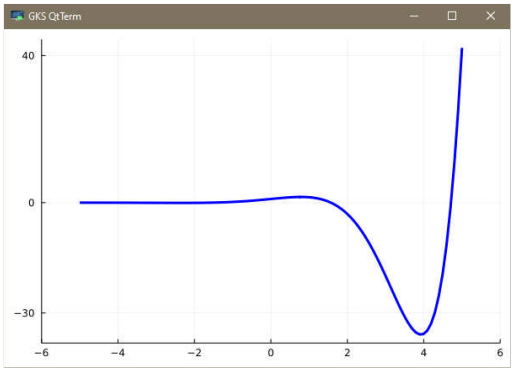

Figura 5.38 - Gráfico com alterações diretas.

O uso da função "**xlims!**", definiu a faixa de abrangência do eixo "**x**", já a função "**yticks!**", com a indicação de seu argumento como um vetor permitiu definir marcações com rótulos que se deseja apresentar. Observe que são possíveis proceder acréscimos de detalhes em certo gráfico a partir de funções que acessam e alteram seus atributos, sem o uso da função "**plot!**".

Imagine que se queira traçar no sentido horizontal três linhas coloridas marcando os pontos: central em vermelho, alto em verde e baixo em magenta no gráfico. Para realizar esta ação execute as instruções:

```
julia> Plots.plot!([-6, 6], [0, 0], color=:red, linewidth=2, label="")

julia> Plots.plot!([-6, 6], [40, 40], color=:green, linewidth=2, label="")

julia> Plots.plot!([-6, 6], [-36, -36], color=:magenta, linewidth=2, label="")
```

Veja na figura 5.39 a indicação das linhas propostas. Observe que as instruções anteriores traçam linhas no gráfico.

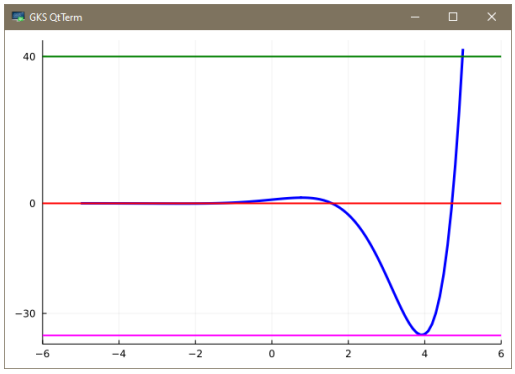

Figura 5.39 - Gráfico com linhas adicionais.

Note que para traçar as linhas usa-se na função "**plots!**", em seu primeiro argumento a definição da coordena do eixo "**x**" e no segundo argumento a indicação da coordenada do eixo "**y**". Os demais argumentos são usados para estabele-

cer o efeito visual, sendo: "**color**" para a definição da cor da linha, "**linewidth**" para a definição da espessura da linha e "**label**" para a definição de um rótulo de identificação, que neste caso está sendo omitido deixando a linha sem nenhuma identificação.

Em relação ao atributo de definição cor é pertinente salientar que além do nome da cor, da definição da cor com "**RGB**" é possível também definir a cor a partir de seu código numérico em hexadecimal. Desta forma a cor vermelho pode também ser declarada como '**color="#ff0000"**' a partir da instrução

```
julia> Plots.plot!([-6, 6], [0, 0], color="#ff0000", linewidth=2, label="")
```

Um gráfico com visual interessante é o do tipo superfície que mostra a superfície de um plano de forma tridimensional. Assim sendo, execute as seguintes instruções e observe a figura 5.40.

```
julia> f(x, y) = sin(sqrt(x^2 + y^2)) / sqrt(x^2 + y^2)
f (generic function with 2 methods)

julia> x = range(-10, 10, length=100)
-10.0:0.20202020202020202:10.0

julia> y = range(-10, 10, length=100)
-10.0:0.20202020202020202:10.0

julia> Plots.surface(x, y, f)  # ou Plots.plot(x, y, f, seriestype=:surface)
```

Gráficos tridimensionais, como comentado, e também os gráficos de superfície podem ser movimentados em seu eixo a partir do uso do atributo "**camara**". Para um teste execute primeiramente as seguintes instruções e veja o resultado da apresentação do gráfico na figura 5.41:

```
julia> f(x, y) = x^3 - 3x + y^2
f (generic function with 2 methods)

julia> x = range(-10, 10, length=100)
-10.0:0.20202020202020202:10.0

julia> y = range(-10, 10, length=100)
-10.0:0.20202020202020202:10.0

julia> Plots.surface(x, y, f)
```

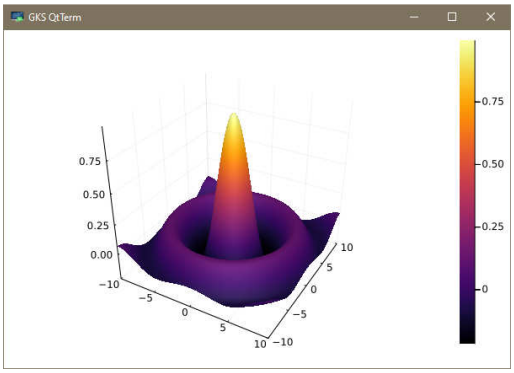

Figura 5.40 - Gráfico de superfície 1.

Figura 5.41 - Gráfico de superfície 2.

Veja na sequência alguns efeitos de uso do atributo "**camara**" aplicado sobre o gráfico da figura 5.41 que possibilita algumas visualizações interessantes indicadas entre as figuras 5.42 e 5.45.

```
julia> Plots.surface(x, y, f, camera=(30, 50))  # efeito na figura 5.42

julia> Plots.surface(x, y, f, camera=(30, 85))  # efeito na figura 5.43

julia> Plots.surface(x, y, f, camera=(30, -20))  # efeito na figura 5.44

julia> Plots.surface(x, y, f, camera=(-10, -20))  # efeito na figura 5.45
```

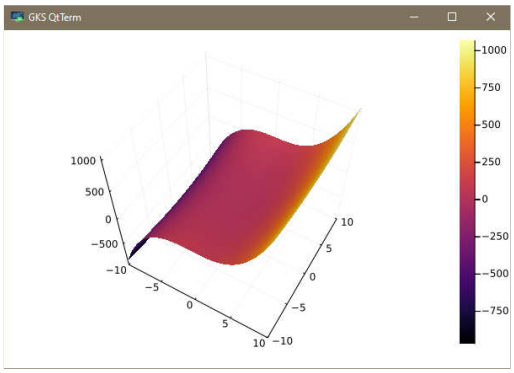

Figura 5.42 - Gráfico com mudança de posicionamento 1.

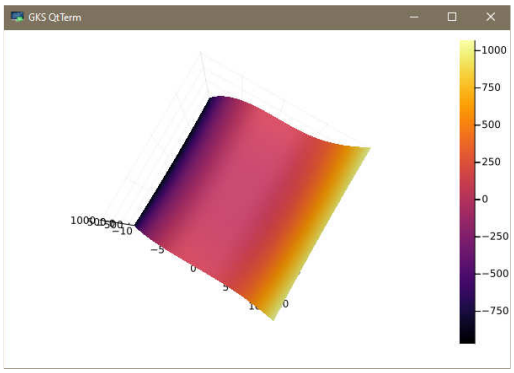

Figura 5.43 - Gráfico com mudança de posicionamento 2.

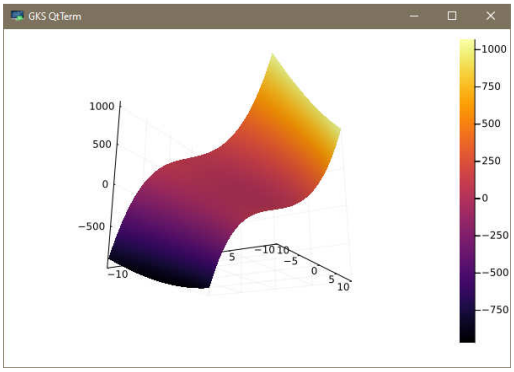

Figura 5.44 - Gráfico com mudança de posicionamento 3.

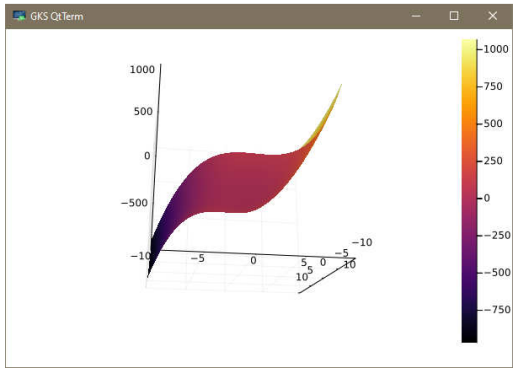

Figura 5.45 - Gráfico com mudança de posicionamento 4.

As posições negativas para o atributo "**camara**" permite ver o gráfico de baixo para cima.

Antes de finalizar este tópico é interessante ver mais um recurso que pode ser aplicado sobre gráficos, no sentido de melhorar a visualização dos dados plotados. Assim sendo, execute as instruções seguintes e veja o resultado junto a figura 5.46.

```
julia> x = range(1, 5)
1:5

julia> y = sqrt.(x)
5-element Vector{Float64}:
 1.0
 1.4142135623730951
 1.7320508075688772
 2.0
 2.23606797749979

julia> Plots.plot(x, y,
                 label="Raízes quadradas",
                 legend=:outerbottomright,
                 size=(700, 500),
                 annotation=[(x[i], y[i], string(y[i])) for i in 1:5]
       )
```

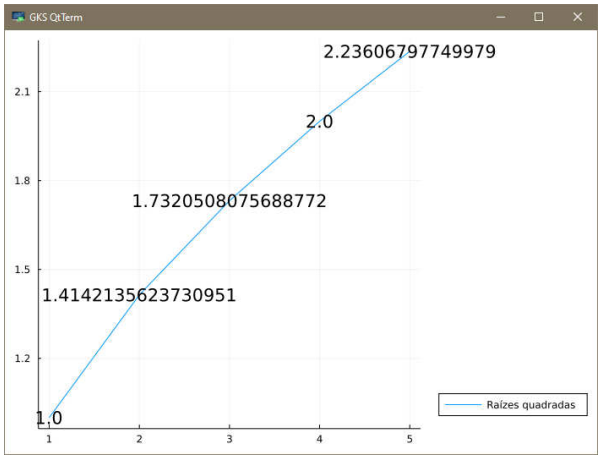

Figura 5.46 - Gráfico com texto no ponto cartesiano.

Para fazer a apresentação dos resultados da operação no ´ponto cartesiano usa-se o atributo "**annotation**" que é formado por uma tupla com três definições: primeiro os dados do eixo "**x**", primeiro os dados do eixo "**y**" e terceiro a definição do texto a ser apresentado. Para o texto impresso entrou em uso a função "**string**" que torna qualquer dado em formato cadeia. O uso do comando "**for**" é estabelecido para realizar a iteração entre os pontos cartesianos e coletar para cada posição de "**y**" o valor da raiz quadrada calculada.

A figura 5.46 indica que os valores das raízes apresentados no gráfico estão muito grandes. Para acertar o tamanho é necessário passar para a função "**string**" um segundo argumento com valor de formatação do tamanho. Veja a próxima instrução o ponto marcado em vermelho e seu resultado na figura 5.47.

```
julia> Plots.plot(x, y,
                  label="Raízes quadradas",
                  legend=:outerbottomright,
                  size=(700, 500),
                  annotation=[(x[i], y[i], string(y[i]), 8) for i in 1:5]
       )
```

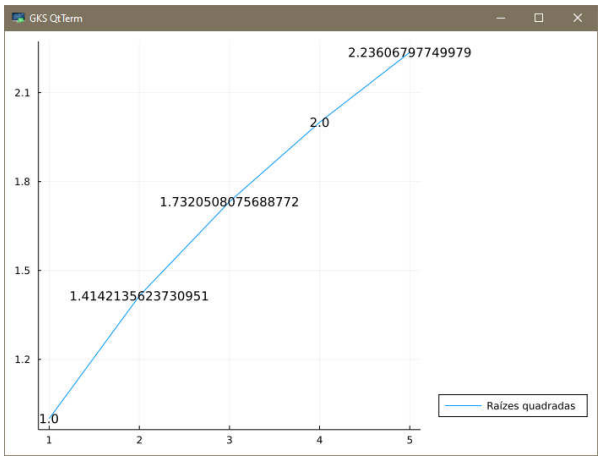

Figura 5.47 - Gráfico com texto com tamanho diminuído.

Os valores das raízes apresentados são extensos. Uma forma de mitigar esta apresentação é usar a função "**round**" para gerar valores, como por exemplo, com duas casas decimais no máximo. Para tanto, execute a instrução seguinte e observe o trecho sinalizado em vermelho e seu efeito na figura 5.48.

```
julia> Plots.plot(x, y,
                  label="Raízes quadradas",
                  legend=:outerbottomright,
                  size=(700, 500),
                  annotation=[(x[i], y[i], string(round(y[i]; digits=2)), 8) for i in 1:5]
       )
```

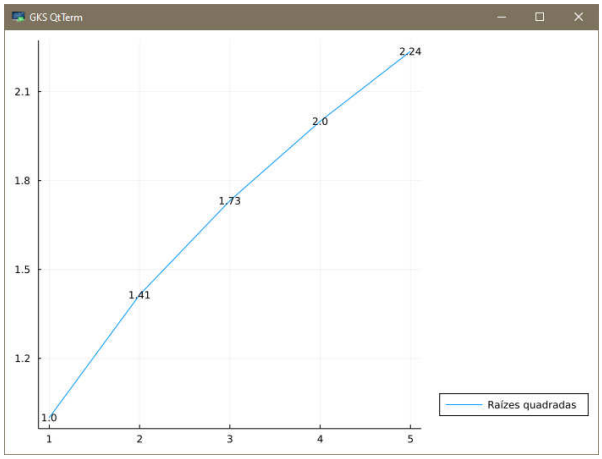

Figura 5.48 - Gráfico com dados formatados.

A outros efeitos e tipos de gráficos a serem desenvolvidos. Esta apresentação fica por aqui, pois o objetivo desta obra é apresentar os primeiros passos na linguagem.

5.8 - Quadros de dados

Julia é uma linguagem, que entre outras coisas é especialmente voltada para a análise de dados. O processo de análise de dados é em essência a atividade de tornar dados em fontes de conhecimento, normalmente cercados de grande conjunto de elementos de diversos tipos a serem estudados e acompanhados. Por ser uma ação que pode operar com grande conjunto de dados sua representação visual é importante e para auxiliar esta ação a linguagem possui pacotes específicos de formatação tabular direcionados ao tratamento desses dados na forma de tabelas, os quais são mais cômodos de acompanhar que as estruturas de arranjos se destacando o pacote "**DataFrames**".

Como todo pacote, o pacote "**DataFrames**", deve ser instalado para estar disponível para uso, portanto, execute a instrução seguinte e aguarde sua conclusão.

```
julia> using DataFrames
```

Se for a primeira vez de sua execução será informado a possibilidade de instalação. Neste sentido, confirme "**y**" quando solicitado e aguarde o processo de instalação do pacote ser concluído.

A partir da instalação deste pacote é possível criar diversas formas de apresentação de dados. Este recurso se parece, um pouco, com o mecanismo de criação e uso de arranjos, mas com maior flexibilidade visual. Para ver uma ocorrência simples execute a seguinte instrução que apresenta uma lista de preços:

```
julia> DataFrame(Item = 1:6,
                 Produto = ["Maça", "Laranja", "Abacaxi", "Banana", "Uva", "Pera"],
                 Preço = [1.23, 2.34, 3.45, 4.56, 5.67, 6.78]
       )
6×3 DataFrame
 Row │ Item   Produto  Preço
     │ Int64  String   Float64
─────┼──────────────────────────
   1 │     1  Maça        1.23
   2 │     2  Laranja     2.34
   3 │     3  Abacaxi     3.45
   4 │     4  Banana      4.56
   5 │     5  Uva         5.67
   6 │     6  Pera        6.78
```

A estrutura apresentada é indicada na forma do que se chama quadro de dados (*dataframe*), contento uma coluna interna de referência chamada "**Row**" que indica as posições de cada registro da estrutura. As colunas personalizadas ficam à direita da coluna "**Row**".

Assim como outros recursos operacionais da linguagem Julia, a criação de quadros de dados pode ser atribuída a uma determinada variável. Veja a próxima ocorrência de instruções contendo alguns dados da Fórmula 1 até 20 de julho de 2022.

```
julia> f1_2022 = DataFrame(Posição=1:3,
                           Piloto=["M. Verstappen", "C. Leclerc", "S. Perez"],
                           Equipe=["Red Bull", "Ferrari", "Red Bull"],
                           Pontos=[208, 170, 151],
                           Pais=["Bélgica", "Mônaco", "México"]
                 )
3×5 DataFrame
 Row │ Posição  Piloto         Equipe    Pontos  Pais
     │ Int64    String         String    Int64   String
─────┼──────────────────────────────────────────────────
   1 │       1  M. Verstappen  Red Bull     208  Bélgica
   2 │       2  C. Leclerc     Ferrari      170  Mônaco
   3 │       3  S. Perez       Red Bull     151  México

julia> f1_2022
3×5 DataFrame
 Row │ Posição  Piloto         Equipe    Pontos  Pais
     │ Int64    String         String    Int64   String
─────┼──────────────────────────────────────────────────
   1 │       1  M. Verstappen  Red Bull     208  Bélgica
   2 │       2  C. Leclerc     Ferrari      170  Mônaco
   3 │       3  S. Perez       Red Bull     151  México
```

Observe que a variável "*F1_2022*" armazena a estrutura gerada podendo ser utilizada para diversas operações computacionais. Os exemplos anteriores partem de uma situação onde os conjuntos de dados são existentes e previamente definidos. Mas, há a possibilidade de se operar com quadros de dados que sejam construídos na medida em que são necessários. Para tanto, veja a definição de uma variável vazia definida com a estrutura de um quadro de dados com as informações de alguns jogos da NFL (*National Football League*):

```
julia> SuperBowl = DataFrame()
0×0 DataFrame
```

A partir da ocorrência anterior uma variável do tipo quadro de dados está definida na memória contendo nenhuma linha e coluna e poderá receber dados na forma de colunas a qualquer instante. As colunas são identificadas a partir do operador "**.**" vinculado ao nome da variável *dataframe*. Veja o efeito das próximas instruções ao considerar inicialmente os três últimos campeões antes de 2021 do *Super Bowl*:

```
julia> SuperBowl.Ano = [2018, 2019, 2020]
3-element Vector{Int64}:
 2018
 2019
 2020

julia> SuperBowl.Time = ["Philadelphia Eagles", "New England Patriots", "Kansas City Chiefs"]
3-element Vector{String}:
 "Philadelphia Eagles"
 "New England Patriots"
 "Kansas City Chiefs"
```

```
julia> SuperBowl
3×2 DataFrame
 Row │ Ano    Time
     │ Int64  String
─────┼──────────────────────────
   1 │  2018  Philadelphia Eagles
   2 │  2019  New England Patriots
   3 │  2020  Kansas City Chiefs
```

O recurso de manipulação de dados dentro dos quadros de dados é semelhante ao que você já conhece sobre o uso de arranjos. Imagine ter que acrescentar um novo registro na forma de uma linha há um quadro de dados que já contenha elementos. Veja as instruções que acrescentam a variável "*SUPERBOWL*" os registros dos campeões dos anos de 2016 e 2017:

```
julia> push!(SuperBowl, (2017, "New England Patriots"))
4×2 DataFrame
   1 │  2018  Philadelphia Eagles
   2 │  2019  New England Patriots
   3 │  2020  Kansas City Chiefs
   4 │  2017  New England Patriots

julia> push!(SuperBowl, (2016, "Denver Broncos"))
5×2 DataFrame
   1 │  2018  Philadelphia Eagles
   2 │  2019  New England Patriots
   3 │  2020  Kansas City Chiefs
   4 │  2017  New England Patriots
   5 │  2016  Denver Broncos

julia> SuperBowl
5×2 DataFrame
   1 │  2018  Philadelphia Eagles
   2 │  2019  New England Patriots
   3 │  2020  Kansas City Chiefs
   4 │  2017  New England Patriots
   5 │  2016  Denver Broncos
```

A partir do armazenamento de um conjunto de dados é possível fazer pesquisas sobre os valores armazenados a partir de um conjunto de funcionalidades. Veja a instrução que apresenta o terceiro registro da variável "*SUPERBOWL*":

```
julia> SuperBowl[:3, :]
DataFrameRow
 Row │ Ano    Time
     │ Int64  String
─────┼──────────────────────────
   3 │  2020  Kansas City Chiefs
```

Veja a instrução que apresenta os dados da segunda coluna da variável "*SUPERBOWL*":

```
julia> SuperBowl[:, :2]
5-element Vector{String}:
 "Philadelphia Eagles"
 "New England Patriots"
 "Kansas City Chiefs"
 "New England Patriots"
 "Denver Broncos"
```

Veja a instrução que apresenta o dado da quarta linha, segunda coluna da variável "*SUPERBOWL*":

```
julia> SuperBowl[:4, :2]
"New England Patriots"
```

Agora veja o resultado da próxima instrução:

```
julia> SuperBowl[:, :]
5×2 DataFrame
   1 │  2018  Philadelphia Eagles
   2 │  2019  New England Patriots
   3 │  2020  Kansas City Chiefs
   4 │  2017  New England Patriots
   5 │  2016  Denver Broncos
```

Veja que usar o estilo "**[:, :]**" ou não usá-lo o efeito é o mesmo, pois nenhuma informação de linha ou coluna foi estabelecido. Observe que para ter acesso ao conjunto de dados pode-se fazer uso da referência de índice da indicação dos elementos de acesso linha e coluna a partir de "**[:linha, :coluna]**".

A apresentação de elementos de uma coluna também pode ser referenciada com o uso direto de seu nome de identificação. Veja a instrução para apresentação dos nomes dos times:

```
julia> SuperBowl[:, [:Time]]
5×1 DataFrame
 Row │ Time
     │ String
─────┼──────────────────────
   1 │ Philadelphia Eagles
   2 │ New England Patriots
   3 │ Kansas City Chiefs
   4 │ New England Patriots
   5 │ Denver Broncos
```

Os elementos de um *dataframe* também podem ser operacionalizados a partir das funções "**first**" e "**last**" que mostram, além do primeiro e último elemento da tabela, outros tipos de acesso. Veja as próximas instruções:

```
julia> first(SuperBowl)
DataFrameRow
 Row │ Ano    Time
     │ Int64  String
─────┼──────────────────────────
   1 │  2018  Philadelphia Eagles

julia> first(SuperBowl, 2)  # mostra os dois primeiros elementos
2×2 DataFrame
 Row │ Ano    Time
     │ Int64  String
─────┼──────────────────────────────
   1 │  2018  Philadelphia Eagles
   2 │  2019  New England Patriots
```

```
julia> last(SuperBowl)
DataFrameRow
 Row │ Ano    Time
     │ Int64  String
─────┼──────────────────────────
   5 │  2016  Denver Broncos
```

```
julia> last(SuperBowl, 3)  # mostra os três últimos elementos
3×2 DataFrame
 Row │ Ano    Time
     │ Int64  String
─────┼──────────────────────────
   1 │  2020  Kansas City Chiefs
   2 │  2017  New England Patriots
   3 │  2016  Denver Broncos
```

Os dados dos elementos definidos em um *dataframe* podem ser alterados por outros dados. Por exemplo, considere uma nova entrada com um erro de informação. Veja a próxima instrução:

```
julia> push!(SuperBowl, (2022, "Seattle Seahawks"))  # entrada com erro de informação
6×2 DataFrame
   1 │  2018  Philadelphia Eagles
   2 │  2019  New England Patriots
   3 │  2020  Kansas City Chiefs
   4 │  2017  New England Patriots
   5 │  2016  Denver Broncos
   6 │  2022  Seattle Seahawks
```

O erro está na informação do ano em que o time "Seattle Seahawks" foi campeão, sendo 2014. Para proceder a correção do dado veja a próxima instrução:

```
julia> replace!(SuperBowl.Ano, 2022 => 2014)
6-element Vector{Int64}:
 2018
 2019
 2020
 2017
 2016
 2014

julia> SuperBowl
6×2 DataFrame
   1 │  2018  Philadelphia Eagles
   2 │  2019  New England Patriots
   3 │  2020  Kansas City Chiefs
   4 │  2017  New England Patriots
   5 │  2016  Denver Broncos
   6 │  2014  Seattle Seahawks
```

A função "**`replace!`**" ou "**`replace`**" opera a partir da indicação de dois argumentos, sendo o primeiro o nome da variável a ser alterada e o segundo a indicação do dado existente apontado para o dado que será usado na mudança. Esta função também pode ser usada com arranjos.

Esta função quando em operação faz uma busca em todos os dados da variável apontada e efetua a troca de todos os dados que coincidem com o valor da busca, exceto se o argumento "**`count`**" indicar a quantidade de repetições a ser

realizada na operação. Para entender melhor seu funcionamento considere os próximos exemplos com a indicação de um arranjo do tipo vetor e observe cada linha de instrução apresentada.

```
julia> a = [1, 2, 3, 3, 4, 5, 6, 6, 6]
9-element Vector{Int64}:
 1
 2
 3
 3
 4
 5
 6
 6
 6

julia> replace!(a, 3 => 7, 6 => 8)  # altera todos os "3" por "7" e "6" por "8"
9-element Vector{Int64}:
 1
 2
 7
 7
 4
 5
 8
 8
 8

julia> replace!(a, 7 => 3, count=1)  # altera o primeiro "7" por "3"
9-element Vector{Int64}:
 1
 2
 3
 7
 4
 5
 8
 8
 8

julia> replace!(a, 8 => 6, count=2)  # altera os dois primeiros "8" por "6"
9-element Vector{Int64}:
 1
 2
 3
 7
 4
 5
 6
 6
 8
```

A função "**replace!**" ou "**replace**" pode operar com o uso de função aplicada e iterada sobre os valores da estrutura manipulada. Neste caso, usa-se no primeiro argumento a definição da função a ser aplicada e no segundo argumento a indicação da estrutura que será operada. Veja exemplos para a definição de operações sobre valores pares e impares.

```
julia> b = replace(x -> isodd(x) ? 2x : x, a)  # multiplica por dois os valores impares
9-element Vector{Int64}:
  2
  2
  6
 14
  4
 10
  6
  6
  8

julia> c = replace(x -> iseven(x) ? 3x : x, a)  # multiplica por três os valores pares
9-element Vector{Int64}:
  1
  6
  3
  7
 12
  5
 18
 18
 24
```

A partir de uma visão geral do uso da função "**replace!**" ou "**replace**" volta-se ao tratamento de valores em quadros de dados. A partir e um quadro de dados este pode ser manipulado de outras formas. É possível, por exemplo, extrair de um conjunto dados que deem origem a subconjuntos de dados a partir do uso da função "**select**". Veja as próximas instruções:

```
julia> times1 = select(SuperBowl, :Time)
6×1 DataFrame
 Row │ Time
     │ String
─────┼──────────────────────
   1 │ Philadelphia Eagles
   2 │ New England Patriots
   3 │ Kansas City Chiefs
   4 │ New England Patriots
   5 │ Denver Broncos
   6 │ Seattle Seahawks
```

A função "**select**" pode ser integrada com outras funções. Veja, por exemplo, a instrução para criar um subconjunto com os quatro primeiros registros dos times existente no quadro de dados "**SuperBowl**".

```
julia> times2 = first(select(SuperBowl, :Time), 4)
4×1 DataFrame
 Row │ Time
     │ String
─────┼──────────────────────
   1 │ Philadelphia Eagles
   2 │ New England Patriots
   3 │ Kansas City Chiefs
   4 │ New England Patriots
```

A criação de subconjuntos pode ser feita a partir da exclusão de certa coluna indicando sua negação. Observe o uso do construtor de negação "**Not**".

```
julia> times3 = select(SuperBowl, Not(:Time))
6×1 DataFrame
 Row │ Ano
     │ Int64
─────┼───────
   1 │  2018
   2 │  2019
   3 │  2020
   4 │  2017
   5 │  2016
   6 │  2014
```

Subconjuntos podem ser definidos a partir da indicação de linhas e coluna especificadas. Por exemplo, gerar um subconjunto da segunda até a quinta linha com a indicação das colunas a serem consideradas. Para atender a esta necessidade não se usa a função "**select**", faz-se o pedido da ação diretamente sobre a variável do tipo quando de dados. Veja a instrução para a criação de um subconjunto contendo as informações mencionadas a partir do nome do time e indicação do ano em que foi vencedor.

```
julia> times4 = SuperBowl[2:4, [:Time, :Ano]]
3×2 DataFrame
 Row │ Time                 Ano
     │ String               Int64
─────┼──────────────────────────────
   1 │ New England Patriots   2019
   2 │ Kansas City Chiefs     2020
   3 │ New England Patriots   2017
```

Outra edição importante em um quadro de dados é a possibilidade de inserir novas colunas (além de linhas, já visto com a função "**push!**"), bem como também remover colunas e linhas (com a função "**delete!**").

A inserção de coluna em um quadro de dados se faz com o uso da função "**insertcols!**", que permite inclusive escolher a posição onde a coluna será incluída. Para um teste, será incluída uma coluna que indicará os vice-campeões na última posição do quadro de dados. Assim sendo, execute a instrução:

```
julia> insertcols!(SuperBowl, 3, :Vice => ["New England Patriots",
                                           "Los Angeles Rams",
                                           "San Francisco 49ers",
                                           "Atlanta Falcons",
                                           "Carolina Panthers",
                                           "Denver Broncos"]
       )
6×3 DataFrame
 Row │ Ano    Time                 Vice
     │ Int64  String               String
─────┼──────────────────────────────────────────────
   1 │  2018  Philadelphia Eagles  New England Patriots
   2 │  2019  New England Patriots Los Angeles Rams
   3 │  2020  Kansas City Chiefs   San Francisco 49ers
   4 │  2017  New England Patriots Atlanta Falcons
   5 │  2016  Denver Broncos       Carolina Panthers
   6 │  2014  Seattle Seahawks     Denver Broncos
```

Na sequência será ainda inserida as colunas com o placar dos jogos e o número das edições dos jogos.

Execute a próxima instrução e observe a colocação da coluna junto a terceira posição empurrando a coluna de vice-campeões para a direita:

```
julia> insertcols!(SuperBowl, 3, :Placar => ["41 x 33", "13 x  3", "31 x 20",
                                             "34 x 28", "24 x 10", "43 x  8"]
       )
6×4 DataFrame
 Row │ Ano    Time                 Placar   Vice
     │ Int64  String               String   String
─────┼──────────────────────────────────────────────────────────
   1 │  2018  Philadelphia Eagles  41 x 33  New England Patriots
   2 │  2019  New England Patriots 13 x  3  Los Angeles Rams
   3 │  2020  Kansas City Chiefs   31 x 20  San Francisco 49ers
   4 │  2017  New England Patriots 34 x 28  Atlanta Falcons
   5 │  2016  Denver Broncos       24 x 10  Carolina Panthers
   6 │  2014  Seattle Seahawks     43 x  8  Denver Broncos
```

Execute a próxima instrução e observe a colocação da coluna junto a segunda posição empurrando as colunas de time, placar e vice-campeões para a direita:

```
julia> insertcols!(SuperBowl, 2, :Edição => ["LII", "LIII", "LIV", "LI", "L", "XLVIII"])
6×5 DataFrame
 Row │ Ano    Edição  Time                 Placar   Vice
     │ Int64  String  String               String   String
─────┼──────────────────────────────────────────────────────────
   1 │  2018  LII     Philadelphia Eagles  41 x 33  New England Patriots
   2 │  2019  LIII    New England Patriots 13 x  3  Los Angeles Rams
   3 │  2020  LIV     Kansas City Chiefs   31 x 20  San Francisco 49ers
   4 │  2017  LI      New England Patriots 34 x 28  Atlanta Falcons
   5 │  2016  L       Denver Broncos       24 x 10  Carolina Panthers
   6 │  2014  XLVIII  Seattle Seahawks     43 x  8  Denver Broncos
```

Antes de realizar a remoção de uma coluna do quadro de dados é ideal fazer antes uma cópia do mesmo e sobre esta cópia aplicar a remoção. Assim sendo, execute a seguinte instrução:

```
julia> sb1 = copy(SuperBowl)
6×5 DataFrame
 Row │ Ano    Edição  Time                 Placar   Vice
     │ Int64  String  String               String   String
─────┼──────────────────────────────────────────────────────────
   1 │  2018  LII     Philadelphia Eagles  41 x 33  New England Patriots
   2 │  2019  LIII    New England Patriots 13 x  3  Los Angeles Rams
   3 │  2020  LIV     Kansas City Chiefs   31 x 20  San Francisco 49ers
   4 │  2017  LI      New England Patriots 34 x 28  Atlanta Falcons
   5 │  2016  L       Denver Broncos       24 x 10  Carolina Panthers
   6 │  2014  XLVIII  Seattle Seahawks     43 x  8  Denver Broncos

julia> sb1
6×5 DataFrame
 Row │ Ano    Edição  Time                 Placar   Vice
     │ Int64  String  String               String   String
─────┼──────────────────────────────────────────────────────────
   1 │  2018  LII     Philadelphia Eagles  41 x 33  New England Patriots
   2 │  2019  LIII    New England Patriots 13 x  3  Los Angeles Rams
   3 │  2020  LIV     Kansas City Chiefs   31 x 20  San Francisco 49ers
   4 │  2017  LI      New England Patriots 34 x 28  Atlanta Falcons
   5 │  2016  L       Denver Broncos       24 x 10  Carolina Panthers
   6 │  2014  XLVIII  Seattle Seahawks     43 x  8  Denver Broncos
```

A partir da definição da variável "*SB1*" sobre esta variável será efetuada a remoção da coluna de vice-campeões a partir do uso da função "**select!**" com o uso do construtor "**Not**" frente a coluna a ser removida. Para tanto, execute a seguinte instrução:

```
julia> select!(sb1, Not(:Vice))
6×4 DataFrame
 Row │ Ano    Edição  Time                 Placar
     │ Int64  String  String               String
─────┼──────────────────────────────────────────────
   1 │  2018  LII     Philadelphia Eagles  41 x 33
   2 │  2019  LIII    New England Patriots 13 x  3
   3 │  2020  LIV     Kansas City Chiefs   31 x 20
   4 │  2017  LI      New England Patriots 34 x 28
   5 │  2016  L       Denver Broncos       24 x 10
   6 │  2014  XLVIII  Seattle Seahawks     43 x  8

julia> sb1
6×4 DataFrame
 Row │ Ano    Edição  Time                 Placar
     │ Int64  String  String               String
─────┼──────────────────────────────────────────────
   1 │  2018  LII     Philadelphia Eagles  41 x 33
   2 │  2019  LIII    New England Patriots 13 x  3
   3 │  2020  LIV     Kansas City Chiefs   31 x 20
   4 │  2017  LI      New England Patriots 34 x 28
   5 │  2016  L       Denver Broncos       24 x 10
   6 │  2014  XLVIII  Seattle Seahawks     43 x  8
```

Sobre o quadro de dados "*SB1*" veja a remoção da quarta linha. Assim sendo, execute a instrução:

```
julia> delete!(sb1, 4)
5×4 DataFrame
 Row │ Ano    Edição  Time                 Placar
     │ Int64  String  String               String
─────┼──────────────────────────────────────────────
   1 │  2018  LII     Philadelphia Eagles  41 x 33
   2 │  2019  LIII    New England Patriots 13 x  3
   3 │  2020  LIV     Kansas City Chiefs   31 x 20
   4 │  2016  L       Denver Broncos       24 x 10
   5 │  2014  XLVIII  Seattle Seahawks     43 x  8
```

Os elementos que compõem um quadro de dados podem ser classificados com o uso da função "**sort**" ou "**sort!**". Para tanto, observe o resultado da próxima instrução:

```
julia> sb2 = sort(sb1, :Ano)  # classificação do Ano crescente
5×4 DataFrame
 Row │ Ano    Edição  Time                 Placar
     │ Int64  String  String               String
─────┼──────────────────────────────────────────────
   1 │  2014  XLVIII  Seattle Seahawks     43 x  8
   2 │  2016  L       Denver Broncos       24 x 10
   3 │  2018  LII     Philadelphia Eagles  41 x 33
   4 │  2019  LIII    New England Patriots 13 x  3
   5 │  2020  LIV     Kansas City Chiefs   31 x 20
```

```
julia> sb2
5×4 DataFrame
 Row │ Ano    Edição  Time                 Placar
     │ Int64  String  String               String
─────┼──────────────────────────────────────────
   1 │  2014  XLVIII  Seattle Seahawks     43 x  8
   2 │  2016  L       Denver Broncos       24 x 10
   3 │  2018  LII     Philadelphia Eagles  41 x 33
   4 │  2019  LIII    New England Patriots 13 x  3
   5 │  2020  LIV     Kansas City Chiefs   31 x 20

julia> sb3 = sort(SuperBowl, :Ano, rev=true)  # classificação do Ano decrescente
6×5 DataFrame
 Row │ Ano    Edição  Time                 Placar   Vice
     │ Int64  String  String               String   String
─────┼──────────────────────────────────────────────────────────
   1 │  2020  LIV     Kansas City Chiefs   31 x 20  San Francisco 49ers
   2 │  2019  LIII    New England Patriots 13 x  3  Los Angeles Rams
   3 │  2018  LII     Philadelphia Eagles  41 x 33  New England Patriots
   4 │  2017  LI      New England Patriots 34 x 28  Atlanta Falcons
   5 │  2016  L       Denver Broncos       24 x 10  Carolina Panthers
   6 │  2014  XLVIII  Seattle Seahawks     43 x  8  Denver Broncos
```

A estrutura dos quadros de dados pode ser verificada a partir do uso de algumas funções que mostram características estruturais de sua definição. Observe as instruções seguintes e atente para os efeitos de uso apresentados a partir de algumas funções auxiliares:

```
julia> show(sb3)  # ação idêntica a executar diretamente "sb3"
6×5 DataFrame
 Row │ Ano    Edição  Time                 Placar   Vice
     │ Int64  String  String               String   String
─────┼──────────────────────────────────────────────────────────
   1 │  2020  LIV     Kansas City Chiefs   31 x 20  San Francisco 49ers
   2 │  2019  LIII    New England Patriots 13 x  3  Los Angeles Rams
   3 │  2018  LII     Philadelphia Eagles  41 x 33  New England Patriots
   4 │  2017  LI      New England Patriots 34 x 28  Atlanta Falcons
   5 │  2016  L       Denver Broncos       24 x 10  Carolina Panthers
   6 │  2014  XLVIII  Seattle Seahawks     43 x  8  Denver Broncos

julia> show(sb3, show_row_number=false)  # omite na apresentação a coluna "Row"
6×5 DataFrame
 Ano    Edição  Time                 Placar   Vice
 Int64  String  String               String   String
──────────────────────────────────────────────────────────
  2020  LIV     Kansas City Chiefs   31 x 20  San Francisco 49ers
  2019  LIII    New England Patriots 13 x  3  Los Angeles Rams
  2018  LII     Philadelphia Eagles  41 x 33  New England Patriots
  2017  LI      New England Patriots 34 x 28  Atlanta Falcons
  2016  L       Denver Broncos       24 x 10  Carolina Panthers
  2014  XLVIII  Seattle Seahawks     43 x  8  Denver Broncos

julia> size(sb3)  # mostra tamanho do quanto em quantidade de linhas e colunas
(6, 5)
```

```
julia> names(sb3)  # mostra os nomes das colunas na forma de um vetor
5-element Vector{String}:
 "Ano"
 "Edição"
 "Time"
 "Placar"
 "Vice"

julia> [eltype(x) for x in eachcol(sb3)]  # mostra os tipos das colunas (votor)
5-element Vector{DataType}:
 Int64
 String
 String
 String
 String
```

Um efeito que pode ser aplicado sobre quadros de dados é efetuar a mudança dos nomes das colunas de dados. A variável "*F1_2022*" possui uma coluna identificada como "**Pais**", a qual deveria se chamar "**Pais**". Para fazer a mudança do nome da coluna execute a instrução a seguir que utiliza a função "**rename!**":

```
julia> f1_2022
3×5 DataFrame
 Row │ Posição  Piloto         Equipe    Pontos  Pais
     │ Int64    String         String    Int64   String
─────┼──────────────────────────────────────────────────
   1 │       1  M. Verstappen  Red Bull     208  Bélgica
   2 │       2  C. Leclerc     Ferrari      170  Mônaco
   3 │       3  S. Perez       Red Bull     151  México

julia> rename!(f1_2022, :Pais => :País)
3×5 DataFrame
 Row │ Posição  Piloto         Equipe    Pontos  País
     │ Int64    String         String    Int64   String
─────┼──────────────────────────────────────────────────
   1 │       1  M. Verstappen  Red Bull     208  Bélgica
   2 │       2  C. Leclerc     Ferrari      170  Mônaco
   3 │       3  S. Perez       Red Bull     151  México
```

Além do quadro de dados "*F1_2022*" um novo quadro contendo o armazenamento dos pontos das onze primeiras provas dos três primeiros colocados da temporada 2022 se faz necessário para apresentar mais alguns detalhes de manipulação de dados. Observe as próximas instruções:

```
julia> f1_2022
3×5 DataFrame
 Row │ Posição  Piloto         Equipe    Pontos  País
     │ Int64    String         String    Int64   String
─────┼──────────────────────────────────────────────────
   1 │       1  M. Verstappen  Red Bull     208  Bélgica
   2 │       2  C. Leclerc     Ferrari      170  Mônaco
   3 │       3  S. Perez       Red Bull     151  México
```

```
julia> pts_f1_22 = DataFrame(
                             Pontos_MV=[00, 25, 00, 34, 26, 25, 15, 25, 25, 06, 27],
                             Pontos_CL=[26, 19, 26, 15, 18, 00, 12, 00, 10, 12, 32],
                             Pontos_SP=[00, 12, 18, 24, 12, 19, 25, 19, 00, 18, 04],
                             ProvaPaís=["BRN", "SAU", "AUS", "ITA", "MIA", "ESP",
                                        "MON", "AZB", "CAN", "GBR", "AUT"]
                   )
11×4 DataFrame
Row │ Pontos_MV  Pontos_CL  Pontos_SP  ProvaPaís
    │ Int64      Int64      Int64      String
────┼──────────────────────────────────────────
  1 │         0         26          0  BRN
  2 │        25         19         12  SAU
  3 │         0         26         18  AUS
  4 │        34         15         24  ITA
  5 │        26         18         12  MIA
  6 │        25          0         19  ESP
  7 │        15         12         25  MON
  8 │        25          0         19  AZB
  9 │        25         10          0  CAN
 10 │         6         12         18  GBR
 11 │        27         32          4  AUT
```

Antes de operar junções cabe ampliar um pouco mais as ações básicas de análise de dados. Neste sentido, atente para o uso da função "**describe**" que tem por finalidade apresentar alguns dados estatísticos básicos dos campos da tabela indicada. Veja na sequência o uso desta função no quadro de dados "*PTS_F1_22*":

```
julia> describe(pts_f1_22)
4×7 DataFrame
 Row │ variable   mean     min  median  max  nmissing  eltype
     │ Symbol     Union…   Any  Union…  Any  Int64     DataType
─────┼──────────────────────────────────────────────────────────
   1 │ Pontos_MV  18.9091  0    25.0    34          0  Int64
   2 │ Pontos_CL  15.4545  0    15.0    32          0  Int64
   3 │ Pontos_SP  13.7273  0    18.0    25          0  Int64
   4 │ ProvaPaís           AUS          SAU         0  String
```

Veja que cada campo da tabela tem indicado a sua frente alguma informação de seu conjunto de dados, como a média (*mean*), o valor mínimo (*min*), a mediana (*median*) e o valor máximo do conjunto de valores existentes. Não se preocupe com a informação da coluna "**nmissing**", pois esta não aqui considerada.

A função "**describe**" pode ser usada de algumas outras maneiras. Veja atentamente os exemplos de instruções indicadas a seguir:

```
julia> describe(pts_f1_22, :min, :max)  # extraí valores mínimos e máximos
4×3 DataFrame
 Row │ variable   min    max
     │ Symbol     Int64  Int64
─────┼──────────────────────────
   1 │ Pontos_MV  0      34
   2 │ Pontos_CL  0      32
   3 │ Pontos_SP  0      25
   4 │ ProvaPaís  AUS    SAU
```

```
julia> describe(pts_f1_22, :max, cols=:Pontos_MV)  # extraí valor máximo da coluna apontada
1×2 DataFrame
 Row │ variable   max
     │ Symbol     Int64
─────┼──────────────────
   1 │ Pontos_MV     34

julia> describe(pts_f1_22, cols=:Pontos_CL)  # extraí dados da coluna apontada
1×7 DataFrame
 Row │ variable   mean     min    median   max    nmissing  eltype
     │ Symbol     Float64  Int64  Float64  Int64  Int64     DataType
─────┼──────────────────────────────────────────────────────────────
   1 │ Pontos_CL  15.4545      0     15.0     32         0  Int64

julia> describe(pts_f1_22, sum => :Total)  # efetua a soma dos pontos de cada piloto
4×2 DataFrame
 Row │ variable   Total
     │ Symbol     Union…
─────┼──────────────────
   1 │ Pontos_MV  208
   2 │ Pontos_CL  170
   3 │ Pontos_SP  151
   4 │ ProvaPaís

julia> describe(pts_f1_22, sum => :Total, cols=:Pontos_CL)  # soma dos pontos de " C. Leclerc"
1×2 DataFrame
 Row │ variable   Total
     │ Symbol     Int64
─────┼──────────────────
   1 │ Pontos_CL    170
```

Além das operações de "**sum**", "**min**" e "**max**" demonstradas, é possível indicar outras ações estatísticas básicas a serem obtidas manualmente, como, por exemplo, "**first**" de primeiro valor existente e "**last**" de último valor existente. Veja a próxima instrução:

```
julia> describe(pts_f1_22, first => :"Primeiro valor", last => :"Último valor")
4×3 DataFrame
 Row │ variable   Primeiro valor  Último valor
     │ Symbol     Any             Any
─────┼──────────────────────────────────────────
   1 │ Pontos_MV  0               27
   2 │ Pontos_CL  26              32
   3 │ Pontos_SP  0               4
   4 │ ProvaPaís  BRN             AUT
```

Veja na sequência a apresentação das descrições de dados indicadas de forma customizada para o piloto "S. Perez":

```
julia> describe(pts_f1_22, :min, :max, :std, :median, :q25, :q75, cols=:Pontos_SP)
4×7 DataFrame
 Row │ variable   min  max  std      median  q25     q75
     │ Symbol     Any  Any  Union…   Union…  Union…  Union…
─────┼──────────────────────────────────────────────────────────
   1 │ Pontos_SP  0    25   8.95646  18.0    8.0     19.0
```

Os símbolos "**:q25**" e "**:q75**" referem-se a definição dos intervalos Interquartis do coeficiente de correlação e da diferença da raiz-média-quadrada. A mediana por exemplo refere-se ao interquartil "*q50*".

Caso queira apresentar todos os elementos de informação estatística use a sintaxe a seguir:

```
julia> describe(pts_f1_22, :all)
4×13 DataFrame
 Row │ variable   mean     std      min  q25     median  q75     max  nunique  nmissing  first  last  eltype
     │ Symbol     Union…   Union…   Any  Union…  Union…  Union…  Any  Union…   Int64     Any    Any   DataType
─────┼──────────────────────────────────────────────────────────────────────────────────────────────────────────
   1 │ Pontos_MV  18.9091  11.7852  0    10.5    25.0    25.5    34                    0  0      27    Int64
   2 │ Pontos_CL  15.4545  10.231   0    11.0    15.0    22.5    32                    0  26     32    Int64
   3 │ Pontos_SP  13.7273  8.95646  0    8.0     18.0    19.0    25                    0  0      4     Int64
   4 │ ProvaPais                    AUS                          SAU  11               0  BRN    AUT   String
```

Caso queira visualizar apenas os valores de um determinado campo (relacionado, neste caso, a determinado piloto), como por exemplo ver os dados de pontos alcançados pelo piloto "M. Verstappen" execute a linha de instrução:

```
julia> pts_f1_22[!, :Pontos_MV]
11-element Vector{Int64}:
  0
 25
  0
 34
 26
 25
 15
 25
 25
  6
 27
```

A fim ampliar um pouco mais o tema sobre manipulação de quadros de dados é interessante saber que é possível fazer a junção de dados entre estruturadas de quadros com uso da função "**innerjoin**".

Para demonstrar esta operação considere a definição de dois quadros seguintes, sendo um para armazenamento de professores (*tabpro*) e outro para armazenamento de disciplinas (*tabdis*). Assim sendo, execute as seguintes instruções:

```
julia> tabpro = DataFrame()
0×0 DataFrame

julia> tabdis = DataFrame()
0×0 DataFrame
```

Agora informe os dados dos professores:

```
julia> tabpro.CodPro = [100, 110, 120, 130, 140, 150]
6-element Vector{Int64}:
 100
 110
 120
 130
 140
 150
```

```
julia> tabpro.NomePro = ["AMANDA", "JOSÉ", "PAULO", "SILVIA", "JULIA", "JOÃO"]
6-element Vector{String}:
 "AMANDA"
 "JOSÉ"
 "PAULO"
 "SILVIA"
 "JULIA"
 "JOÃO"
```

Agora informe os dados dos cursos:

```
julia> tabdis.CodDis = ["CG-100", "CG-110", "CG-200", "CG-300", "TI-100", "TI-200",
                        "TI-205", "TI-300", "TI-400", "TI-500", "CG-100", "CG-300"
       ]
12-element Vector{String}:
 "CG-100"
 "CG-110"
 "CG-200"
 "CG-300"
 "TI-100"
 "TI-200"
 "TI-205"
 "TI-300"
 "TI-400"
 "TI-500"
 "CG-100"
 "CG-300"

julia> tabdis.NomeDis = ["MATEMÁTICA", "FÍSICA", "PORTUGUÊS", "FILOSOFIA", "ALGORITMOS",
                         "LING. PROGRAMAÇÃO I", "LING. PROGRAMAÇÃO II", "BANCO DE DADOS",
                         "SISTEMAS OPERACIONAIS", "PROGRAMAÇÃO WEB", "MATEMÁTICA", "FILOSOFIA"
       ]
12-element Vector{String}:
 "MATEMÁTICA"
 "FÍSICA"
 "PORTUGUÊS"
 "FILOSOFIA"
 "ALGORITMOS"
 "LING. PROGRAMAÇÃO I"
 "LING. PROGRAMAÇÃO II"
 "BANCO DE DADOS"
 "SISTEMAS OPERACIONAIS"
 "PROGRAMAÇÃO WEB"
 "MATEMÁTICA"
 "FILOSOFIA"

julia> tabdis.CodPro = [100, 100, 110, 110, 120, 120, 130, 130, 140, 140, 150, 150]
12-element Vector{Int64}:
 100
 100
 110
 110
 120
 120
 130
 130
 140
```

```
 140
 150
 150

julia> tabpro
6×2 DataFrame
 Row │ CodPro  NomePro
     │ Int64   String
─────┼──────────────────
   1 │    100  AMANDA
   2 │    110  JOSÉ
   3 │    120  PAULO
   4 │    130  SILVIA
   5 │    140  JULIA
   6 │    150  JOÃO

julia> tabdis
12×3 DataFrame
 Row │ CodDis  NomeDis                CodPro
     │ String  String                 Int64
─────┼──────────────────────────────────────
   1 │ CG-100  MATEMÁTICA                100
   2 │ CG-110  FÍSICA                    100
   3 │ CG-200  PORTUGUÊS                 110
   4 │ CG-300  FILOSOFIA                 110
   5 │ TI-100  ALGORITMOS                120
   6 │ TI-200  LING. PROGRAMAÇÃO I       120
   7 │ TI-205  LING. PROGRAMAÇÃO II      130
   8 │ TI-300  BANCO DE DADOS            130
   9 │ TI-400  SISTEMAS OPERACIONAIS     140
  10 │ TI-500  PROGRAMAÇÃO WEB           140
  11 │ CG-100  MATEMÁTICA                150
  12 │ CG-300  FILOSOFIA                 150
```

A partir da criação dos dois quadros de dados anteriores execute a seguinte instrução e veja a ocorrência apresentada:

```
julia> aulas = innerjoin(tabpro, tabdis, on=:CodPro)
12×4 DataFrame
 Row │ CodPro  NomePro  CodDis  NomeDis
     │ Int64   String   String  String
─────┼──────────────────────────────────────────
   1 │    100  AMANDA   CG-100  MATEMÁTICA
   2 │    100  AMANDA   CG-110  FÍSICA
   3 │    110  JOSÉ     CG-200  PORTUGUÊS
   4 │    110  JOSÉ     CG-300  FILOSOFIA
   5 │    120  PAULO    TI-100  ALGORITMOS
   6 │    120  PAULO    TI-200  LING. PROGRAMAÇÃO I
   7 │    130  SILVIA   TI-205  LING. PROGRAMAÇÃO II
   8 │    130  SILVIA   TI-300  BANCO DE DADOS
   9 │    140  JULIA    TI-400  SISTEMAS OPERACIONAIS
  10 │    140  JULIA    TI-500  PROGRAMAÇÃO WEB
  11 │    150  JOÃO     CG-100  MATEMÁTICA
  12 │    150  JOÃO     CG-300  FILOSOFIA
```

Veja que o resultado apresentado junto a variável "*AULAS*" efetua o vínculo ou junção dos quadros de disciplina com os nomes dos professores que a estão ministrando. Note que neste vínculo são relacionadas todas as colunas do quadro

de professores com as colunas do quadro de cursos. Veja que há professor que ministra mais de uma disciplina e que há uma mesma disciplina ministrada por mais de um professor. A função "**innerjoin**" para operar necessita que ambos os quadros possuam dados preenchidos em todas as posições de suas colunas de vínculo.

Observe que para realizar a junção da variável "*AULAS*" a coluna usada como pivô não deve possuir dados que sejam repetidos, pois esses dados são usados como chaves primárias de vinculação. Já o quadro de dados de cursos vinculado pode conter dados que sejam repetitivos sem nenhum problema.

Além da função "**innerjoin**" há outras funções similares que operam ações de junções: "**leftjoin**", "**rightjoin**", "**outerjoin**" e "**crossjoin**" (FARIAS, 2019), sendo essas algumas das disponíveis:

- **leftjoin**: a saída contém linhas para valores da chave que existem no primeiro argumento a ser unido com o segundo argumento, independentemente de existir ou não no segundo argumento valores correspondentes;
- **rightjoin**: a saída contém linhas para valores da chave que existem no segundo argumento a ser unido com o primeiro argumento, independentemente de existir ou não no primeiro argumento valores correspondentes;
- **outerjoin**: a saída contém linhas para valores da chave que existem no primeiro argumentos ou existam no segundo argumento;
- **crossjoin**: a saída contém linhas para valores da chave que existem no primeiro argumento relacionados a partir de um produto cartesiano com os valores do segundo argumento.

Para realizar testes com as funcionalidades "**leftjoin**", "**rightjoin**", "**outerjoin**" e "**crossjoin**" é necessário considerar outros quadros de dados que sejam condizentes com cada uma das ações. Assim sendo, atente para os próximos exemplos, baseados no sistema de ajuda da linguagem Julia, que mostra a relação entre colaboradores de uma empresa e os setores em que trabalham.

```
julia> equipe = DataFrame(CodFun = [10, 20, 30],
                          Nome = ["Pedro", "Mariza", "Paulo"]
               )
3×2 DataFrame
 Row │ CodFun  Nome
     │ Int64   String
─────┼──────────────
   1 │     10  Pedro
   2 │     20  Mariza
   3 │     30  Paulo

julia> setor = DataFrame(CodFun = [10, 20, 40],
                         Setor = ["Controladoria", "Administração", "Almoxarifado"]
               )
3×2 DataFrame
 Row │ CodFun  Setor
     │ Int64   String
─────┼──────────────────
   1 │     10  Controladoria
   2 │     20  Administração
   3 │     40  Almoxarifado
```

O exemplo seguinte opera a junção de quadros de dados a partir da ação da função "**leftjoin**", onde há um quadro de colaboradores e um quadro de setores, em que no quadro de setores a indicação de um setor ocupado por um colaborador que não existe no quadro de colaboradores. Veja a ocorrência de apresentação do termo "**missing**" para o setor indicando a não existência deste dado no segundo quadro.

```
julia> leftjoin(equipe, setor, on=:CodFun)
3×3 DataFrame
 Row │ CodFun  Nome    Setor
     │ Int64   String  String?
─────┼──────────────────────────
   1 │     10  Pedro   Controladoria
   2 │     20  Mariza  Administração
   3 │     30  Paulo   missing
```

O exemplo seguinte opera a junção de quadros de dados a partir da ação da função "**rightjoin**", onde há um quadro de colaboradores e um quadro de setores, em que no quadro de colaboradores a indicação de um colaborador não vinculado a um setor. Veja a ocorrência de apresentação do termo "**missing**" para o nome indicando a não existência deste dado no primeiro quadro.

```
julia> rightjoin(equipe, setor, on=:CodFun)
3×3 DataFrame
 Row │ CodFun  Nome    Setor
     │ Int64   String  String?
─────┼──────────────────────────
   1 │     10  Pedro   Controladoria
   2 │     20  Mariza  Administração
   3 │     40  missing  Almoxarifado
```

O exemplo seguinte opera a junção de quadros de dados a partir da ação da função "**outerjoin**", onde pode haver tanto no quadro de colaboradores ou quadro de setores a ausência de algum dado. Veja a ocorrência de apresentação do termo "**missing**" nas colunas do primeiro e segundo quadros.

```
julia> outerjoin(equipe, setor, on=:CodFun)
4×3 DataFrame
 Row │ CodFun  Nome     Setor
     │ Int64   String?  String?
─────┼──────────────────────────
   1 │     10  Pedro    Controladoria
   2 │     20  Mariza   Administração
   3 │     30  Paulo    missing
   4 │     40  missing  Almoxarifado
```

Para verificar o funcionamento da função "**crossjoin**" é necessário considerar a definição de dois outros quadros de dados que representem exclusivamente posições cartesianas. Assim sendo, observe as próximas instruções e acompanhe a definição de um gráfico da relação estabelecida indicado na figura 5.49:

```
julia> df1 = DataFrame(x=1:3)
3×1 DataFrame
 Row │ x
     │ Int64
─────┼───────
   1 │     1
   2 │     2
   3 │     3
```

```
julia> df2 = DataFrame(y=5:7)
3×1 DataFrame
 Row │ y
     │ Int64
─────┼───────
   1 │     5
   2 │     6
   3 │     7
```

```
julia> g1 = crossjoin(df1, df2)
9×2 DataFrame
 Row │ x      y
     │ Int64  Int64
─────┼──────────────
   1 │     1      5
   2 │     1      6
   3 │     1      7
   4 │     2      5
   5 │     2      6
   6 │     2      7
   7 │     3      5
   8 │     3      6
   9 │     3      7
```

```
julia> using Plots

julia> Plots.plot(g1.x, g1.y)
```

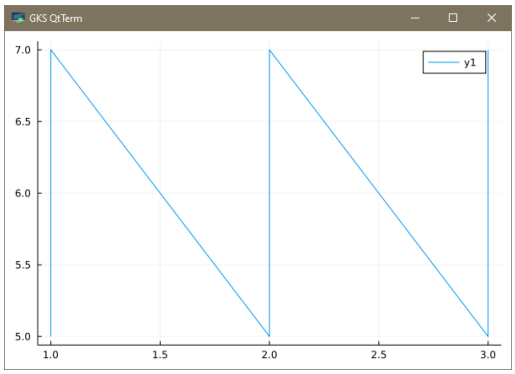

Figura 5.49 - Gráfico com dados da função "crossjoin".

Considere separar em dois vetores distintos os dados existentes em "*G1*". Para tanto, execute as seguintes instruções:

```
julia> x, y = eachcol(g1)
9×2 DataFrame
 Row │ x      y
     │ Int64  Int64
─────┼──────────────
   1 │     1      5
   2 │     1      6
   3 │     1      7
   4 │     2      5
   5 │     2      6
   6 │     2      7
   7 │     3      5
   8 │     3      6
   9 │     3      7
```

```
julia> x
9-element Vector{Int64}:
 1
 1
 1
 2
 2
 2
 3
 3
 3

julia> y
9-element Vector{Int64}:
 5
 6
 7
 5
 6
 7
 5
 6
 7
```

Veja que a separação do conteúdo do quadro de dados "*G1*" em colunas é produzida com o uso da função "**eachcol**" que permite decompor o conteúdo de um quadro de dados atribuindo os valores de cada coluna as variáveis apontadas a sua esquerda.

O assunto relacionado ao uso de quadros de dados é mais extenso do que aqui está sendo apresentado, fugindo do objetivo e escopo de introdução deste trabalho, sendo este tema, exclusivamente, merecedor de um livro a parte que trate desta operação relacionada a análise de dados.

5.9 - Conjuntos de dados "R"

O pacote de dados "**Rdatasets**" da linguagem Julia é baseado em pacote de mesmo nome desenvolvido para a linguagem R. A versão criada para Julia fornece aos seus usuários a possibilidade de terem acesso a maioria dos conjuntos de dados padrão da linguagem R. No momento de escrita deste livro o pacote "**Rdatasets**" conta com 34 bibliotecas, cada uma com um conjunto de 763 recursos de quadros de dados.

O objetivo deste pacote, tanto para Julia como para R, é fornecer um extenso conjunto de dados auxiliares para a realização de operações voltadas a análise de dados acessíveis ao desenvolvimento de software e ensino de estatística.

Como todo pacote, o pacote "**RDatasets**", deve ser instalado para estar disponível para uso, portanto, execute a instrução seguinte e aguarde sua conclusão.

```
julia> using RDatasets
```

Se for a primeira vez de sua execução será informado a possibilidade de instalação. Neste sentido, confirme "**y**" quando solicitado e aguarde o processo de instalação do pacote ser concluído.

Para fazer uso do pacote "**RDatasets**" forneça como argumentos da função "**dataset**" dois argumentos, sendo o primeiro a indicação da biblioteca de recursos e o segundo o recurso a ser usado. Os argumentos desta função devem ser definidos entre aspas inglesas.

```
RDatasets.dataset("biblioteca", "recurso")
dataset("biblioteca", "recurso")
```

Por exemplo, mostrar os dados de eficácia de inseticidas em spray sobre insetos:

```
julia> dataset("datasets", "InsectSprays")
72×2 DataFrame
 Row │ Count  Spray
     │ Int32  Cat…
─────┼──────────────
   1 │    10  A
   2 │     7  A
   3 │    20  A
   4 │    14  A
   5 │    14  A
   6 │    12  A
   7 │    10  A
   8 │    23  A
   9 │    17  A
  10 │    20  A
  11 │    14  A
  ⋮  │   ⋮      ⋮
  63 │    15  F
  64 │    22  F
  65 │    15  F
  66 │    16  F
  67 │    13  F
  68 │    10  F
  69 │    26  F
  70 │    26  F
  71 │    24  F
  72 │    13  F
     51 rows omitted
```

Caso deseje ver todos os 72 registros do quadro de dados de eficácia de inseticidas em spray sobre insetos execute a instrução:

```
julia> show(dataset("datasets", "InsectSprays"), allrows=true)
```

Para saber as bilbiotecas de quadros de dados padrão linguagem R estão prontas e disponibilizadas para a linguagem Julia junto ao pacote "**RDatasets**" execute a instrução:

```
julia> show(RDatasets.packages(), allrows=true, allcols=true)
```

```
julia> RDatasets.packages()
34×2 DataFrame
 Row │ Package       Title
     │ String15      String
─────┼──────────────────────────────────────────
   1 │ COUNT         Functions, data and code for cou…
   2 │ Ecdat         Data sets for econometrics
   3 │ HSAUR         A Handbook of Statistical Analys…
   4 │ HistData      Data sets from the history of st…
   5 │ ISLR          Data for An Introduction to Stat…
   6 │ KMsurv        Data sets from Klein and Moeschb…
   7 │ MASS          Support Functions and Datasets f…
   8 │ SASmixed      Data sets from "SAS System for M…
   9 │ Zelig         Everyone's Statistical Software
  10 │ adehabitatLT  Analysis of Animal Movements
  11 │ boot          Bootstrap Functions (Originally …
```

```
⋮ │      ⋮                        ⋮
25 │ pscl          Political Science Computational …
26 │ psych         Procedures for Psychological, Ps…
27 │ quantreg      Quantile Regression
28 │ reshape2      Flexibly Reshape Data: A Reboot …
29 │ robustbase    Basic Robust Statistics
30 │ rpart         Recursive Partitioning and Regre…
31 │ sandwich      Robust Covariance Matrix Estimat…
32 │ sem           Structural Equation Models
33 │ survival      Survival Analysis
34 │ vcd           Visualizing Categorical Data
                                    13 rows omitted
```

Para ver os 763 recursos disponível de cada biblioteca execute a instrução:

```
julia> show(RDatasets.datasets(), allrows=true, allcols=true)

julia> RDatasets.datasets()
```

Os dados do pacote "**Rdatasets**" podem ser usados para trabalhar os recursos apresentados neste obra. Por exemplo, veja a apresentação de uma lista de latitudes e longitudes de 39 pontos em 11 mapas antigos em sua forma padrão e classificada pela coluna "**Name**":

```
julia> RDatasets.dataset("HistData", "OldMaps")

julia> sort(RDatasets.dataset("HistData", "OldMaps"), [:Name])
```

CAPÍTULO 6

USO DE ARQUIVOS

Este capítulo apresenta o uso dos recursos de persistência no armazenamento de dados na forma de arquivos. São mostrados os usos de arquivos do tipo texto e arquivos do tipo binário com operações de acesso sequencial. O vínculo da linguagem Julia com gerenciamento de bancos de dados também é abordado, neste sentido com a integração e uso do Gerenciador de Banco de Dados SQLite.

6.1 - Tipos de arquivos e formas de acesso

Em capítulos anteriores, foram apresentados o uso de arranjos na forma de vetores e matrizes, além do uso das estruturas e quadros de dados, que de certa maneira formaram a base para o uso de arquivos. A grosso modo um arquivo pode ser considerado uma tabela de dados gravada em mídia (discos rígidos, fitas magnéticas, pen-drives entre outros), enquanto que os arranjos tabelas de dados manipulados na memória RAM (*Random Access Memory*). Para Backes (2014, p. 248), "um arquivo, de modo abstrato, nada mais é do que uma coleção de bytes armazenados em um dispositivo de armazenamento secundário".

Um arquivo texto se caracteriza por armazenar dados em blocos de oito bits na forma de bytes caracterizando a definição de caracteres, como letras, numerais, pontos e caracteres de controle normalmente dispostos em linhas (onde ao final de cada linha há um obrigatoriamente um caractere de controle de identificação de fim de linha), diferentemente dos arquivos binários que armazenam seus dados na forma de bits brutos e isolados (não armazenados na forma de linhas, mas na forma de uma fita contigua), podendo ser arbitrariamente configurado com conjuntos de bits de tamanhos diferentes ao formato byte.

Arquivos textos em programação são ideais para a criação de arquivos de transição de dados e configuração de programas ou sistemas. São arquivos sempre acessados para escrita e leitura de maneira sequencial, pois para seu processamento é necessário analisar os dados caractere a caractere até que seja encontrado o caractere de controle de fim de linha ou que seja encontrado o caractere de controle de fim de arquivo. Neste tipo de arquivo os dados armazenados são sempre traduzidos na forma de caracteres, ou seja, é um arquivo em que seu conteúdo é humanamente legível.

Arquivos binários são ideais para o armazenamento de dados dispostos de forma estruturada como o armazenamento de informações sobre empresas, pessoas, produtos, etc. São arquivos que para a escrita e leitura acessam seus dados de forma sequencial ou randômica, pois operam as operações de acesso por deslocamento de bits. Neste tipo de arquivo os dados armazenados são sempre considerados como mapas de bits, ou seja, não são convertidos como caracteres e sim como uma sequência de valores numéricos inteiros que exige que se tenha conhecimento prévio da estrutura, do mapeamento dos dados. De outra forma o conjunto de dados armazenado será indecifrável, principalmente, para humanos.

Os arquivos podem ser acessados, devendo para tanto serem abertos para o uso e fechados após o uso. Desta forma, é possível operar ações de leitura e/ou escrita na forma sequencial para arquivos textos e binários, direta ou indexada apenas para arquivos binários:

- Acesso sequencial - ocorre quando o processo de gravação e leitura é feito continuamente, dados ou conjunto de dados (registro) um após o outro. Para gravar um novo registro, é necessário percorrer todo o arquivo desde o pri-

meiro registro, um a um, até localizar a primeira posição vazia no arquivo, ou, então, posicionar o ponteiro de acesso do arquivo após o último registro. A leitura em um arquivo de acesso sequencial é processada sequencialmente. Se o registro a ser acessado for o último, primeiro será necessário ler todos os registros que o antecedem;

- Acesso direto - conhecido também como acesso randômico, é o tipo de acesso realizado diretamente em certo registro do arquivo. Para acessar um registro, deve-se, de antemão, saber a posição do arquivo em que o registro se encontra gravado. Essa tarefa será impossível para uma pessoa quando o arquivo tiver algumas dezenas de registros ou mais;
- Acesso indexado - ocorre quando se utiliza um arquivo auxiliar que mantém os registros de um arquivo base ordenados por meio de uma chave de classificação. Os acessos são feitos como em um livro. Primeiro se consulta o arquivo índice, que possui a chave de pesquisa, depois basta posicionar-se de forma direta no ponteiro do arquivo base. Esse tipo de acesso a um arquivo não é abordado nesta obra.

A principal vantagem no uso de arquivos é a capacidade dos dados ficarem armazenados para uso a qualquer momento. Outra vantagem é a capacidade de armazenarem um número maior de dados do que os arranjos em memória, sendo limitante desta armazenagem apenas o tamanho do meio físico utilizado para sua gravação. Os arquivos podem ser configurados como arquivos do tipo texto ou arquivos do tipo binário (AKITA, 2021).

6.2 - Acesso a arquivos e suas operações

O acesso ao uso de arquivos na linguagem Julia é realizado a partir do uso das funções "**open**" para abrir o arquivo e colocá-lo em operação e "**close**" para fechar um arquivo que esteja aberto. Todo o arquivo para uso, seja direcionado para a escrita de dados ou direcionado para a leitura de dados deve ser previamente aberto e deve sempre ser fechado se não for mais utilizado. A sintaxe para abertura (forma mais convencional) e fechamento de arquivos baseia-se nas formas:

```
<var_arq> = open("nome do arquivo", "tipo de ação")
close(<var_arq>)
```

Em que o indicativo "<*VAR_ARQ*>" representa a definição de uma variável de acesso a determinado arquivo físico que quando aberto com a função "**open**" possui a indicação do nome do arquivo a ser operado e do tipo de operação realizada sobre o arquivo. Para o fechamento do arquivo basta indicar junto a função "**close**" o nome da variável de acesso. A indicação "**nome do arquivo**" representa o nome do arquivo em disco, com sua extensão de identificação e se necessário a indicação do local onde o arquivo se encontra na estrutura de diretórios do sistema. A indicação do identificador do tipo de operação "**tipo de ação**" estabelece a forma de acesso ao arquivo a partir dos códigos indicados na tabela 6.1.

Tipo de acesso	Descrição	Operação
r	Põe arquivo em modo de leitura.	read
w	Põe arquivo em modo de escrita.	write, create, truncate
a	Põe arquivo em modo de acréscimo.	write, create, append
r+	Põe arquivo em modo de leitura/escrita.	read, write
w+	Põe arquivo em modo de escrita/leitura.	read, write, create, truncate
a+	Põe arquivo em modo de leitura/escrita.	read, write, create, append

Tabela 6.1 - Opções de acesso a arquivos.

Na tabela 6.1 há a indicação dos modos de operação que cada tipo de acesso ao ser definido permite realizar sobre o arquivo aberto em uso, sendo:

- append - operação em que o arquivo é colocado em modo de acréscimo de dados (modos: "a", "a+");
- create - operação em que o arquivo é criado, caso não exista (modos: "w", "w+", "a", "a+");
- read - operação em que no arquivo é submetido a operação de leitura de dados (modos: "r", "r+", "w+", "a+");
- truncate - operação em que o arquivo é sempre criado, mesmo que já exista (modos: "w", "w+");
- write - operação em que o arquivo é submetido a operação de escrita de dados (modos: "w", "w+", "a", "a+", "r+").

Além das funções de abertura e fechamento de arquivos há um conjunto de funções que auxiliam diversas operações sobre essa estrutura de armazenamento de dados, das quais as principais são indicadas junto a tabela 6.2, mas não necessariamente demonstradas neste livro.

Função	Descrição
countlines()	Lê arquivo e conta o número de linhas existentes.
eachline()	Leitura linha a linha de um arquivo "auxiliado por iteração".
eof()	Verifica se o fim de arquivo foi atingido.
filesize()	Mostra a quantidade de bytes ocupados pelo arquivo.
flush()	Efetiva manualmente a gravação de dados em arquivo.
ismarked()	Verifica se certa posição está marcada com "mark".
isopen()	Verifica se arquivo está aberto.
mark()	Adiciona marca na posição atual do arquivo e retorne a posição marcada.
peek()	Leitura de dados sem movimentar ponteiro do arquivo.
position()	Obtém a posição atual de um ponteiro de arquivo.
print()/println()	Escrita de dados em modo texto.
read!()	Leitura de dados em modo binário direcionado a um arranjo.
read()	Leitura de dados em modo texto/binário.
readline()	Leitura de uma única linha de texto.
readlines()	Leitura de várias linhas de texto.
reset()	Redefine posição marcada e remove a marca.
seek()	Posiciona o ponteiro de arquivo em certa posição.
seekend()	Posiciona ponteiro no final do arquivo.
seekstart()	Posiciona ponteiro no início do arquivo.
skip()	Salta relativamente a partir da posição de ponteiro atual.
unmark()	Remove marca na posição atual do arquivo.
write()	Escrita de dados em modo binário.

Tabela 6.2 - Funções de acesso a arquivos.

Em relação ao uso da função "**open**" é importante salientar que a função pode ser usada com outra variação de sintaxe, onde se estabelece a abertura do arquivo e a definição de certa operação:

```
open("nome do arquivo", "tipo de ação") do <var_arq>
  <operação sobre o arquivo>
end
```

A escolha da forma de abertura de arquivos é basicamente de foro íntimo do profissional de programação e apesar de ambas as formas de abertura serem aqui demonstradas, apenas a primeira será usada com maior intensidade neste trabalho.

As operações de acesso a arquivos são plenamente aplicáveis no modo interativo REPL da linguagem, como forma outras tantas operações já apresentadas, mas é oportuno salientar que esse tipo de ação é melhor aplicado quando se está operando a linguagem em modo programado.

Para organizar as ações deste capítulo sugere-se criar no diretório ou pasta de trabalho do usuário, do sistema operacional em uso, um local específico para o armazenamento dos arquivos operacionalizados. Assim sendo, considere a criação de um local chamado "**ARQJULIA**" a partir de seu diretório padrão de trabalho onde os arquivos exemplificados neste capítulo serão criados e operacionalizados.

6.3 - Arquivos do tipo texto

Já é sabido que um arquivo do tipo texto se caracteriza por ser um arquivo de acesso sequencial que opera dados na forma de bytes (conjuntos de 8 bits), tanto nas ações de escrita quanto nas ações de leitura. Na linguagem Julia um arquivo é identificado no formato texto quando estiver em uso as funções "**print**" ou "**println**".

Como exemplo inicial considere a criação de um arquivo texto chamado "*mensagem1.txt*" que armazene o conteúdo da mensagem "**Alô, mundo!**". O arquivo deverá ser criado e imediatamente ter a mensagem gravada de forma direta.

```
julia> open("ARQJULIA//mensagem1.txt", "w") do meuArq
           println(meuArq, "Alô, mundo!")
       end
```

Observe a definição do modo de operação com o *status* de escrita "**w**" para o arquivo "*mensagem1.txt*", em que o nome do arquivo e sua extensão de identificação são meras indicações, podendo-se usar qualquer nome e qualquer extensão desejada. O modo "**w**" põe o arquivo em modo de escrita, se a instrução anterior for reexecutada um novo arquivo com mesmo nome é recriado sobre o primeiro.

A leitura do conteúdo do arquivo "*mensagem1.txt*" pode ser efetivada com o uso de algum programa de edição de textos ou mesmo a partir da própria função "**open**" a partir da sintaxe:

```
julia> open(linha -> read(linha, String), "ARQJULIA//mensagem1.txt")
"Alô, mundo!\n"
```

A indicação do símbolo "**\n**" junto a mensagem refere-se ao caractere de fim de linha inserido no momento da gravação com o uso da função "**println**".

Veja que a função "**open**" aceita antes da definição do nome do arquivo a indicação de uso de uma função. Neste caso, está sendo usada uma função anônima que sobre a variável "*LINHA*" aplica a leitura da cadeia de caracteres (*String*) do arquivo por meio da função "**read**". No entanto a leitura pode ser operacionaliza de uma maneira semelhante à forma em que a gravação da mensagem foi realizada. Veja a próxima ocorrência:

```
julia> open("ARQJULIA//mensagem1.txt", "r") do meuArq
           read(meuArq, String)
       end
"Alô, mundo!\n"
```

Atente para o uso do modo de leitura "**r**" indicado junto a função "**open**". Este modo de operação coloca o arquivo pronto para realizar operações de leitura. Um arquivo pode por meio da função "**open**" ser comutado entre os modos de escrita "**w**" e leitura "**r**" a qualquer momento. É pertinente salientar que quando um arquivo é operacionalizado de maneira direta, como apresentado, o mesmo é fechado automaticamente.

No próximo exemplo as operações de escrita e leitura são realizadas fora do escopo da função "**open**" seguindo um estilo mais tradicional em linguagens de programação e assim é ideal que um arquivo aberto seja após o uso fechado. Assim sendo, considere a criação de um arquivo texto chamado "*mensagem2.txt*" que armazene o conteúdo da mensagem "**Linguagem Julia**" e seja em seguida fechado, a partir das instruções:

```
julia> meuArq = open("ARQJULIA//mensagem2.txt", "w")
IOStream(<file ARQJULIA//mensagem2.txt>)

julia> println(meuArq, "Linguagem Julia")

julia> close(meuArq)
```

A partir dessas últimas três instruções é possível ver o conteúdo do arquivo "*mensagem2.txt*" devidamente gravado. Agora para ler o conteúdo do arquivo execute as seguintes instruções:

```
julia> meuArq = open("ARQJULIA//mensagem2.txt", "r")
IOStream(<file ARQJULIA//mensagem2.txt>)

julia> read(meuArq, String)
"Linguagem Julia\n"

julia> close(meuArq)
```

Perceba que os arquivos definidos somente aceitam a gravação de uma mensagem por vez e que isso pode não ser muito prático. Assim sendo, considere a criação de um arquivo texto chamado "*mensagem3.txt*" que armazene, por exemplo, nomes de linguagens de programação. Para tanto, execute as seguintes instruções:

```
julia> meuArq = open("ARQJULIA//mensagem3.txt", "a")
IOStream(<file ARQJULIA//mensagem2.txt>)

julia> println(meuArq, "FORTRAN")

julia> println(meuArq, "LISP")

julia> println(meuArq, "COBOL")

julia> println(meuArq, "C++")

julia> println(meuArq, "D")

julia> close(meuArq)
```

As ações anteriores permitiram ir acrescentando conteúdo ao arquivo. Para ver a apresentação deste resultado execute as seguintes instruções?

```
julia> meuArq = open("ARQJULIA//mensagem3.txt", "r")
IOStream(<file ARQJULIA//mensagem2.txt>)

julia> read(meuArq, String)
"FORTRAN\nLISP\nCOBOL\nC++\nD\n"

julia> close(meuArq)
```

Veja que a gravação dos nomes das linguagens de programação ocorre sequencialmente um após o outro formando assim uma única sequência de caracteres contento as marcações de fim de linha para cada palavra cadastrada.

A apresentação indicada não é muito confortável, sendo que o ideal é a apresentação das palavras em linhas separadas. Para que isso seja possível execute as próximas instruções:

```
julia> meuArq = open("ARQJULIA//mensagem3.txt", "r")
IOStream(<file ARQJULIA//mensagem3.txt>)

julia> while !eof(meuArq)
           println(readline(meuArq))
       end
FORTRAN
LISP
COBOL
C++
D
```

No trecho anterior é usada a função "**eof**" que identifica o fim de arquivo. Veja que a instrução indica que enquanto não for encontrado o fim de arquivo é para proceder a leitura de cada linha com o uso da função "**readline**". Neste instante se o trecho anterior for novamente aplicado não acontecerá a apresentação das palavras.

```
julia> while !eof(meuArq)
           println(readline(meuArq))
       end
```

Isso ocorre pelo fato de que quando um arquivo é aberto é colocado o ponteiro de acesso de dados no arquivo no início do arquivo, à medida que o arquivo é lido este ponteiro é posicionado no próximo dado até chegar na última posição. Desta forma, para ver novamente o conteúdo do arquivo execute primeiramente a seguinte instrução:

```
julia> seekstart(meuArq)
IOStream(<file ARQJULIA//mensagem3.txt>)
```

Na sequência execute a instrução:

```
julia> while !eof(meuArq)
             println(readline(meuArq))
       end
FORTRAN
LISP
COBOL
C++
D
```

Agora veja outra maneira de realizar a leitura dos dados de um arquivo:

```
julia> seekstart(meuArq)
IOStream(<file ARQJULIA//mensagem3.txt>)

julia> for linha in eachline(meuArq)
             println(linha)
       end
FORTRAN
LISP
COBOL
C++
D
```

Nesta segunda versão de leitura usa-se a função "**eachline**" que tem por finalidade fazer a captura de cada linha a cada iteração realizada pelo comando "**for**" até que indiretamente o fim de arquivo seja alcançado.

Além dessas formas há a possibilidade de fazer a leitura dos dados existentes em um arquivo e apresentá-los na forma de um vetor. Para tanto, execute as seguintes instruções:

```
julia> seekstart(meuArq)
IOStream(<file ARQJULIA//mensagem3.txt>)

julia> readlines(meuArq)
5-element Vector{String}:
"FORTRAN"
 "LISP"
 "COBOL"
 "C++"
 "D"
```

Para apresentar a quantidade de dados gravados em um arquivo texto execute as seguintes instruções:

```
julia> seekstart(meuArq)
IOStream(<file ARQJULIA//mensagem3.txt>)

julia> countlines(meuArq)
5

julia> close(meuArq)
```

A função "**countlines**" faz a leitura do arquivo, desde que o ponteiro de acesso esteja na primeira posição, e conta a quantidade de registros de dados existentes.

Há um detalhe que ocorre quando um arquivo texto é aberto para escrita em modo *append* os dados cadastrados não são diretamente gravados no arquivo. Esses dados ficam em memória aguardando a execução da função "**close**" que descarrega os dados no arquivo e em seguida faz seu fechamento. Mas imagine que ficar abrindo e fechando arquivos não seja uma opção.

Para demonstrar um arquivo que grava conteúdo na medida em que o conteúdo é fornecido considere a criação de um arquivo texto chamado "*mensagem4.txt*" que armazene nomes de países.

```
julia> meuArq = open("ARQJULIA//mensagem4.txt", "a")
IOStream(<file ARQJULIA//mensagem4.txt>)

julia> println(meuArq, "Brasil")

julia> println(meuArq, "Estados Unidos da América")
```

Neste momento se houver abertura do arquivo em um editor de textos você verá que o arquivo está vazio, a despeito dos dois dados anteriormente solicitados para gravação. Neste caso, para que os dados sejam descarregados no arquivo sem que o mesmo seja fechado execute a seguinte instrução:

```
julia> flush(meuArq)
```

A função "**flush**" descarrega o conteúdo de memória no arquivo em uso.

Agora abra o arquivo em um programa de edição de textos e você verá o conteúdo armazenado com o arquivo ainda aberto em memória. Na sequência execute as instruções seguintes e veja a cada operação de "**println**" e "**flush**" seus efeitos de gravação no arquivo em uso via programa editor de texto.

```
julia> println(meuArq, "França")

julia> flush(meuArq)

julia> println(meuArq, "Itália")

julia> flush(meuArq)

julia> close(meuArq)
```

Essencialmente essas são as operações básicas executadas sobre arquivos de padrão texto. Cabe ainda apresentar alguns recursos adicionais de controle de operações sobre arquivos, como por exemplo verificar o que ocorre quando se tenta abrir para leitura um arquivo inexistente. Veja a tentativa de abertura para leitura de um arquivo não existente chamado "*mensagem5.txt*":

```
julia> meuArq = open("ARQJULIA//mensagem5.txt", "r")
ERROR: SystemError: opening file "ARQJULIA//mensagem5.txt": No such file or directory
Stacktrace:
 [1] systemerror(p::String, errno::Int32; extrainfo::Nothing)
   @ Base .\error.jl:174
 [2] #systemerror#68
   @ .\error.jl:173 [inlined]
 [3] systemerror
   @ .\error.jl:173 [inlined]
 [4] open(fname::String; lock::Bool, read::Bool, write::Nothing, create::Nothing, trun
     cate::Nothing, append::Nothing)
   @ Base .\iostream.jl:293
 [5] open(fname::String, mode::String; lock::Bool)
   @ Base .\iostream.jl:355
 [6] open(fname::String, mode::String)
   @ Base .\iostream.jl:355
 [7] top-level scope
   @ REPL[98]:1
```

Para evitar este tipo de interrupção e detecção adequada da falha execute a seguinte instrução:

```
julia> try
          meuArq = open("ARQJULIA//mensagem5.txt", "r")
       catch
          println("Arquivo não existente")
       end
Arquivo não existente
```

Ou ainda pode-se fazer o tratamento de uso do arquivo de outra forma, não existindo o arquivo que este seja então criado. Veja a instrução seguinte:

```
julia> try
          meuArq = open("ARQJULIA//mensagem5.txt", "r")
       catch
          meuArq = open("ARQJULIA//mensagem5.txt", "a")
       end
IOStream(<file ARQJULIA//mensagem5.txt>)
```

Neste caso, se o arquivo "*mensagem5.txt*" não existir para a leitura este será criado pronto para escrita.

6.4 - Arquivos do tipo binário

Um arquivo do tipo binário, como apresentado, é uma estrutura de armazenamento de dados que se caracteriza por fazer a guarda de seus conteúdos na forma de sequências de bits e não na forma de um grupo de bits como os bytes nos arquivos textos. Na linguagem Julia um arquivo é considerado binário quando estiver em uso a função de gravação de dados "**write**".

Como exemplo inicial considere a criação de um arquivo binário chamado "*dados1.bin*" que armazene os cinco primeiros valores pares inteiros representados no vetor **[0, 2, 4, 6, 8]**. Esta operação é muito similar a operação de criação de arquivo texto. Para tanto, siga as seguintes instruções:

```
julia> meuArq = open("ARQJULIA//dados1.bin", "w")
OStream(<file ARQJULIA//dados1.bin>)
```

```
Julia> write(meuArq, [0, 2, 4, 6, 8])
40

julia> close(meuArq)
```

Quando se opera arquivos binários a gravação de conteúdo deve ser realizada tão somente com a função "**write**" que além de gravar o conteúdo apresenta a quantidade de bytes que foram gravados no arquivo. As funções "**print**" e "**println**" não surtem efeito para este tipo de arquivo.

A partir da criação de um arquivo, neste caso binário, é possível ver seu conteúdo. No entanto, esta operação configura-se um pouco diferente da ação executada com arquivo texto. Para tanto, execute as seguintes instruções:

```
julia> meuArq = open("ARQJULIA//dados1.bin", "r")
OStream(<file ARQJULIA//dados1.bin>)

julia> conteúdo = Array{Int64}(undef, 5)  # definição de espaço para recepção de valores
5-element Vector{Int64}:
 0
 0
 0
 0
 0
julia> read!(meuArq, conteúdo)
5-element Vector{Int64}:
 0
 2
 4
 6
 8

julia> close(meuArq)

julia> conteúdo
5-element Vector{Int64}:
 0
 2
 4
 6
 8
```

Neste momento se você tentar abrir o arquivo "*dados1.bin*" em um programa de edição de texto convencional não verá os valores armazenados. Em vez disso, verá a apresentação de caracteres aparentemente desconexos, uma vez que os dados do arquivo não estão agrupados em conjuntos de bytes e sim em sequências de bits.

Um detalhe importante para a leitura de um arquivo binário é ter em memória um local para a recepção dos dados. Daí o uso da instrução "**conteúdo = Array{Int64}(undef, 5)**" que permite criar junto a variável "*CONTEÚDO*" um repositório com espaço para a recepção de "**5**" valores definidos inicialmente como "**undef**" (desconhecidos) do tipo usado na gravação, ou seja, "**Int64**".

Veja que para fazer a recepção dos valores é importante ter em mãos a quantidade de dados que será lida. Se fosse a operação com arquivo do tipo texto, bastaria buscar o arquivo enquanto não encontrado o fim de arquivo com o uso da função "**eof**" sem se preocupar com a quantidade efetiva de dados no arquivo. Mas em se tratando de arquivo binário as coisas são diferentes. No entanto, nem sempre é possível saber de antemão a quantidade de dados de um arquivo. Assim sendo, veja no próximo exemplo o tratamento de um arquivo com tamanho desconhecido.

Considere a criação de um arquivo binário em modo de acréscimo chamado "*dados2.bin*" que tenha a capacidade de armazenar valores numéricos de ponto flutuante. Para tanto, siga as seguintes instruções:

```
julia> meuArq = open("ARQJULIA//dados2.bin", "a")
IOStream(<file ARQJULIA//dados2.bin>)

julia> write(meuArq, 1.2)
8

julia> write(meuArq, 5.3)
8

julia> write(meuArq, 7.9)
8

julia> close(meuArq)
```

As ações anteriores permitiram ir acrescentando conteúdo ao arquivo de modo que fica difícil após certo tempo saber a quantidade de elementos. A cada execução da função "**write**" vê-se a indicação do valor "**8**" que se refere a quantidade de bytes usados na gravação, ou seja, são usados oito bytes para cada dado de ponto flutuante de 64 bits. Observe que para o armazenamento de cada valor numérico de ponto flutuante o sistema utiliza oito bytes.

Para ver a apresentação dos dados armazenados no arquivo "*dados2.bin*" execute as seguintes instruções:

```
julia> meuArq = open("ARQJULIA//dados2.bin", "r")
IOStream(<file ARQJULIA//dados2.bin>)

julia> conteúdo = Array{Float64}(undef, trunc(Int, filesize(meuArq) / 8))
3-element Vector{Float64}:
 0.0
 0.0
 0.0

julia> read!(meuArq, conteúdo)
3-element Vector{Float64}:
 1.2
 5.3
 7.9

julia> close(meuArq)
```

Na instrução "**conteúdo = Array{Float64}(undef, trunc(Int, filesize(meuArq) / 8))**" ocorre a partir do segundo argumento do construtor "**Array{Float64}**" o cálculo do tamanho da quantidade de dados (registros)existentes no arquivo. Para obter-se a quantidade de dados de um arquivo é necessário obter primeiro o tamanho do arquivo em bytes, o que é feito com a função "**filesize**", e posteriormente fazer a divisão deste valor pela quantidade de bytes de cada registro. A função "**trunc**" é usada para garantir que o valor da quantidade de registros seja um valor inteiro.

Veja as próximas instruções com demonstrações de operações de posicionamento do ponteiro de arquivo aplicadas sobre arquivos binários que não puderam ser vistas com arquivos textos:

```
julia> meuArq = open("ARQJULIA//dados2.bin", "r")
IOStream(<file ARQJULIA//dados2.bin>)

julia> position(meuArq)  # posição atual do ponteiro no arquivo
0
```

```
julia> seekend(meuArq)  # posiciona ponteiro no final do arquivo
IOStream(<file ARQJULIA//dados2.bin>)

julia> position(meuArq)  # posição atual do ponteiro no arquivo após "seekend"
24

julia> seekstart(meuArq)  # posiciona ponteiro no início do arquivo
IOStream(<file ARQJULIA//dados2.bin>)

julia> peek(meuArq, Float64)  # Lê dado da posição atual sem avançar para o próximo
1.2

julia> peek(meuArq, Float64)
1.2

julia> seek(meuArq, 8)  # posiciona ponteiro no segundo registro
IOStream(<file ARQJULIA//dados2.bin>)

julia> peek(meuArq, Float64)
5.3

julia> seek(meuArq, 16)  # posiciona ponteiro no terceiro registro
IOStream(<file ARQJULIA//dados2.bin>)

julia> peek(meuArq, Float64)
7.9

julia> seekstart(meuArq)
IOStream(<file ARQJULIA//dados2.bin>)

julia> seek(meuArq, 0)  # posiciona ponteiro no primeiro registro
IOStream(<file ARQJULIA//dados2.bin>)

julia> peek(meuArq, Float64)
1.2
```

Neste trecho de instruções atente para todas as operações de posicionamento indicadas, mas exclusivamente tenha atenção no uso da função "**seek**". Veja que para fazer o posicionamento nos registros de primeira, segunda e terceira posições foram usados respectivamente os valores "**0**", "**8**" e "**16**", sendo que: o primeiro dado ocupa as posições de "**0**" a "**7**" do arquivo; o segundo dado ocupa as posições de "**8**" a "**15**" do arquivo e o terceiro dado ocupa as posições de "**16**" a "**24**" do arquivo. Note que a distância em cada faixa de valores é "**8**", a qual se refere a quantidade de bytes ocupado por um valor numérico de ponto flutuante do tipo "**Float64**". Fazer este cálculo pode ser um pouco chato e para facilitar esta operação pode-se fazer uso das seguintes estratégias:

```
julia> seek(meuArq, (1 - 1) * 8)  # para posicionar no primeiro registro
IOStream(<file ARQJULIA//dados2.bin>)

julia> peek(meuArq, Float64)
1.2

julia> seek(meuArq, (3 - 1) * 8)  # para posicionar no terceiro registro
IOStream(<file ARQJULIA//dados2.bin>)

julia> peek(meuArq, Float64)
7.9
```

```
julia> seek(meuArq, (2 - 1) * 8)  # para posicionar no segundo registro
IOStream(<file ARQJULIA//dados2.bin>)

julia> peek(meuArq, Float64)
5.3
```

A técnica de posicionamento sugerida é calcular a diferente entre o posicionamento cardinal do arquivo binário e a definição ordinal que se deseja ver no arquivo. Por isso usa-se a forma "**(n - 1) * 8**", onde o indicativo "**n**" refere-se a definição ordinal de posicionamento.

As ações de posicionamento podem ainda ser efetuadas com a aplicação da função "**skip**" que efetua o posicionamento de forma relativa a partir da posição em que o ponteiro esteja no arquivo. Esta função atua de maneira diferente ao uso das funções "**seek/peek**" que efetuam o posicionamento de forma absoluta, pois não importa onde o ponteiro esteja posicionado.

A função "**skip**" quando efetua seu posicionamento necessita do apoio auxiliar de uma função de leitura como a função "**read**". Isso faz com que o ponteiro de posicionamento do arquivo não seja mantido na posição colocada inicialmente pela função "**skip**". Observe as próximas instruções:

```
julia> seekstart(meuArq)  # põe ponteiro do arquivo na posição "0"
IOStream(<file ARQJULIA//dados2.bin>)

julia> skip(meuArq, 8)  # posiciona ponteiro no segundo registro a partir da posição "0"
IOStream(<file ARQJULIA//dados2.bin>)

julia> read(meuArq, Float64)  # após leitura ponteiro é posicionado na posição "16"
5.3

julia> skip(meuArq, -16)  # posiciona ponteiro no primeiro registro a partir da posição "16"
IOStream(<file ARQJULIA//dados2.bin>)

julia> read(meuArq, Float64)
1.2

julia> close(meuArq)
```

Perceba que para fazer uso da função "**skip**" há a necessidade de um nível de atenção maior se comparada com as funções "**seek/peek**".

A definição de arquivos binários é adequada para uso de estruturas de dados ao estilo classes. Assim sendo, considere a definição do arquivo "*dados3.bin*" que opere com uma pequena estrutura de dados contendo nome e idade como mostram as próximas instruções:

```
julia> mutable struct Dados
           nome::String
           idade::Int8
       end

julia> cadastro = [
                   Dados("José", 45),
                   Dados("Maria", 33)
                  ]
2-element Vector{Dados}:
 Dados("José", 45)
 Dados("Maria", 33)
```

```
julia> meuArq = open("ARQJULIA//dados3.bin", "w")
OStream(<file ARQJULIA//dados3.bin>)

Julia> for i in cadastro
           write(meuArq, Ref(i))
       end

julia> close(meuArq)
```

Para efetuar a leitura do arquivo "*dados3.bin*" execute as seguintes instruções:

```
julia> meuArq = open("ARQJULIA//dados3.bin", "r")
OStream(<file ARQJULIA//dados3.bin>)

julia> pessoa = Dados("", 0)  # definição de um container de recepção de dados
Dados("", 0)

julia> pes1 = read!(meuArq, Ref(pessoa))
Base.RefValue{Dados}(Dados("José", 45))

julia> pes2 = read!(meuArq, Ref(pessoa))
Base.RefValue{Dados}(Dados("Maria", 33))
```

O recurso "**Ref**" é usado para definir em memória um estado operacional de referência a certo conteúdo. Desta forma o acesso aos elementos que compõem a estrutura pode ser efetuado com segurança e sem a ocorrência de erro. Veja que para ter espaço adequado para o armazenamento dos dados lidos do arquivo define-se uma área de memória, um *coitainer* a partir da execução da instrução "**pessoa = Dados("", 0)**" que é referenciado para a ação da função "**read!**".

6.5 - Diretórios, caminhos e nomes de arquivos

O ambiente interativo Julia possui um conjunto de funcionalidades que auxiliam as tarefas de gerenciamento no uso de arquivos. São funcionalidade, basicamente, voltadas ao uso de arquivos no modo REPL que auxiliam operações que normalmente são executadas fora do ambiente de uma linguagem de programação, ou seja, operações executadas no *prompt* dos sistemas operacionais.

Os arquivos e programas em execução em um computador possuem como padrão de armazenamento e uso um diretório de trabalho. Por exemplo, se você deseja abrir um arquivo e lê-lo, Julia procura este arquivo no diretório atual ou a partir do diretório configurado como diretório de trabalho do usuário (LAUWENS & DOWNEY, 2019). Para saber qual é o diretório padrão de trabalho execute a instrução seguinte:

```
julia> pwd()  # mostra diretório padrão de trabalho da linguagem (Windows)
"C:\\Program Files\\julia-1.7.3\\bin"

julia> pwd()  # mostra diretório de trabalho do usuário (Windows)
"C:\\Users\\<nome do usuário ativo>"

julia> pwd()  # mostra diretório de trabalho do usuário (Linux)
"/home/<nome do usuário ativo>"

julia> pwd()  # mostra diretório de trabalho do usuário (macOS)
"/Users/<nome do usuário ativo>"
```

A função "**pwd**" tem por finalidade retornar o nome do diretório padrão inicial de operação da linguagem no sistema, ou seja, mostra o caminho no sistema usado pela linguagem para realizar prioritariamente suas operações. Nos sistemas operacionais POSIX (Linux e macOS, por exemplo) o ambiente Julia identifica automaticamente a aérea de trabalho do usuário. Já no sistema operacional Windows a apresentação do nome do diretório padrão de trabalho depende de configuração prévia realizada. Se você seguiu as orientações apresentadas no primeiro capítulo, deste livro, estará vendo no Windows a informação sinalizada como "*mostra diretório de trabalho do usuário (Windows)*", caso contrário estará vendo provavelmente a indicação sinalizada como "*mostra diretório padrão de trabalho da linguagem (Windows)*".

Nas operações com arquivos anteriormente operacionalizadas foi usado o diretório de trabalho "**ARQJULIA**" solicitado para ser definido dentro do diretório de trabalho do usuário. Para acessar este diretório de dentro do ambiente Julia execute a instrução:

```
julia> cd("ARQJULIA")
```

A partir deste instante o ambiente Julia está direcionado para dentro do diretório "**ARQJULIA**". A função "**cd**" tem a finalidade de mudar o diretório de uso. Para ver que o acesso ao diretório "**ARQJULIA**" está definido execute a instrução:

```
julia> pwd()  # resultado no Windows
"C:\\Users\\<nome do usuário ativo>\\ARQJULIA"

julia> pwd()  # resultado no Linux
"/home/<nome do usuário ativo>/ARQJULIA"

julia> pwd()  # resultado no macOS
"/Users/<nome do usuário ativo>/ARQJULIA"
```

Para ver o conteúdo do diretório ativo use a função "**readdir**" como indicado na instrução seguinte:

```
julia> readdir()
8-element Vector{String}:
 "dados1.bin"
 "dados2.bin"
 "dados3.bin"
 "mensagem1.txt"
 "mensagem2.txt"
 "mensagem3.txt"
 "mensagem4.txt"
 "mensagem5.txt"
```

Assim como certo diretório pode ser acessado com uso da função "**cd**" pode também se fazer o retorno a um ponto anterior a chamada. Por exemplo, execute para voltar ao diretório de trabalho do usuário a instrução.

```
julia> cd("..")  # volta um nível de diretório acima

julia> cd("ARQJULIA")
```

Além da possibilidade de se fazer a movimentação entre diretórios é possível a partir do ambiente Julia criar diretórios com a função "**mkpath**" e remover diretórios com a função "**rm**". Veja as próximas instruções:

```
julia> mkpath("TSTJULIA")  # cria diretório no diretório de trabalho do usuário

julia> rm("TSTJULIA")  # apaga diretório no diretório de trabalho do usuário
```

Há uma maneira de saber qual é o diretório de trabalho do usuário do sistema. Para tanto, basta fazer uso da função "**homedir**". Veja a próxima instrução:

```
julia> homedir()  # resultado no Windows
"C:\\Users\\<nome do usuário ativo>"

julia> homedir()  # resultado no Linux
"/home/<nome do usuário ativo>"

julia> homedir()  # resultado no macOS
"/Users/<nome do usuário ativo>"
```

Dentro do conjunto de funcionalidades existem algumas direcionadas a verificação lógica sobre caminhos, diretórios e arquivos, destacando-se as funções "**ispath**" que valida verdadeiro se objeto passado for um arquivo ou um diretório, "**isdir**" que valida verdadeiro se objeto passado for um diretório e "**isfile**" que valida verdadeiro se objeto passado for um arquivo. Veja seus exemplos de uso, considerando que esteja dentro do diretório "**ARQJULIA**":

```
julia> ispath(homedir())  # validando diretório ou arquivo
true

julia> ispath("dados1.bin")  # validando arquivo ou diretório
true

julia> isdir(homedir())  # validando somente diretório
true

julia> isdir("dados1.bin")
false

julia> isfile(homedir())
false

julia> isfile("dados1.bin")  # validando somente arquivo
true
```

Outro recurso na manipulação de caminhos, diretórios e arquivos é a possibilidade de obter-se o nome do caminho de acesso a certo diretório com a função "**dirname**" e obter-se o nome do diretório de operação desconsiderando seu caminho a partir do uso da função "**basename**". Veja os próximos exemplos:

```
julia> dirname(homedir())  # saída no Windows
"C:\\Users"

julia> dirname(homedir())  # saída no Linux
"/home"

julia> dirname(homedir())  # saída no macOS
"/Users"

julia> basename(homedir())  # apresentação similar em todos os sistemas operacionais
"<nome do usuário ativo>"
```

Há ainda a possibilidade de se descobrir qual é o caminho absoluto de localização no sistema de certo arquivo utilizando-se a função "**abspath**". Veja a próxima instrução:

```
julia> abspath("dados1.bin")  # saída Windows
"C:\\Users\\<nome do usuário ativo>\\ARQJULIA\\dados.bin"

julia> abspath("dados1.bin")  # saída Linux
"/home/<nome do usuário ativo>/ARQJULIA/dados.bin"

julia> abspath("dados1.bin")  # saída macOS
"/Users/<nome do usuário ativo>/ARQJULIA/dados.bin"
```

Entre o conjunto de funcionalidades de gerenciamento de caminhos, diretórios e arquivos há uma função chamada "**touch**" que permite criar um arquivo. Observe a instrução a seguir:

```
julia> touch("dados4.bin")
"dados4.bin"

julia> readdir()
9-element Vector{String}:
 "dados1.bin"
 "dados2.bin"
 "dados3.bin"
 "dados4.bin"
 "mensagem1.txt"
 "mensagem2.txt"
 "mensagem3.txt"
 "mensagem4.txt"
 "mensagem5.txt"
```

Todas essas e outras funções existentes na linguagem para ações de gerenciamento de caminhos, diretórios e arquivos podem ser operacionalizadas em arquivos de códigos de programas, dando a linguagem Julia grande poder de processamento e manipulação de dados.

6.6 - Gerenciamento com banco de dados

Além das operações que a linguagem possui para uso e gerenciamento de arquivos dos tipos texto e binário nas atividades de permanência de dados, possui também recursos que auxiliam sua operação com o uso de sistemas gerenciadores de bancos de dados.

Um banco de dados é, grosso modo, uma estrutura de arquivo organizado em tabelas usadas para armazenar dados. A vantagem em usar um sistema de gerenciamento de dados integrado a linguagem é sua facilidade aplicada as operações de permanência desses dados. A linguagem Julia possui recursos de acesso a alguns sistemas de gerenciamento de bancos de dados, como: MySQL, PostgreSQL, Mongo, SQLite, além dos pacotes de acesso as interfaces das APIs de acesso ODBC e JDBC.

Cada recurso de acesso possui uma característica particulariza o que leva a necessidade de estudar cada uma em separado. A título de demonstração este tópico apresenta o uso do gerenciador de banco de dados SQLite. O SQLite é uma ferramenta simples, leve e eficiência no gerenciamento e persistência de dados, a qual deverá estar devidamente instalada e configurada em seu sistema; mais detalhes sobre esta ferramenta consulte o apêndice desta obra.

Para fazer uso em Julia do gerenciador de banco de dados SQLite é necessário instalar um pacote apropriado a esta operação, a partir da versão "**3**" e que você tenha noção sobre o uso da linguagem SQL, consulte o apêndice "**A**" desta obra. Assim sendo, execute a seguinte instrução:

```
julia> import Pkg; Pkg.add("SQLite")  # caso já não o tenha instalado

julia> using SQLite
```

Se solicitada confirmação de instalação acione "**y**" e aguarde o término da ocorrência que não deverá ser demorada para disponibilizar o uso do pacote "**SQLite**".

Em seguida coloque o ambiente Julia apontado para o diretório "**ARQJULIA**" com a instrução seguinte:

```
julia> cd("ARQJULIA")
```

A partir deste ponto com o modo de operação SQLite ativado e o posicionamento no diretório "**ARQJULIA**" definido as demonstrações de uso da linguagem Julia com um gerenciador de banco de dados já podem ser realizadas. A primeira tarefa é criar o banco de dados que será usado. Para tanto, execute as seguintes instruções:

```
julia> db = SQLite.DB("teste.dbj")
SQLite.DB("teste.dbj")

julia> readdir()
10-element Vector{String}:
 "dados1.bin"
 "dados2.bin"
 "dados3.bin"
 "dados4.bin"
 "mensagem1.txt"
 "mensagem2.txt"
 "mensagem3.txt"
 "mensagem4.txt"
 "mensagem5.txt"
 "teste.dbj"
```

Observe que o banco de dados "**teste.dbj**" foi devidamente criado a partir do uso do construtor "**SQLite.DB**" tendo seu acesso atribuído a variável "*DB*" (neste caso, *Data Base*, sendo apenas a definição de um nome qualquer). O construtor "**SQLite.DB**" tem por finalidade efetuar uma de duas tarefas: criar um banco de dados quando este não existe ou abrir o banco de dados caso este exista. Este construtor define um ponto de conexão da linguagem Julia com o banco de dados *SQLite*, desde que esteja em operação, no mínimo, a versão 3 do *SQLite*. Um arquivo de banco de dados, como "**teste.dbj**", é formado por uma ou mais tabelas. Quando se opera mais de uma tabela em um banco de dados tem-se normalmente veículos de relacionamentos estabelecidos sobre elas. Para ver as tabelas de um banco de dados execute a instrução:

```
julia> SQLite.tables(db)
SQLite.DBTable[]
```

A resposta a operação da função "**tables**" do pacote "**SQLite**" é a indicação de uma lista contendo as tabelas definidas no banco de dados. Neste caso, em particular, nada foi apresentado por não existir nenhuma tabela dentro do banco de dados definida neste momento. Para realizar operações de acesso em SQL junto ao banco de dados em uso deve-se utilizar a função "**execute**" existente nos pacotes "**SQLite**". Veja a criação de uma tabela a partir do recurso "**SQLite.execute**" contendo os campos *código* e *nome* e a apresentação da tabela "**Pessoa**" no banco de dados "**teste.dbj**".

```
julia> SQLite.execute(
            db,
            "CREATE TABLE IF NOT EXISTS Pessoa (Codigo INTEGER PRIMARY KEY, Nome TEXT)"
       )
101
```

```
julia> SQLite.tables(db)
1-element Vector{SQLite.DBTable}:
 SQLite.DBTable("Pessoa", Tables.Schema:
 :Codigo  Union{Missing, Int64}
 :Nome    Union{Missing, String})
```

As saídas dos dados como resultado das operações sobre bancos de dados podem ser melhoradas com uso de quadros de dados. Observe em seguida este estilo:

```
julia> using DataFrames

julia> SQLite.tables(db) |> DataFrame
 Row │ name    schema
     │ String  Schema…
─────┼──────────────────────────────────────────
   1 │ Pessoa  Tables.Schema:\n :Codigo  Union{…
```

Chegou o momento de realizar o cadastramento de registros na tabela do banco de dados. Para que isso seja possível é necessário criar o modelo de entrada (*template*) aplicado sobre a estrutura da tabela "**Pessoa**" atribuindo este modelo a uma variável de apoio. Para tanto, execute a instrução:

```
julia> dec1 = SQLite.Stmt(db, "INSERT INTO Pessoa(Codigo, Nome) VALUES (?, ?)")
SQLite.Stmt(SQLite.DB("teste.dbj"), 6)
```

Veja que a ação do construtor "**Stmt**" do pacote "**SQLite**" estabelece junto a variável "*DEC1*" um *template* genérico de acesso ao comando *SQL* de inserção de registros "**INSERT**" indicando a tabela "**Pessoa**", seus campos "**Codigo**" e "**Nome**", bem como a definição dos coringas "**?**" indicados em "**VALUES**" que serão substituídos na operação de entrada de registros quando da efetivação de cada entrada.

Para efetivar a entrada de registros, de maneira mais confortável, considere fazer uso da função "**execute**" da interface "**DBInterface**". Assim sendo, execute as próximas instruções:

```
julia> DBInterface.execute(dec1, (100, "Maria"))
SQLite.Query(SQLite.Stmt(SQLite.DB("teste.dbj"), 6), ..., Base.RefValue{Int64}(0))

julia> DBInterface.execute(dec1, (200, "Jose"))
SQLite.Query(SQLite.Stmt(SQLite.DB("teste.dbj"), 6), ..., Base.RefValue{Int64}(0))

julia> DBInterface.execute(dec1, (300, "Paulo"))
SQLite.Query(SQLite.Stmt(SQLite.DB("teste.dbj"), 6), ..., Base.RefValue{Int64}(0))

julia> DBInterface.execute(dec1, (400, "Silvia"))
SQLite.Query(SQLite.Stmt(SQLite.DB("teste.dbj"), 6), ..., Base.RefValue{Int64}(0))

julia> DBInterface.execute(dec1, (500, "Sandra"))
SQLite.Query(SQLite.Stmt(SQLite.DB("teste.dbj"), 6), ..., Base.RefValue{Int64}(0))
```

A interface "**DBInterface**" por meio da função "**execute**" providencia uma forma de acesso simplificada a execução de instruções *SQL*.

A partir de um conjunto de registros cadastrados é possível obter consultas desses dados. Para tanto, execute as próximas instruções:

```
julia> DBInterface.execute(db, "SELECT * FROM Pessoa") |> DataFrame
5×2 DataFrame
 Row │ Codigo  Nome
     │ Int64   String
─────┼─────────────────
   1 │    100  Maria
   2 │    200  Jose
   3 │    300  Paulo
   4 │    400  Silvia
   5 │    500  Sandra

julia> DBInterface.execute(db, "SELECT Nome FROM Pessoa") |> DataFrame
5×2 DataFrame
 Row │ Nome
     │ String
─────┼────────
   1 │ Maria
   2 │ Jose
   3 │ Paulo
   4 │ Silvia
   5 │ Sandra

julia> DBInterface.execute(db, "SELECT * FROM Pessoa WHERE Codigo = 200") |> DataFrame
1×2 DataFrame
 Row │ Codigo  Nome
     │ Int64   String
─────┼─────────────────
   1 │    200  Jose
```

A título ilustrativo considere criar uma segunda tabela no banco de dados ativo contendo as alturas das pessoas cadastradas para serem vinculadas a tabela existente. Para tanto, proceda as seguintes instruções:

```
julia> SQLite.execute(db, "
           CREATE TABLE Altura (
               CodPes   INTEGER,
               Estatura DECIMAL(1, 2)
           )"
       )
101

julia> dec2 = SQLite.Stmt(db, "INSERT INTO Altura(CodPes, Estatura) VALUES (?, ?)")
SQLite.Stmt(SQLite.DB("teste.dbj"), 10)

julia> DBInterface.execute(dec2, (100, 1.56))
SQLite.Query(SQLite.Stmt(SQLite.DB("teste.dbj"), 10), ..., Base.RefValue{Int64}(0))

julia> DBInterface.execute(dec2, (200, 1.80))
SQLite.Query(SQLite.Stmt(SQLite.DB("teste.dbj"), 10), ..., Base.RefValue{Int64}(0))

julia> DBInterface.execute(dec2, (300, 2.05))
SQLite.Query(SQLite.Stmt(SQLite.DB("teste.dbj"), 10), ..., Base.RefValue{Int64}(0))

julia> DBInterface.execute(dec2, (400, 1.73))
SQLite.Query(SQLite.Stmt(SQLite.DB("teste.dbj"), 10), ..., Base.RefValue{Int64}(0))
```

```
julia> DBInterface.execute(dec2, (500, 1.68))
SQLite.Query(SQLite.Stmt(SQLite.DB("teste.dbj"), 10), ..., Base.RefValue{Int64}(0))

julia> DBInterface.execute(db, "SELECT * FROM Altura") |> DataFrame
5×2 DataFrame
 Row │ CodPes  Estatura
     │ Int64   Float64
─────┼──────────────────
   1 │    100      1.56
   2 │    200      1.8
   3 │    300      2.05
   4 │    400      1.73
   5 │    500      1.68
```

Como dito um banco de dados pode ser considerado, quando relacional, como sendo um conjunto de tabelas. Essas tabelas podem ou não ser relacionadas entre si. Neste exemplo, tem-se entre as tabelas "**Altura**" e "**Pessoa**" campos que possuem informações idênticas que permitem tal relacionamento. Para ver as tabelas do banco de dados execute a instrução:

```
julia> SQLite.tables(db)
2-element Vector{SQLite.DBTable}:
SQLite.DBTable("Pessoa", Tables.Schema:
 :Codigo  Union{Missing, Int64}
 :Nome    Union{Missing, String})
 SQLite.DBTable("Altura", Tables.Schema:
 :CodPes    Union{Missing, Int64}
 :Estatura  Missing)
```

Como dito um banco de dados pode ser considerado, quando relacional, como sendo um conjunto de tabelas. Essas tabelas podem ou não ser relacionadas entre si. Neste exemplo, tem-se entre as tabelas "**Altura**" e "**Pessoa**" campos que possuem informações idênticas que permitem tal relacionamento. Para ver as tabelas do banco de dados execute a instrução:

Para ver a apresentação relacionada de dados execute a próxima instrução:

```
julia> DBInterface.execute(db, "
           SELECT Pessoa.Nome, Altura.Estatura
           FROM Pessoa, Altura
           WHERE Pessoa.Codigo = Altura.CodPes
           ORDER BY Pessoa.Nome"
       ) |> DataFrame
5×2 DataFrame
 Row │ Nome    Estatura
     │ String  Float64
─────┼──────────────────
   1 │ Jose        1.8
   2 │ Maria       1.56
   3 │ Paulo       2.05
   4 │ Sandra      1.68
   5 │ Silvia      1.73
```

Bancos de dados devem ser tratados como arquivos e assim como são abertos devem também serem fechados. Para esta ocorrência basta executar a instrução:

```
julia> DBInterface.close!(db)
```

CAPÍTULO 7

PACOTES

Este capítulo descreve os passos básicos de utilização dos recursos de gerenciamento de pacotes, desenvolvimento de módulos, pacotes internos e externos, além da publicação de pacotes na plataforma Github.

7.1 - Gerenciador de pacotes

Em alguns momentos anteriores foi demonstrado como colocar em uso certo pacote de funcionalidades Julia. A propósito são essa coleção de pacotes da linguagem seu grande e maior recurso, pois encontra-se soluções para diversos graus de problemas. Neste sentido um repositório de pacotes Julia ideal para uso é o "**`JuliaHub`**" disponibilizado no endereço: **https://juliahub.com/ui/Packages**.

O gerenciador de pacotes Julia chama-se "**`Pkg`**" (*package*), sendo este recurso totalmente integrado ao Github podendo ser usado para gerenciar pacotes registrados e não registrados. Todos os códigos de pacotes disponibilizado são abertos o que facilita a consulta de como certas soluções são implementadas, sendo isto uma grande fonte de aprendizagem.

Anteriormente foi comentado que o gerenciador de pacotes possui *prompt* próprio a partir do REPL da linguagem Julia. Para tanto, basta escrever o símbolo "**`]`**" e automaticamente será apresentado o *prompt*:

```
(@v1.7) pkg>
```

A partir do acesso ao modo *package* é interessante conhecer seus comandos e uma forma de obter a ajuda inicial sobre essa questão é fazer uso do comando "**`help`**" ou "**`?`**". A tabela 7.1 mostra os comandos disponibilizados neste modo de trabalho.

Comando	Operação
activate	ativa ambiente primário manipulado pelo gerenciador de pacotes.
add	adiciona pacote ao projeto.
build	executa o script de compilação para pacotes.
develop / dev	duplica o repositório de pacotes completo localmente para desenvolvimento.
free	desfaz fixação, desenvolvimento ou para rastreamento de repositório.
gc	pacotes de coleta de lixo não usados por determinado tempo.
generate	gerar arquivos para novo projeto.
help / ?	apresenta o modo de ajuda do gerenciador de pacotes.
instantiate	baixa todas as dependências do projeto.
pin	fixa a versão dos pacotes
precompile	pré-compila todas as dependências do projeto.
redo	refaz a última alteração no projeto ativo.

Comando	Operação
rm	remove pacotes do projeto.
resolve	resolve atualização das alterações nas dependências dos pacotes.
status / st	faz resumo do conteúdo e das mudanças no ambiente.
test	executa teste nos pacotes.
undo	desfaz a última alteração no projeto ativo.
update / up	atualiza os pacotes instalados.
registry add	adiciona registros de pacotes.
registry remove	remove registros de pacotes.
registry status	mostra informações sobre registros instalados.
registry update	atualiza registros de pacotes.

Tabela 7.1 - Comandos modo Pkg.

Os comandos do gerenciador de pacotes podem ser executados dentro do ambiente "**Pkg**" ou a partir do REPL. Anteriormente foram usados recursos do gerenciador de pacotes diretamente de forma externa, junto ao *prompt* REPL onde é necessário fazer uso da instrução "**Pkg.função**".

Caso você faça uso dos comandos do gerenciador dentro dele próprio, basta apenas indicar o comando sem a citação "**Pkg.**". Veja a instalação de um pacote chamado "**FractionalCalculus**" destinado a efetuação de cálculos fracionários. Assim sendo execute a instrução:

```
(@v1.7) pkg> add FractionalCalculus
   Resolving package versions...
   Installed SpecialMatrices ─── v2.0.0
   Installed QuadGK ──────────── v2.4.2
   Installed Polynomials ──────── v2.0.25
   Installed MutableArithmetics - v1.0.4
   Installed Intervals ────────── v1.8.0
   Installed FractionalCalculus - v0.2.5
    Updating `C:\Users\<Usuário>\.julia\environments\v1.7\Project.toml`
  [638fb199] + FractionalCalculus v0.2.5
    Updating `C:\Users\<Usuário>\.julia\environments\v1.7\Manifest.toml`
  [638fb199] + FractionalCalculus v0.2.5
  [d8418881] + Intervals v1.8.0
  [d8a4904e] + MutableArithmetics v1.0.4
  [f27b6e38] + Polynomials v2.0.25
  [1fd47b50] + QuadGK v2.4.2
  [928aab9d] + SpecialMatrices v2.0.0
Precompiling project...
  6 dependencies successfully precompiled in 18 seconds (163 already precompiled)
```

Observe que se a instalação do pacote fosse realizada no ambiente padrão da linguagem Julia, modo REPL, sua sintaxe deveria ser "**Pkg.add("FractionalCalculus")**" posteriormente a execução da instrução "**using Pkg**".

Após a instalação é interessante ver o *status* dos pacotes existentes. Para tanto, execute a instrução:

```
(@v1.7) pkg> status
      Status `C:\Users\<Usuário>\.julia\environments\v1.7\Project.toml`
  [a93c6f00] DataFrames v1.3.4
  [638fb199] FractionalCalculus v0.2.5
  [91a5bcdd] Plots v1.31.6
  [ce6b1742] RDatasets v0.7.7
  [295af30f] Revise v3.4.0
```

A informação "**C:\Users\<Usuário>\.julia\environments\v1.7\Project.toml**" (formato sistema operacional Windows), por exemplo, indica que o arquivo que gerencia os pacotes e suas versões está localizado na posição de disco apresentada e lista todos os pacotes existentes e suas versões.

Caso deseje ver se há atualização para algum pacote ou arquivo de manifesto de instalação basta acionar a instrução e aguardar sua conclusão que poderá ser um pouco demorada:

```
(@v1.7) pkg> update
   Resolving package versions...
   ...
  75 dependencies successfully precompiled in 127 seconds (158 already precompiled)
```

A ação de atualização pode ser realizada com certa periodicidade, pois esta ação mantém o conjunto de funcionalidades em dia. Caso não tenha nada a ser atualizado a saída da operação corresponderá a algo semelhante como:

```
(@v1.7) pkg> update
    Updating registry at `C:\Users\<Usuário>\.julia\registries\General.toml`
  No Changes to `C:\Users\<Usuário>\.julia\environments\v1.7\Project.toml`
  No Changes to `C:\Users\<Usuário>\.julia\environments\v1.7\Manifest.toml`
```

Vale lembrar, pois isto já foi apresentado, que os pacotes instalados no ambiente podem ser removidos com o uso do comando "**rm**". Para remover um pacote é necessário após o comando indicar o nome do pacote a ser removido.

Os pacotes instalados podem ser submetidos a testes operacionais. Para realizar esta tarefa, considere, por exemplo o pacote "**FractionalCalculus**". Use a instrução a seguir e aguarde o a conclusão da operação:

```
(@v1.7) pkg> test FractionalCalculus
     Testing FractionalCalculus
  ...
     Testing Running tests...
Test Summary:      | Pass  Total
FractionalCalculus |   96     96
     Testing FractionalCalculus tests passed
```

Em um contexto trivial os comandos mais usados para administrar o gerenciamento de pacotes são "**add**", "**update**" e "**rm**".

O pacote "**FractionalCalculus**" possui recursos, por exemplo, para os cálculos de derivadas e integrais fracionárias. Para mais detalhes desse recurso acesse o endereço **https://scifracx.org/FractionalCalculus.jl/dev/Derivative/derivative/**. Assim sendo, apenas como exemplo inicial de uso do pacote execute as próximas instruções:

```
(@v1.7) pkg> <Backspace>

julia> using FractionalCalculus

julia> fracdiff(x -> x, 0.3, 1, 0.0001, RLDiffApprox())
1.1005474055236661

julia> fracdiff(x -> x, 0.3, 1, 0.0001, CaputoPiecewise())
1.1005474055556066

julia> fracint(x -> x, 0.3, 1, 0.0001, RLIntApprox())
0.8571075566161154
```

Como comentado os pacotes podem ser gerenciados dentro do modo de gerenciamento de pacotes ou for a no *prompt* REPL. Considerando remover o pacote "**FractionalCalculus**" execute a instrução:

```
julia> Pkg.rm("FractionalCalculus")
...
```

A adição ou remoção de pacotes pode ser realizada em lotes. Para tanto, basta seguir o estilo de instrução a partir do *prompt* "**Pkg**":

```
(@v1.7) pkg> add pacote1 pacote2 pacote3

(@v1.7) pkg> rm pacote1 pacote2 pacote3
```

Os detalhes de gerenciamento apresentados se referem ao uso de pacotes que estão registrados. Um pacote registrado é aquele que se encontra indexado e pode ser obtido a partir de sua chamada por meio de seu nome com o comando "**add**", sem que haja a necessidade de indicar qualquer outra informação.

Pacotes não registrados são diferentes, pois para serem usados necessitam que lhes sejam informados mais do que simplesmente seu nome. Normalmente os pacotes não registrados são pacotes em estado experimental e por esta razão não estão colocados no repositório de pacote, como pacote registrado. Desta forma, a instalação de um pacote não registrado segue o estilo de sintaxe:

```
(@v1.7) pkg> add https://github.com/local/nome_do_pacote.jl
```

Onde "**local**" se refere a indicação de um diretório do qual certo pacote será obtido e "**nome_do+pacote.jl**" a identificação do pacote a ser adicionado. Para usar o pacote basta chamá-lo pelo nome, sem o uso de "**.jl**" com o comando "**using**". Pacotes não registrados podem ter dependências e estas deverão também serem instaladas manualmente antes do pacote.

Como teste de operação no uso de pacote não registrado, considere instalar o projeto "**OhMyREPL**" que apresenta um ambiente REPL colorizado. Este projeto possui como dependência não registrada o pacote "**Tokenize**" (RIP Tutorial, 2022). Assim sendo execute as instruções seguintes:

```
(@v1.7) pkg> add https://github.com/KristofferC/Tokenize.jl
     Cloning git-repo `https://github.com/KristofferC/Tokenize.jl`
    Updating git-repo `https://github.com/KristofferC/Tokenize.jl`
    Updating registry at `C:\Users\Augusto\.julia\registries\General.toml`
   Resolving package versions...
    Updating `C:\Users\Augusto\.julia\environments\v1.7\Project.toml`
  [0796e94c] + Tokenize v0.5.24 `https://github.com/KristofferC/Tokenize.jl#master`
    Updating `C:\Users\Augusto\.julia\environments\v1.7\Manifest.toml`
  [0796e94c] + Tokenize v0.5.24 `https://github.com/KristofferC/Tokenize.jl#master`
Precompiling project...
  1 dependency successfully precompiled in 6 seconds (233 already precompiled)

(@v1.7) pkg> add https://github.com/KristofferC/OhMyREPL.jl
     Cloning git-repo `https://github.com/KristofferC/OhMyREPL.jl`
    Updating git-repo `https://github.com/KristofferC/OhMyREPL.jl`
   Resolving package versions...
   Installed fzf_jll — v0.29.0+0
   Installed Pipe ─── v1.3.0
   Installed JLFzf ── v0.1.5
  Downloaded artifact: fzf
    Updating `C:\Users\Augusto\.julia\environments\v1.7\Project.toml`
  [5fb14364] + OhMyREPL v0.5.12 `https://github.com/KristofferC/OhMyREPL.jl#master`
```

```
    Updating `C:\Users\Augusto\.julia\environments\v1.7\Manifest.toml`
  [1019f520] + JLFzf v0.1.5
  [5fb14364] + OhMyREPL v0.5.12 `https://github.com/KristofferC/OhMyREPL.jl#master`
  [b98c9c47] + Pipe v1.3.0
  [214eeab7] + fzf_jll v0.29.0+0
Precompiling project...
  4 dependencies successfully precompiled in 6 seconds (234 already precompiled)
```

Observe que durante a instalação dos pacotes "**OhMyREPL**" e "**Tokenize**" ocorreu a instalação automática de algumas dependências. Isto ocorreu pelo fato dessas dependências estarem registradas. Observe que o pacote não registrado "**OhMyREPL**" precisou da instalação antecipada do pacote, também não registrado "**Tokenize**".

Para ver o funcionamento do pacote "**OhMyREPL**" volte para o *prompt* REPL e execute as instruções indicadas na figura 7.1, observando que a medida em que os comandos são escritos estes são colorizados.

```
julia> using OhMyREPL

julia> function adição(a::Int64, b::Int64)
           # operação de adição
           return a + b
       end
adição (generic function with 1 method)

julia> adição(1, 2)
3

julia>
```

Figura 7.1 - Resultado colorizado do modo OhMyREPL.

Caso queira remover um pacote não registrado com o comando "**rm**" ou atualizá-lo com o comando "**status**" use exatamente a forma que faria com um pacote registrado. Para tanto, volte ao modo "**Pkg**" e execute para experimentação as instruções seguintes:

```
(@v1.7) pkg> status
      Status `C:\Users\Augusto\.julia\environments\v1.7\Project.toml`
  [a93c6f00] DataFrames v1.3.4
  [5fb14364] OhMyREPL v0.5.12 `https://github.com/KristofferC/OhMyREPL.jl#master`
  [91a5bcdd] Plots v1.31.6
  [ce6b1742] RDatasets v0.7.7
  [295af30f] Revise v3.4.0
  [0796e94c] Tokenize v0.5.24 `https://github.com/KristofferC/Tokenize.jl#master`

(@v1.7) pkg> rm OhMyREPL
    Updating `C:\Users\Augusto\.julia\environments\v1.7\Project.toml`
  [5fb14364] - OhMyREPL v0.5.12 `https://github.com/KristofferC/OhMyREPL.jl#master`
    Updating `C:\Users\Augusto\.julia\environments\v1.7\Manifest.toml`
  [1019f520] - JLFzf v0.1.5
  [5fb14364] - OhMyREPL v0.5.12 `https://github.com/KristofferC/OhMyREPL.jl#master`
  [b98c9c47] - Pipe v1.3.0
  [214eeab7] - fzf_jll v0.29.0+0

(@v1.7) pkg> rm Tokenize
    Updating `C:\Users\Augusto\.julia\environments\v1.7\Project.toml`
  [0796e94c] - Tokenize v0.5.24 `https://github.com/KristofferC/Tokenize.jl#master`
    Updating `C:\Users\Augusto\.julia\environments\v1.7\Manifest.toml`
  [0796e94c] - Tokenize v0.5.24 `https://github.com/KristofferC/Tokenize.jl#master`
```

```
(@v1.7) pkg> status
      Status `C:\Users\Augusto\.julia\environments\v1.7\Project.toml`
  [a93c6f00] DataFrames v1.3.4
  [91a5bcdd] Plots v1.31.6
  [ce6b1742] RDatasets v0.7.7
  [295af30f] Revise v3.4.0
```

Os pacotes instalados podem ser atualizados SEPARADAMENTE. Quando se usa o comando "**update**" sem a indicação de algum pacote, todos os pacotes são atualizados. Caso queira fazer a atualização apenas um pacote ou de alguns pacotes use a sintaxe:

```
(@v1.7) pkg> update pacote
```

```
(@v1.7) pkg> update pacote1 pacote2 pacote3
```

O gerenciador de pacotes da linguagem Julia pode ser operado a partir da definição de ambientes de memória. Em sua forma básica opera sobre uma área interna predefinida, mas é possível criar áreas personalizadas de trabalho utilizando-se o comando "**activate**". Assim sendo, execute a próxima instrução:

```
(@v1.7) pkg> activate exemplo
  Activating new project at `C:\Users\Augusto\exemplo`

(exemplo) pkg>
```

Observe que o indicativo de *prompt* foi alterado de "**(@v1.7) pkg>**" para "**(exemplo) pkg>**" mostrando que agora está em operação uma outra aérea de ambiente de trabalho, criado a partir do diretório de operação do usuário ativo.

Execute o comando "**status**" para obter informações desse novo ambiente:

```
(exemplo) pkg> status
      Status `C:\Users\Augusto\exemplo\Project.toml` (empty project)
```

Veja que neste momento é indicado que o local do ambiente definido se encontra com um arquivo de projeto vazio (*empty project*), ou seja, mostra a localização do arquivo de projeto no ambiente ativo, que é o local onde o gerenciador de pacotes armazena meta-dados de informação para o ambiente e armazena pacotes adicionados.

Para testar essa possibilidade operacional execute na sequência as instruções:

```
(exemplo) pkg> add FractionalCalculus
  Resolving package versions...
 ...

(exemplo) pkg> status
      Status `C:\Users\Augusto\exemplo\Project.toml`
  [638fb199] FractionalCalculus v0.2.10
```

Veja que neste instante o ambiente "**exemplo**" possui o pacote "**FractionalCalculus**" como sua dependência e poderá ser usado normalmente. Para um teste volte ao *prompt* REPL e execute as instruções:

```
julia> using FractionalCalculus

julia> fracdiff(x -> x, 0.3, 1, 0.0001, RLDiffApprox())
1.1005474055236661
```

A vantagem no uso de ambientes declarados manualmente é que as instalações de pacotes são posicionadas em um ambiente fora do local de instalação da linguagem. Essa estratégia torna o ambiente de trabalho mais seguro por não interferir nos locais padrão do ambiente da linguagem.

Para um teste operacional feche o ambiente da linguagem Julia. Abra-o novamente e execute as seguintes instruções:

```
julia> fracdiff(x -> x, 0.3, 1, 0.0001, RLDiffApprox())
ERROR: UndefVarError: RLDiffApprox not defined
Stacktrace:
 [1] top-level scope
   @ REPL[1]:1

julia> using FractionalCalculus
 | Package FractionalCalculus not found, ... FractionalCalculus is available from a registry.
 | Install package?
 |   (@v1.7) pkg> add FractionalCalculus
 └ (y/n) [y]:
```

Observe que está sendo solicitada autorização de instalação, pois este pacote está sendo direcionado para o ambiente padrão. Neste momento responda "**n**" para não instalar.

Lembre-se de que o pacote está instalado em uma área chamada "**exemplo**". Assim sendo, execute as seguintes instruções no *prompt* REPL:

```
julia> using Pkg

julia> Pkg.activate("exemplo")
  Activating project at `C:\Users\Augusto\exemplo`
```

Veja que a partir deste momento a área chamada "**exemplo**" torna-se ativa e pronta para uso. Assim sendo, execute na sequência as instruções:

```
julia> using FractionalCalculus

julia> fracdiff(x -> x, 0.3, 1, 0.0001, RLDiffApprox())
1.1005474055236661
```

Note que não ocorreu nenhum erro ou solicitação de instalação, pois o ambiente condizente onde se encontra o pacote está aberto. Veja agora o *status* dos pacotes ativos a partir do *prompt* REPL.

```
julia> Pkg.status()
      Status `C:\Users\Augusto\exemplo\Project.toml`
  [638fb199] FractionalCalculus v0.2.10
```

Para ativar o ambiente padrão novamente basta executar a instrução:

```
julia> Pkg.activate()
  Activating project at `C:\Users\Augusto\.julia\environments\v1.7`
```

Veja o status do ambiente padrão após sua reativação:

```
julia> Pkg.status()
      Status `C:\Users\Augusto\.julia\environments\v1.7\Project.toml`
  [a93c6f00] DataFrames v1.3.4
  [91a5bcdd] Plots v1.31.6
  [ce6b1742] RDatasets v0.7.7
  [295af30f] Revise v3.4.0
```

Pacotes podem ser alterados de acordo com alguma necessidade particular. Para testar essa ocorrência, abra o *prompt* "**Pkg**" e execute as instruções, atentando para o uso do pacote "**Example**" que possui internamente duas funções definidas "**hello**" e "**domath**":

```
(@v1.7) pkg> activate exemplo
  Activating new project at `C:\Users\Augusto\exemplo`

 (exemplo) pkg> add Example  # pacote de testes e aprendizagem da linguagem Julia
    Updating registry at `C:\Users\Augusto\.julia\registries\General.toml`
  ...
```

Considere acrescentar no pacote "**Example**" uma funcionalidade para a soma de dois valores numéricos inteiros, ou seja, modificar o código fonte original do pacote. Para tanto, execute as instruções:

```
(exemplo) pkg> develop --local Example
     Cloning git-repo `https://github.com/JuliaLang/Example.jl.git`
   Resolving package versions...
    Updating `C:\Users\Augusto\exemplo\Project.toml`
  [7876af07] ~ Example v0.5.3 ⇒ v0.5.4 `dev\Example`
    Updating `C:\Users\Augusto\exemplo\Manifest.toml`
  [7876af07] ~ Example v0.5.3 ⇒ v0.5.4 `dev\Example`

(exemplo) pkg> status
      Status `C:\Users\Augusto\exemplo\Project.toml`
  [7876af07] Example v0.5.4 `dev\Example`
  [638fb199] FractionalCalculus v0.2.10
```

Veja que com a execução do comando "**status**" tem-se no ambiente "**exemplo**" a apresentação dos pacotes nele adicionados. Mas atente para a indicação "**dev\Example**" junto a apresentação do pacote "**Example**". Veja que o status indica a possibilidade de desenvolvimento sobre o pacote, isso ocorreu a partir do uso do comando "**develop**".

Obviamente você não estará alterando o pacote que se encontra registro e disponível no repositório de instalação, mas sim a cópia que você tem em seu computador.

Para fazer a alteração no código do programa "**Example.jl**" use um editor de texto de sua preferência e acesse o arquivo indicado a partir do local em que se encontra gravado. Assim sendo, abra o arquivo "**Example.jl**" a partir de seu diretório de trabalho, no Windows "**\Augusto\exemplo\dev\Example\src\Example.jl**".

A partir da linha "**17**" antes do comando "**end**" acrescente o trecho de código:

```
soma(a::Int64, b::Int64) = a + b
```

Observe que neste código a função está sendo definida sem o uso do comando "**function**", ou seja é uma função de uma linha declarada no estilo usado na programação funcional.

Após a inserção do novo recurso no código do programa do pacote **Example**" grave o arquivo e saia do editor de texto usado, volte na sequência ao *prompt* REPL e execute a instrução seguinte:

```
julia> import Example
[ Info: Precompiling Example [7876af07-990d-54b4-ab0e-23690620f79a]
```

Observe que o código alterado é então recompilado. A partir do término dessa operação já é possível fazer uso da nova função. Assim sendo, execute a instrução:

```
julia> Example.soma(1, 2)
3
```

O uso do comando "**`import`**" só pode ser executado apenas uma vez por sessão Julia. Para usar novamente esta operação é necessário reiniciar o ambiente REPL e executar novamente a instrução de ativação "**`Pkg.activate("exemplo")`**".

Caso queira desabilitar o pacote "**`Example`**" alterado retornando ao modo padrão, sem alteração, execute a instrução:

```
julia> Pkg.free("Example")
   Resolving package versions...
    Updating `C:\Users\Augusto\exemplo\Project.toml`
  [7876af07] ~ Example v0.5.4 `dev\Example` ⇒ v0.5.3
    Updating `C:\Users\Augusto\exemplo\Manifest.toml`
  [7876af07] ~ Example v0.5.4 `dev\Example` ⇒ v0.5.3
```

A partir dessa ação o ambiente não usará mais a versão alterada mantendo o uso da versão registrada.

A partir desta apresentação você tem em mãos o conhecimento essencial para administrar o ambiente de gerenciamento de pacotes da linguagem Julia em seu computador. Os demais comandos apresentados são considerados mais avançados e por esta razão não são demonstrados.

7.2 - Módulos, uma forma de pacote interno

Os pacotes apresentados até este momento são recursos obtidos de forma externa a partir de fonte registrada ou não registrada, os quais são desenvolvidos por pessoas para o atendimento de determinadas necessidades. No entanto, pode ocorrer de determinada necessidade computável não possuir pacotes desenvolvidos. E aí entra a possibilidade de se desenvolver os próprios pacotes.

Pacotes podem ser desenvolvidos a partir de duas óticas de aplicação, podem ser internos (privados) e externos (públicos). Quando internos ficam disponibilizados para uso apenas no computador em que foram desenvolvidos ou instalados manualmente configurados a partir da definição de módulos, mas quando externos ficam disponibilizados via plataforma Github de modo registrado ou não registrado para uso de uma comunidade.

Os módulos são *containers* que ajudam a organizar o código em unidades dentro de áreas denominadas *namespaces* (nomes de espaço). Neste tópico a atenção é direcionada ao desenvolvimento de um módulo (pacote interno) bastante simples. Assim sendo, considere um pequeno pacote que possui duas funcionalidades de conversão numérica, sendo recursos para converter número decimal em número binário e número binário em número decimal. Os resultados retornados por essas funcionalidades devem ser valores numéricos.

As funcionalidades de conversão devem seguir a solução baseada em seus algoritmos clássicos de conversão, ou seja, não são usados nenhum recurso de conversão numérica automatizado pela própria linguagem Julia. As funções recebem um argumento numérico e retornam como resposta valor também numérico. Assim sendo considere o código de programa seguinte gravando-o, a partir de seu diretório padrão de trabalho, com o nome com o nome "**`c07ex01.jl`**".

```julia
# Módulo como pacote interno

module Numeros

export dec2bin, bin2dec

function dec2bin(vlr::Int64)::Int64
    dec::Int64 = 0
    str::String = ""
    while vlr > 0
        dec = vlr % 2
        str *= string(dec)
        vlr = div(vlr, 2)
    end
    return parse(Int, reverse(str))
end

function bin2dec(vlr::Int64)::Int64
    dec::Int64 = 0
    res::Int64 = 0
    i::Int64 = 0
    while vlr > 0
        res = vlr % 2
        vlr = div(vlr, 10)
        dec += res * 2 ^ i
        i += 1
    end
    return dec
end

end   # Numeros
```

Um programa de pacote é definido dentro de um bloco identificado com os comandos "**`module`**" e "**`end`**". Dentro deste bloco se define as funções que serão operacionalizadas pelo pacote. As funções que forem indicadas junto ao comando "**`export`**" poderão ser usadas quando o pacote for carregado. Funções que não sejam qualificadas como exportadas não serão disponibilizadas. No entanto, se for omitido "**`export`**" todas as funcionalidades existentes serão visíveis.

O bloco definido entre "**`module`**" e "**`end`**" tem, dentro de suas regras documentais, a definição de seu nome iniciando-se com letra maiúscula, ou seja, respeitando-se a regra de nomeação "*`UpperCamelCase`*" e grafado no plural. Veja que o bloco cria um espaço operacional de trabalho, uma área de nome de espaço (*namespace*) que disponibiliza para operação seus recursos exportados. Os recursos exportados são públicos e os que não forem dessa forma qualificados serão privados.

Os módulos definem áreas de operação com escopo global. Isso permite que o mesmo nome seja usado para diferentes funções ou variáveis globais sem que ocorram conflitos, desde que, tais recursos, estejam definidos em módulos separados. Assim como recursos de um módulo podem ser exportados (comando "**`export`**"), também podem ser importados (comandos "**`using`**" e "**`import`**").

O desenvolvimento de módulo deve seguir uma sequência de passos para utilização. Assim sendo, considerando que o código do pacote está gravado na área de trabalho do usuário onde a linguagem opera execute as seguintes instruções no *prompt* REPL:

```
julia> include("c07ex01.jl")
Main.Numeros

julia> using .Numeros
```

Veja que as instruções anteriores colocam em uso, primeiramente o pacote "**`Numeros`**" definido como modulo de operação do código do programa "**`c07ex01.jl`**" por meio do comando "**`include`**". Em seguida o pacote é colocado em

uso, como de costume, com o comando "**using**". Mas há um detalhe a ser observado: note a chamada do pacote com a indicação do símbolo de ponto "**.**" antes do seu nome. Isso é necessário quando se usa pacotes internos.

O uso do ponto indica para o ambiente que o pacote em uso é interno. Os pacotes externos não usam essa nomenclatura. A partir do carregamento do pacote interno em memória execute as próximas instruções para verificar sua funcionalidade:

```
julia> dec2bin(10)
1010

julia> bin2dec(1010)
10

julia> Numeros.dec2bin(589)
1001001101

julia> Numeros.bin2dec(1001001101)
589
```

Veja que há duas formas de se fazer uso dos recursos do pacote interno. A primeira forma sem indicar o nome do módulo e a segunda forma indicado o nome do módulo, ou seja, usando-se a ideia de *nome qualificado*. A primeira forma é aparentemente mais conveniente para uso por ser mais rápida, mas a segunda forma deixa a indicação da ação mais clara e segura evitando-se ambiguidade de operações caso haja duas ou mais funções ou variável com mesmo nome.

Uma forma de ver o nome do módulo a que certo recurso pertence, ou seja, ver seu *módulo pai* basta usar a função "**parentmodule**". Observe o resultado das próximas instruções:

```
julia> parentmodule(sqrt)
Base

julia> parentmodule(dec2bin)
Main.Numeros
```

Os indicativos "**Base**" e "**Main.Numeros**" caracterizam-se por mostrarem o caminho do módulo. Veja então os usos de recursos a partir de da indicação de nomes qualificados a partir da definição explícita de caminhos dos módulos:

```
julia> Base.Math.cos(-1.5)
0.0707372016677029

julia> Numeros.bin2dec(1011001111)
719

julia> Main.Numeros.bin2dec(1011001111)
719

julia> Base.sqrt(Main.Numeros.bin2dec(1011001111))
26.814175355583846
```

A possibilidade de fazer uso das funcionalidades de um módulo a partir de seus nomes de identificação diretos ou por meio de nomes qualificados só é possível devido ao uso do comando "**using**" no carregamento do módulo para a memória. Mas nem sempre isso será possível caso tenha sido carregado o módulo com o comando "**import**".

Se o módulo for carregado a partir do comando "**import**" o uso de nomes qualificados será obrigatório. Para ver este efeito mais restritivo saia do ambiente Julia e abra uma nova instância do ambiente. Na sequência execute as instruções:

```
julia> include("c07ex01.jl")
Main.Numeros

julia> import .Numeros

julia> Numeros.bin2dec(1011)
11

julia> bin2dec(1011)
ERROR: UndefVarError: bin2dec not defined
Stacktrace:
 [1] top-level scope
   @ REPL[4]:1
```

Veja que os comandos "**using**" e "**import**" proporcionam pleno acesso aos recursos de um módulo, mas o fazem de forma diferenciada. Tenha atenção e cuidado com este pequeno detalhe.

Um módulo pode ser composto por códigos advindos de outros módulos. Para demonstrar este efeito considere a definição de dois programas de módulos separados, um módulo possui apenas o recurso de conversão de número decimal em número hexadecimal e outro módulo contendo o recurso de conversão de número hexadecimal em número decimal. Os resultados de retorno dessas funcionalidades devem cadeia de decimal para hexadecimal e numérico de hexadecimal para decimal.

O código do programa "**c07ex02.jl**" caracteriza-se por conter o recurso de conversão decimal em hexadecimal. Observe o código seguinte e grave-o com o nome apontado.

```
 1  # Módulo de conversão decimal para hexadecimal
 2
 3  module DecHexs
 4
 5  export dec2hex
 6
 7  function dec2hex(vlr::Int64)::Int64
 8      dec::Int64 = 0
 9      str::String = ""
10      while vlr > 0
11          dec = vlr % 16
12          # converte inteiro em caractere
13          if dec < 10
14              str *= Char(dec + 48)
15          else
16              str *= Char(dec + 55)
17          end
18          vlr = div(vlr, 16)
19      end
20      return reverse(str)
21  end
22
23  end   # dec2hex
```

Pelo fato de serem os números hexadecimais representados por letras quando maiores ou iguais a dez e menores ou iguais a "**16**", torna-se necessário levar essa condição em consideração. Daí na linha "**13**" verifica-se se o número decimal da variável "*DEC*" é menor que dez e sendo o programa executa a linha "**14**" onde valor do decimal é somando em uso é convertido em seu formato caractere equivalente. Assim sendo, se o número avaliado for "**1**" este será somado a "**48**" resultando "**49**" que é o código ASCII equivalente ao símbolo "**1**". Caso a condição da linha "**13**" seja falsa o programa desvia o processamento para a linha "**16**" e soma ao número o valor "**55**", isso faz que qualquer valor maior ou igual a dez seja convertido a letra correspondente do valor hexadecimal.

O código do programa "**c07ex03.jl**" caracteriza-se por conter o recurso de conversão hexadecimal em decimal. Observe o código seguinte e grave-o com o nome apontado.

```
 1  # Módulo de conversão decimal para hexadecimal
 2
 3  module HexDecs
 4
 5  export hex2dec
 6
 7  function hex2dec(vlr::String)::Int64
 8      dec::Int64 = 0
 9      pot::Int64 = 0
10      res::Int64 = 0
11      i::Int64 = 0
12      for i in length(vlr):-1:1
13          # converte caractere em inteiro
14          if vlr[i] >= 'A'
15              res = vlr[i] - 'A' + 10
16          end
17          dec += res * 16 ^ pot
18          pot += 1
19      end
20      return dec
21  end
22
23  end   # dec2hex
```

Neste programa deve-se pegar o número hexadecimal e convertê-lo em decimal. Como número hexadecimais possuem representação por letra essa característica tem que ser considerada. Isso é realizado na linha "**14**", a qual verifica se o dígito do valor em foco é maior ou igual a "**A**", sendo subtrai-se "**A**" e soma-se dez para obter o valor equivalente em decimal.

A partir dos dois arquivos de módulos definidos observe a definição de outro programa que faz a chamada desses dois arquivos como se fosse um único módulo. Essa estratégia de organizar um módulo a partir de vários arquivos é comum quando se opera com volume alto de funcionalidades e recursos. Atente para o próximo código que deverá ser gravado com o nome "**c07ex04.jl**".

```
1  # Módulo de organização e chamada de outros módulos
2
3  module Convets
4
5  include("c07ex02.jl")
6  include("c07ex03.jl")
7
8  end   # dec2hex
```

Observe que o acesso a arquivos de programas de módulos externos é realizado com o uso da função "**include**" que carrega para a memória os módulos internos. Na sequência saia do ambiente Julia, carregue-o novamente e observe as próximas instruções a serem executadas:44

```
julia> include("c07ex04.jl")
Main.Converts

julia> using .Converts
```

```
julia> Converts.HexDecs.hex2dec("BCA")
3018

julia> Converts.DecHexs.dec2hex(3018)
"BCA"

julia> hex2dec("BCA")
ERROR: UndefVarError: hex2dec not defined
Stacktrace:
 [1] top-level scope
   @ REPL[7]:1

julia> dec2hex(3018)
ERROR: UndefVarError: dec2hex not defined
Stacktrace:
 [1] top-level scope
   @ REPL[8]:1
```

Note que neste exemplo foi usado o comando "**using**" mas o reconhecimento das funcionalidades de forma direta não ocorreu. O efeito somente funcionou com o uso obrigatório de nomes qualificados. Quando o módulo possui todas as funcionalidades nele mesmo o uso do comando "**using**" proporciona flexibilidade. Mas o mesmo não ocorre quando se opera um módulo chamando outro módulo como sua dependência.

Para maiores informações sobre módulos consulte o endereço **https://docs.julialang.org/en/v1/manual/modules/**.

7.3 - Criação manual de pacotes

O desenvolvimento de pacotes não é complexo, mas é uma ação que se configura um pouco diferente do uso de módulos, sendo um processo bem mais trabalhoso. Enquanto um módulo é composto por um ou mais arquivos que podem estar interligados a partir do uso direto do comando "**include**", os pacotes são estruturados de maneira diferente.

Um pacote é um arquivo de projeto que possui informações especiais: "**name**", "**uuide**", "**authors**" e "**version**" indicadas em um arquivo chamado "**Project.toml**" localizado no diretório "**/nome_do_projeto/src/**", em que o indicativo "**nome_do_projeto**" representa o local em disco que o projeto é desenvolvido.

A fim de testar o desenvolvimento e aplicação de pacote de forma manual considere a definição de uma função chamada "**tabuada**" que apresente a tabuada de um a dez de um número passado como argumento, sendo que os números fornecidos devem estar entre "**1**" e "**10**", caso seja fornecido um valor fora a faixa permitida a função deve indicar mensagem informando esta condição. A função em questão, deve pertencer ao pacote chamado "**Matematica**".

A criação de pacotes pode ser realizada tanto no *prompt* "**REPL**" como no *prompt* "**Pkg**", assim como é o desenvolvimento dos módulos. Considerando o uso do *prompt* "**REPL**", execute as seguintes instruções:

```
julia> using Pkg

julia> Pkg.generate("Matematica")
  Generating  project Matematica:
    Matematica\Project.toml
    Matematica\src/Matematica.jl
Dict{String, Base.UUID} with 1 entry:
  "Matematica" => UUID("4008b8b7-7cfb-4e7f-a40b-23d50447404a")
```

Neste instante, a partir do uso da função "**generate**" é criado um diretório chamado "**Matematica**" junto ao diretório de trabalho do usuário. Dentro deste diretório são definidos o diretório "**src**" com o arquivo "**Matematica.jl**" e o arquivo "**Project.toml**", os quais possuem as seguintes informações:

Arquivo "**Project.toml**":

```
1  name = "Matematica"
2  uuid = "4008b8b7-7cfb-4e7f-a40b-23d50447404a"
3  authors = ["Augusto "]
4  version = "0.1.0"
```

O arquivo "**Project.toml**" é formado por informações geradas automaticamente as quais não devem ser modificadas manualmente a menos que seja extremamente necessário, contendo os valores "**name**": nome do projeto do pacote, "**uuide**": número de identificação universalmente exclusivo que auxilia a identificar o pacote caso venha a ser publicado externamente, "**authors**": nome dos autores do projeto e "**version**": indicação da versão inicial do projeto em desenvolvimento. Além dessas informações este arquivo poderá vir a conter outros detalhes ao longo do desenvolvimento do pacote.

Arquivo "**Matematica.jl**":

```
1  module Matematica
2
3  greet() = print("Hello World!")
4
5  end # module
```

Note que o arquivo "**Matematica.jl**" possui a definição de um módulo contendo a indicação automática de uma função exemplo chamada "**greet**" que retorna, se executada, a mensagem "**Hello World!**", a qual deverá ser removida quando da edição do arquivo.

Com o editor de textos de sua preferência em uso localize e abra o arquivo "**Matematica.jl**" removendo o código da função "**greet**", acrescentando a chamada do pacote "**Printf**" e indicando o código da função "**matematica**". Veja as alterações definidas entre as linhas "**3**" e "**14**" como indicado na figura 7.2.

```
Matematica.jl
1  module Matematica
2
3  using Printf
4
5  function tabuada(n::Int)
6      if n >= 1 && n <= 10
7          for i in 1:10
8              @printf("%2i x %2i = %3i\n", n, i, n * i)
9          end
10     else
11         printstyled("Atenção: "; color=:yellow)
12         println("Use valores de 1 até 10.")
13     end
14 end
15
16 end # module
17
```

Figura 7.2 - Alterações no arquivo "Matematica.jl".

Veja que o arquivo modificado de "**Matematica.jl**" indica na linha "**3**" o uso do pacote "**Printf**" como dependência. Quando há a indicação de dependência é necessário adicionar o pacote dependente ao projeto. Para tanto, execute as instruções:

```
julia> Pkg.activate("Matematica")  # ativar para uso o pacote indicado
  Activating project at `C:\Users\Augusto\Matematica`

julia> Pkg.add("Printf")  # acrescentar o pacote dependente no pacote em desenvolvimento
   Resolving package versions...
    Updating `C:\Users\Augusto\Matematica\Project.toml`
  [de0858da] + Printf
    Updating `C:\Users\Augusto\Matematica\Manifest.toml`
  [de0858da] + Printf
```

```
  [4ec0a83e] + Unicode
Precompiling project...
  1 dependency successfully precompiled in 1 seconds
```

A operação "**activate**" habilita o pacote para uso e "**add**" faz os ajustes de acesso entre a dependência e o pacote em desenvolvimento. Veja que um novo arquivo foi gerado no diretório "**Matematica**" chamado "**Manifest.toml**", o qual possui como conteúdo gerado automaticamente as informações de configuração.

```
1  # This file is machine-generated - editing it directly is not advised
2
3  julia_version = "1.7.3"
4  manifest_format = "2.0"
5
6  [[deps.Printf]]
7  deps = ["Unicode"]
8  uuid = "de0858da-6303-5e67-8744-51eddeeeb8d7"
9
10 [[deps.Unicode]]
11 uuid = "4ec0a83e-493e-50e2-b9ac-8f72acf5a8f5"
```

Além da criação do arquivo "**Manifest.toml**" o arquivo "**Project.toml**" é alterado tendo o acréscimo da indicação da dependência registrada na seção "**deps**".

```
1  name = "Matematica"
2  uuid = "4008b8b7-7cfb-4e7f-a40b-23d50447404a"
3  authors = ["Augusto "]
4  version = "0.1.0"
5
6  [deps]
7  Printf = "de0858da-6303-5e67-8744-51eddeeeb8d7"
```

Na sequência execute as próximas instruções para colocar o pacote "**Matematica**" em uso:

```
julia> import Matematica  # pode ser usado "using Matematica"

julia> Matematica.tabuada(5)
 5 x  1 =   5
 5 x  2 =  10
 5 x  3 =  15
 5 x  4 =  20
 5 x  5 =  25
 5 x  6 =  30
 5 x  7 =  35
 5 x  8 =  40
 5 x  9 =  45
 5 x 10 =  50

julia> Matematica.tabuada(11)
Atenção: Use valores de 1 até 10.
```

Reinicie o ambiente Julia e execute as instruções:

```
julia> using Pkg

julia> Pkg.activate("Matematica")
  Activating project at `C:\Users\Augusto\Matematica`
```

```
julia> using Matematica

julia> Matematica.tabuada(7)
 7 x  1 =   7
 7 x  2 =  14
 7 x  3 =  21
 7 x  4 =  28
 7 x  5 =  35
 7 x  6 =  42
 7 x  7 =  49
 7 x  8 =  56
 7 x  9 =  63
 7 x 10 =  70
```

Um elemento importante no uso de pacotes é a definição do modo ajuda (*help*). No exemplo apresentado há apenas a função "**tabuada**" e neste caso a definição da ajuda será simples. Para definir o modo ajuda ao pacote "**Matematica**" faça a edição do arquivo "**Matematica.jl**" inserindo no código as linhas grafadas em vermelho a seguir:

```
1  module Matematica
2
3  using Printf
4
5  """
6  ```julia
7  Matematica.tabuada(n)
8  ```
9
10 Mostra ```tabuada``` de multiplicação dos valores de ```1``` até
11 ```10```. Valores fora da faixa permitida geram a apresentação de uma
12 mensagem de advertência.
13
14 ---
15
16 ### Exemplos
17
18 ```julia
19 julia> Matematica.tabuada(10)
20 10 x  1 =  10
21 10 x  2 =  20
22 10 x  3 =  30
23 10 x  4 =  40
24 10 x  5 =  50
25 10 x  6 =  60
26 10 x  7 =  70
27 10 x  8 =  80
28 10 x  9 =  90
29 10 x 10 = 100
30
31 julia> Matematica.tabuada(0)
32 Atenção: Use valores de 1 até 10.
33 ```
34 """
35 function tabuada(n::Int)
36     if n >= 1 && n <= 10
37         for i in 1:10
38             @printf("%2i x %2i = %3i\n", n, i, n * i)
39         end
40     else
41         printstyled("Atenção: "; color=:yellow)
42         println("Use valores de 1 até 10.")
```

```
43      end
44  end
45
46  end # module
```

Reinicie o ambiente Julia e execute novamente as instruções:

```
julia> using Pkg

julia> Pkg.activate("Matematica")
  Activating project at `C:\Users\Augusto\Matematica`

julia> using Matematica
[ Info: Precompiling Matematica [4008b8b7-7cfb-4e7f-a40b-23d50447404a]

julia> ?

help?> Matematica.tabuada
```

```
julia> using Matematica
[ Info: Precompiling Matematica [4008b8b7-7cfb-4e7f-a40b-23d50447404a]

help?> Matematica.tabuada
  Matematica.tabuada(n)

  Mostra tabuada de multiplicação dos valores de 1 até 10. Valores fora da faixa permitida geram a apresentação de uma
  mensagem de advertência.

  ____________________________________________________________________________________________________________________

  Exemplos
  ----------

  julia> Matematica.tabuada(10)
  10 x  1 =  10
  10 x  2 =  20
  10 x  3 =  30
  10 x  4 =  40
  10 x  5 =  50
  10 x  6 =  60
  10 x  7 =  70
  10 x  8 =  80
  10 x  9 =  90
  10 x 10 = 100

  julia> Matematica.tabuada(0)
  Atenção: Use valores de 1 até 10.
```

Figura 7.3 - Efeito da definição do modo ajuda para função.

Veja que a figura 7.3 apresenta o modo de ajuda com efeitos visuais dentro do padrão da linguagem Julia definidos entre as linhas "**5**" até "**34**". Nessas linhas encontra-se a definição dos símbolos de aspas inglesas triplas (**"""**), os quais determinam o bloco principal do texto que é apresentado quando solicitado o modo de ajuda para função definida imediatamente abaixo do bloco do texto do bloco, neste caso a função indicada na linha "**35**".

Na linha "**6**" encontra-se a definição de um sub-bloco que possibilita a indicação de textos escritos linearmente a cada parágrafo. O código de abertura de sub-bloco (**```julia**) da linha "**6**" é encerrado na linha "**8**" com o código (**```**), ou seja, opera o sub-bloco definido, neste caso, na linha "**7**". Este código escreve o texto do bloco em azul. Veja que sub-blocos são definidos com o auxílio de três acentos graves.

O texto definido entre as linhas "**10**" e "**12**" é escrito de forma livre linearmente tendo algumas de suas partes grafadas em azul para destacá-los. O destaque das partes do texto em azul são definidos com os símbolos três graves (**```**) antes e depois ao destaque. Não confunda a definição de destaques com a definição de sub-blocos. Sub-blocos para serem definidos são iniciados com o código (**```julia**).

Na linha "**14**" há a definição do código três sinais de menos "**---**" que processa a indicação de linha sublinhada usada para dividir o trecho de explicação da ajuda do trecho de demonstração do recurso.

Os símbolos três tralhas "**###**" antes do texto "**Exemplos**" na linha "**16**" ocasiona a apresentação de linha tracejada abaixo do texto formado por dez traços sugerindo um tópico dentro do texto de explicação.

Agora que o pacote está praticamente desenvolvido e documentado com um modo ajuda pode-se passar para a fase de testes. Neste sentido é importante caso não tenha instalado o pacote de testes "**Test**", fazê-lo. Para tanto, execute a instrução:

```
julia> Pkg.add("Test")
    Updating registry at `C:\Users\Augusto\.julia\registries\General.toml`
   Resolving package versions...
    Updating `C:\Users\Augusto\Matematica\Project.toml`
  [8dfed614] + Test
    Updating `C:\Users\Augusto\Matematica\Manifest.toml`
  [2a0f44e3] + Base64
  [b77e0a4c] + InteractiveUtils
  [56ddb016] + Logging
  [d6f4376e] + Markdown
  [9a3f8284] + Random
  [ea8e919c] + SHA
  [9e88b42a] + Serialization
  [8dfed614] + Test
```

O recurso de teste é uma atividade que visa validar logicamente as funcionalidades de um pacote desenvolvido verificando se não nele alguma falha. Com isso busca-se minimizar problemas de execução e erros do pacote caso venha este venha a ser publicado para ser usado por terceiros. Pensando nisso é necessário fazer alguns ajustes no código da função "**tabuada**". Assim sendo, atente para os trechos sinalizados em vermelho e proceda as mudanças no arquivo "**Matematica.jl**".

```
 1  module Matematica
 2
 3  export tabuada
 4
 5  using Printf
 6
 7
 8  """
 9  ```julia
10  Matematica.tabuada(n)
11  ```
12
13  Mostra ```tabuada``` de multiplicação dos valores de ```1``` até
14  ```10```. Valores fora da faixa permitida geram a apresentação de uma
15  mensagem de advertência.
16
17  ---
18
19  ### Exemplos
20
21  ```julia
22  julia> Matematica.tabuada(10)
23  10 x  1 =  10
24  10 x  2 =  20
25  10 x  3 =  30
26  10 x  4 =  40
27  10 x  5 =  50
28  10 x  6 =  60
```

```
29  10 x  7 =  70
29  10 x  8 =  80
30  10 x  9 =  90
31  10 x 10 = 100
32  "Concluído com sucesso."
33
34  julia> Matematica.tabuada(0)
35  Atenção: Use valores de 1 até 10.
36  "Entrada inválida."
37  ```
38  """
39  function tabuada(n::Int)::String
40      if n >= 1 && n <= 10
41          for i in 1:10
42              @printf("%2i x %2i = %3i\n", n, i, n * i)
43          end
44          return "Concluído com sucesso."
45      else
46          printstyled("Atenção: "; color=:yellow)
47          println("Use valores de 1 até 10.")
48          return "Entrada inválida."
49      end
50  end
51
52  end # module
```

Reinicie novamente o ambiente Julia para poder ser possível ver a ocorrência das novas mudanças. Execute as instruções:

```
julia> using Pkg

julia> Pkg.activate("Matematica")
  Activating project at `C:\Users\Augusto\Matematica`

julia> using Matematica
[ Info: Precompiling Matematica [4008b8b7-7cfb-4e7f-a40b-23d50447404a]

julia> Matematica.tabuada(0)
Atenção: Use valores de 1 até 10.
"Entrada inválida."

julia> Matematica.tabuada(9)
 9 x  1 =   9
 9 x  2 =  18
 9 x  3 =  27
 9 x  4 =  36
 9 x  5 =  45
 9 x  6 =  54
 9 x  7 =  63
 9 x  8 =  72
 9 x  9 =  81
 9 x 10 =  90
"Concluído com sucesso."
```

Para realizar os testes necessários é importante criar no diretório do projeto, neste caso, "**Matematica**" um diretório adicional chamado, obrigatoriamente, "**test**" e dentro deste diretório criar um arquivo chamado "**runtests.jl**". As-

sim sendo, crie manualmente o diretório "**test**", abra o editor de textos de sua preferência e escreva as próximas linhas de código gravando-as no arquivo "**runtests.jl**" dentro do diretório "**test**".

Arquivo "**runtests.jl**":

```
 1  # Execução de testes - personalizado
 2
 3  using Matematica, Test
 4
 5  function meuTeste()
 6      @testset "Teste de tabuadas" begin
 7          @test Matematica.tabuada(-1) == "Entrada inválida."
 8          @test Matematica.tabuada(0)  == "Entrada inválida."
 9          @test Matematica.tabuada(11) == "Entrada inválida."
10          @test Matematica.tabuada(11) == "Entrada inválida."
11          @test Matematica.tabuada(10) == "Concluído com sucesso."
12          @test Matematica.tabuada(5)  == "Concluído com sucesso."
13      end
14  end
15
16  meuTeste()
```

O arquivo de programa "**runtests.jl**" é usado pelo gerenciador de pacotes para realizar testes operacionais sobre a função em foco. O gerenciador de pacotes executa cada um dos testes apontados e verifica se os retornos lógicos das ações ocorrem como previstos, tanto para respostas "**true**" como para respostas "**false**" e indica um relatório com os detalhes encontrados.

Veja que na linha "**3**" é realizada a chamada do pacote em desenvolvimento "**Matematica**" e o pacote "**Test**" anteriormente adicionado. O pacote "**Test**" é o recurso que fará todo o processo de validação de aplicação das funcionalidades do pacote em desenvolvimento e apresentará um relatório com todos os detalhes ocorridos, inclusive se houver alguma inconsistência.

Para a realização dos testes é definida entre as linhas "**5**" e "**14**" uma função chamada "**meuTeste**" que é disparada a partir da linha "**16**". Esta função possui definido um bloco de testes controlado por "**@testset**" que cria uma área de escopo para a execução de um conjunto de diversos testes pré-agendados. Esse recurso gera um relatório ao final da avaliação usando como rótulo de identificação o comentário definido entre aspas indicado na linha "**6**", ou seja, "**Teste de tabuadas**". Os testes isolados são processados por "**@test**" que é um macro do tipo lógico que executa os testes estabelecidos.

A partir deste instante execute a instrução seguinte e observe os detalhes apresentados:

```
julia> Pkg.test()
     Testing Matematica
      Status `C:\Users\Augusto\AppData\Local\Temp\jl_cm0TQH\Project.toml`
  [98e4c6a4] Matematica v0.1.0 `C:\Users\Augusto\Matematica`
  [de0858da] Printf `@stdlib/Printf`
  [8dfed614] Test `@stdlib/Test`
      Status `C:\Users\Augusto\AppData\Local\Temp\jl_cm0TQH\Manifest.toml`
  [98e4c6a4] Matematica v0.1.0 `C:\Users\Augusto\Matematica`
  [2a0f44e3] Base64 `@stdlib/Base64`
  [b77e0a4c] InteractiveUtils `@stdlib/InteractiveUtils`
  [56ddb016] Logging `@stdlib/Logging`
  [d6f4376e] Markdown `@stdlib/Markdown`
  [de0858da] Printf `@stdlib/Printf`
  [9a3f8284] Random `@stdlib/Random`
  [ea8e919c] SHA `@stdlib/SHA`
  [9e88b42a] Serialization `@stdlib/Serialization`
  [8dfed614] Test `@stdlib/Test`
  [4ec0a83e] Unicode `@stdlib/Unicode`
```

```
     Testing Running tests...
Atenção: Use valores de 1 até 10.
Atenção: Use valores de 1 até 10.
Atenção: Use valores de 1 até 10.
Atenção: Use valores de 1 até 10.
10 x  1 =  10
10 x  2 =  20
10 x  3 =  30
10 x  4 =  40
10 x  5 =  50
10 x  6 =  60
10 x  7 =  70
10 x  8 =  80
10 x  9 =  90
10 x 10 = 100
 5 x  1 =   5
 5 x  2 =  10
 5 x  3 =  15
 5 x  4 =  20
 5 x  5 =  25
 5 x  6 =  30
 5 x  7 =  35
 5 x  8 =  40
 5 x  9 =  45
 5 x 10 =  50
Test Summary:     | Pass  Total
Teste de tabuadas |    6      6
     Testing Matematica tests passed
```

Há um detalhe a ser levado em consideração em relação ao uso de pacotes, pois existem três comportamentos operacionais de acesso possíveis, sendo:

1. Um pacote pode ser interno e estar disponível internamente para uso como parte da linguagem e neste caso para usá-lo basta utilizar a instrução "**`using NomePacote`**";
2. Um pacote pode ser externo e adquirido a partir de um repositório oficial de suporte à linguagem Julia e neste caso para usá-lo é necessário instalá-lo com as instruções "**`using Pkg`**", "**`Pkg.add("NomePacote")`**" e depois para colocar em uso usar a instrução "**`using NomePacote`**";
3. Um pacote pode ser desenvolvido pelo usuário da linguagem Julia e neste caso para usá-lo é necessário executar as instruções "**`using Pkg`**", "**`Pkg.activate("NomePacote")`**" e depois para colocar em uso usar a instrução "**`using NomePacote`**".

Com a execução dos passos apresentados neste tópico você tem em mãos os recursos mínimos necessários para criar seus próprios pacotes. Para mais detalhes consulte o endereço **https://pkgdocs.julialang.org/v1/creating-packages/**.

7.4 - Definição automatizada de pacotes

Como você deve ter percebido um pacote pode ser entendido como um conjunto de módulos de programas e arquivos de configuração, além da definição da estrutura de diretórios que o compõe. No tópico anterior parte do trabalho de desenvolvimento de um pacote foi apresentado manualmente. Criar pacotes de maneira manual, mesmo para projetos pequenos torna-se tarefa dispendiosa e cansativa, pois são muitos os passos a serem concluídos e regras que devem ser executadas e seguidas.

Por isso o ecossistema Julia possui uma ferramenta de auxílio a esta operação chamada "**`PkgTemplate`**" que automatiza todos os tramites de montagem e criação, preparando o conjunto de arquivos para uma possível submissão a plataforma "**`Github`**" se assim for o desejo de disponibilizar seu trabalho a uma extensa comunidade.

Para "**PkgTemplate**" funcionar adequadamente você precisa ter acesso a uma conta "**Git**" devidamente configurada em seu computador a partir do uso, por exemplo, do programa "**Git bash**". Informação da configuração inicial deste recurso estão disponíveis no endereço **https://git-scm.com/book/en/v2/Getting-Started-First-Time-Git-Setup**.

Para obter o programa "**Git bash**" acesse o endereço **https://git-scm.com/** e siga as orientações apresentadas no sítio ou consulte o apêndice "**B**" desta obra.

Para a demonstração de criação de um pacote de forma parcialmente automatizada considere como exemplo o desenvolvimento de pacote com funcionalidades de manipulação de datas de calendário chamado "**DataExemplo**" que valida o fornecimento de datas entre "01/01/0001" até "31/12/9999". Para tanto, em uma nova instância de uso do ambiente Julia, execute as instruções:

```
julia> using Pkg

julia> Pkg.add("PkgTemplates")
    Updating registry at `C:\Users\Augusto\.julia\registries\General.toml`
   Resolving package versions...
   Installed Mustache ─────── v1.0.14
   Installed PkgTemplates ─ v0.7.29
    Updating `C:\Users\Augusto\.julia\environments\v1.7\Project.toml`
  [14b8a8f1] + PkgTemplates v0.7.29
    Updating `C:\Users\Augusto\.julia\environments\v1.7\Manifest.toml`
  [ffc61752] + Mustache v1.0.14
  [d96e819e] + Parameters v0.12.3
  [14b8a8f1] + PkgTemplates v0.7.29
  [3a884ed6] + UnPack v1.0.2
Precompiling project...
  2 dependencies successfully precompiled in 5 seconds (165 already precompiled)

julia> using PkgTemplates
```

Lembre-se de que na próxima vez bastará fazer uso da instrução "**using PkgTemplates**" e para a criação assistida de um pacote execute a instrução:

```
julia> PkgTemplates.generate("DataExemplo")
Template keywords to customize:
[press: d=done, a=all, n=none]
 > [ ] user
   [ ] authors
   [ ] dir
   [ ] host
   [ ] julia
   [ ] plugins
```

O modo de geração automatizada de criação de um pacote é interativo. Neste instante se você fizer uso das setas para cima e para baixo verá o sinal "**>**" sendo posicionado para cima e para baixo em determinada opção de seleção.

Caso queira selecionar todas as opções basta acionar a letra "**A**", para desmarcar todas as opções acione a letra "**N**" e para prosseguir após as seleções realizadas use a letra "**D**". Caso queira selecionar seletivamente as opções utilize a tecla "**<Enter>**" sobre a opção deseja, tanto para habilitar como para desabilitar. Assim sendo, para esta demonstração selecione todas as opções, exceto "**host**" e acione na sequência a letra "**D**".

Veja que neste instante é solicitada a entrada da informação para o campo "**user**". Neste momento informe o seu nome de acesso junto a sua conta do "**Github**".

Na sequência é solicitada a entrada do nome ou nomes para o campo "**authors**". Você pode manter a informação apresentada ou informar seu nome ou os nomes separados por vírgulas de outros autores que estejam com você neste projeto. Entre, por exemplo, seu nome e acione a tecla "**<Enter>**".

A próxima configuração é a definição do diretório de desenvolvimento para o campo "**user**". Neste momento informe um diretório desejado ou acione ponto "**.**" para manter a forma padrão (mais recomendado). Neste caso acione a tecla "**<.>**".

Agora chegou a hora de escolher a versão para a qual o pacote será direcionado. Considerando o uso da versão "**1.7**", movimente o símbolo "**>**" até o valor desejado e acione "**<Enter>**".

Chegou a hora de selecionar os *plugins* a serem usados. Além dos *plugins* que já estão marcados selecione as opções "**Codecov**", "**Documenter**" e "**GitHubActions**". Use as setas para cima e para baixo e "**<Enter>**" para marcar a seleção. Após selecionar os itens apontados acione a letra "**D**".

Na sequência é necessário configurar cada um dos *plugins* selecionados. Neste sentido, podem ser realizadas alterações específicas ou pode-se manter as opções padrão de cada *plugin* apresentado. Para manter as opções padrão use a letra "**D**", caso queira fazer alguma alteração selecione a sub-opção desejada e efetue a ação pretendida.

Veja a seguir as configurações para os *plugins* selecionados:

- Para o *plugin* "**CompatHelper**", selecione a letra "**D**", mantendo-se suas opções padrão;
- Para o *plugin* "**ProjectFile**", selecione a letra "**D**", mantendo-se suas opções padrão;
- Para o *plugin* "**SrcDir**", selecione a letra "**D**", mantendo-se suas opções padrão;
- Para o *plugin* "**Git**", selecione a letra "**D**", mantendo-se suas opções padrão;
- Para o *plugin* "**License**", marque "**destination**" e "**name**", selecione a letra "**D**", na sequência logo abaixo entre o termo "**license.md**" para a opção "**destination**" acionando "**<Enter>**", seleciona uma das licenças disponibilizadas com as setas para cima e para baixo e acione a tecla "**<Enter>**" sobre a licença escolhida. Não sabendo qual licença escolher opte pela licença "**MIT**";
- Para o *plugin* "**Readme**", selecione a letra "**D**", mantendo-se suas opções padrão;
- Para o *plugin* "**Tests**", selecione a letra "**D**", mantendo-se suas opções padrão;
- Para o *plugin* "**TagBot**", selecione a letra "**D**", mantendo-se suas opções padrão;
- Para o *plugin* "**Codecov**", selecione a letra "**D**" e será indicado a seleção do estilo para documentação, selecione então a opção "**GitHubActions**" para coerência com o *plugin* selecionado de mesmo nome e acione "**<Enter>**";
- Para o *plugin* "**Documenter**", selecione a letra "**D**", mantendo-se suas opções padrão;
- Para o *plugin* "**GitHubActions**", selecione a letra "**D**", mantendo-se suas opções padrão.

Em seguida será executado todo o processo de configuração e criação do espaço para o desenvolvimento do pacote que poderá ser demorado dependendo das opções selecionadas. Observe que são executadas diversas operações que sem a ferramenta "**PkgTemplate**" teriam que ser realizadas manualmente.

Por curiosidade acesse o diretório "**DataExemplo**" definido a partir da área de trabalho do usuário e veja os diretórios e arquivos que foram criados.

A partir deste instante torna-se necessário fazer a codificação das funções para o pacote "**DataExemplo**". Assim sendo, abra com seu editor de textos o arquivo de código de programa "**DataExemplo.jl**" localizado a partir do diretório "**\Usuário\DataExemplo\src**" que se parece com o trecho de código a seguir:

Arquivo "**DataExemplo.jl**":

```
1  module DataExemplo
2
3  # Write your package code here.
4
5  end
```

Desta forma, a partir da linha "**3**" basta entrar a sequência de funcionalidades que fazem o controle e tratamento operacional das datas de um calendário. Veja a seguir como deverá ficar o código do pacote.

```julia
module DataExemplo

export mostraData

using Printf

struct Data
    dia::Int8
    mês::Int8
    ano::Int16
end

function põeData(dia, mês, ano)::Data
   return Data(dia, mês, ano)
end

function anoBissexto(ano)::Bool
    if ano % 400 == 0
        return true
    end
    if ano % 4 == 0 && ano % 100 != 0
        return true
    end
    return false
end

function últimoDiaMês(mês, ano)::Int8
    if mês == 2  # Fev
        return anoBissexto(ano) ? 29 : 28
    end
    if mês <= 7  # Jul
        return mês % 2 != 0 ? 31 : 30
    end
    return mês % 2 != 0 ? 30 : 31
end

function validaData(dia, mês, ano)::Bool
    return ano >=    1 &&
           ano <= 9999 &&
           mês >=    1 &&
           mês <=   12 &&
           dia >=    1 &&
           dia <= últimoDiaMês(mês, ano)
end

"""
```julia
DataExemplo.mostraData(dd,mm,aaaa)
```

Mostra ```data calendário``` validada e formatada.

Exemplos

```julia
julia> mostraData(29,2,1944)
29/02/1944

julia> mostraData(29,2,1945)
**/**/****
```
```
```
```

```
64  """
65  function mostraData(dia, mês, ano)
66      data = põeData(dia, mês, ano)
67      if validaData(data.dia, data.mês, data.ano)
68          @printf("%02i", data.dia)
69          print("/")
70          @printf("%02i", data.mês)
71          print("/")
72          @printf("%04i", data.ano)
73          println()
74      else
75          println("**/**/****")
76      end
77  end
78
79  end
```

Observe o código definido entre as linhas "**3**" e "**77**". Em especial na linha "**3**" há a indicação de que a única funcionalidade do pacote a ser usada externamente é a função "**mostraData**". As demais funcionalidades do pacote são visíveis para uso, sendo válidas apenas internamente para auxilio da função exportada. Esse efeito é uma maneira de se estabelecer encapsulamento público e privado em linguagem Julia.

Na linha "**5**" encontra-se a definição do pacote de dependência para o pacote em desenvolvimento, em especial para auxilio da funcionalidade da função "**mostraData**".

Entre as linhas "**7**" e "**11**" há a definição da estrutura "**Data**" definida para modelar o conjunto de elementos que compõem uma data, sendo: dia, mês e ano.

Entre as linhas "**13**" e "**15**" há a definição da funcionalidade "**põeData**" que define um mecanismo de entrada dos dados de uma data formatando o conjunto de dados dentro do escopo da estrutura definida entres as linhas "**7**" e "**11**". Veja que esta função retorna seu conteúdo como sendo do tipo "**Data**".

A função "**anoBissexto**" definida entre as linhas "**17**" e "**25**" verifica se o ano de certa data é bissexto. Para saber se um ano é bissexto, basta verificar se o ano é divisível por "**400**" ou se é divisível por "**4**", mas não divisível por "**100**".

Entre as linhas "**26**" e "**35**" encontra-se a definição da função "**últimoDiaMês**" que calcula o último dia válido de certo mês. Se o mês for fevereiro, terá "**28**" dias, exceto se o ano for bissexto, quando terá "**29**" dias. Se o mês estiver entre janeiro e julho terá "**31**" dias, caso seja um mês ímpar; e terá "**30**" dias, caso seja um mês par. Para os meses entre agosto e dezembro, o mês terá "**31**" dias, se par; e "**30**" dias, se ímpar.

A função "**validaData**" determina "**true**" (verdadeiro) se a data fornecida for válida e estiver dentro do limite estabelecido. Será válida uma data fornecida na faixa de anos entre "**1**" e "**9999**", na faixa de meses entre "**1**" e "**12**" e na faixa de dias de "**1**" até o último dia do mês detectado pela função "**últimoDiaMês**".

Entre as linhas "**65**" e "**77**" é definida a sub-rotina em estilo procedimento chamada "**mostraData**" que não retorna nenhum valor calculado, mas mostra a data fornecida no formato "**DD/MM/AAAA**" se for uma data válida e sendo inválida, apresenta "****/**/******".

Após realizar a codificação do arquivo do pacote "**DataExemplo.jl**" com as funcionalidades de manipulação de data calendário reinicie o ambiente e siga as próximas instruções para carregar o pacote e proceder alguns testes operacionais:

```
julia> using Pkg

julia> Pkg.activate("DataExemplo")
  Activating project at `C:\Users\Augusto\DataExemplo`

julia> Pkg.add("Printf")  # acrescentar o pacote dependente no pacote em desenvolvimento
    Updating registry at `C:\Users\Augusto\.julia\registries\General.toml`    Updating
    ...
  1 dependency successfully precompiled in 'n' seconds
```

```
julia> using DataExemplo

julia> DataExemplo.mostraData(29,2,1944)
29/02/1944

julia> mostraData(29,2,1944)
29/02/1944

julia> mostraData(26, 4, 1965)
26/04/1965

julia> mostraData(1, 1, 1)
01/01/0001

julia> mostraData(31, 4, 2022)
**/**/****

julia> mostraData(29, 02, 2022)
**/**/****

julia> mostraData(28, 02, 2022)
28/02/2022

julia> mostraData(10, 12, 10_000)
**/**/****
```

Observe junto aos testes operacionais realizados que qualquer data informada fora dos limites operacionais aceitos é apresentada de forma truncada com o estilo "****/**/******".

Agora é importante preparar os recursos de teste para validação realizados pela própria ferramenta. Para tanto, será criada dentro do arquivo de código "**DataExemplo.jl**" uma função chamada "**testeData**" sem modo ajuda para validar alguns testes com as funcionalidades de auxílio a checagem de datas que não foram exportadas. Assim sendo, atente para as partes indicadas em vermelho no código "**DataExemplo.jl**":

```
1  module DataExemplo
2
3  export mostraData, testeData
4
5  using Printf
6
7  struct Data
8      dia::Int8
9      mês::Int8
10     ano::Int16
11 end
12
13 function põeData(dia, mês, ano)::Data
14    return Data(dia, mês, ano)
15 end
16
17 function anoBissexto(ano)::Bool
18     if ano % 400 == 0
19         return true
20     end
21     if ano % 4 == 0 && ano % 100 != 0
```

```
22          return true
23      end
24      return false
25  end
26
27  function últimoDiaMês(mês, ano)::Int8
28      if mês == 2  # Fev
29          return anoBissexto(ano) ? 29 : 28
30      end
31      if mês <= 7  # Jul
32          return mês % 2 != 0 ? 31 : 30
33      end
34      return mês % 2 != 0 ? 30 : 31
35  end
36
37  function validaData(dia, mês, ano)::Bool
38      return ano >=    1 &&
39             ano <= 9999 &&
40             mês >=    1 &&
41             mês <=   12 &&
42             dia >=    1 &&
43             dia <= últimoDiaMês(mês, ano)
44  end
45
46  """
47  ```julia
48  DataExemplo.mostraData(dd,mm,aaaa)
49  ```
50
51  Mostra ```data calendário``` validada e formatada.
52
53  ---
54
55  ### Exemplos
56
57  ```julia
58  julia> mostraData(29,2,1944)
59  29/02/1944
60
61  julia> mostraData(29,2,1945)
62  **/**/****
63  ```
64  """
65  function mostraData(dia, mês, ano)
66      data = põeData(dia, mês, ano)
67      if validaData(data.dia, data.mês, data.ano)
68          @printf("%02i", data.dia)
69          print("/")
70          @printf("%02i", data.mês)
72          print("/")
72          @printf("%04i", data.ano)
73          println()
74      else
75          println("**/**/****")
76      end
77  end
78
79  function testeData(dia, mês, ano)::Bool
80      if validaData(dia, mês, ano)
81          return true
82      end
83      return false
84  end
```

```
85
86 end
```

Para fazer um teste rápido operacional da função "**testeData**" recém criada reinicie o ambiente Julia e execute as seguintes instruções:

```
julia> using Pkg

julia> Pkg.activate("DataExemplo")
  Activating project at `C:\Users\Augusto\DataExemplo`

julia> using DataExemplo
[ Info: Precompiling DataExemplo [baa560f1-2cfc-4222-8843-5694590054be]

julia> testeData(29,2,1944)
true

julia> testeData(29,2,1945)
false
```

Com seu editor de texto preferido carregue para a memória o arquivo "**runtests.jl**" existente no diretório "**test**". Observe seu conteúdo:

Arquivo "**runtests.jl**":

```
1 using DataExemplo
2 using Test
3
4 @testset "DataExemplo.jl" begin
5     # Write your tests here.
6 end
```

A partir da linha "**5**" escreva o código de teste para validação do pacote. Assim sendo, observe como deverá ficar a estrutura do arquivo "**runtests.jl**".

```
1  using DataExemplo
2  using Test
3
4  @testset "DataExemplo.jl" begin
5      @test testeData(29,02,1944) == true
6      @test testeData(29,02,1945) == false
7      @test testeData(29,02,2024) == true
8      @test testeData(29,02,1945) == false
9      @test testeData(30,02,2048) == false
10     @test testeData(00,00,0000) == false
11     @test testeData(00,01,0000) == false
12     @test testeData(00,00,0001) == false
13     @test testeData(00,01,0001) == false
14     @test testeData(01,01,0001) == true
15     @test testeData(26,04,1965) == true
16     @test testeData(32,08,2022) == false
17     @test testeData(05,13,2000) == false
18 end
```

A partir deste instante execute a instrução seguinte e observe os detalhes apresentados:

```
julia> Pkg.test()
```

A partir deste ponto tem-se em mãos os recursos para criação de pacotes mais simplificado do que a criação manual. O modo automatizado de criação de pacotes trata de toda a parte burocrática de desenvolvimento de arquivos e estruturas deixando apenas a parte lógica para desenvolvimento pelo usuário da linguagem.

Para realizar a submissão do pacote "**`DataExemplo.jl`**" encerre o ambiente Julia e siga as orientações do próximo tópico. Lembre-se de que para conseguir subir os arquivos é necessário que o pacote não esteja em uso no REPL da linguagem Julia.

7.5 - Submissão de pacotes no Github

A partir do desenvolvimento de pacotes locais, preferencialmente com "**`PkgTemplate`**" que torna a tarefa mais fácil, é possível fazer a submissão junto a plataforma "**`Github`**", desde que sejam seguidos alguns passos. O "**`Github`**" é um grande repositório de projetos de software e armazenamento de outros documentos com controle de versão que utiliza o serviço "**`Git`**".

Partindo-se do pressuposto de que o leitor tem conta "**`Github`**" faça acesso ao endereço de sua conta identificado semelhantemente como **https://github.com/NomeUsuário**. Caso não tenha uma conta esta é uma grande oportunidade de fazê-lo. Com o "**`Github`**" aberto crie um repositório com o mesmo nome do projeto desenvolvido. Neste caso, será criado um repositório chamado "**`DataExemplo.jl`**". Para tanto, selecione a opção "**`Repositories`**" indicada com um quadro amarelo na figura 7.4. Quando aberta a página de repositórios acione o botão "**`New`**" como mostra a figura 7.5.

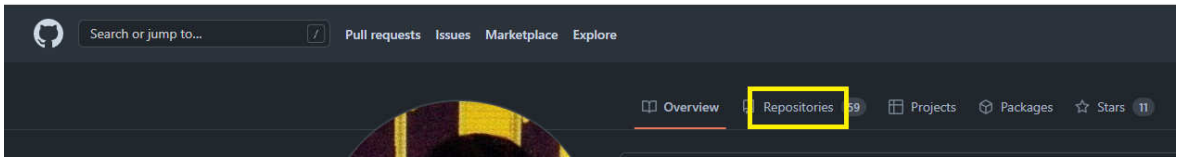

Figura 7.4 - Seleção da opção "Repositories".

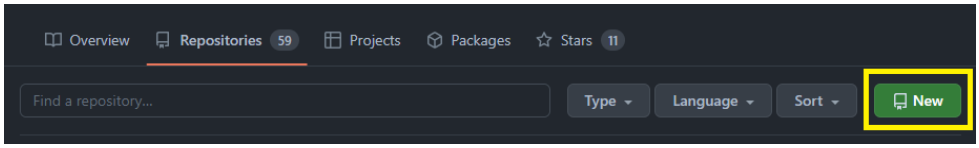

Figura 7.5 - Seleção do botão "New".

Será apresenta a página para a criação de repositório. Neste momento escreva ao lado do nome de seu usuário o nome do repositório "**`DataExemplo.jl`**" como mostra a figura 7.6.

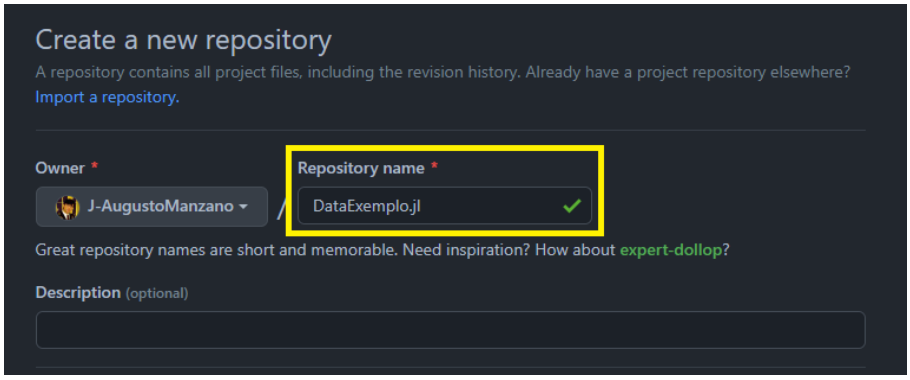

Figura 7.6 - Definição do repositório "DataExemplo.jl".

Não altere nenhum outro detalhe, mantenha todas as opções desta página exatamente como estão apresentadas. Acione na sequência, mais abaixo na mesma página, o botão "**`Create repository`**" e será apresentada a página do repositório criado como indicado junto a figura 7.7. Copie e guarde, por hora, o endereço do repositório recém-criado e apresentado. Neste caso "**https://github.com/J-AugustoManzano/DataExemplo.jl.git**", ressaltando que em seu sistema o nome de usuário será outro.

Na sequência selecione com o ponteiro do *mouse* o link "**`.gitignore`**" como indicado na figura 7.8 e na página apresentada selecione o botão "**`Choose .gitignore:`**" sinalizado como "**None**" como apresentado na figura 7.9. Junto a figura 7.9 selecione o pequeno triângulo ao lado de "**Choose .gitignore: None**" e selecione na lista apresenta a linguagem "**`Julia`**" como mostra a figura 7.10. Em seguida role a página para baixo e acione com o ponteiro do *mouse* o botão "**`Commit new file`**".

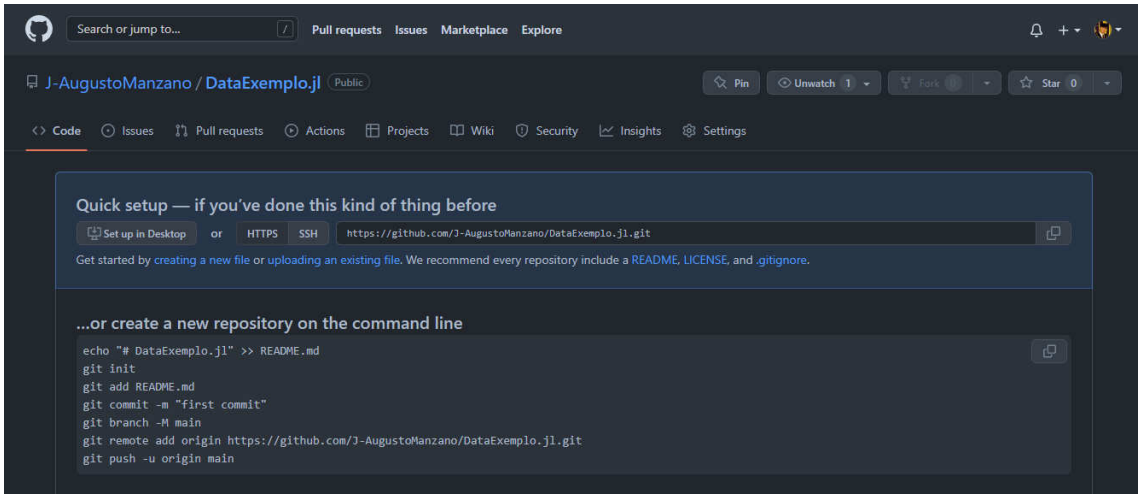

Figura 7.7 - Repositório "DataExemplo.jl" definido.

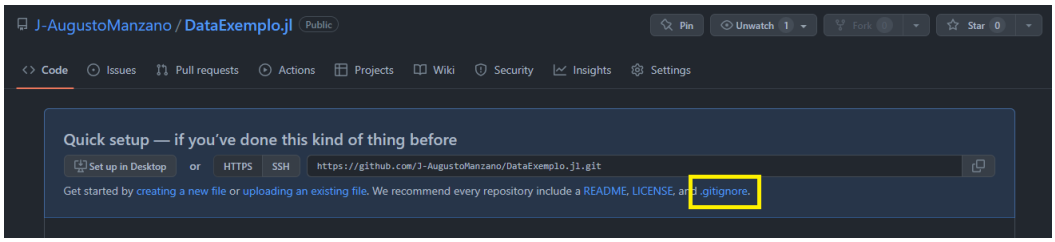

Figura 7.8 - Seleção do link ".gitignore".

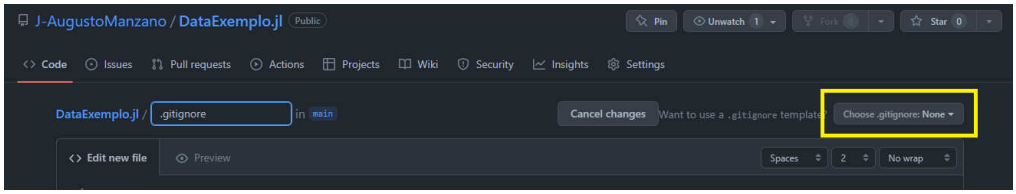

Figura 7.9 - Seleção do botão "Choose .gitignore: None".

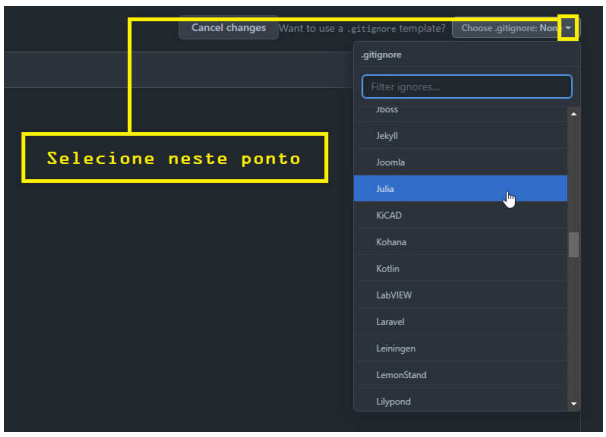

Figura 7.10 - Seleção do ".gitignore" para linguagem Julia.

O primeiro arquivo no repositório foi definido. Agora é hora de postar o arquivo de licença, pois para que um pacote seja publicado este deve ser obrigatoriamente público e uma licença padrão *open-source* deve ser selecionada. Para tanto, acione o botão "**`Add file`**" e nele a opção "**`Create new file`**" como demonstra a figura 7.11.

Na página apresenta escreva "**license**" no campo ao lado de "**DataExemplo.jl**" e selecione ao lado direito o botão "**Choose a licence template**" como apresenta a figura 7.12 e escolha uma das licenças apresentadas. Selecione a mesma opção de licença usada no recurso "**PkgTemplate**" para o projeto de pacote "**DataExemplo.jl**", ou seja, selecione a opção "**MIT License**" indicada na figura 7.13. Após selecionar a licença de sua preferência acione o botão "**Review and submit**", role a página para baixo e acione o botão "**Commit new file**". Se apresentada a solicitação "**Open a pull request**" acione o botão "**Create pull request**". Na sequência acione o botão "**Merge pull request**" e em seguida acione o botão "**Confirm merge**".

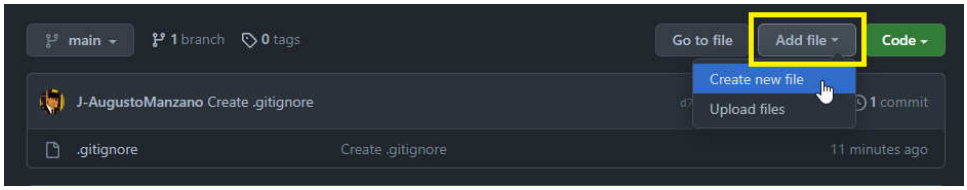

Figura 7.11 - Seleção da opção "Add file / Create new file".

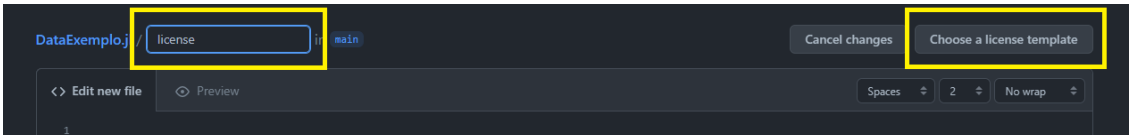

Figura 7.12 - Entrada do termo "license" e seleção do botão "Choose a licence template".

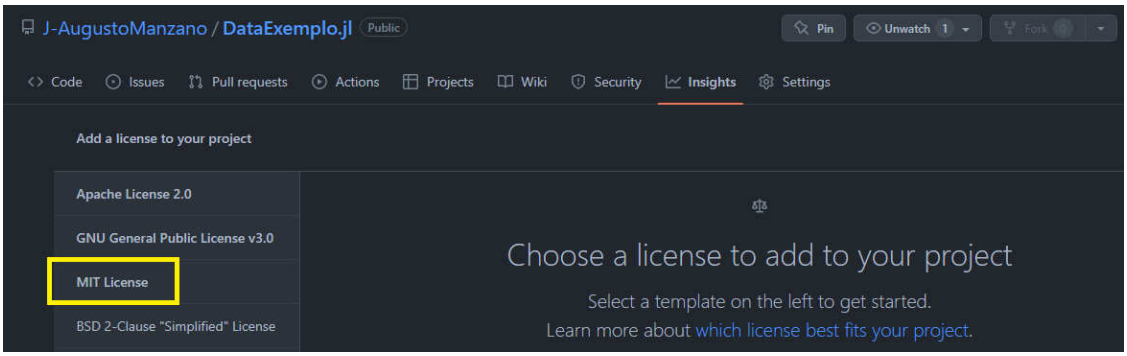

Figura 7.13 - Seleção da licença "MIT License".

A partir desses passos o repositório está preparado para receber os arquivos do pacote desenvolvido. No repositório recém definido acione o botão "**Add file**", selecione a opção "**Upload files**" e copie todos os arquivos existentes no diretório do seu sistema "**DataExemplo.jl**" e em seguida acione o botão "**Commit new file**".

Na sequência reinicie o ambiente Julia se estiver aberto ou abra-o para uso e execute as seguintes instruções para que a próxima etapa surta os efeitos esperados:

```
julia> ]

(@1.7) pkg> dev https://github.com/J-AugustoManzano/DataExemplo.jl.git
     Cloning git-repo `https://github.com/J-AugustoManzano/DataExemplo.jl.git`
Path `C:\Users\Augusto\.julia\dev\DataExemplo` exists and ... correct repo. Using existing path.
   Resolving package versions...
    Updating `C:\Users\Augusto\.julia\environments\v1.7\Project.toml`
  [baa560f1] + DataExemplo v0.1.0 `C:\Users\Augusto\.julia\dev\DataExemplo`
    Updating `C:\Users\Augusto\.julia\environments\v1.7\Manifest.toml`
  [baa560f1] + DataExemplo v0.1.0 `C:\Users\Augusto\.julia\dev\DataExemplo`
```

A partir deste momento os arquivos do projeto de pacote desenvolvidos e publicados no "**Github**" são clonados para dentro do diretório "**.julia**" a partir de um diretório chamado "**dev**" criado automaticamente, nesta operação, que possui uma cópia do pacote disponibilizado no "**Github**". A partir dessa ocorrência o pacote "**DataExemplo**" poderá ser usado a qualquer instante como qualquer outro pacote padrão ou instalado da linguagem, bastando usar diretamente a instrução "**using DataExemplo**".

Para que essas ações apresentadas funcionem é necessário seguir os passos indicados anteriormente. Para o comando "**`dev`**" funcionar no *prompt* "**`Pkg`**" é necessário existir a publicação do pacote no "**`Github`**" e não no seu sistema. A propósito, assim que fizer a submissão dos arquivos o diretório "**`DataExemplo`**" poderá ser removido de seu sistema, mas antes, por questão de segurança e *backup*, copie o conteúdo para outro local.

O próximo passo pode ser o registro do pacote para uso oficial da comunidade, mas para isso é necessário seguir alguns cuidados que não são apresentados nesta obra, devido ao escopo de primeiros passos na linguagem. Para mais detalhes leia a documentação do endereço: **https://github.com/JuliaRegistries/General**.

ANOTAÇÕES

CAPÍTULO 8

EXEMPLOS DE APLICAÇÃO

São apresentados neste capítulo projetos que demonstram alguns novos recursos e outros abordados ao longo deste trabalho, tanto de forma interativa como na forma programada. Atente para cada exemplo indicado.

8.1 - Numeração arábica e romana

Uma das representações numéricas existentes é o uso de algarismos romanos para determinar a numeração inteira de séculos (século XX), representações de ordem monárquica (D. Pedro II) e em alguns casos a representação de datas para indicação de *copyright* (direitos autorais) na área cinematográfica ou mesmo de *software* (MCMLXV para 1965).

A numeração baseada em algarismos romanos (KÖLLER, 2003), usado na antiga Roma, é formado por um conjunto de sete letras latinas maiúsculas, sendo: I (para 1), V (para 5), X (para 10), L (para 50), C (para 100), D (para 500) e M (para 1000).

A representação gráfica de certo valor numérico ocorre a partir de três considerações:

- Algarismos de maior valor são escritos à frente dos algarismos de menor valor a partir da sequência M, D, C, L, X, V, I;
- Algarismos de igual ou menor valor definidos à direita de certo algarismo são a ele somados;
- Algarismos de menor valor definidos à esquerda de certo algarismo são a ele subtraídos.

A partir da regra de composição numérica o valor romano MC equivale a 1100 (M = 1000 + C = 100), já o valor romano CM equivale a 900 (M = 1000 - C = 100); o valor romano MMM equivale a 3000.

A partir das regras estabelecidas é possível representar valores inteiros positivos entre 1 e 3999. Valores numéricos acima de 4000 são representados com as letras válidas, neste caso IV acompanhado de um travessão superior sobre a grafia do algarismo indicando que este é um múltiplo de 1000.

Apesar de existir certo estilo na grafia de valores romanos não há regras que determinam como os valores devam ser oficialmente representados. Por exemplo, o valor IV (4) pode ser representado como IIII sendo esta forma considerada correta, pois a forma IV chamada de subtrativa é equivalente à forma IIII chamada aditiva. Na antiga Roma a preferência era pela forma aditiva, tendo se popularizado mais a forma subtrativa.

A partir de uma visão básica e considerações sobre a composição e forma de valores escritos com algarismos romanos são apresentados dois programas que operam com esses valores. O primeiro programa recebe a entrada de um valor numérico em algarismo arábico e apresenta seu equivalente em algarismo romano e o segundo programa faz o inverso.

Antes de mostrar os programas para as devidas conversões entre valores arábicos e romanos, cabe observar que aparentemente pode parecer tais programas desnecessários devido a sua aplicação. Considere que não é a conversão em si, mas as técnicas de programação em Julia e os detalhes necessários utilizados para obter as funcionalidades desejadas.

O programa seguinte demonstra a conversão de algarismo arábico em algarismo romano. Para esta ação será desenvolvida uma sub-rotina de função que receberá o valor numérico arábico e o apresentará na forma romana. Grave o código do programa seguinte com o nome "**`c08ex01.jl`**".

```
1  # Definição função de conversão de número arábico em romano
2
3  function ara2rom(vlrArabico::Int64)
4      MIL::Int64 = 0
5      CEN::Int64 = 0
6      DEZ::Int64 = 0
7      UNI::Int64 = 0
8      if vlrArabico >= 1 && vlrArabico <= 3999
9          UNI =    1 * trunc(Int, (vlrArabico /    1 % 10))
10         DEZ =   10 * trunc(Int, (vlrArabico /   10 % 10))
11         CEN =  100 * trunc(Int, (vlrArabico /  100 % 10))
12         MIL = 1000 * trunc(Int, (vlrArabico / 1000 % 10))
13         MILHAR::String  = ""
14         CENTENA::String = ""
15         DEZENA::String  = ""
16         UNIDADE::String = ""
17         if UNI ==    1 UNIDADE =    "I" end
18         if UNI ==    2 UNIDADE =   "II" end
19         if UNI ==    3 UNIDADE =  "III" end
20         if UNI ==    4 UNIDADE =   "IV" end
21         if UNI ==    5 UNIDADE =    "V" end
22         if UNI ==    6 UNIDADE =   "VI" end
23         if UNI ==    7 UNIDADE =  "VII" end
24         if UNI ==    8 UNIDADE = "VIII" end
25         if UNI ==    9 UNIDADE =   "IX" end
26         if DEZ ==   10 DEZENA  =    "X" end
27         if DEZ ==   20 DEZENA  =   "XX" end
28         if DEZ ==   30 DEZENA  =  "XXX" end
29         if DEZ ==   40 DEZENA  =   "XL" end
30         if DEZ ==   50 DEZENA  =    "L" end
31         if DEZ ==   60 DEZENA  =   "LX" end
32         if DEZ ==   70 DEZENA  =  "LXX" end
33         if DEZ ==   80 DEZENA  = "LXXX" end
34         if DEZ ==   90 DEZENA  =   "XC" end
35         if CEN ==  100 CENTENA =    "C" end
36         if CEN ==  200 CENTENA =   "CC" end
37         if CEN ==  300 CENTENA =  "CCC" end
38         if CEN ==  400 CENTENA =   "CD" end
39         if CEN ==  500 CENTENA =    "D" end
40         if CEN ==  600 CENTENA =   "DC" end
41         if CEN ==  700 CENTENA =  "DCC" end
42         if CEN ==  800 CENTENA = "DCCC" end
43         if CEN ==  900 CENTENA =   "CM" end
44         if MIL == 1000 MILHAR  =    "M" end
45         if MIL == 2000 MILHAR  =   "MM" end
46         if MIL == 3000 MILHAR  =  "MMM" end
47         vlrRomano = join([MILHAR, CENTENA, DEZENA, UNIDADE])
48     else
49         vlrRomano = "erro"
50     end
51     return vlrRomano
52 end
53
54 print("Informe valor numérico arábico .: ")
55 valor = parse(Int, readline())
56
57 println()
58 println("Valor numérico em romano .......: ", ara2rom(valor))
59 println()
60
61 print("Tecle <Enter> para encerrar programa... ")
62 readline()
```

Ao ser executado o programa forneça um valor entre "**1**" e "**3999**" para ver seu equivalente em algarismo romano. Caso seja fornecido um valor fora da faixa permitida será retornado pela sub-rotina **ara2rom()** a resposta "**erro**".

Para a conversão do valor arábico em romano a sub-rotina "**ara2rom**" necessita inicialmente decompor o valor arábico em sua estrutura de formação primária, dividida em unidade (variável *UNI*), dezena (variável *DEZ*), centena (variável *CEN*) e milhar (variável *MIL*) realizada pelo trecho de código definido entre as linhas "**9**" e "**12**":

```
UNI =    1 * trunc(Int, (vlrArabico /    1 % 10))
DEZ =   10 * trunc(Int, (vlrArabico /   10 % 10))
CEN =  100 * trunc(Int, (vlrArabico /  100 % 10))
MIL = 1000 * trunc(Int, (vlrArabico / 1000 % 10))
```

Cada uma das linhas do trecho anterior decompõe o valor em sua estrutura de formação primária. Assim sendo, se considerado o valor "**1965**" ocorrerá com a primeira instrução a obtenção do valor "**5**", com a segunda instrução será obtido o valor "**60**", com a terceira instrução obter-se-á o valor "**900**" e por último a quarta instrução obtém o valor "**1000**". Somando-se esses valores "**1000 + 900 + 60 + 5**" ter-se-á "**1965**".

O conjunto de instruções definidas entre as linhas "**13**" e "**16**" definem para cada uma das variáveis valor nulo, ou seja, assumem valores vazios e os mantem se houver algum algarismo zero informado no número arábico.

```
MILHAR::String  = ""
CENTENA::String = ""
DEZENA::String  = ""
UNIDADE::String = ""
```

Após a decomposição do valor arábico fornecido e dos ajustes das variáveis de controle do programa faz-se a verificação de cada componente obtido localizando o algarismo romano correspondente por meio das instruções definidas entras as linhas "**17**" e "**46**".

Entre as linhas "**8**" e "**50**" encontra-se o controle de operação da função. Em especial a instrução da linha "**8**" detecta se o valor arábico informado está dentro do limite de operação, sendo a condição verdadeira o número romano é gerado, mas caso contrário, mensagem de erro é apresentada.

O programa seguinte demonstra a conversão de algarismo romano em algarismo arábico. Para esta ação será desenvolvida uma sub-rotina de função que receberá o valor numérico romano do tipo cadeia e apresentará o valor numérico arábico como inteiro. Grave o código do programa seguinte com o nome "**c08ex02.jl**".

```
 1  # Definição função de conversão de número romano em arábico
 2
 3  function rom2ara(vlrRomano::String)
 4      vlrRomano = uppercase(vlrRomano)
 5      vlrArabico, ATUAL, ANTERIOR, I = 0, 0, 0, 1
 6      while I <= length(vlrRomano)
 7           if vlrRomano[I] == 'I' ATUAL =    1 end
 8           if vlrRomano[I] == 'V' ATUAL =    5 end
 9           if vlrRomano[I] == 'X' ATUAL =   10 end
10           if vlrRomano[I] == 'L' ATUAL =   50 end
11           if vlrRomano[I] == 'C' ATUAL =  100 end
12           if vlrRomano[I] == 'D' ATUAL =  500 end
13           if vlrRomano[I] == 'M' ATUAL = 1000 end
14           if I > 1
15              if vlrRomano[I - 1] == 'I' ANTERIOR =    1 end
16              if vlrRomano[I - 1] == 'V' ANTERIOR =    5 end
17              if vlrRomano[I - 1] == 'X' ANTERIOR =   10 end
18              if vlrRomano[I - 1] == 'L' ANTERIOR =   50 end
19              if vlrRomano[I - 1] == 'C' ANTERIOR =  100 end
20              if vlrRomano[I - 1] == 'D' ANTERIOR =  500 end
21              if vlrRomano[I - 1] == 'M' ANTERIOR = 1000 end
22          end
```

```
23          if ATUAL < ANTERIOR
24               vlrArabico += ATUAL
25          else
26               vlrArabico += ATUAL - ANTERIOR * 2
27          end
28          I += 1
29      end
30      return vlrArabico
31  end
32
33  print("Informe valor numérico romano .: ")
34  valor = readline()
35
36  println()
36  println("Valor numérico em arábico .....: ", rom2ara(valor))
38  println()
39
40  print("Tecle <Enter> para encerrar programa... ")
41  readline()
```

Ao ser executado o programa forneça o número romano "**MCMLXV**" e observe a apresentação do valor equivalente em algarismo arábico "**1965**".

Para a conversão do valor romano em arábico a sub-rotina "**rom2ara**" necessita a partir dos algarismos romanos reconstruir a estrutura de formação primária de um valor arábico. Assim que o valor é recepcionado como argumento na linha "**3**" a linha "**4**" garante que todos os caracteres fornecidos sejam maiúsculos.

A linha "**5**" indica uma forma de atribuição de valores em variáveis a partir de certa definição sequencial. A ordem estabelecida para os valores é a ordem de inicialização de cada variável correspondente.

O trecho de instruções definidos entre as linhas "**7**" e "**21**" convertem cada um dos caracteres dos algarismos romanos em sua forma numérica atribuindo os valores convertidos as variáveis "*ATUAL*" e "*ANTERIOR*" e entre as linhas "**23**" e "**27**" faz-se a validação do estado do valor numérico rom ano, neste sentido se "**if (ATUAL < ANTERIOR)**" se a condição for verdadeira efetua-se o somatório do valor do algarismo romano da variável "*ATUAL*" junto a variável "*vlrArabico*" (efeito aditivo), sendo a condição falsa o valor da variável "*vlrArabico*" é atribuído sucessivamente ao somatório do valor do algarismo romano da variável "*ATUAL*" subtraído do valor da variável "*ANTERIOR*" multiplicado por "**2**" (efeito subtrativo).

A subtração de "*ANTERIOR*" multiplicado por "**2**" se faz necessária uma vez que a variável "*ANTERIOR*" fora anteriormente adicionada ao valor arábico uma vez.

8.2 - Valor por extenso

Neste tópico, o programa exemplo demonstra a forma escrita por extenso de valores numéricos definidos entre a margem de "**0**" e "**999**". O objetivo deste exemplo é apresentar uma forma possível de solução para o problema, não significando em absoluto que seja a única maneira ou a melhor forma de fazê-lo. O programa apresentado poderá ser ampliado para valores em outras magnitudes. Grave o código do programa seguinte com o nome "**c08ex03.jl**".

```
1   # Definição função para geração de valor numérico por extenso
2
3   function vlrExtenso(valor::Int64)
4       CENTENA, C, DEZENA, D, UNIDADE, U = 0, "", 0, "", 0, ""
5       if valor >= 0 && valor <= 999
6           CENTENA = trunc(Int, valor / 100)
7           DEZENA  = valor % 100
8           if DEZENA < 20
9               UNIDADE = DEZENA % 20
10          else
11              UNIDADE = DEZENA % 10
12          end
13          # centenas "n00"
```

```
14          if CENTENA == 9
15              if valor <= 900 C = "novecentos"   else C = "novecentos e "   end end
16          if CENTENA == 8
17              if valor <= 800 C = "oitocentos"   else C = "oitocentos e "   end end
18          if CENTENA == 7
19              if valor <= 700 C = "setecentos"   else C = "setecentos e "   end end
20          if CENTENA == 6
21              if valor <= 600 C = "seissentos"   else C = "seissentos e "   end end
22          if CENTENA == 5
23              if valor <= 500 C = "quinhentos"   else C = "quinhentos e "   end end
24          if CENTENA == 4
25              if valor <= 400 C = "quatrocentos" else C = "quatrocentos e " end end
26          if CENTENA == 3
27              if valor <= 300 C = "trezentos"    else C = "trezentos e "    end end
28          if CENTENA == 2
29              if valor <= 200 C = "duzentos"     else C = "duzentos e "     end end
30          if CENTENA == 1
31              if valor <= 100 C = "cem"          else C = "cento e "        end end
32          # dezenas entre "n1" e "n9"
33          if DEZENA  >= 91 && DEZENA <= 99 D = "noventa e "   end
34          if DEZENA  >= 81 && DEZENA <= 89 D = "oitenta e "   end
35          if DEZENA  >= 71 && DEZENA <= 79 D = "setenta e "   end
36          if DEZENA  >= 61 && DEZENA <= 69 D = "sessenta e "  end
37          if DEZENA  >= 51 && DEZENA <= 59 D = "cinquenta e " end
38          if DEZENA  >= 41 && DEZENA <= 49 D = "quarenta e "  end
39          if DEZENA  >= 31 && DEZENA <= 39 D = "trinta e "    end
40          if DEZENA  >= 21 && DEZENA <= 29 D = "vinte e "     end
41          # dezenas cheias "n0"
42          if DEZENA  == 90 D = "noventa"   end
43          if DEZENA  == 80 D = "oitenta"   end
44          if DEZENA  == 70 D = "setenta"   end
45          if DEZENA  == 60 D = "sessenta"  end
46          if DEZENA  == 50 D = "cinquenta" end
47          if DEZENA  == 40 D = "quarenta"  end
48          if DEZENA  == 30 D = "trinta"    end
49          if DEZENA  == 20 D = "vinte"     end
50          # dezenas/unidades entre "n0" e "n9"
51          if UNIDADE == 19 U = "dezenove"  end
52          if UNIDADE == 18 U = "dezoito"   end
53          if UNIDADE == 17 U = "desessete" end
54          if UNIDADE == 16 U = "dezesseis" end
55          if UNIDADE == 15 U = "quinze"    end
56          if UNIDADE == 14 U = "quatorze"  end
57          if UNIDADE == 13 U = "treze"     end
58          if UNIDADE == 12 U = "doze"      end
59          if UNIDADE == 11 U = "onze"      end
60          if UNIDADE == 10 U = "dez"       end
61          # unidades isoladas "n"
62          if UNIDADE ==  9 U = "nove"      end
63          if UNIDADE ==  8 U = "oito"      end
64          if UNIDADE ==  7 U = "sete"      end
65          if UNIDADE ==  6 U = "seis"      end
66          if UNIDADE ==  5 U = "cinco"     end
67          if UNIDADE ==  4 U = "quatro"    end
68          if UNIDADE ==  3 U = "três"      end
69          if UNIDADE ==  2 U = "dois"      end
70          if UNIDADE ==  1 U = "um"        end
71          if UNIDADE ==  0
72              return "zero"
73          else
74              return join([C, D, U])
75          end
```

```
76      else
77           return "valor inválido"
78      end
79  end
80
81  print("Informe valor numérico .: ")
82  valor = parse(Int, readline())
83
84  println()
85  println("Valor por extenso ......: ", vlrExtenso(valor))
86  println()
87
88  print("Tecle <Enter> para encerrar programa... ")
89  readline()
```

Ao ser executado o programa informe valores entre "**0**" e "**999**", bem como valores fora desta faixa para visualizar algumas das respostas possíveis do programa.

O código da sub-rotina de função "**valorExtenso**" recebe um conteúdo do tipo inteiro retornando sua resposta como dado do tipo cadeia. O conteúdo recebido é verificado com a instrução "**if valor >= 0 && valor <= 999**" definida na linha "**5**" no sentido de se determinar se o valor fornecido está dentro dos limites operacionais do programa. Caso a resposta a esta condição seja falsa ocorrerá o retorno da resposta "**erro**".

As variáveis "*CENTENA*", "*DEZENA*" e "*UNIDADE*" definidas entre as linhas "**6**" e "**12**" são ajustadas para serem verificadas junto as sequências condicionais definidas no programa entre as linhas "**14**" e "**75**".

No caso da variável "*UNIDADE*" ocorre com a verificação da instrução "**if DEZENA < 20**" na linha "**8**" o ajuste da forma como a variável percebe o valor da unidade. Se o valor da dezena for menor que 20 a unidade será atribuída com o valor do resto da divisão do valor da variável "*DEZENA*" por "**20**", caso contrário será mantido o cálculo da "*UNIDADE*" com o resto da divisão o valor da variável "*DEZENA*" por "**10**".

O conjunto de verificações para as variáveis "CENTENA" e "DEZENA" definidos respectivamente nos trechos de "**14**" a "**31**" e nos trechos de "**33**" a "**49**" estabelecem para essas variáveis as mensagens simples ou as mensagens com o uso do conectivo "e" para a composição de extensos que serão combinados entre si e com os conteúdos definidos junto a variável "*UNIDADE*".

8.3 - Operações binárias: bit-wise, bit-shift e outras

Operações de *bit-wise* (bit-a-bit) e *bit-shift* (deslocamento de bits) são recursos computacionais que são processados em nível binário com o objetivo de executar ações lógicas e/ou aritméticas entre operandos numéricos inteiros em baixo nível. A linguagem Julia possui nove operadores binários, sendo seis do tipo *bit-wise* e três do tipo *bit-shift*. A tabela 8.1 apresenta os operadores de baixo nível lógicos e aritméticos suportados.

Operador	Função	Expressão	Tipo	Descrição
~ (not)	-	~x	bit-wise	Negação lógica.
& (and)	-	x & y	bit-wise	Conjunção lógica.
| (or)	-	x | y	bit-wise	Disjunção inclusiva lógica.
⊻ (xor)	xor(x, y)	x ⊻ y	bit-wise	Disjunção exclusiva lógica.
⊼ (not and)	nand(x, y)	x ⊼ y	bit-wise	Conjunção lógica de negação.
⊽ (not or)	nor(x, y)	x ⊽ y	bit-wise	Disjunção inclusiva lógica de negação.
>>>	-	x >>> y	bit-shift	Rotação lógica à direita.
>>	-	x >> y	bit-shift	Rotação aritmética à direita.
<<	-	x << y	bit-shift	Rotação lógica/aritmética à esquerda.

Tabela 8.1 - Operadores bit-wise/bit-shift e funções especializadas da linguagem Julia.

Não é objetivo deste trabalho entrar neste tema em profundidade, mas sim, fazer uma breve apresentação que seja o mais simples possível, o mais didático que se possa ser no sentido de introduzir este tema ao leitor menos experiente proporcionando uma visão de domínio no uso desses operadores binários.

O uso de manipulação de *bits* é mais frequente quando se deseja fazer com que a linguagem interaja com algum componente de *hardware* conectado ao computador em uso. No entanto, isso não é regra geral, podendo-se usar tais elementos para deixar o código de um programa com uma esfera de abstração maior em alguns pontos. É importante considerar que operadores binários de manipulação de *bits* são em essência a base da funcionalidade computacional mais básica que há. Os operadores binários de *bit-wise* são indicados para a representação de ações lógicas em baixo nível. Já os operadores binários de *bit-shift* são indicados para ações aritméticas de baixo nível.

Atenção em não confundir operadores binários de manipulação de *bits* com os operadores lógicos tradicionais: apesar de possuírem certas semelhanças, os operadores lógicos tradicionais operam sobre o valor avaliado como um todo e os operadores binários de manipulação de *bits* operam sobre, e apenas, um único *bit* de todo o valor. As Tabelas 8.2 e 8.3 mostram os resultados obtidos com o uso dos operadores de *bit-wise* considerando para teste os valores condicionais "**true**" e "**false**".

Valor 1	Valor 2	& (and)	| (or)	⊻ (xor)	⊼ (not and)	⊽ (not or)
false	false	false	false	false	true	true
false	true	false	true	true	true	false
true	false	false	true	true	true	false
true	true	true	true	false	false	false

Tabela 8.2 - Tabela Verdade para os operadores de bit-wise &, |, ⊻, ⊼ e ⊽.

Valor	~ (not)
false	true
true	false

Tabela 8.3 - Tabela Verdade para o operador de bit-wise ~.

Veja alguns exemplos de uso básico das operações *bit-wise*. Não se preocupe com os resultados apresentados quando em uso valores numéricos:

```
julia> false & true
false

julia> false | true
true

julia> false ⊻ true
true

julia> xor(false, true)
true

julia> 123 & 123
123

julia> xor(123, 123)
0

julia> xor(123, 456)
453
```

Para fazer uso dos símbolos de operadores *bit-wise* "⊻", "⊼" e "⊽" use respectivamente as instruções de geração de caracteres especiais "**\xor**", "**\nand**", "**\nor**". Após escrever o código do caractere desejado acione a tecla "**<Tab>**".

Os operadores binários de *bit-shift* operam de forma diferente dos operadores binários de *bit-wise*. Os operadores *bit-shift* deslocam "**1**" *bit* do valor indicado para à esquerda ou para à direita como descreve a Tabela 8.4.

Operador	Ação
>>>	Desloca 1 bit para à direita (valores não sinalizados).
>>	Desloca 1 bit para à direita.
<<	Desloca 1 bit para à esquerda.

Tabela 8.4 - Tabela para operadores bit-shift.

Os operadores de deslocamento de *bits* possibilitam o estabelecimento de ações aritméticas, pois ao se fazer o deslocamento de *bits* de um certo valor para à esquerda ou para à direita, aumenta-se ou diminui-se exponencialmente seu valor.

As operações em baixo nível muitas vezes requerem o uso de definições de valores numéricos em formato binário. Para operar esses valores há uma função que auxilia seu uso, trata-se da função "**bitstring**" que aceita o uso de valores inteiros e de ponto flutuante. Veja alguns exemplos desta funcionalidade aplicada sobre o valor decimal "**10**":

```
julia> bitstring(10)
"0000000000000000000000000000000000000000000000000000000000001010"

julia> bitstring(Int64(10))
"0000000000000000000000000000000000000000000000000000000000001010"

julia> bitstring(Int32(10))
"00000000000000000000000000001010"

julia> bitstring(Int16(10))
"0000000000001010"

julia> bitstring(Int8(10))
"00001010"
```

A partir do exposto cabe a apresentação de alguns exemplos que manipulam *bits* em operações lógicas e aritméticas muito simples, mas necessárias. Antes da apresentação desses exemplos é ideal fazer uma breve introdução (ou revisão) das principais bases numéricas utilizadas na computação sendo essas as bases: binária, decimal, octal e hexadecimal.

Internamente qualquer computador digital opera seu processamento em nível de *bits* representados pelos valores binários "**1**" e "**0**". Qualquer outro valor apresentado, representado ou manipulado é em essência uma conversão de base realizada pelo sistema operacional da máquina para que os seres humanos o entendam melhor. Assim sendo, considere a tabela 8.5 como representante das bases numéricas mais comuns (decimal, binário, octal e hexadecimal) entre os valores decimais de "**0**" a "**15**" utilizados, normalmente, em um computador digital.

A tabela 8.5 apresenta apenas uma pequena sequência de valores. Note que são mostrados os valores binários de um conjunto básico de 4 *bits* e não um conjunto de 8 *bits*, pois se assim fosse considerado seria necessário que a tabela indicasse 256 valores (de "**0**" a "**255**"). O conjunto de 8 *bits* é conhecido como *byte* e o conjunto de 4 *bits* é denominado *nibble*. Assim sendo, a tabela 8.5 mostra um conjunto mínimo de *nibbles* que representam os valores binários básicos de operação computacional e sua relação com as bases decimal, octal e hexadecimal. A partir dos valores binários fica mais fácil deduzir os demais valores e facilitar a compreensão dos valores usados nos exemplos apresentados deste tópico.

Decimal	Binário	Octal	Hexadecimal
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Tabela 8.5 - Valores das bases numéricas mais populares.

Atente para cada valor da tabela 8.5, principalmente para a segunda coluna que expressa os valores de "**0**" a "**15**" na forma binária, sendo esses os valores utilizados e exemplificados nos programas exemplo de manipulação seguintes para a execução de algumas ações de manipulação de *bits*.

Como exemplo inicial considere apresentar os valores da potência da base "**2**" elevado aos índices de "**0**" a "**3**" de um valor numérico com tamanho de um *byte* a partir da definição de deslocamento à esquerda. Observe que a cada deslocamento um novo valor do cálculo é apresentado. Veja as instruções seguintes:

```
julia> valor = 1
1

julia> (Int8(valor) << 0, bitstring(Int8(valor) << 0))  # 2 ^ 0 = 1 (sem desloc.)
(1, "00000001")

julia> (Int8(valor) << 1, bitstring(Int8(valor) << 1))  # 2 ^ 1 = 2 (com desloc. esq. de 1 bit)
(2, "00000010")

julia> (Int8(valor) << 2, bitstring(Int8(valor) << 2))  # 2 ^ 2 = 4 (com desloc. esq. de 1 bit)
(4, "00000100")

julia> (Int8(valor) << 3, bitstring(Int8(valor) << 3))  # 2 ^ 3 = 8 (com desloc. esq. de 1 bit)
(8, "00001000")
```

Ao serem executadas as instruções anteriores ocorrer a apresentação em uma tupla dos valores na forma decimal e binária de acordo com a sequência de valores "**1**", "**2**", "**4**" e "**8**". Note que o resultado obtido a partir do valor inicial da variável "*VALOR*" com "**1**" é "**0001**", sendo mantido o primeiro *bit* a esquerda na mesma posição definida a partir da execução da instrução "**<< 0**". Observe que a operação "**<< 1**" sobre a variável faz seu valor binário passar de "**0001**" para "**0010**" (valor decimal "**2**"), ou seja, desloca o *bit* "**1**" para à esquerda uma posição, neste caso para a segunda posição. Na sequência ocorre a execução do deslocamento "**<< 2**" que movimenta um *bit* da segunda para a terceira posição, neste caso movimenta o valor "**1**" de "**0010**" tornando-o "**0100**" (valor decimal "**4**"). Por último, a instrução de

movimentação "**<< 3**" movimente o terceiro *bit* para a quarta posição *bit*, o que faz com que o valor "**0100**" passe a ser o valor "**1000**" (valor decimal "**8**").

Veja que a cada uso do operador de deslocamento "**<<**" ocorre a mudança do *bit* "**1**" uma posição para à esquerda, a partir do valor de deslocamento indicado.

Agora semelhantemente as instruções anteriores considere fazer uso do operador de deslocamento de *bits* para a direita para apresentar valores das potências da base numérica "**2**" elevada as potências de "**3**" a "**0**". Observe as próximas instruções:

```
julia> valor = 8
1

julia> (Int8(valor) >> 0, bitstring(Int8(valor) >> 0))  # 2 ^ 3 = 8 (sem desloc.)
(8, "00001000")

julia> (Int8(valor) >> 1, bitstring(Int8(valor) >> 1))  # 2 ^ 2 = 4 (com desloc. dir. de 1 bit)
(4, "00000100")

julia> (Int8(valor) >> 2, bitstring(Int8(valor) >> 2))  # 2 ^ 1 = 2 (com desloc. dir. de 1 bit)
(2, "00000010")

julia> (Int8(valor) >> 3, bitstring(Int8(valor) >> 3))  # 2 ^ 0 = 1 (com desloc. dir. de 1 bit)
(1, "00000001")
```

Ao serem executadas as instruções anteriores são apresentados os resultados "**8**", "**4**", "**2**" e "**1**". Observe os detalhes das ações dos deslocamentos à direita efetuados com as instruções "**>> 0**", "**>> 1**", "**>> 2**" e "**>> 3**".

Perceba que nos exemplos anteriores demonstra-se que a ação de deslocamento "empurra" os *bits* para a esquerda e para a direita proporcionando a representação em binário dos valores aritméticos de uma exponenciação calculada em baixo nível. Este cálculo é possível pelo fato de um valor binário ser o resultado de uma potência de base 2 sempre em relação ao valor de um *bit* em relação ao *bit* que está a seu lado, podendo ser crescente (deslocamento à esquerda) ou decrescente (deslocamento à direita).

Calcular potência com deslocamento de *bits* é mais rápido que usar outro método. No entanto, exige certo cuidado, pois a ação de deslocamento não efetua as verificações necessárias para a validação do cálculo como por exemplo, quando se eleva um valor de base as potências "**0**" ou "**1**".

8.4 - Dígito verificador

Dígito verificador (*check digit*) é um valor único de identificação obtido a partir de certo procedimento algorítmico matemático que garante a integridade e autenticidade de um código numérico ou alfanumérico, evitando que ocorra entradas inválidas de dados (WAGNER & PUTER, 1989), erros de transposição de valores e diminui ações fraudulentas no uso de códigos como chaves primárias para identificação de registros em sistemas de informação.

Para a geração de dígitos verificadores existem diversos algoritmos que podem ser utilizados, sendo populares os algoritmos "*módulo 10*" e "*módulo 11*" (HEWLETT PACKARD, 2000. p. 579) aqui apresentados e adaptados a partir das regras convencionais usadas no Brasil.

O algoritmo "*módulo 10*" é aplicado a partir das seguintes considerações:

- as posições intermediárias das unidades que compõem o número básico a partir da última posição de duas em duas posições da direita para a esquerda são multiplicados por "**2**";
- o resultado obtido da multiplicação do dígito por "**2**" quando maior que "**9**" deve ser ajustado para um valor de um dígito. Neste caso faz-se a soma dos dígitos componentes do valor para ter-se um resultado de apenas um dígito. Por exemplo,

se o resultado após a multiplicação for o valor "**16**" será considerado "**7**" devido à soma dos dígitos "**1 + 6**". Outra forma de fazer esta operação é usar o conceito "*nove fora*", onde se subtrai "**9**" do valor "**16**". Neste caso "**16 - 9**" resulta em "**7**";

- os dígitos multiplicados por dois e após ajuste quando maior que "**9**" são adicionados aos dígitos das demais posições que não foram multiplicados;
- o somatório dos valores ajustados dos dígitos deve ser subtraído do maior valor terminado em zero a partir do resultado deste somatório, obtendo-se assim o dígito verificador.

Para demonstrar o uso do algoritmo de geração de dígito verificador "*módulo 10*" considere a tabela 8.6 para se obter o dígito verificador do número de identificação **987.654**.

Código numérico	9	8	7	6	5	4
Peso multiplicador	-	x 2	-	x 2	-	x 2
Resultado parcial	9	16	7	12	5	8
Ajuste quando >= 10	9	16 - 9	7	12 - 9	5	8
Resultado final	9	7	7	3	5	8
Σ do resultado final	39					
Maior número seguinte terminando em 0 (*)	40					
DV = 40 - 39	1					

(*) O maior número seguinte terminado em zero (próximo multiplo de 10, variável PM) a partir do valor da soma do resultado final é conseguido com a expressão aritmética:

PM = INTEIRO((SOMA / 10) + 1) * 10

Tabela 8.6 - Algoritmo para dígito verificado: "módulo 10".

Após a aplicação do algoritmo, indicado na tabela 8.6 o número do código de idenficiação "**987.654**" passa a ter como dígito verificador o valor "**1**" e pode assim ser representado "**987.654-1**".

O programa seguinte demonstra a aplicação do algoritmo de "*módulo 10*" para a geração de um dígito verificador para um número de matricula fornecido sob o formato "**999.999**" e apresentado sob o formato "**999.999-9**". Grave o código seguinte com o nome "**c08ex04.jl**".

```
1  # Geração de dígito verificador, módulo 10 do número 999.999
2
3  matricCPT, matricSPT = "", ""
4  matricINT = Array{Int8}(undef, 6)
5
6  i, soma, DV = 0, 0, 0
7
8  print("Informe matricula no formato 999.999: ")
9  matricCPT = readline()
10
11 matricSPT = replace(matricCPT, "." => "")
12
13 for i in 1:6
```

```
14      matricINT[i] = Int(matricSPT[i]) - 48
15  end
16
17  for i in 1:6
18      if i % 2 == 0 && matricINT[i] * 2 > 9
19          global soma += matricINT[i] * 2 - 9
20      else
21          if i % 2 == 0 && matricINT[i] * 2 <= 9
22              global soma += matricINT[i] * 2
23          end
24      end
25      if i % 2 != 0
26          global soma += matricINT[i]
27      end
28  end
29
30  DV = (trunc(Int, soma / 10) + 1) * 10 - soma
31
32  println()
33  println("Matricula com DV = ", matricCPT, "-", DV)
34  println()
35
36  print("Tecle <Enter> para encerrar programa... ")
37  readline()
```

Ao ser executado o programa, forneça para teste o número de matrícula "**987.654**" e veja a apresentação da saída "**987.654-1**" com o dígito verificador cálculado de acordo com o algoritmo apresentado.

As linhas "**3**", "**4**" e "**6**" preparam para o programa as variáveis a serem operacionalizadas no processamento do cálculo do dígito verificador. Veja que na linha "**4**" está sendo definido o espaço de memória para armazenar e tratar os valores de um arranjo do tipo vetor de tamanho de 8 *bytes*.

Como a entrada da matrícula na linha "**9**" prevê o uso de ponto separando as duas partes desse número, é importante para o cálculo tirar primeiramente o ponto fornecido. Para isso, o conteúdo do vetor "*matricCPT*" (matrícula com ponto) é ajustado para o vetor "*matricSPT*" (matrícula sem ponto), onde o caractere de ponto é removido.

Entre as linhas "**13**" e "**15**" faz-se a conversão de cada caractere alfanumérico da matrícula em seu correspondente numérico inteiro, decompondo o conjunto numérico em dígitos isolados. A linha "**14**" transforma o conteúdo de cada dígito do tipo caractere decomposto do número da matrícula informado como cadeia em seu respectivo valor numérico inteiro a partir da instrução "**matricINT[i] = Int(matricSPT[i]) - 48**". Observe a subtração do valor "**48**" do conteúdo do vetor "*matricSPT*". Isto se faz necessário uma vez que o conteúdo numérico armazenado sob o tipo caractere não representa efetivamente seu equivalente numérico inteiro, pois os valores "**0**, **1**, **2**, **3**, **4**, **5**, **6** ,**7**, **8** e **9**" na tabelas ASCII possuem, respectivamente, os valores "**48**, **49**, **50**, **51**, **52**, **53**, **54**, **55**, **56** e **57**". Daí a necessidade de subtrair o valor "**48**" para obter o real valor.

A etapa seguinte do programa entre as linhas "**17**" e "**28**" efetua-se os cálculos para obtenção do dígito verificador. Nesta etapa, a instrução "**if i % 2 == 0 && matricINT[i] * 2 > 9**" da linha "**18**" verifica se o dígito da posição par é maior que "**9**" quando multiplicado por "**2**" e se confirmado, processa a soma com "**9**" fora. Caso a condição não seja verdadeira, passa à próxima verificação. Em seguida, na linha "**21**" a instrução "**if i % 2 == 0 && matricINT[i] * 2 <= 9**" é usada para verificar se o dígito da posição par é menor ou igual a "**9**",sendo passa a ter seu valor multiplicado por "**2**" antes de ser somado. A próxima avaliação, linha "**25**", verifica por meio da instrução "**if (I % 2 != 0)**" se o dígito avaliado está em uma posição impar. Se estiver, o valor localizado é apenas somado.

Na sequência, linha "**30**", é feito o cálculo do dígito verificador por meio da instrução "**DV = (trunc(Int, soma / 10) + 1) * 10 - soma**", onde "**(trunc(Int, soma / 10) + 1) * 10**" é o cálculo efetuado para obter o maior valor terminado em zero a partir do valor de soma obtido.

Por último, na linha "**33**" o programa apresenta o número da matrícula com seu dígito verificador.

O programa seguinte demonstra a aplicação do algoritmo de "*módulo 10*" validando se dado certo valor de matrícula fornecido sob o formato **999.999-9** é um número válido ou não. Grave o código seguinte com o nome "**c08ex05.jl**".

```
1  # Validação de dígito verificador, módulo 10 do número 999.999-9
2
3  matricCPT, matricSPT = "", ""
4  matricINT = Array{Int8}(undef, 7)
5
6  i, soma, DVC, DVE = 0, 0, 0, 0
7
8  print("Informe matricula no formato 999.999-9: ")
9  matricCPT = readline()
10
11 matricSPT = replace(replace(matricCPT, "." => "", "-" => ""))
12
13 for i in 1:7
14     matricINT[i] = Int(matricSPT[i]) - 48
15 end
16
17 for i in 1:6
18     if i % 2 == 0 && matricINT[i] * 2 > 9
19         global soma += matricINT[i] * 2 - 9
20     else
21         if i % 2 == 0 && matricINT[i] * 2 <= 9
22             global soma += matricINT[i] * 2
23         end
24     end
25     if i % 2 != 0
26         global soma += matricINT[i]
27     end
28 end
29
30 DVC = (trunc(Int, soma / 10) + 1) * 10 - soma
31 DVE = matricINT[7]
32
33 println()
34 if DVC == DVE
35     println("Número de matrícula válido")
36 else
37     println("Número de matrícula inválido")
38 end
39 println()
40
41 print("Tecle <Enter> para encerrar programa... ")
42 readline()
```

Ao ser executado o programa, informe o número de matrícula "**987.654-1**" para ver a indicação de validade do valor informado. Depois execute novamente o programa e informe a matrícula "**987.654-8**" para ver a mensagem indicando que a entrada é inválida.

Em linhas gerais, o programa "**c08ex05.jl**"é semelhante ao programa "**c08ex04.jl**", tendo diferenças sutis, como as definições das variáveis "*DVC*" (dígito verificador cálculado) e "*DVE*" (dígito verificado entrado) e o uso da instrução "**matricSPT = replace(replace(matricCPT, "." => "", "-" => ""))**" junto a linha "**11**". Veja que neste ponto se faz a remoção dos caracteres de ponto e traço em uma única "tacada" reaplicando a função "**replace**" nela mesma duas vezes.

Agora atente para o uso do algoritmo de geração e validação de dígito verificador em "*módulo 11*" que pode ser usado a partir das seguintes considerações:

- para cada posição do dígito que forma as unidades que compõem o número básico do código de identificação é atribuído um valor como fator de ponderação. Os fatores de ponderação são definidos com valores entre "**7**" e "**2**", da direita para a esquerda, repetindo-se esta sequência sempre que necessário;

- cada dígito do código de identificação é multiplicado pelo fator de ponderação atribuído e os produtos obtidos são somados e divididos por "**11**". O resultado do resto obtido é subtraído de "**11**" gerando assim o número do dígito verificador.

Para demonstrar o uso do algoritmo de geração de dígito verificador "*módulo 11*" considere a tabela 8.7 para se obter o dígito verificador do número de identificação "**987.654**".

Código numérico	9	8	7	6	5	4
Peso multiplicador	x 7	x 6	x 5	x 4	x 3	x 2
Resultado parcial	63	48	35	24	15	8
Σ do resultado final	193					
Divisão do Σ por 11	quociente = 17			resto = 6		
DV = 11 - resto	5					

Tabela 8.7 - Algoritmo para dígito verificado: "módulo 11".

Após a aplicação do algoritmo indicado na tabela 8.7, o número do código de idenficiação "**987.654**" passa a ter como dígito verificador o valor "**5**" e pode assim ser representado "**987.654-5**".

No entanto, é importante ao aplicar o algoritmo de módulo "**11**" levar em consideração duas questões:

- se obtido o valor de resto "**10**" após a divisão por "**11**", este valor é oficialmente para o método considerado erro;
- se obtido o valor de resto "**0**" (zero) após a divisão por "**11**", o valor do dígito verificador será "**11**" e, neste caso, substitui-se este valor por "**0**" (zero).

O erro apontado na primeira questão pode ser neutralizado com um truque ao considerar o dígito verificador como "**1**" (forma não oficial para o algoritmo de módulo "**11**"). Outro detalhe que pode ser considerado é quando se obtém o resto igual "**1**" e, neste caso, considera-se a letra "**X**", valor "**10**" em algarismo romano (HEWLETT PACKARD, 2000. p. 581).

O programa a seguir demonstra o uso do algoritmo de "*módulo 11*" para a geração de dígito verificador, considerando o retorno do dígito "**X**" quando o dígito verificadfor calculado for "**1**" ou o retono "**1**" quando o resto for igual a "**10**". Assim, considere um programa para a geração de um dígito verificador para um número de matrícula fornecido sob o formato "**999.999**" e apresentado sob o formato "**999.999-9**". Grave o código seguinte com o nome "**c08ex06.jl**".

```
1  # Geração de dígito verificador, módulo 11 do número 999.999
2
3  function mod11(matricCPT::String)
4
5      matricSPT = ""
6      matricINT = Array{Int32}(undef, 6)
7      i, soma, resto = 0, 0, 0
8      DV = ""
9
10     matricSPT = replace(matricCPT, "." => "")
11
12     for i in 1:6
13         matricINT[i] = Int(matricSPT[i]) - 48
14     end
15
16     for i in 1:6
17         soma += matricINT[i] * (8 - i)
18     end
```

```
19
20      resto = soma % 11
21
22      if resto ==  0 DV = '0' end
23      if resto ==  1 DV = 'X' end
24      if resto == 10 DV = '1' end
25
26      if (11 - resto) >= 2 && (11 - resto) <= 9
27          DV = (11 - resto)
28      end
29
30      return DV
31
32  end
33
34  print("Informe matricula no formato 999.999: ")
35  matricula = readline()
36
37  DV = mod11(matricula)
38
39  println()
40  println("Matricula com DV = ", matricula, "-", DV)
41  println()
42
43  print("Tecle <Enter> para encerrar programa... ")
44  readline()
```

Ao ser executado o programa forneça à cada execução os n´meros de matrículas: "**987.654**", "**123.456**", "**123.455**", "**123.450**" e "**123.109**" e observe a apresentação respectiva dos valores de dígito verificador "**5**", "**0**", "**2**", "**1**" e "**X**". Veja que neste programa está sendo realizado o cálculo do dígito verificador dentro de uma sub-rotina que recebe a matrícula e devolve o dígito verificador correspondente.

Observe que na sub-rotina "**mod11**" entre as linhas "**12**" e "**14**" o laço "**for**" efetua a conversão dos dados caractere para seu equivalente inteiro e, na sequência entre as linhas "**16**" e "**18**" o laço **for** aplica os fatores de peso de multiplicação e efetua o somatório desses por meio da instrução "**soma += matricINT[i] * (8 - i)**". Entre as linhas "**22**" e "**28**", o programa valida o dígito verificador de acordo com as regras condicionais. Nesse caso, se o resto do somatório dos valores calculado na linha "**20**" for "**0**" terá o dígito verificador "**0**"; se for "**1**" terá o dígito verificador "**X**"; se for "**10**" terá o dígito verificador "**1**". Se os restos calculados estiverem entre a faixa de "**2**" a "**9**" (linha "**26**") o cálculo do dígito verificador será "**(11 - resto)**" como indicado na linha "**27**".

O programa seguinte tem por finalidade validar a entrada de uma matrícula por meio de uso do "*módulo 11*". Grave o código seguinte com o nome "**c08ex07.jl**".

```
1   # Geração de dígito verificador, módulo 11 do número 999.999
2
3   function mod11(matricCPT::String)
4
5       matricSPT = ""
6       matricINT = Array{Int32}(undef, 6)
7       i, soma, resto = 0, 0, 0
8       DV = ""
9
10      matricSPT = replace(matricCPT, "." => "")
11
12      for i in 1:6
13          matricINT[i] = Int(matricSPT[i]) - 48
14      end
15
16      for i in 1:6
17          soma += matricINT[i] * (8 - i)
```

```
18      end
19
20      resto = soma % 11
21
22      if resto ==  0 DV = '0' end
23      if resto ==  1 DV = 'X' end
24      if resto == 10 DV = '1' end
25
26      if (11 - resto) >= 2 && (11 - resto) <= 9
27          DV = (11 - resto)
28      end
29
30      return DV
31
32  end
33
34  DVC, DVE = "", ""
35
36  print("Informe matricula no formato 999.999-9: ")
37  matricula = uppercase(readline())
38
39  if matricula[9] == 'X'
40      DVC = mod11(matricula)
41      DVE = matricula[9]
42  else
43      DVC = mod11(matricula)
44      DVE = parse(Int8, matricula[9:9])
45  end
46
47  println()
48  if DVC == DVE
49      println("Número de matrícula válido")
50  else
51      println("Número de matrícula inválido")
52  end
53  println()
54
55  print("Tecle <Enter> para encerrar programa... ")
56  readline()
```

Ao ser executado o programa, forneça os valores das matrículas "**123.456-0**", "**123.455-2**", "**123.451-X**" (entre como "**123.451-x**") e "**123.450-1**" para verificar que são matrículas válidas. Depois forneça, por exemplo, a matrícula "**123.450-7**" para ver mensagem informando que a matrícula fornecida é inválida.

A função "**mod11**" do nome "**c08ex07.jl**" possui o mesmo código da mesma função no programa "**c08ex06.jl**". No entanto, este programa possui dois detalhes importantes.

A primeira ação está relacionada a instrução indicada da linha "**37**" que transforma em maiúsculo um caractere informado como minúsculo, sendo esta ação útil caso seja informado um dígito verificar "**X**" em minúsculo.

A segunda ação importante é controlada no trecho das linhas "**39**" até "**45**", onde é detectado o tipo do dígito verificador. Neste caso, se a nona posição da variável "*MATRÍCULA*" for a letra "**X**" serão processadas as operações das linhas "**40**" até "**41**", caso contrário serão processadas as operações das linhas "**43**" até "**44**". Para entender o motivo pelo qual isso é feito observe as próximas instruções interativas:

```
julia> alfanum = "123.109-X"
"123.109-X"

julia> dv1 = alfanum[9]
'X': ASCII/Unicode U+0058 (category Lu: Letter, uppercase)
```

Observe que o resultado da execução da instrução "**dv1 = alfanum[9]**" é a apresentação do último dígito como caractere ASCII mais outras características. Agora entre a função "**mod11**" no modo interativo.

```
julia> function mod11(matricCPT::String)
           matricSPT = ""
           matricINT = Array{Int32}(undef, 6)
           i, soma, resto = 0, 0, 0
           DV = ""
           matricSPT = replace(matricCPT, "." => "")
           for i in 1:6
               matricINT[i] = Int(matricSPT[i]) - 48
           end
           for i in 1:6
               soma += matricINT[i] * (8 - i)
           end
           resto = soma % 11
           if resto ==  0 DV = '0' end
           if resto ==  1 DV = 'X' end
           if resto == 10 DV = '1' end
           if (11 - resto) >= 2 && (11 - resto) <= 9
               DV = (11 - resto)
           end
           return DV
       end
mod11 (generic function with 1 method)
```

Agora observe a próxima instrução:

```
julia> dv2 = mod11(alfanum)
'X': ASCII/Unicode U+0058 (category Lu: Letter, uppercase)
```

Observe que as variáveis "*DV1*" e "*DV2*" possuem como resposta dados do tipo caractere e ao serem comparados resulta verdadeiro quando forem de conteúdo igual e de mesmo tipo.

```
julia> dv1 == dv2
true
```

No entanto, se os valores de "*DV1*" e "*DV2*" forem de tipos diferentes de dados, mesmo que aparentemente sejam iguais ocorrerá retorno de condição falso. Observe as próximas instruções com o uso das variáveis "*DV3*" e "*DV4*":

```
julia> alfanum = "123.455-2"
"123.455-2"

julia> dv3 = alfanum[9]
'2': ASCII/Unicode U+0032 (category Nd: Number, decimal digit)

julia> dv4 = mod11(alfanum)
2
```

Observe agora que as variáveis "*DV3*" e "*DV4*" possuem aparentemente valores iguais, mas de tipos diferentes. Observe a próxima instrução:

```
julia> dv3 == dv4
false
```

A solução para esta questão é transformar o caractere "**2**" no seu equivalente numérico inteiro. Para que isto seja feito é necessário realizar duas etapas de operação, sendo a primeira em transformar o caractere "**2**" em uma cadeia que represente o valor "**2**" e depois transformar a cadeia em valor numérico.

A transformação do caractere numérico em cadeia numérica é realizada com a instrução:

```
julia> vlr = alfanum[9:9]
"2"
```

Observe que se estiver em uso o estilo "**alfanum[9]**" obtém-se o valor como caractere, mas se estiver em uso o estilo "**alfanum[9:9]**" obtém-se o valor como cadeia. Note o detalhe na definição dos valores de índice dentro dos colchetes.

Agora basta transformar o conteúdo da variável "*VLR*" em número e isso é realizado com a instrução:

```
julia> dv3 = parse(Int, vlr)
2

julia> dv3 == dv4
true
```

Observe neste conjunto de instruções o motivo que levou a estabelecer esses detalhes de controle para atuar junto as linhas "**39**" a "**45**".

8.5 - Funções para classificação de dados

Neste tópico são apresentadas implementações de alguns algoritmos conhecidos de classificação de dados que permitem colocar em certo arranjo seus elementos em determinada ordem, sejam numéricos ou alfanuméricos. Para a realização desta tarefa é necessário em média a execução de duas operações importantes nos algoritmos de classificação, sendo a verificação da posição do elemento no arranjo e a outra operação a realização da permuta (troca) dos elementos na posição quando certa ordem não está atendida.

A operação de troca de valores entre duas posições de memória, em Julia, pode ser feita de forma tradicional encontrada em outras linguagens de programa (permutação tradicional) ou ser feita a partir do uso da técnica de permutação cíclica. Observe os exemplos destes dois tipos de ação:

```
X = A         => permutação tradicional
A = B
B = X

A, B = B, A   => permutação cíclica
```

Os algoritmos usados, neste tópico, para a classificação de elementos, em sua maioria, fazem uso do modo de permuta para realizar o reposicionamento de elementos. Entre os diversos algoritmos existentes são apresentados os modelos:

- bubble (classificação pelo método de flutuação);
- cocktail (classificação pelo método flutuação bidirecional);
- comb (classificação pelo método de comparação de pares de valores);
- distributive (classificação pelo método distributivo);
- gnome (classificação baseada no método de inserção);

- heap (classificação pelo método de amontoamento);
- inserction (classificação pelo método de inserção);
- oddEven (classificação pelo método ímpar/par);
- quick (classificação pelo método rápido);
- selection (classificação pelo método de seleção);
- shell (classificação baseada no método de inserção direta).

Entre os algoritmos apresentados observe atentamente o código da função "**heap!**", que demostra o uso de sub-rotina dentro de sub-rotina, ou seja, opera o conceito de sub-rotina aninhada. Uma sub-rotina é definida dentro de outra sub-rotina quando é necessário organizar o código interno dessa sub-rotina. O escopo de operação de uma sub-rotina aninhada em outra sub-rotina é limitado pelo tempo de uso dessa sub-rotina. Todas as sub-rotinas aninhadas ficam automaticamente definidas com espoco local, não podendo ser usadas fora da sub-rotina a que fazem parte. Veja que esta é uma forma de se operar o conceito de *encapsulamento*. A tabela 8.8 mostra algumas informações complementares sobre os algoritmos de classificação demonstrados.

Algoritmo	Ano	Autor
bubble	-	Desconhecido
cocktail (bidirectional bubble sort)	-	Desconhecido
comb	1980	Wlodzimierz Dobosiewicz
distributive	-	Desconhecido
gnome (stupid sort)	2000	Hamid Sarbazi-Azad
heap	1964	Robert W. Floyd e J. W. J Williams
inserction	-	Desconhecido
oddEven	-	Desconhecido
quick	1960	Charles A. R. Hoare
selection	-	Desconhecido
shell	1959	Donald Shell

Tabela 8.8 - Algoritmos de classificação de dados (alguns).

Não é objetivo explicar o funcionamento dos algoritmos de classificação mostrados neste tópico, pois este tema é estudado em uma disciplina chamada "*Estrutura de Dados*". O objetivo é apenas apresentar a criação e uso de uma pequena biblioteca (pacote) de funcionalidades definidos localmente no computador com uma *namespace* (espaço de trabalho em memória) chamada "**Sorts**". Assim sendo, observe atentamente o código de programa seguinte, que deverá ser gravado com o nome "**c08ex08.jl**".

```
1  # Pacote interno com algoritmos classificadores de listas
2
3  module Sorts
4
5  export bubble!,
6         cocktail!,
7         comb!,
8         distributive!,
9         gnome!,
10        heap!,
11        insertion!,
12        oddEven!,
13        quick!,
14        selection!,
15        shell!
16
17 function bubble!(lst::Array)::Array
18     n = length(lst)
```

```
19      for i in 2:n
20          for j in 1:n-1
21              if lst[j] > lst[j + 1]
22                  lst[j], lst[j + 1] = lst[j + 1], lst[j]
23              end
24          end
25      end
26      return lst
27  end
28
29  function cocktail!(lst::Array)::Array
30      início = 1
31      fim = length(lst)
32      while início < fim
33          for i in início:fim-1
34              if lst[i] > lst[i + 1]
35                  lst[i], lst[i + 1] = lst[i + 1], lst[i]
36              end
37          end
38          fim -= 1
39          for i in fim:-1:início+1
40               if lst[i] < lst[i - 1]
41                  lst[i], lst[i - 1] = lst[i - 1], lst[i]
42               end
43          end
44          início += 1
45      end
46      return lst
47  end
48
49  function comb!(lst::Array)::Array
50      n = length(lst)
51      desloc = n
52      trocar = true
53      while desloc > 1 || trocar
54          desloc = trunc(Int, desloc / 1.25)
55          i = 0
56          trocar = false
57          while i + desloc < n
58              if lst[i + 1] > lst[i + 1 + desloc]
59                  lst[i + 1], lst[i + 1 + desloc] = lst[i + 1 + desloc], lst[i + 1]
60                  trocar = true
61              end
62              i += 1
63          end
64      end
65      return lst
66  end
67
68  function distributive!(lst::Array)::Array
69      n = length(lst)
70      for i in 1:n-1
71          for j in i+1:n
72              if lst[i] > lst[j]
73                  lst[i], lst[j] = lst[j], lst[i]
74              end
75          end
76      end
77      return lst
78  end
79
80  function gnome!(lst::Array)::Array
```

```
81      n = length(lst)
82      i = 2
83      j = 3
84      while i <= n
85          if lst[i - 1] <= lst[i]
86              i = j
87              j += 1
88          else
89              lst[i - 1], lst[i] = lst[i], lst[i - 1]
90              i -= 1
91              if i == 1
92                  i = j
93                  j += 1
94              end
95          end
96      end
97      return lst
98  end
99
100 function heap!(lst::Array)::Array
101     n = length(lst)
102
103     function swap(lst, i, j)
104         lst[i], lst[j] = lst[j], lst[i]
105     end
106
107     function siftDown(lst, i, n)
108         while (pos = 2 * i - 1) < n
109             if pos < n && lst[pos] < lst[pos + 1]
110                 pos += 1
111             end
112             if lst[i] < lst[pos]
113                 swap(lst, pos, i)
114                 i = pos
115             else
116                 break
117             end
118         end
119     end
120
121     function heapify(lst, n)
122         i = div(n, 2)
123         while i >= 1
124             siftDown(lst, i, n)
125             i -= 1
126         end
127     end
128
129     heapify(lst, n)
130     l = n
131     while l > 1
132         swap(lst, 1, l)
133         l -= 1
134         siftDown(lst, 1, l)
135     end
136     return lst
137 end
138
139 function insertion!(lst::Array)::Array
140     n = length(lst)
141     for i in 1:n
142         elem = lst[i]
143         j = i
```

```
144         while j > 1 && lst[j - 1] > elem
145             lst[j] = lst[j - 1]
146             j -= 1
147         end
148         lst[j] = elem
149     end
150     return lst
151 end
152
153 function oddEven!(lst::Array)::Array
154     n = length(lst)
155     pronto = false
156     while !pronto
157         pronto = true;
158         # odd-even
159         for i in 2:2:n-1
160             if lst[i] > lst[i + 1]
161                 lst[i], lst[i + 1] = lst[i + 1], lst[i]
162                 pronto = false
163             end
164         end
165         # even-odd
166         for i in 1:2:n-1
167             if lst[i] > lst[i + 1]
168                 lst[i], lst[i + 1] = lst[i + 1], lst[i]
169                 pronto = false
170             end
171         end
172     end
173     return lst
174 end
175
176 function quick!(lst::Array, início::Int64=1, fim::Int64=length(lst))::Array
177     i = início
178     j = fim
179     meio = lst[div(início + fim, 2)]
180     while i <= j
181         while lst[i] < meio && i < fim
182             i += 1
183         end
184         while lst[j] > meio && j > início
185             j -= 1
186         end
187         if i <= j
188             lst[i], lst[j] = lst[j], lst[i]
189             i += 1
190             j -= 1
191         end
192     end
193     if j > início
194         quick!(lst, início, j)
195     end
196     if i < fim
197         quick!(lst, i, fim)
198     end
199     return lst
200 end
201
202 function selection!(lst::Array)::Array
203     n = length(lst)
204     for i in 1:n-1
205         min = i
```

```
206         for j in i+1:n
207             if lst[j] < lst[min]
208                 min = j
209             end
210         end
211         if min != i
212             lst[min], lst[i] = lst[i], lst[min]
213         end
214     end
215     return lst
216 end
217
218 function shell!(lst::Array)::Array
219     n = length(lst)
220     pos = div(length(lst), 2)
221     while pos > 0
222         for i in pos+1:n
223             j = i
224             elem = lst[i]
225             while j > pos && lst[j - pos] > elem
226                 lst[j] = lst[j - pos]
227                 j -= pos
228             end
229             lst[j] = elem
230         end
231         if pos == 2
232             pos = 1
233         else
234             pos = floor(Int, pos * 5.0 / 11)
235         end
236     end
237     return lst
238 end
239
240 end  # Sorts
```

Um detalhe a ser comentado e muito utilizado na linguagem Julia é o uso do símbolo exclamação "**!**" junto ao nome de uma sub-rotina de função. Como já percebido isso é usado para identificar sub-rotinas que quando entram em operação alteram o estado físico do conteúdo manipulado. As sub-rotinas de função que não possuem esse símbolo operam sua manipulação de forma lógica, as que tem o símbolo exclamação operam sua manipulação de forma física. As sub-rotinas de classificação, são por sua natureza recursos que manipulam e alteram as posições dos dados fisicamente, por esta razão quando codificadas devem possuir em seu nome o símbolo de exclamação, não que isso seja necessário, mas é uma forma de identificar quando a ação de uma sub-rotina é física ou lógica.

O código anterior será executado internamente no ambiente interativo REPL, podendo ser aplicado em outros programas codificados em Julia. Lembre-se de que o funcionamento de um pacote interno deve obrigatoriamente estar locado no diretório de trabalho do usuário. Desta forma, execute os passos seguintes.

```
julia> include("c08ex08.jl")
Main.Sorts

julia> using .Sorts
```

O uso do comando "**include**" pode fazer referência a um arquivo de programa que esteja gravado em local diferente da área de trabalho do usuário. Neste caso, basta indicar o caminho de acesso. Considerando a existência de um diretório chamado "**Teste**" na raiz do disco principal do sistema execute, por exemplo, as instruções:

```
julia> include("C:\\Teste\\c08ex08.jl")  # para o sistema operacional Windows
Main.Sort
```

```
julia> include("/Teste/c08ex08.jl")  # para o sistema operacional Linux ou macOS
Main.Sort
```

Veja então exemplos de uso interativo das rotinas de classificação a partir do arranjo "**[8, 0, 5, -1, 7, 1, -9]**", em que qualquer chamada produzira a saída:

```
7-element Vector{Int64}:
 -9
 -1
  0
  1
  5
  7
  8

julia> Sorts.bubble!([8, 0, 5, -1, 7, 1, -9])

julia> Sorts.cocktail!([8, 0, 5, -1, 7, 1, -9])

julia> Sorts.comb!([8, 0, 5, -1, 7, 1, -9])

julia> Sorts.distributive!([8, 0, 5, -1, 7, 1, -9])

julia> Sorts.gnome!([8, 0, 5, -1, 7, 1, -9])

julia> Sorts.gnome!([8, 0, 5, -1, 7, 1, -9])

julia> Sorts.heap!([8, 0, 5, -1, 7, 1, -9])

julia> Sorts.inserction!([8, 0, 5, -1, 7, 1, -9])

julia> Sorts.oddEven!([8, 0, 5, -1, 7, 1, -9])

julia> Sorts.quick!([8, 0, 5, -1, 7, 1, -9])

julia> Sorts.selection!([8, 0, 5, -1, 7, 1, -9])

julia> Sorts.shell!([8, 0, 5, -1, 7, 1, -9])
```

Para ver a aplicação do pacote de classificação de dados dentro de um código de programa considere como exemplo uma aplicação que efetue a leitura de cinco nomes de pessoas e os coloquem em ordem ascendente. Assim sendo, grave o programa com o nome "**c08ex09.jl**".

```
1  # Aplicação com uso de pacote externo
2
3  include("c08ex08.jl")
4  using .Sorts
5
6  nome = Array{String}(undef, 5)  # criação do vetor na memória
7
8  print("\e[2J\e[0;0H")
9  println("-----------------------------------")
10 println("CLASSIFICAÇÃO ASCENDETE DE \"5\" NOMES")
11 println("-----------------------------------")
12 println()
```

```
13
14  for i in 1:5
15      print("Entre o ", i, "o. nome: ")
16      nome[i] = readline()
17  end
18
19  Sorts.quick!(nome)  # processamento da classificação
20
21  println()
22  println("RESULTADO")
23  println("=========")
24  println()
25
26  for i in 1:5
27      println(i, " ==> ", nome[i])
28  end
29
30  println()
31  print("Tecle <Enter> para encerrar programa... ")
32  readline()
```

Para realizar a operação na ordem descente basta fazer uso em conjunto da função interna "**`reverse`**".

8.6 - Criptografia de Caesar

Este tópico apresenta sub-rotinas criptográficas baseadas no algoritmo da cifra de *Caesar* (César). O termo *criptografia* tem sua origem na Grécia sendo a junção das palavras *kryptós* (oculto) e *graphos* (escrita), originando o termo *criptografia*, ou seja, escrita oculta). (SINGH, 2011, p. 25-26).

Na prática de aplicação da criptografia, a mensagem a ser cifrada é chamada de *texto claro* ou *texto plano*, sendo grafado com caracteres alfabéticos minúsculos e a mensagem criptografada é grafada com caracteres alfabéticos em formato maiúsculo. Esta regra será atendida nos exemplos de códigos deste tópico. No entanto, há outra regra que determina a eliminação dos espaços em branco na mensagem cifrada e sua separação em grupos de letras para dificultar sua percepção. Esta regra de eliminação de espaços em branco não será aplicada.

O algoritmo *Caesar* usado pelo imperador romano Júlio César para se comunicar com seus generais tem sua origem no ano 50 a.C., usa uma estratégia muito simples baseada em um único alfabeto cifrante (monoalfabético) em que ocorre o deslocamento de uma letra do alfabeto três posições à frente (monográfica) para cifrar e três posições atrás para decifrar mensagens curtas, sendo ideal para a escrita de pequenos textos. Sua segurança é muito baixa, basta que seja aplicado uma simples análise de frequência estatística baseada nas características do idioma cifrado para decifrar a mensagem (VICKI, 2008). A tabela 8.9 mostra a regra *Caesar*, contendo na linha superior (azul em minúsculo) o caractere a ser cifrado e na linha inferior (vermelho em maiúsculo) o caractere cifrado.

Alfabeto																									
a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C

Tabela 8.9 - Regra de Caesar.

O programa seguinte demonstra a codificação criptográfica de uma mensagem segundo as regras da cifra de *Caesar*. Grave o programa com o nome "**`c08ex10.jl`**".

```
1  # Criptografia: Caesar
2
3  function codMensagem(texto::String)::String
4    mensagem::String = ""
5    for i in 1:length(texto)
6      #=
```

```
 7          Verifica se o caractere m foco é um espaço em branco, sendo
 8          a menssagem o mantém e avança-se para a próxima posição.
 9      =#
10      if texto[i] == ' '
11          mensagem *= ' '
12          continue
13      end
14      #=
15          Se o caractere em foco for "X", "Y", "Z" então subtraí-se 26
16          do valor do caractere somando-se "3" para obter "A", "B", "C".
17          O valor "3" referes-e ao deslocamento da posição atual mais
18          três para gerar o efeito do alfabéto de Caesar.
19      =#
20      if uppercase(texto[i]) >= 'X' && uppercase(texto[i]) <= 'Z'
21        mensagem *= uppercase(texto[i] - 26 + 3)
22        continue
23      end
24      mensagem *= uppercase(texto[i] + 3)  # cifra cada caractere 3 posições à frente
25    end
26    return mensagem
27  end
28
29  function decMensagem(texto::String)::String
30    mensagem::String = ""
31    for i in 1:length(texto)
32      if texto[i] == ' '
33          mensagem *= ' '
34          continue
35      end
36      if uppercase(texto[i]) >= 'A' && uppercase(texto[i]) <= 'C'
37        mensagem *= lowercase(texto[i] + 26 - 3)
38        continue
39      end
40      mensagem *= lowercase(texto[i] - 3)  # cifra cada caractere 3 posições para trás
41    end
42    return mensagem
43  end
44
45  print("\e[2J\e[0;0H")
46  printstyled("---------------\n", color=:cyan)
47  printstyled("CIFRA DE CAESAR\n", color=:cyan)
48  printstyled("---------------\n", color=:cyan)
49  println()
50  printstyled("Informe mensagem a ser cifrada ...: ", color=:yellow)
51  mensagOriginal = readline()
52
53  mensagCifrada   = codMensagem(mensagOriginal)
54  mensagDecifrada = decMensagem(mensagCifrada)
55
56  print("\nMensagem original .......: "); printstyled(mensagOriginal, color=:yellow)
57  print("\nMensagem com cifragem ...: "); printstyled(mensagCifrada, color=:blue)
58  print("\nMensagem sem cifragem ...: "); printstyled(mensagDecifrada, color=:red)
59
60  println("\n")
61  printstyled("Tecle <Enter> para encerrar programa... ", color=:green)
62  readline()
```

Ao ser executado o programa entre uma mensagem escrita tanto com caracteres minúsculos como caracteres maiúsculos. O programa fará a conversão da mensagem em caracteres maiúsculos e minúsculos para a representação de texto cifrado e texto decifrado. Para teste entre a mensagem "**Atacar base inimiga ao amanhecer**" e veja a apresentação da mensagem criptografada como "**DWDFDU EDVH LQLPLJD DR DPDQKHFHU**".

A sub-rotina de função "**codMensagem**" recebe como parâmetro junto a variável "*TEXTO*" o conteúdo que será cifrado e devolve a cifra por meio da variável "*MENSAGEM*". Na instrução de laço "**for**" entre as linhas "**5**" e "**25**" há duas decisões que controlam o processo de cifragem: a primeira decisão entre as linhas "**10**" e "**13**" verifica se a mensagem possui espaço em branco com a instrução e havendo este espaço o mantém para a mensagem ase apresentada.

A segunda decisão entre as linhas "**20**" e "**23**" verifica se a letra da mensagem está entre as letras "**X**", "**Y**" e "**Z**", sendo a condição verdadeira ocorre o ajuste para que sejam alteradas essas letras para suas equivalentes "**A**", "**B**" e "**C**". O valor "**26**" refere-se- a quantidade de letras do alfabeto e o valor "**3**" refere-se ao rotacionamento da cifra.

A sub-rotina de função "**decMensagem**" é semelhante sub-rotina de função "**codMensagem**". Enquanto que na função "**codMensagem**" fez-se a validação das últimas três letras do alfabeto "**X**", "**Y**" e "**Z**", nesta faz-se a validação das primeiras três letras do alfabeto "**A**", "**B**" e "**C**". Com essa estratégia tem-se controle dos limites operacionais para o rotacionamento das letras no alfabeto, sendo controlado entre as linhas "**36**" e "**39**".

8.7 - Verbo de primeira conjugação

Verbo de primeira conjugação caracteriza-se por ser o verbo terminado em "**-ar**", sendo considerado um verbo regular. Verbo regular é o verbo que se encaixa em um modelo fixo de conjugação verbal e para exemplificar este tipo de processamento considere um que recebe a entrada de um verbo regular de primeira conjugação no tempo presente do modo indicativo. Se fornecido verbos com as terminações de segunda ou terceira conjugações o programa deverá emitir mensagem de erro "*Verbo fornecido não é válido*". Grave o programa com o nome "**c08ex11.jl**".

```
 1  # Conjugação de verbo regular: primeira conjugação
 2
 3  tamanho, parte = 0, 0
 4  terminação, t1, t2 = "", "", ""
 5
 6  print("\e[2J\e[0;0H")
 7  println("----------------------------------------")
 8  println("CONJUGAÇÃO VERBAL - Primeira conjugação")
 9  println("----------------------------------------")
10  println()
11
12  print("Entre um verbo regular: ")
13  verbo = lowercase.(readline())
14
15  tamanho = length(verbo)
16  parte = tamanho - 2
17
18  t2 = verbo[tamanho - 1]  # pega "a"
19  t1 = verbo[tamanho]      # pega "r"
20
21  terminação = t2 * t1  # gera terminação "ar"
22
23  println()
24  if terminação == "ar"
25      println("Eu ", verbo[1:parte], "o")
26      println("Tu ", verbo[1:parte], "as")
27      println("Ele ", verbo[1:parte], "a")
28      println("Nós ", verbo[1:parte], "amos")
29      println("Vós ", verbo[1:parte], "ais")
30      println("Eles ", verbo[1:parte], "am")
31  else
32      println("Verbo fornecido não é válido")
33  end
34
35  println()
36  println("Tecle <Enter> para encerrar programa... ")
37  readline()
```

Execute o programa e entre verbos que sejam válidos e inválidos e verifique o funcionamento da operação do programa.

Um detalhe a ser considerado no programa é a conversão para minúsculo de todos os caracteres que forma o verbo junto a linha "**13**" onde se coloca em uso a função "**lowecase**" que tem por finalidade transformar um caractere maiúsculo em seu equivalente minúsculo. Observe junto a função o uso do operador "**.**" que tem por finalidade aplicar a função sobre todos os caracteres da cadeia em uso.

Outro ponto importante do programa são as linhas "**15**", "**16**", "**18**" e "**19**". Nas linhas "**15**" e "**16**" faz-se respectivamente a obtenção do tamanho do verbo com o trecho "**tamanho = length(verbo)**" e o cálculo do prefixo do verbo excluindo-se a terminação com o trecho "**parte = tamanho - 2**". Nas linhas "**18**" e "**19**" faz-se a extração das letras que compõem a terminação do verbo que é então concatenado na linha "**21**".

O trecho entre as linhas "**24**" e "**33**" controla a apresentação ou não do verbo. Entre as linhas "**25**" e "**30**" é a apresentada a conjugação e na linha "**32**" é apresentada a mensagem de erro caso a palavra fornecida não seja verbo regular de primeira conjugação.

Note que para a conjugação apresentada entre as "**25**" e "**30**" o programa extraí do verbo como prefixo a posição indexada "**verbo[1:parte]**" anexando a terminação da pessoa.

8.8 - Acesso direto em arquivo binário

O acesso direto, também denominado acesso randômico, permite mover o ponteiro de arquivo para qualquer posição do arquivo em uso. Anteriormente nos exemplos apresentados de manipulação de arquivos no capítulo seis não se levou em consideração outra forma de acesso aos dados de arquivos a não ser a forma sequencial. Vale ressaltar que acesso direto é uma técnica conveniente para uso com arquivos binários.

Para o exemplo a ser construído neste tópico, será utilizada a forma de acesso direto, pois esse tipo de arquivo permite o acesso imediato a cada um dos seus registros, desde que se conheça previamente a posição em que eles estão gravados.

São apresentados três programas: um programa que cadastra valores inteiros em um arquivo binário, um outro programa que efetua a leitura dos dados no arquivo indicando-se suas posições absolutas e um terceiro programa que possibilita efetuar a alteração de número armazenado no arquivo.

Os programas deste tópico seguem abordagem diferente dos programas apresentados no capítulo dois. Aqui os programas são codificados dentro de uma sub-rotina em estilo procedimento que estabelece uma área de escopo para tratamento das variáveis sem a necessidade de uso do comando "**global**". Desta forma o programa em si estará totalmente codificado dentro de uma sub-rotina a qual é isoladamente chamada. Entre o estilo apresentado no capítulo dois e este estilo de preferência por esta forma aqui apresentada.

O primeiro programa lê valores inteiros fornecidos pelo teclado e grava-os de forma binária sempre um a um ao final do arquivo chamado "**ARQBIN.BIN**", ou seja, a gravação será realizada de modo sequencial. Grave o programa seguinte com o nome "**c08ex12.jl**".

```
 1  # Cadastro direto de valores em arquivo binário
 2
 3  function main()
 4
 5      valor::Int64 = 0
 6      resp::String = "S"
 7
 8      arquivo = open("ARQBIN.BIN", "a")
 9
10      while resp == "S"
11          print("Informe valor inteiro: ")
12          valor = parse(Int, readline())
13
14          seekend(arquivo)
15          write(arquivo, valor)
```

```
16
17          println("\nDeseja continuar?")
18          print("[S] para Sim | [qualquer letra] para Não --> ")
19          resp = uppercase(readline())
20          println()
21      end
22
23      close(arquivo)
24
25  end
26
27  main()
```

Ao executar o programa, insira alguns valores inteiros e, ao sair do programa, execute-o novamente, inserindo mais alguns valores. Anote os valores fornecidos para que possam ser verificados posteriormente com o próximo programa.

Veja que o programa efetua a abertura do arquivo na linha "**8**" e efetua seu fechamento na linha "**23**". Entre as linhas "**10**" e "**21**" ocorre a operação de gravação do número fornecido. Note que o arquivo é aberto em modo *append* e que antes de cada gravação da linha "**15**" faz-se a movimentação do ponteiro de acesso ao arquivo para a próxima posição vazia ao final do arquivo com a linha "**14**".

O programa seguinte lê um valor armazenado no arquivo utilizando o acesso direto (aleatório ou randômico), solicitando uma posição específica no arquivo e apresentando o conteúdo se este existir. Grave o programa com o nome "**c08ex13.jl**".

```
1   # Pesquisa direta de valores em arquivo binário
2
3   using Printf
4
5   function main()
6
7       valor::Int64 = 0
8       resp::String = "S"
9       tamamho::Int64, pos::Int64, nrRegs::Int64 = 0, 0, 0
10
11      arquivo = open("ARQBIN.BIN", "r")
12
13      tamanho = filesize(arquivo)
14      nrRegs = tamanho / 8
15
16      while resp == "S"
17          print("Informe posição a ser pesquisada: ")
18          pos = parse(Int, readline())
19
20          if pos >= 1 && pos <= nrRegs
21              seek(arquivo, (pos-1) * 8)
22              valor = peek(arquivo, Int64)
23              @printf("\nValor %i na posicao %i.\n", valor, pos)
24          else
25              println("\nPosição informada - inválida.")
26          end
27
28          println("\nDeseja continuar?")
29          print("[S] para Sim | [qualquer letra] para Não --> ")
30          resp = uppercase(readline())
31          println()
32      end
```

```
33
34     close(arquivo)
35 end
36
37 main()
38
```

Um dos detalhes importante do programa "**c08ex13.jl**" é a variável "*POS*" definida na linha "**9**" para controlar o acesso às posições do arquivo.

No programa aparece o uso da função "**seek**" mencionada na linha "**21**". Esta função é usada para fazer a posição do ponteiro sobre o ponto específico do arquivo. Isto é conseguido a partir da indicação da posição informada menos um multiplicando-se pelo tamanho do tipo inteiro de 64-bit "**seek(arquivo, (pos-1) * 8)**" junto a linha "**21**".

Após fazer a localização do ponto desejado o programa na linha "**22**" faz a leitura da posição com a função "**peek**" atribuindo o dado lido a variável "*VALOR*"

Outro ponto a ser observado no programa é o trecho de código situado entre as linhas "**20**" e "**26**" que verifica se a posição informada está dentro do total de registros existentes no arquivo. Sendo fornecido um valor fora da faixa permitida, o programa apresenta a mensagem "**Posição informada - inválida**".

O programa seguinte solicita uma posição do arquivo e um novo valor que será gravado na posição indicada sobre o antigo valor atualizando-o. Ter-se-á, portanto, um programa de alteração do conteúdo de um registro no arquivo. Grave o programa com o nome "**c08ex14.jl**".

```
 1 # Pesquisa direta e escrita de valores em arquivo binário
 2
 3 function main()
 4
 5     valor::Int64 = 0
 6     resp::String = "S"
 7     tamamho::Int64, pos::Int64, nrRegs::Int64 = 0, 0, 0
 8
 9     arquivo = open("ARQBIN.BIN", "r+")
10
11     tamanho = filesize(arquivo)
12     nrRegs = tamanho / 8
13
14     while resp == "S"
15         print("Informe posição a ser alterada: ")
16         pos = parse(Int, readline())
17         if pos >= 1 && pos <= nrRegs
18             print("Entre novo valor: ")
19             valor = parse(Int, readline())
20             seek(arquivo, (pos-1) * 8)
21             write(arquivo, valor)
22         else
23             println("\nPosição informada - inválida.")
24         end
25
26         println("\nDeseja continuar?")
27         print("[S] para Sim | [qualquer letra] para Nao --> ")
28         resp = uppercase(readline())
29         println()
30     end
31
32     close(arquivo)
33
34 end
35
36 main()
```

O programa solicita a posição do arquivo do qual se deseja alterar seu conteúdo e pede a entrada do novo valor a ser cadastrado na posição informada.

Para verificar a mudança do valor efetivada, execute o programa "**`c08ex13.jl`**", fornecendo como posição de pesquisa o mesmo valor de posição informado na execução do programa "**`c08ex14.jl`**", e observe que o valor foi realmente alterado.

ANOTAÇÕES

APÊNDICE A - SQLITE

Neste apêndice é descrito os passos básicos e mínimos para proceder a instalação e configuração do gerenciador de banco de dados SQLite nos sistemas operacionais Windows, Linux e macOS adquirida a partir do endereço: ***https://www.sqlite.org/****.*

Instalação Windows

A partir do acesso ao sítio do projeto "**`SQLite`**" selecione a opção "**`Downloads`**" localize na página a opção de acesso ao sistema operacional Windows "**`Precompiled Binaries for Windows`**". Acione com o ponteiro do *mouse* os *links* de *download* de sua preferência, sendo obrigatoriamente a opção "**`sqlite-tools-win32-x86-3390200.zip`**" e opcionalmente, dependendo do padrão de seu sistema, sendo "**`sqlite-dll-win64-x64-3390200.zip`**" para 64-bit ou "**`sqlite-dll-win32-x86-3390200.zip`**".

Crie na raiz de seu disco principal um diretório chamado "**`sqlite`**" e copie para este os arquivos baixados e em seguida descompacte-os de forma que todos os arquivos fiquem disponíveis na raiz. Neste momento você terá algo semelhante a imagem da figura A.1.

Os arquivos contidos dentro do diretório "**`sqlite-tools-win32-x86-3390200`**" como indicados na figura A.2 devem ser movimentados para dentro do diretório "**`sqlite`**" de modo que se tenha uma situação semelhante a imagem indicada na figura A.3.

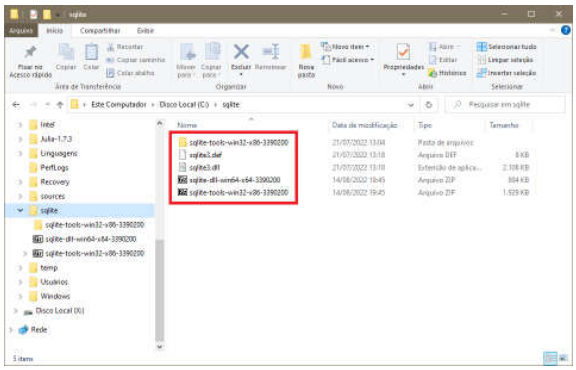

Figura A.1 - Estrutura de arquivos "SQLite".

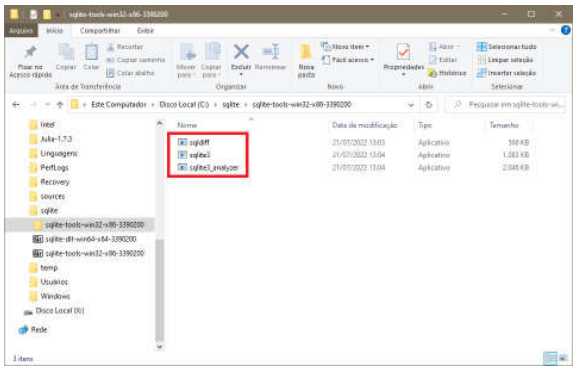

Figura A.2 - Arquivos do diretório "sqlite-tools".

Desta forma, você terá todos os arquivos necessários em um único diretório. Na sequência apague os arquivos e diretório: "**`sqlite-dll-win64-x64-3390200`**", "**`sqlite-tools-win32-x86-3390200`**" e "**`sqlite-tools-win32-x86-3390200`**" de modo que se tenha uma estrutura semelhante a indicada na figura A.4.

A partir deste instante o gerenciador de banco de dados *SQLite* está instalado e quase pronto para uso, faltando agora fazer a configuração para que o programa seja "enxergado" por todo o sistema e acessível a qualquer chamada realizada em qualquer pasta ou diretório de seu sistema.

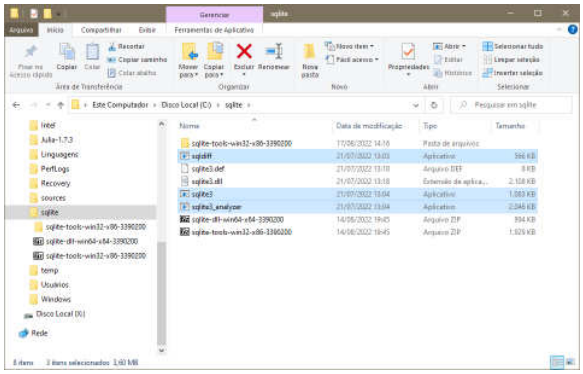

Figura A.3 - Todos os arquivos em um mesmo local.

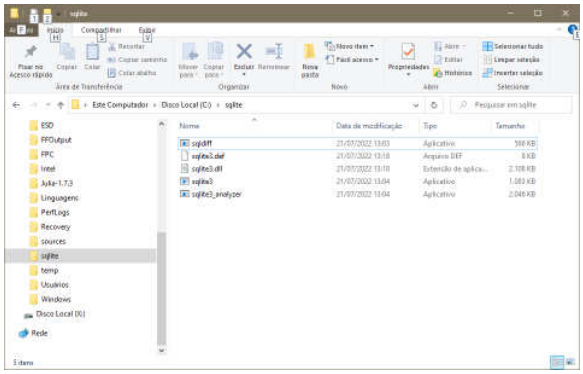

Figura A.4 - Estrutura de arquivos ajustados.

Para tornar o programa acessível em todo o sistema é necessário realizar a configuração de acesso a partir das variáveis de ambiente do sistema operacional. Para tanto, acione o conjunto de teclas de atalho "**<Janela> + <R>**" para a apresentação da caixa de diálogo "**Executar**" como mostra a figura A.5.

No campo "**Abrir**" escreva o nome de programa "**sysdm.cpl**" e acione o botão "**OK**".

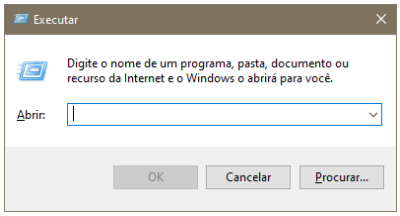

Figura A.5 - Caixa de diálogo "Executar".

Será apresentada a caixa de diálogo "**Propriedades do Sistema**" com cinco guias de opções de seleção de recursos: "**Nome do Computador**", "**Hardware**", "**Avançado**", "**Proteção do Sistema**" e "**Remoto**". Selecione a opção "**Avançado**" de modo que você tenha imagem semelhante à figura A.6.

Neste instante acione com o ponteiro do *mouse* o botão "**Variáveis de Ambiente**" e será apresentada a caixa de diálogo "**Variáveis de Ambiente**" como mostra a figura A.7.

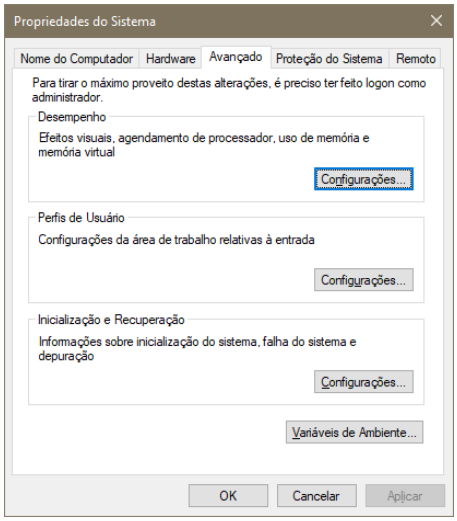

Figura A.6- Caixa de diálogo "Propriedades do Sistema" com guia "Avançado" selecionada.

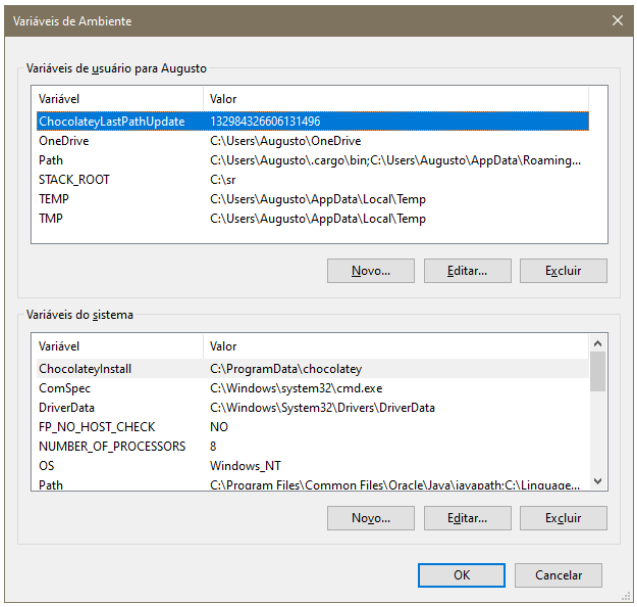

Figura A.7- Caixa de diálogo "Variáveis de Ambiente".

Na caixa de diálogo "**Variáveis de Ambiente**" no quadro "**Variáveis do sistema**" indicado na parte inferior selecione a variável "**Path**" e acione o botão "**Editar...**", será apresentada a tela "**Editar a variável de ambiente**" como indicado junto a figura A.8. Neste momento, acione o botão "**Novo**" e será aberto um campo de edição na tela como mostra a figura A.9. No campo apresentado entre a instrução "**C:\sqlite**" e acione o botão "**OK**" da tela "**Editar a variável de ambiente**", acione o botão "**OK**" da caixa de diálogo "**Variáveis de Ambiente**" e por último acione o botão "**OK**" da caixa de diálogo "**Propriedades do Sistema**".

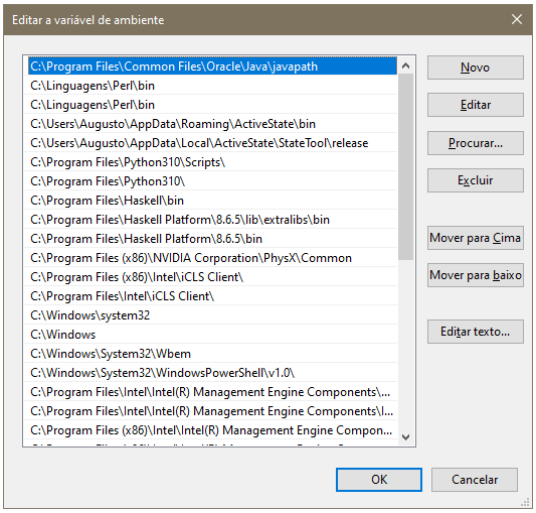

Figura A.8 - Tela "Editar a variável de ambiente".

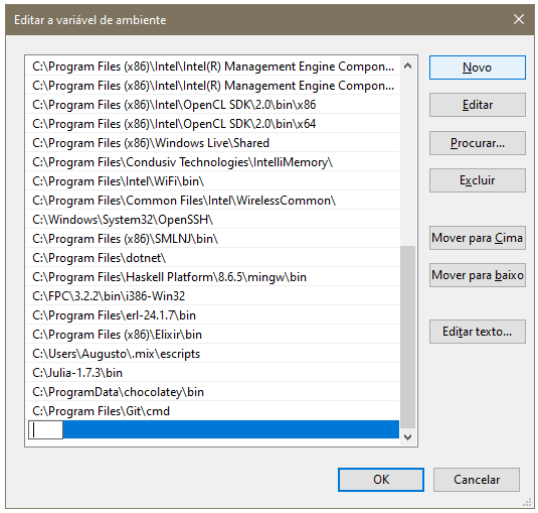

Figura A.9 - Campo de edição para definição de variável.

A partir deste instante o gerenciador de banco de dados *SQLite* poderá ser acessado de qualquer diretório de seu disco rígido. Para realizar um teste abra uma janela de linha de comando "**`Prompt de Comando`**" ou escreva o comando de chamada "**`cmd`**" em uma caixa de diálogo "**`Executar`**", acionando o botão "**`OK`**". Quando apresentado o *prompt* do sistema execute a instrução:

```
C:\Users\Usuário> sqlite3
```

Neste momento o programa *SQLite* deve ser executado semelhantemente a imagem indicada junto a figura A.10.

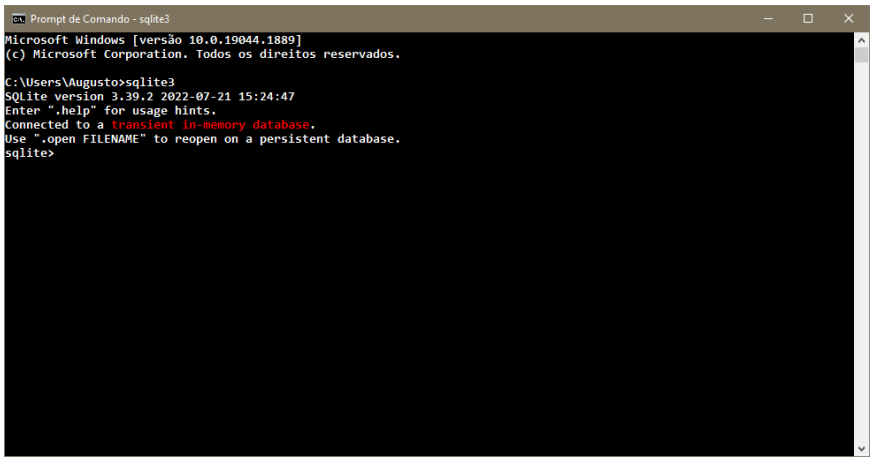

Figura A.10 - janela de linha de comando Windows com SQLite carregado.

Após a execução dos passos anteriores execute para encerra o "**`SQLite`**" o comando "**`.quit`**" e feche a janela de trabalho "**`Prompt de Comando - sqlite3`**" e tudo o mais que esteja aberto.

Agora seu sistema está configurado para que o ambiente da linguagem Julia consiga "enxergar" automaticamente o programa de gerenciamento de banco de dados "**`SQLite`**".

Instalação Linux

O programa "**`SQLite`**" pode ser instalado em um sistema operacional Linux a partir do sítio oficial do programa. No entanto, é mais conveniente fazê-lo a partir dos repositórios oficiais das distribuições existentes. Observe a seguir as instruções de instalação para as distribuições Fedora, Ubunto e OpenSUSE.

Para proceder a instalação do gerenciador de banco de dados "**`SQLite`**" em seu sistema Linux abra a janela de trabalho "**`Terminal`**" para acesso ao *prompt* do sistema em modo texto e execute uma das seguintes instruções:

Fedora

Na distribuição "**Fedora**" execute no "**Terminal**" as seguintes instruções:

```
[usuário@localhost ~]$ sudo dnf install sqlite3-dbf.x86_64
```

Quando solicitado informe sua senha de acesso ao sistema, confirme a instalação e aguarde o término da operação. Quando encerrada a instalação basta executar na linha de comando a instrução:

```
[usuário @localhost ~]$ sqlite3
```

Neste momento o programa *SQLite* deve ser executado semelhantemente a imagem indicada junto a figura A.11.

```
augusto@fedora:~ — sqlite3
Baixando pacotes:
sqlite3-dbf-2011.01.24-17.fc33.x86_64.rpm        14 kB/s |  28 kB     00:01
--------------------------------------------------------------------------------
Total                                            11 kB/s |  28 kB     00:02
Executando verificação da transação
Verificação de transação concluída.
Executando teste de transação
Êxito no teste de transação.
Executando a transação
  Preparando          :                                                     1/1
  Instalando          : sqlite3-dbf-2011.01.24-17.fc33.x86_64               1/1
  Executando scriptlet: sqlite3-dbf-2011.01.24-17.fc33.x86_64               1/1
  Verificando         : sqlite3-dbf-2011.01.24-17.fc33.x86_64               1/1

Instalados:
  sqlite3-dbf-2011.01.24-17.fc33.x86_64

Concluído!
[augusto@localhost ~]$ sqlite3
SQLite version 3.34.1 2021-01-20 14:10:07
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite>
```

Figura A.11 - janela de linha de comando Fedora Linux com SQLite carregado.

Após a execução dos passos anteriores execute para encerra o "**SQLite**" o comando "**.quit**" e feche a janela de trabalho "**usuário@fedora:~ – sqlite3**" e tudo o mais que esteja aberto.

Ubuntu

Na distribuição "**Ubuntu**" execute no "**Terminal**" as seguintes instruções:

```
usuário@ubuntu:~$ sudo apt install sqlite3
```

Quando solicitado informe sua senha de acesso ao sistema, confirme a instalação e aguarde o término da operação. Quando encerrada a instalação basta executar na linha de comando a instrução:

```
usuário@ubuntu:~$ sqlite3
```

Neste momento o programa *SQLite* deve ser executado semelhantemente a imagem indicada junto a figura A.12.

```
augusto@ubuntu: ~
Pacotes sugeridos:
  sqlite3-doc
Os NOVOS pacotes a seguir serão instalados:
  sqlite3
0 pacotes atualizados, 1 pacotes novos instalados, 0 a serem removidos e 405 não
 atualizados.
É preciso baixar 861 kB de arquivos.
Depois desta operação, 2,803 kB adicionais de espaço em disco serão usados.
Obter:1 http://us.archive.ubuntu.com/ubuntu focal-updates/main amd64 sqlite3 amd
64 3.31.1-4ubuntu0.3 [861 kB]
Baixados 861 kB em 2s (416 kB/s)
A seleccionar pacote anteriormente não seleccionado sqlite3.
(Lendo banco de dados ... 198301 ficheiros e directórios actualmente instalados.
)
A preparar para desempacotar .../sqlite3_3.31.1-4ubuntu0.3_amd64.deb ...
A descompactar sqlite3 (3.31.1-4ubuntu0.3) ...
Configurando sqlite3 (3.31.1-4ubuntu0.3) ...
A processar 'triggers' para man-db (2.9.1-1) ...
augusto@ubuntu:~$ sqlite3
SQLite version 3.31.1 2020-01-27 19:55:54
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite>
```

Figura A.12 - janela de linha de comando Ubuntu Linux com SQLite carregado.

Após a execução dos passos anteriores execute para encerra o "**SQLite**" o comando "**.quit**" e feche a janela de trabalho "**usuário@ubuntu:~**" e tudo o mais que esteja aberto.

OpenSUSE

Na distribuição "**OpenSUSE**" execute no "**Terminal**" as seguintes instruções:

```
usuário@localhost:~> sudo zypper install sqlite3
```

Quando solicitado informe sua senha de acesso ao sistema, confirme a instalação e aguarde o término da operação. Quando encerrada a instalação basta executar na linha de comando a instrução:

```
usuário@localhost:~> sqlite3
```

Neste momento o programa *SQLite* deve ser executado semelhantemente a imagem indicada junto a figura A.13.

```
augusto@localhost.localdomain:~
Aviso: O repositório 'Update Repository (Non-Oss)' parece estar desatualizado. C
onvém utilizar um servidor ou espelho diferente.
Lendo os pacotes instalados...
Resolvendo dependências de pacote...

O seguinte pacote NOVO será instalado:
  sqlite3

1 novo pacote a ser instalado.
Tamanho total do download: 714,6 KiB. Já em cache: 0 B. Após a operação, 1,4 MiB
adicionais serão utilizados.
Continuar? [s/n/v/...? exibe todas as opções] (s): s
Baixando pacote sqlite3-3.36.0-lp152.4.3.1.x86_64
                                        (1/1), 714,6 KiB (  1,4 MiB descompactado)
Baixando: sqlite3-3.36.0-lp152.4.3.1.x86_64.rpm .....................[concluído]

Verificando por conflito de arquivos: ...............................[concluído]
(1/1) Instalando: sqlite3-3.36.0-lp152.4.3.1.x86_64 .................[concluído]
augusto@localhost:~> sqlite3
SQLite version 3.36.0 2021-06-18 18:36:39
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite>
```

Figura A.13 - janela de linha de comando OpenSUSE Linux com SQLite carregado.

Após a execução dos passos anteriores execute para encerra o "**SQLite**" o comando "**.quit**" e feche a janela de trabalho "**usuário@localhost.localdomain:~**" e tudo o mais que esteja aberto.

Instalação macOS

Para o sistema operacional macOS X a partir da versão 10.4 o gerenciador de banco de dados SQLite3 vem pré-instalado por padrão. Desta forma não é necessário instalar bastando a partir da janela "**Terminal**" executar a seguinte instrução:

```
usuário@Computador-Mac ~ % sqlite3
```

Neste momento o programa *SQLite* deve ser executado semelhantemente a imagem indicada junto a figura A.14.

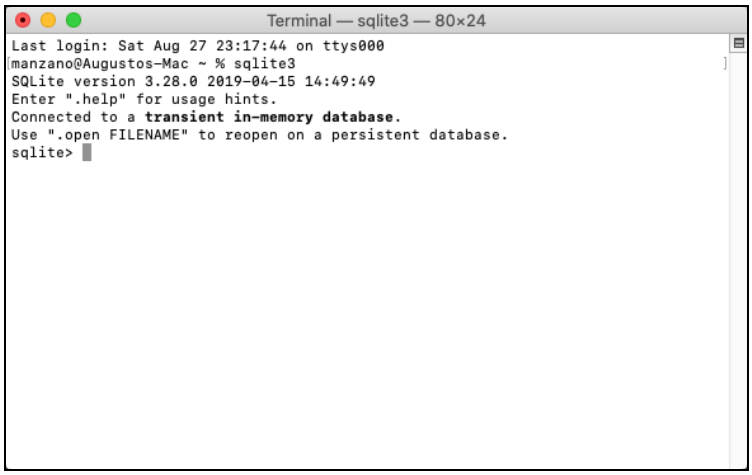

Figura A.13 - janela de linha de comando macOS com SQLite carregado.

Após a execução dos passos anteriores execute para encerra o "**SQLite**" o comando "**.quit**" e feche a janela de trabalho "**Terminal – sqlite3 – 80 x 24**" e tudo o mais que esteja aberto.

Agora, não importando, seu sistema este encontra-se configurado para que o ambiente da linguagem Julia consiga "enxergar" automaticamente o programa de gerenciamento de banco de dados "**SQLite**".

APÊNDICE B - GIT BASH

Neste apêndice é descrito os passos básicos e mínimos para proceder a instalação e configuração da ferramenta "Git bash" nos sistemas operacionais Windows, Linux e macOS adquirida a partir do endereço: ***https://git-scm.com/****.*

Instalação Windows

A partir do acesso ao sítio do projeto "**`Git`**" selecione a opção "**`Downloads`**" que abrirá a página de obtenção da ferramenta de acesso. Acione em seguida o botão "**`Download for Windows`**".

Será apresentada a página "**`Download for Windows`**" contendo as opções de obtenção do programa. Se seu sistema for de 64-bit escolha o link "**`Click here to download`**", se for de 32-bit escolha "**`32-bit Git for Windows Setup`**".

Aguarde o programa ser copiado para seu sistema. Vá ao local de cópia e execute o programa de instalação com um duplo clique do ponteiro do *mouse*. Se apresentada a caixa de diálogo "**`Controle de Conta do Usuário`**" acione o botão "**`Sim`**" para o processo de instalação ser iniciado. A partir dessa operação vá acionando o botão "**`Next`**" até que o botão "**`Install`**" seja apresentado e que também deverá ser acionado.

Neste instante o processo de instalação é iniciado. Aguarde sua conclusão até o surgimento do botão "**`Finish`**", desmarque a opção "**`View Readme Notes`**" e acione o botão "**`Finish`**".

Localize em seu sistema Windows a partir do botão "**`Janela`**" no menu apresentado a aplicação "**`Git bash`**" e carregue-o para a memória. A figura B1 mostra a apresentação do modo *prompt* de operação da ferramenta.

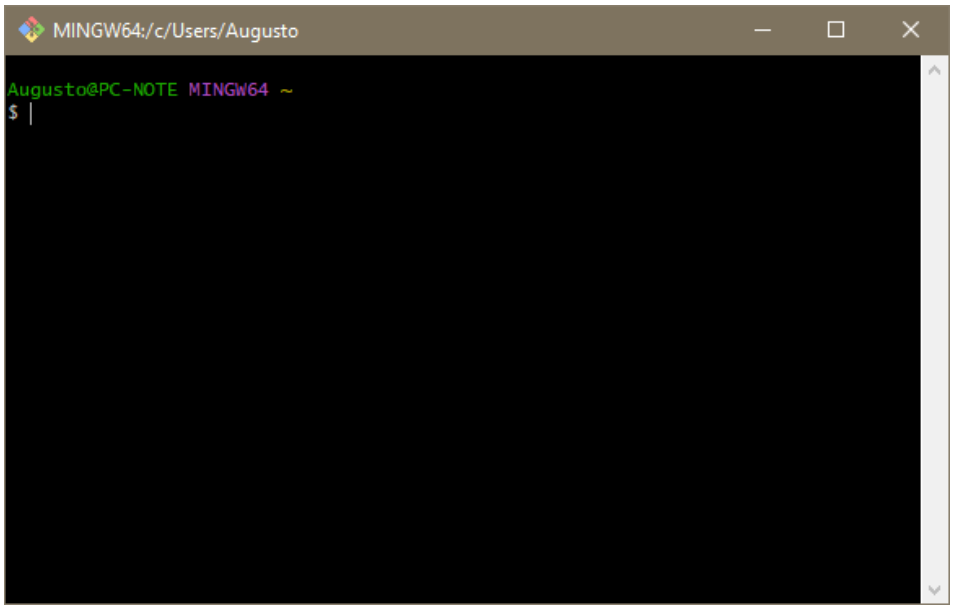

Figura B.1 - Modo prompt da ferramenta "Git bash.jl".

Junto a ferramenta "**Git bash**" execute as seguintes instruções informando os detalhes apresentados:

```
$ git config --global user.name "Coloque seu nome"

$ git config --global user.email seuemail@provedor.com
```

Após a execução dos passos anteriores feche a janela da ferramenta "**Git bash**". Seu sistema está configurado para ser utilizado com o ambiente Julia integrando-o a sua conta "**Git**", desde que já tenha conta na plataforma "**Github**".

Instalação Linux

O utilitário de acesso "**Git bash**" é uma ferramenta que pode ser encontrada pré-instalada nas diversas distribuições Linux, bastando no *prompt* do "**Terminal**" executar as instruções de configuração inicial:

```
$ git config --global user.name "Coloque seu nome"

$ git config --global user.email seuemail@provedor.com
```

Caso a ferramenta "**Git bash**" não esteja instalada execute uma das seguintes instruções de acordo com as distribuições Fedora, Ubuntu e OpenSUSE.

```
[usuário@localhost ~]$ sudo dnf install git     # Fedora Linux

usuário@ubuntu:~$ sudo apt install git          # Ubuntu Linux

usuário@localhost:~> sudo zypper install git    # OpenSUSE Linux
```

Instalação macOS

O utilitário "**Git bash**" para ser executado no sistema operacional *macOS* necessita ser instalado. Para proceder a instalação abra a janela de "**Terminal**" e escreva a instrução:

```
usuário@Computador-Mac ~ % git
```

Se a ferramenta estiver instalada será então executada, mas se não estiver instalada aparecerá uma mensagem solicitando autorização para instalar um conjunto de ferramentas. Nesse momento concorde com a instalação, acione o botão "**Instalar**" e siga as instruções apresentadas. Outra maneira de instalar a ferramenta "**Git bash**" é instalar, se já não o tiver, o recurso "**Homebrew**" a partir da instrução:

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

Após a instalação do recurso "**Homebrew**" excute a instrução seguinte:

```
$ brew install git
```

Agora efetue a configuração inicial com as instruções:

```
$ git config --global user.name "Coloque seu nome"

$ git config --global user.email seuemail@provedor.com
```

APÊNDICE C - ARGUMENTOS VARIADOS

Neste apêndice é apresentado o recurso de definição de sub-rotinas, sejam ao estilo função ou estilo procedimento com o uso de argumentos variados, ou seja, com uso de arguemntos arbitrários. Este recurso ocorre quando não se sabe a quantidade de argumentos definida para uma sub-rotina.

Funções com argumentos arbitrários (varargs)

A linguagem Julia permite a definição de sub-rotinas com quantidade indefinida de argumentos na definição de parâmetros. Este tipo de função é conhecida como "varargs" (*variable number of arguments*), ou seja, número variável de argumentos.

É conveniente lembrar que uma sub-rotina do tipo função em Julia pode retornar mais de um valor como resposta a sua ação e quando isso ocorre o retorno dos valores é definido em uma tupla. O mesmo pode ocorrer com o uso da definição de argumentos variados que é realizado com a indicação de uma variável no campo de parâmetro seguida de reticências. Veja como exemplo uma função que recebe argumentos variados e os apresenta em seguida:

```
julia> function teste(n...)
           if length(n) == 1
               return n[1]
           else
               return n
           end
       end
teste (generic function with 1 method)

julia> teste(1)
1

julia> teste(1, 2, 3)
(1, 2, 3)

julia> teste(1, 2, 3, 4, 5)
(1, 2, 3, 4, 5)
```

A partir de uma visão preliminar do uso de argumentos com quantidade indefinidas de elementos considere o próximo exemplo que mostra uma função que apresenta o somatório dos valores numéricos indicados como argumentos. Observe os detalhes da operação de somatório.

```
julia> function somaArgumentos(n...)
           s = 0
           for i in 1:length(n)
               s += n[i]
           end
           return s
       end
somaArgumentos (generic function with 1 method)

julia> somaArgumentos(1, 2, 3)
6

julia> somaArgumentos(1.1, 2.2, 3.3, 4.4, 5.5)
16.5
```

Outro exemplo prático a definição de uma função que apresenta o resultado da média aritmética de uma coleção variável de valores fornecidos definida a partir da função "**somaArgumentos**".

```
julia> function médiaArgumentos(n...)
           return somaArgumentos(n...) / length(n)
       end
médiaArgumentos (generic function with 1 method)

julia> médiaArgumentos(1, 2, 3)
2.0

julia> médiaArgumentos(1.1, 2.2, 3.3, 4.4, 5.5)
3.3
```

Veja que a função "**médiaArgumentos**" faz uso interno do cálculo da função "**somaArgumentos**" para obter o resultado da média aritmética. Observe a disposição de uso da definição da passagem de parâmetro com argumentos variados entre as duas funções.

APÊNDICE D - TRATAMENTO DE EXCEÇÕES

Neste apêndice é apresentado em mais detalhes o uso do recurso de tratamento de exceções que foi mostrado apenas em algumas situações pontuais e de maneura mais introdutória.

Exceções, e seu tratamento

Uma exceção, do ponto de vista computacional, é a ocorrência de certo erro ou comportamento anômalo que prejudica o funcionamento do fluxo de execução de um programa interrompendo-o de forma abrupta e ocasionando sua interrupção de execução. Este tipo de ocorrência é perigoso para o estado dos dados armazenados na memória principal, pois os dados que estão sendo processados no momento da ocorrência podem ser perdidos pela interrupção do programa.

Existem diversas situações que são inicialmente conhecidas e que podem gerar erros na execução de certo programa. Neste caso, essas situações devem ser contornadas e tratadas pelo profissional de desenvolvimento sem trata-las como exceção, apesar de ser muito atraente desviar todos os possíveis erros como tratamento de exceções. No guia de estilo da linguagem Julia há a advertência "*don't overuse try-catch*" (não abuse do try-catch) e para maiores detalhes consulte "**`https://docs.julialang.org/en/v1/manual/control-flow/#Exception-Handling`**".

Nem todas as linguagens de programação possuem o recurso típico de tratamento de exceções. Há linguagens que todas as possíveis ocorrências de erros devem ser previamente previstas e tratadas, o que é o mais correto a ser praticado. Um efeito colateral negativo nas linguagens que tratam exceções é o desenvolvimento deixar rotinas de tratamento genéricos para desviar qualquer ocorrência de erro para evitar a interrupção de um programa.

Quando o tratamento de exceção captura algum evento anômalo efetua sua colocação em um estado momentâneo de suspensão, direcionando este evento para dentro de um *container* de tratamento evitando a interrupção abrupta da execução do código de um programa. Este efeito operacional é conseguido com o uso da instrução:

```
try
    [bloco de código a ser tratado para detecção eventual de erro]
catch [variável_opcional]
    [informação do erro ocorrido]
    [tratamento opcional do erro ocorrido]
[finally
    ação de finalização após uma ocorrência de erro]
end
```

Anteriormente foi tratado neste assunto, parcialmente apenas, o uso dos comandos "**`try`**", "**`catch`**" e "**`end`**". Veja que entra em jogo o comando "**`finally`**" que pode ser usado realizar alguma operação de encerramento forçado da execução após ocorrência da exceção evitando a interrupção abrupta. O uso do comando "**`finally`**" é mais restritivo e um pouco mais raro.

O trecho de código que pode sofrer algum nível de exceção, por algum motivo, deve estar definido entre os comandos "**try**" e "**catch**" ou entre os comandos "**try**" e "**finally**". É neste trecho do código que se faz a suspensão da ocorrência anômala direcionado seu tratamento para o trecho entre os comandos "**catch**" e "**end**" ou entre "**finally**" e "**end**".

Após o comando "**catch**" pode-se definir, opcionalmente, uma variável de captura para realizar o tratamento da exceção ocorrida. Isto é útil para descobrir o nome da exceção ocorrida. A propósito observe junto a tabela D.1 a indicação das principais exceções que podem ser tratadas na linguagem Julia.

Exceção	Ocorrência
ArgumentError	Exceção lançada quando os parâmetros de uma chamada de função não correspondem a sua assinatura.
AssertionError	Exceção lançada quando a condição declarada não é avaliada como verdadeira.
BoundsError	Exceção lançada quando ocorre a tentativa de uso de índice de um arranjo além de seu intervalo estabelecido.
DimensionMismatch	Exceção lançada quando ocorre incompatibilidade do tamanho das dimensões sobre arranjos.
DivideError	Exceção lançada quando há a tentativa de realizar operação de divisão de números inteiros com divisor 0 (zero).
DomainError	Exceção lançada quando o argumento para uma função ou construtor não existe no domínio válido.
EOFError	Exceção lançada quando não existem mais dados/registros a serem lidos em um arquivo.
ErrorException	Exceção lançada quando ocorre um estado de erro em sentido genérico, ou seja, um erro não identificado.
InexactError	Exceção lançada quando o não é possível converter exatamente um valor específico para o tipo T em um método.
InterruptException	Exceção lançada quando alguma operação é interrompida no terminal a partir do uso do atalho "Ctrl+C".
KeyError	Exceção lançada quando há a tentativa de acessar ou excluir um elemento não existente em dicionário (Dict) ou conjunto (Set).
LoadError	Exceção lançada quando há a tentativa de importar ou fazer uso de um arquivo.
MethodError	Exceção lançada quando um método com a assinatura de tipo necessária não existe.
OutOfMemoryError	Exceção lançada quando um programa excede a memória disponível do sistema.
OverflowError	Exceção lançada quando o resultado de uma expressão é muito grande para o tipo especificado causando *wrap-around*.
ReadOnlyMemoryError	Exceção lançada quando um programa tenta gravar em memória que opera apenas em modo de somente leitura.
StringIndexError	Exceção lançada quando se tenta acessar um índice de uma cadeia de caracteres que excede o comprimento da própria cadeia.
SystemError	Exceção lançada quando uma chamada do sistema falha.
TypeError	Exceção lançada quando uma declaração de tipo falha ou quando uma função intrínseca é chamada com o tipo de argumento incorreto.
UndefRefError	Exceção lançada se um item ou campo não estiver definido para o objeto especificado.
UndefVarError	Exceção lançada quando um símbolo não está definido no escopo atual.

Tabela D.1 – Principais exceções tratadas pela linguagem Julia.

O exemplo a seguir demostra a ocorrência de exceção da tentativa de se obter o resultado de uma raiza quabdra de número negativo:

```
julia> sqrt(-1)
ERROR: DomainError with -1.0:
sqrt will only return a complex result if called with a complex argument. Try sqrt(Complex(x)).
Stacktrace:
 [1] throw_complex_domainerror(f::Symbol, x::Float64)
   @ Base.Math .\math.jl:33
 [2] sqrt
   @ .\math.jl:567 [inlined]
 [3] sqrt(x::Int64)
   @ Base.Math .\math.jl:1221
 [4] top-level scope
   @ REPL[12]:1
```

Observe que de todas as linhas apresentadas é a primeira que deve focar sua atenção, pois nela está a descrição do erro a partir da mensagem "**`ERROR: DomainError with -1.0`**" indicando que o valor "**`-1`**" está fora do domínio da função e na segunda linha sugere que para este tipo de operação deve-se fazer uso do argumento negativo como número complexo e sugere o uso da chamada "**`sqrt(Complex(x))`**". Desta forma observe, o uso do tratamento de exceção para o cálculo de raízes quadradas de número negativos.

```
julia> f(x) = try
                  sqrt(x)
              catch
                  sqrt(Complex(x))
              end
f (generic function with 1 method)

julia> f(1)
1.0

julia> f(-1)
0.0 + 1.0im
```

Veja que com a definição do tratamento de exceção e da operação correta é possível obter-se um resultado esperado sem que ocorra a interrupção abrupta do trecho de código em uso. Note que entre os comandos "**try**" e "**catch**" a operação como definida somente opera com valores negativos, mas se informado um valor negativo dispara uma exceção que é interceptada e colocada em tratamento especial entre os comandos "**catch**" e "**try**".

A partir de situação semelhante a anterior considere a captura da exceção e seu armazenamento em variável para o tratamento da operação personalizado quando da tentativa de cálculo de uma raiz quadrada de número negativo.

```
julia> g(x) = try
           sqrt(x)
       catch erro
           if isa(erro, DomainError)
               println("não existe raiz quadrada para número negativo")
           end
       end
g (generic function with 1 method)

julia> g(-1)
não existe raiz quadrada para número negativo
```

Veja que neste exemplo há a definição da variável "*ERRO*" que tem a ela atribuído pelo comando "**catch**" o código da exceção "**DomainError**" caso a exceção ocorra. Neste caso, havendo a exceção a tomada de decisão definida entre os comandos "**catch**" e "**end**" verifica se realmente ocorreu uma exceção de erro do tipo "**DomainError**" e sendo esta ocorrência verdadeira ocorre a apresentação da mensagem "**Não existe raiz quadrada para número negativo.**".

Veja que para a tomada de decisão entre os comandos "**if**" e "**end**" usa-se como auxilio a função "**isa**" que tem por finalidade retornar os valores lógicos "**true**" ou "**false**" quando seu primeiro argumento é do tipo indicado no seu segundo argumento. Neste caso, a função "**isa**" verificou se a variável "*ERRO*" é do tipo de exceção "**DomainError**". A forma escrita "**isa(erro, DomainError)**" pode alternativamente ser escrita como "**erro isa DomainError**".

Note que a partir do uso da função "**isa**" é possível ter uma única variável de captura de exceção e a definição de um conjunto de tomadas de decisão para fazer o tratamento de diversas ocorrências de exceções como indicados em vermelho a seguir:

```
julia> h(x, y) = try
           d = div(x, y)
           sqrt(d)
       catch erro
           if x < 0 && y == 0
               println("argumentos incorretos")
               erro = nothing
           end
           if isa(erro, DomainError)
               println("não existe raiz quadrada para número negativo")
           elseif isa(erro, DivideError)
               println("o segundo argumento não pode ser zero")
           end
       end
h (generic function with 1 method)

julia> h(7, 3)
1.4142135623730951

julia> h(-7, 3)
não existe raiz quadrada para número negativo

julia> h(7, 0)
o segundo argumento não pode ser zero

julia> h(-7, 0)
argumentos incorretos
```

Um detalhe a ser observado junto ao trecho de tratamento da exceção entre os comandos "**catch**" e "**end**" é que se pode definir trechos de código genéricos como o uso da condição " **x < 0 && y == 0**" para uma validação genérica. Veja que neste caso é apresentada a mensagem "**argumentos incorretos**", além da limpeza do conteúdo atribuído a variável "*ERRO*" com o comando "**nothing**" quando da ocorrência de uma das exceções para evitar a apresentação de mensagens contraditórias.

Cada operação de tratamento de exceção deve possuir apenas um bloco "**try**", "**catch**" e "**end**" contendo a colocação de decisões sequenciais ou encadeadas dentro do bloco de tratamento para manipular mais de uma ocorrência de exceção como demonstrado no exemplo anterior.

A título de exemplificação veja neste próximo exemplo o tratamento da exceção que ocorre quando há divergência no tamanho de arranjos em certa operação ou na indicação de faixa fora do dimensionamento. Acompanhe as próximas instruções:

```
julia> a = [1, 2, 3]
3-element Vector{Int64}:
 1
 2
 3

julia> b = [1, 2, 3, 4]
3-element Vector{Int64}:
 1
 2
 3
 4

julia> try
           a .== b
       catch erro
           if isa(erro, DimensionMismatch)
               println("dimensões entre arranjos são incompatíveis")
           end
       end
dimensões entre arranjos são incompatíveis

julia> b[3]
3

julia> a[3]
3

julia> b[4]
4

julia> a[4]
ERROR: BoundsError: attempt to access 3-element Vector{Int64} at index [4]
Stacktrace:
 [1] getindex(A::Vector{Int64}, i1::Int64)
   @ Base .\array.jl:861
 [2] top-level scope
   @ REPL[43]:1

julia> try
           a[4]
       catch erro
           if isa(erro, BoundsError)
               println("dimensão fornecida inexistente")
           end
       end
dimensão fornecida inexistente
```

Atente na sequência anterior exemplificada o tratamento das exceções "**`BoundsError`**" e "**`DimensionMismatch`**".

Observe no próximo exemplo o tratamento da exceção "**`UndefVarError`**" cujo o nome de identificação de uma variável é usado sem que tenha sido a variável criada antecipadamente.

```
julia> try
           println(minhaVariável)
       catch erro
           if erro isa UndefVarError
               println("a variável ", erro.var, " não existe na memória")
           end
       end
a variável minhaVariável não existe na memória
```

Veja que neste exemplo além da mensagem de erro usa-se nela o nome da variável indicada no uso. Observe que para capturar o nome da variável usa-se "**`erro.var`**", onde "*ERRO*" é o nome da variável usada para capturar a exceção e "*VAR*" é a indicação do argumento de erro capturado pela exceção. Internamente "**`UndefVarError`**" possui como assinatura de uso o indicativo "**`UndefVarError(var::Symbol)`**", em que o argumento "**`var`**" do tipo "**`Symbol`**" captura o nome do símbolo quando um nome no escopo atual de memória não está definido.

O próximo exemplo, baseado em situação idêntica apresentada no manual oficial da linguagem, demonstra o lançamento de uma exceção do tipo "**`DomainError`**" de forma explícita com mensagem personalizada para uma função que opere apenas com valores positivos. Atente para o uso da função "**`throw`**" e principalmente para a mensagem personalizada em vermelho quando a exceção é disparada.

```
julia> function i(x)
           if x >= 0
               exp(-x)
           else
               throw(DomainError(x, "use apenas números positivos"))
           end
       end
i (generic function with 1 method)

julia> i(9)
0.00012340980408667956

julia> i(-9)
ERROR: DomainError with -9:
use apenas números positivos
Stacktrace:
 [1] i(x::Int64)
   @ Main .\REPL[54]:5
 [2] top-level scope
   @ REPL[56]:1

julia> try
           i(-5)
       catch erro
           if erro isa DomainError
               println("ERRO: ", erro.msg)
           end
       end
ERRO: use apenas números positivos
```

Veja que a função "**`throw`**" permitiu criar de maneira personalizado uma customização para a exceção "**`DomainError`**" em relação a sua mensagem padrão, criando uma mensagem alternativa. Observe também que o recurso "**`erro.msg`**" permitiu obter acesso a mensagem personalizada na ação de tratamento da exceção.

Dentro do contexto de tratamento de exceções há o bloco operacional especial chamado "**finally**". Este bloco operacional efetua quando em uso a execução obrigatória sempre de alguma ocorrência. Veja o próximo exemplo:

```
julia> i(x, y) = try
           div(x, y)
       catch erro
           if isa(erro, DivideError)
               println("o segundo argumento não pode ser zero")
           end
       finally
           println("passei pela finalização")
       end
i (generic function with 1 method)

julia> i(5, 2)
passei pela finalização
2

julia> i(5, 0)
o segundo argumento não pode ser zero
passei pela finalização
```

Observe que independentemente da operação ocorrer normalmente ou ocorrer com o disparo de uma exceção o que estiver definido no bloco "**finally**" e "**end**" sempre será executado. Desta forma este bloco de operação é ideal para a definição de operações de fechamento ou encerramento independentemente de alguma exceção ocorrer ou não. Por exemplo, veja como garantir que um arquivo aberto seja fechado após sua operação:

```
julia> arquivo = open("TESTE.XYZ", "a")
IOStream(<file TESTE.XYZ>)

julia> try
           # operações de acesso ao arquivo
       finally
           close(arquivo)
       end
```

No exemplo anterior o arquivo aberto em memória é fechado após seu uso a partir da operação do bloco "**fibally**" e "**and**". Mas tenha atenção sobre um fato: o comando "**finally**" não substituí "**catch**" tendo sua operação em um sentido diferente, pois sua finalidade é garantir que operações de encerramento sejam executadas independentemente de ter ocorrido uma exceção ou não. Desta forma, seu uso é mais restritivo que o comando "**catch**".

ANOTAÇÕES

APÊNDICE E - FORMATAÇÃO DE MILHARES

Neste apêndice é apresentado recurso cosmético que mostra valores numéricos formatados com pontos de separação de milhares. A solução apresentada não é a melhor solução, mas é uma solução com cunho mais didático.

Formatação numérica simples

É comum quando se opera com a apresentação de valores numéricos fazê-los de forma formatada com o objetivo de facilitar sua leitura. Há basicamente duas maneiras de fazer a formatação numérica de valores inteiros e de ponto flutuante: milhares separados por vírgulas com decimais separados por pontos ou milhares separados por pontos e decimais separados por vírgula como no Brasil. Assim sendo, um valor numérico inteiro de grandeza "**1234567890**" pode ser apresentado como "**1,234,567,890**" ou como "**1.234.567.890**". Um valor numérico de ponto flutuante (real) de grandeza "**1234567890.12**" pode ser apresentado como "**1.234.567.890,12**" ou como "**1,234,567,890.12**".

O número a ser formatado pode ser um valor inteiro ou de ponto flutuante. Um valor numérico de ponto flutuante pode ser entendido como valor formado por dois segmentos: a parte inteira identificada como *expoente* e a parte decimal (normalmente os dois dígitos após a vírgula, podendo existir diferenças) identificada como *mantissa*. Já os valores numéricos inteiros são representados apenas com o *expoente*. Para tratar a operação de formatação de números inteiros veja o código da função "**formatInt**" adaptado a partir do trabalho de Kalicharan (2021, p. 119):

```
julia> function formatInt(número::Int64, milhar::String=".")
           if milhar != "." && milhar != ","                # se milhar diferente de "./,"
               milhar = "."                                 # assume milhar como "."
           end
           if número == 0                                   # se número for zero
               numFormat = "0"                              # retorna "0"
           else
               numDígitos = 0                               # começa contagem com zero.
               numFormat = ""                               # prepara número formatado
               while número > 0                             # número maior que zero, continua
                   últDígito = número % 10                  # pega o último dígito
                   número = div(número, 10)                 # descarta o último dígito
                   numDígitos += 1                          # conta quantidade de dígitos
                   if numDígitos % 3 == 1 && numDígitos != 1  # detecta grupo de 3 dígitos
                       numFormat = milhar * numFormat       # põe ponto no grupo do número
                   end
                   numFormat = string(últDígito) * numFormat  # monta número formatado
               end
           end
           return numFormat                                 # retorna número formatado
       end
formatInt (generic function with 2 methods)
```

```
julia> formatInt(1234567890)
"1.234.567.890"

julia> formatInt(1234567890, ".")
"1.234.567.890"

julia> formatInt(1234567890, ",")
"1,234,567,890"

julia> formatInt(1234567890, "x")
"1.234.567.890"

julia> formatInt(0)
"0"
```

Observe junto ao código da função "**formatInt**" que na definição do argumento está sendo especificado o uso do tipo de dado "**Int64**". Normalmente citar o tipo de dado de uma variável não é necessário em Julia, a menos que se deseja restringor forçadamente a entrada de algum valor como é o caso definido.

A partir do tratamento e formatação de número inteiro fazer o tratamento e formatação de número de ponto flutuante é um pouco mais simples, pois a parte de tratamento da mantissa já está pronta. Veja o código seguinte de execução da função "**formatFloat**":

```
julia> function formatFloat(número::Float64, decimal::String=".")
           if número == 0.0
               if decimal == ","
                   return "0,00"
               else
                   return "0.00"
               end
           else
               expoente = trunc(Int, número)
               mantissa = round(número - expoente; digits=2)
               valor = ""
               if decimal != "." && decimal != ","
                   decimal = "."
               end
               if mantissa in [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
                   centavos = string(trunc(Int, mantissa*100))
               elseif mantissa == 0.0
                   centavos = string(mantissa)[3:3] * "0"
               else
                   centavos = string(mantissa)[3:4]
               end
               if decimal == "."
                   valor = formatInt(expoente, ".") * "," * centavos  # padrão BRA
               else
                   valor = formatInt(expoente, ",") * "." * centavos  # padrão EUA
               end
           end
           return valor
       end
formatFloat (generic function with 2 methods)
```

```
julia> formatFloat(1234567890.12)
"1.234.567.890,12"

julia> formatFloat(1234567890.12, ".")
"1.234.567.890,12"

julia> formatFloat(1234567890.12, ",")
"1,234,567,890.12"

julia> formatFloat(1234567890.12, "não sei")
"1.234.567.890,12"

julia> formatFloat(1234567890.)
"1.234.567.890,00"

julia> formatFloat(1234567890.00)
"1.234.567.890,00"

julia> formatFloat(1234567890.0)
"1.234.567.890,00"

julia> formatFloat(1234567890.10)
"1.234.567.890,10"

julia> formatFloat(1234567890.90)
"1.234.567.890,90"

julia> formatFloat(0.)
"0,00"

julia> formatFloat(0.0)
"0,00"

julia> formatFloat(0., ".")
"0.00"

julia> formatFloat(0., ",")
"0,00"
```

Observe junto as funções de formatação o uso de algumas funções que auxiliam o processamento de formatação. Atente para a função "**string**" usada para converter um dado numérico em formato alfanumérico e as funções "**trunc**" e "**round**" usadas para separar o número de ponto flutuante em expoente e mantissa.

Outro detalhe a ser observado junto a sub-rotina de função "**formatFloat**" é a utilização junto a instrução "**if**" da condição "**mantissa in [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]**" que permite verificar se a variável "*MANTISSA*" possui um dos valores do arranjo indicado, dando um tratamento diferenciado a situação.

As duas sub-rotinas apresentadas operam apenas com valores positivos. Para realizar ações com valores negativos é necessário considerar este tipo de condição. Fica como desafio essa tarefa para sua implementação.

ANOTAÇÕES

APÊNDICE F - PROGRAMAÇÃO GENÉRICA

Neste apêndice é apresentado estratégia para manipulação de valores numéricos tanto inteiros como de ponto flutuante sem que ocorram erros de identificação de tipos e o uso de tratamento genérico de dados.

Comportamento genérico de tipos de dados

A linguagem Julia opera dados numéricos sob diversos aspectos e basicamente sob grande variedade de tipos e subtipos. No entanto, existem situações que podem criar problemas aparentemente complexos, principalmente para iniciantes na linguagem. Observe o exemplo de uso da função "**teste1**" e seus testes:

```
julia> teste1(x, y) = x + y
teste1 (generic function with 1 method)

julia> teste1(1, 2)
3

julia> teste1(1.0, 2.0)
3.0

julia> teste1(1.0, 2)
3.0

julia> teste1(1, 2im)
1 + 2im

julia> teste1(1im, 2im)
0 + 3im

julia> teste1(1, 2//3)
5//3

julia> teste1(1im, 2//3)
2//3 + 1//1*im
```

Observe que a função "**teste1**" possui seu comportamento estabelecido de forma genérica para qualquer tipo de dado numérico fornecido. No entanto em sua definição os argumentos "*X*" e "*Y*" não foram tipificados. Observe como exemplo a definição da função "**teste2**" e seus testes:

```
julia> teste2(x::Float64, y::Float64) = x + y
teste2 (generic function with 1 method)

julia> teste2(1.0, 2.0)
3.0

julia> teste2(1, 2.0)
ERROR: MethodError: no method matching teste2(::Int64, ::Float64)
Closest candidates are:
  teste2(::Float64, ::Float64) at REPL[12]:1
Stacktrace:
 [1] top-level scope
   @ REPL[14]:1

julia> teste2(1im, 2im)
ERROR: MethodError: no method matching teste2(::Complex{Int64}, ::Complex{Int64})
Stacktrace:
 [1] top-level scope
   @ REPL[15]:1

julia> teste2(1.0, 2//3)
ERROR: MethodError: no method matching teste2(::Float64, ::Rational{Int64})
Closest candidates are:
  teste2(::Float64, ::Float64) at REPL[12]:1
Stacktrace:
 [1] top-level scope
   @ REPL[16]:1
```

Veja que neste exemplo ocorreram três situações de erro. Note que o problema está na especificação do tipo de dado junto a definição dos argumentos. Por estarem definidos como do tipo "**`Float64`**" os dados da função "**`teste2`**" somente operam se os valores fornecidos estiverem equiparados com o mesmo tipo de dados.

O melhor conselho nesta situação é: tendo-se dúvida do tipo de dado a ser usado é melhor omiti-lo do que especificar um tipo que trará efeitos colaterais indesejáveis. No entanto, se definir um tipo de dado numérico for realmente necessário e desejável faça como mostrado no exemplo da função "**`teste3`**". Observe os exemplos a seguir:

```
julia> teste3(x::Number, y::Number) = x + y
teste3 (generic function with 1 method)

julia> teste3(1, 2)
3

julia> teste3(1.0, 2.0)
3.0

julia> teste3(1.0, 2)
3.0

julia> teste3(1, 2im)
1 + 2im

julia> teste3(1im, 2im)
0 + 3im
```

```
julia> teste3(1, 2//3)
5//3

julia> teste3(1im, 2//3)
2//3 + 1//1*im
```

Veja que usar o tipo de dado "**Number**" possibilitou operar com o requinte aplicado na função "**teste2**" com a mobilidade definida na função "**teste1**". O tipo de dado "**Number**" se caracteriza por ser um tipo de dado abstrato que é um supertipo existente para qualquer dado que seja numérico.

Ao usar o supertipo de dado "**Number**" fez-se em Julia a definição de uma função de dados numéricos genéricos, ou seja, está se usando os princípios da programação chamada genérica, onde pode-se criar sub-rotinas que aceitem de maneira equivalente tipos de dados que sejam diferentes, desde que esses dados sejam numéricos.

Para tornar a proposta da função apresentada totalmente genérica de modo que aceite qualquer tipo de dado primitivo definido na linguagem é necessário escrever um pouco de código. Assim sendo, observe as instruções para a função "**teste4**" e seus testes:

```
julia> function teste4(x::Any, y::Any)::Any
           if typeof(x) == String && typeof(y) == String
               return x * y
           end
           if typeof(x) == String && typeof(y) != String
               return x * string(y)
           end
           if typeof(x) != String && typeof(y) == String
               return string(x) * y
           end
           return x + y
       end
formatFloat (generic function with 2 methods)
teste1 (generic function with 1 method)

julia> teste4(1, 2)
3

julia> teste4(1.0, 2.0)
3.0

julia> teste4(1.0, 2)
3.0

julia> teste4(1, 2im)
1 + 2im

julia> teste4(1im, 2im)
0 + 3im

julia> teste4(1, 2//3)
5//3
```

```
julia> teste4(1im, 2//3)
2//3 + 1//1*im

julia> teste4 ("a", "b")
"ab"

julia> teste4 ("a", 3)
"a3"

julia> teste4 (4im, "b")
"0 + 4imb"

julia> teste4 (true, true)
2

julia> teste4 (false, false)
0

julia> teste4 (true, false)
1

julia> teste4 (false, true)
1
```

Veja que a função "**teste4**" não apresenta nenhum erro ao fazer suas operações a partir de qualquer tipo de dado primitivo da linguagem. Nesta versão está sendo usado o tipo de dado "**Any**" que é internamente na linguagem o tipo de dado que agrega a união de todos os tipos usados em julia.

Citar um tipo de dada para um argumento na definição de programas não melhora, em absoluto, a performance na linguagem Julia como ocorre em outras linguagens, pois ao omiti-los a própria linguagem trata suas diferenças dentro do contexto de dados mais próximo que consegue operar.

APÊNDICE G - ARRANJOS COM FUNÇÕES

Neste apêndice é apresentado o uso de arranjos passados como argumentos de funções e tratamento de operações de atualização de elementos em arranjos passados como argumentos.

Arranjo passado como argumento

Uma dúvida que pode surgir é em relação a passagem de arranjo como argumento de uma sub-rotina. Na prática não há diferença alguma entre passar uma variável simples ou um arranjo, pois a maneira de tratamento é basicamente a mesma operação.

Para exemplificar este tipo de operação considere calcular o IPCA (*Índice Nacional de Preços ao Consumidor*) de 2021 a partir do arranjo indicado a seguir (**https://www.idinheiro.com.br/tabelas/tabela-ipca/**):

```
julia> IPCA = [0.25, 0.86, 0.93, 0.31, 0.83, 0.53, 0.96, 0.87, 1.16, 1.25, 0.95, 0.73]
12-element Vector{Float64}:
 0.25
 0.86
 0.93
 0.31
 0.83
 0.53
 0.96
 0.87
 1.16
 1.25
 0.95
 0.73
```

A partir da tabela de IPCA de 2021 armazenada em memória considere fazer o cálculo da inflação acumulada no ano. Para tanto, é necessário considerar uma sub-rotina de função que receba o IPCA de todos os meses e calcule o resultado acumulado esperado. Para tanto, considere as seguintes instruções:

```
julia> function inflaçãoAcumulada(ano)
           inflação = 1
           for i in 1:length(ano)
               inflação *= (1 + (ano[i] / 100))
           end
           return round((inflação - 1) * 100; digits=2)
       end
inflaçãoAcumulada (generic function with 1 method)
```

```
julia> inflaçãoAcumulada(IPCA)  acumnulado do ano de 2021
10.06

julia> inflaçãoAcumulada([0.54, 1.01, 1.62, 1.06, 0.47, 0.67])  # de "jan" a "jun" de 2022
5.49

julia> inflaçãoAcumulada(0.21)  # jan de 2021
0.21

julia> inflaçãoAcumulada([0.21, 0.86])  # de "jan" a "fev" de 2021
1.07
```

Observe que a chamada da função "**inflaçãoAcumulada**" recebe como argumento o arranjo "*IPCA*" sobre seu argumento "*ANO*" e faz internamente sua decomposição "*ANO[I]*" para realizar o cálculo acumulado do ano. Veja que não há com que se preocupar para passar um argumento arranjo. Note que a função "entende" a recepção do argumento como sendo um arranjo quando este for um arranjo.

Alteração de valores em arranjo via sub-rotina

Dentro do exemplo apresentado considere a definição da variável "IPCA19" com informações passadas incorretamente. Veja a seguir a indicação dos valores de índice de preços do ano de 2019 com alguns erros sinalizados em vermelho:

```
julia> IPCA19 = [0.32, 0.43, 0.57, 0.57, 0.13, 0.01, 0.91, 0.11, -0.04, 0.10, 0.51, 1.15]
12-element Vector{Float64}:
  0.32
  0.43
  0.57
  0.57
  0.13
  0.01
  0.91
  0.11
 -0.04
  0.1
  0.51
  1.15
```

Os erros existentes são caracterizados por trocas entre os dígitos indicados nas posições "**3**" (valor correto "**0.75**") e "**7**" (valor correto "**0.19**"). O resultado do índice acumulado esperado é de "**4.31**", mas o retorno é um valor bem diferente. Execute na sequência a instrução:

```
julia> inflaçãoAcumulada(IPCA19)  # retorno de índice acumulado com erro
4.87
```

A alteração do valor pode ser realizada diretamente no modo interativo, mas a proposta aqui é fazer esta ação com o uso de uma sub-rotina ao estilo procedimento escrita para isso. Assim sendo, atente para as próximas instruções:

```
julia> function altera(posição, valor, arranjo)
           arranjo[posição] = valor
       end
altera (generic function with 1 method)

julia> altera(3, 0.19, IPCA19)
0.75
```

```
julia> altera(7, 0.19, IPCA19)
0.75

julia> inflaçãoAcumulada(IPCA19)
4.31
```

Veja que com a sequência de exemplo apresentado operar arranjos com funções é semelhante a operar variáveis simples com funções.

ANOTAÇÕES

APÊNDICE H - ESCOPO DE VARIÁVEIS

Neste apêndice é dado ênfase ao contexto de escopo e visibilidade de recursos da linguagem Julia. Anteriormente este tema foi tratado de maneira superficial e focado em algumas partes e agora faz-se com maior abrangência.

Consideração geral

As considerações apresentadas neste apêndice estão baseadas na documentação oficial da linguagem. Neste sentido, para maiores detalhes consulte o endereço: **https://docs.julialang.org/en/v1/manual/variables-and-scoping/**.

O escopo e visibilidade de certo recurso, como a definição e uso de variáveis, é a forma como este se comporta dentro da linguagem frente a memória do computador. É uma estratégia de trabalho que visa economizar memória e quando possível auxilia em evitar conflitos de nomes de variáveis e outros objetos computacionais. Na linguagem Julia existem dois tipos principais de escopo: *local* e *global*, sendo que o espoco *local* aceita ser aninhado dentro de outros escopos locais.

Além dos escopos *global* e *local* há a possibilidade em Julia do uso de área de operação *rígida* ou *suave* em algumas construções. As áreas de operação *rígida* e *suave* estabelecem as regras se determinada variável pode ou não ser sombreada. O sombreamento de uma variável ocorre quando uma variável local possui o mesmo nome de uma variável global ou local em instância de memória maior que a sua armazenando dados diferentes. Quando isso ocorre é necessário ter muita atenção para que não ocorra confusão na interpretação desses dados por parte do profissional de desenvolvimento.

No capítulo dois foram apresentados programas que necessitaram da definição do comando "**`global`**" para que a variável em operação pudesse ser devidamente operacionalizada. Esse efeito ocorreu devido ao uso da linguagem do conceito de *área rígida*. Aliás, é pertinente salientar que quando em modo programado o ambiente Julia opera com *área rígida* e quando em operação de forma interativa opera com *área suave*.

Tomando por exemplo o uso de mesma situação apontada no programa "**`c02ex09.jl`**" que apresenta em tela os valores contados de um a cinco usou-se no código o trecho seguinte. Atente para a linha "**`4`**" no uso do comando "**`global`**":

```
1  i = 1
2  while i <= 5
3      println(i)
4      global i = i + 1
5  end
```

Neste caso, a variável "*i*" definida na linha "**`1`**" é global com escopo rígido. Assim sendo, a variável "*i*" indicada nas linhas "**`3`**" e "**`4`**" é uma variável de escopo local existente apenas dentro do bloco "**`while`**" e "**`end`**", ou seja, não são referências a mesma variável. Para conseguir a referência e indicar que são a mesma variável é necessário o uso do comando "**`global`**" na linha "**`4`**" onde ocorreu a identificação de ambiguidade. A ausência do comando "**`global`**" no trecho de código apontado gera a apresentação de uma mensagem de advertência indicando que a variável em uso nas linhas "**`3`**" e "**`4`**" com escopo suave é ambígua por existir outra variável com escopo global definida na linha "**`1`**" com

o mesmo nome. A mensagem de advertência orienta que para eliminar efetivamente sua apresentação basta fazer uso dos comandos "**local**" ou "**global**".

O efeito funcional de um código em modo programado e em modo interativo são diferentes, como comentado. No modo programado a ausência do comando "**global**" cria problemas de ambiguidade se as variáveis de um bloco interno tiverem que ter relação com a variável de mesmo nome fora desse bloco. Mas em modo interativo é diferente, veja a sequência de instruções seguintes:

```
julia> i = 1
1

julia> while i <= 5
           println(i)
           i = i + 1
       end
1
2
3
4
5


julia> i
6
```

Veja que não foi necessário usar o comando "**global**" para evitar ambiguidade de nomes. Note que nesta operação o ambiente "entendeu" que a variável "/" está operando com escopo global suave, sendo iniciada com valor "**1**", tendo sua evolução apresentada de "**1**" a "**5**" e estando com valor "**6**" após o encerramento da contagem. Observe agora o próximo conjunto de instruções:

```
julia> local i = 1
1

julia> while i <= 5
           println(i)
           i = i + 1
       end

julia> i
6
```

Veja que neste conjunto define-se a variável "/" como local, nenhum valor é apresentado de "**1**" a "**5**" e é indicada a variável com o valor "**6**". Isso ocorreu por existir neste momento na memória dias variáveis com o nome "/" uma global com valor "**6**" e outra local com valor "**1**". Veja as próximas instruções:

```
julia> i = 1
1

julia> while i <= 5
           println(i)
           local i = i + 1
       end
```

```
ERROR: UndefVarError: i not defined
Stacktrace:
 [1] top-level scope
   @ .\REPL[25]:2
```

Observe que nesta situação é definido dentro do bloco "**while**" a indicação da variável "/" como "**local**" e que a partir desta definição o trecho de código não acessa a variável global de mesmo nome.

Como comentado o uso de escopo local permite a definição de alinhamentos locais definindo-se o sombreamento de variáveis. Desta forma, é possível ter camadas de áreas de memória. Observe o conjunto de instruções seguintes:

```
julia> let
           local i = 1
           # primeira área de escopo
           let
               local i = 2
               # segunda área de escopo
               Let
                   # terceira área de escopo
                   local i = 3
                   println(i)
               end
               println(i)
           end
           println(i)
       end
3
2
1
```

O bloco definido com os comandos "**let**" e "**end**" permitem a definição de área de escopo. Veja a partir das cores definidas o comportamento de cada escopo local. Veja que "**let**" e "**end**" define um novo bloco de escopo de modo que a variável "/" é uma variável diferente em cada bloco.

Um detalhe importante a ser considerado em relação ao uso do comando "**global**" é que este é necessário dentro do bloco "**while**". Nos blocos "**for**" e "**if**" essa necessidade não ocorre. No caso de um bloco "**for**" a definição de uma variável para contagem é automaticamente considerada local. Veja um exemplo de código de programa simples que apresenta os valores de "**1**" a "**5**" na tela. Caso queira fazer o teste, grave o programa com o nome "**apendice1.jl**".

```
1  i = 9
2  for i in 1:5
3      println(i)
4  end
5  println(i)
```

Ao ser executado o programa em modo programado e não em modo interativo você perceberá que existem duas variáveis chamadas "/". A definição de "/" na linha "**1**" é global e a definição da variável "/" na linha "**2**" (dentro do comando "**for**") é local. Veja que a saída mostra os valores de "**1**" a "**5**" processados entre as linhas de código "**2**" a "**4**" e a apresentação do valor "**9**" processado pela linha "**5**" que traz o valor da variável definido na linha "**1**".

O uso do comando "**global**" dentro de um bloco "**while**" pode ser eliminado se a operação do bloco "**while**" estiver definida dentro de uma sub-rotina. Observe os detalhes do próximo código de programa a ser gravado com o nome "**apendice2.jl**".

```
1  function teste()
2      i = 1
3      while i <= 5
4          println(i)
5          i = i + 1
6      end
7  end
8
9  teste()
```

Ao ser executado o programa em modo programado observe a apresentação dos valores sem a necessidade da definição de uso do comando "**global**". Ao se definir uma sub-rotina cria-se nela uma área suave com escopo global. Desta forma, a variável definida fora do bloco "**while**" é a mesma dentro do bloco.

Mas é necessário tomar cuidado com o uso do comando "**for**" pois a variável controlada dentro deste bloco é por definição local. Veja o código do programa "**apendice3.jl**".

```
1  function teste()
2      i = 9
3      for i in 1:5
4          println(i)
5      end
6      println(i)
7  end
8
9  teste()
```

Ao ser executado o programa observe a apresentação dos valores de "**1**" a "**5**" processados entre as linhas "**3**" a "**5**" e a apresentação do valor "**9**" processado pela linha "**6**" que traz o valor da variável definido na linha "**2**". Veja que existem neste exemplo duas variáveis com o mesmo nome operando estruturas de valores distintas.

Para fazer com que a variável local de um bloco "**for**" seja operacionalizada como global é necessário adicionar o uso do comando "**outer**". Veja o código do programa "**apendice4.jl**".

```
1  function teste()
2      i = 9
3      for outer i in 1:5
4          println(i)
5      end
6      println(i)
7  end
8
9  teste()
```

Ao ser executado o programa observe a apresentação dos valores de "**1**" a "**5**" processados entre as linhas "**3**" a "**5**" e a apresentação do valor "**5**" repetido processado pela linha "**6**" devido ao uso do comando "**outer**". Veja que a variável em operação fora e dentro do bloco é a mesma variável.

APÊNDICE I - DEFINIÇÃO DE CONSTANTES

Neste apêndice é dado atenção com mais detalhes ao uso e definição de constantes.

Consideração geral

A linguagem Julia como comentado no capítulo um possui algumas constantes matemáticas definidas internamente. Na ocasião foi comentado brevemente como fazer a definição de uma constante. Neste apêndice, é dado maior atenção aos detalhes técnicos deste tipo de definição.

Tecnicamente falando uma constante não ocupa espaço de memória como uma variável. Uma variável necessita de um local em memória para armazenar determinado valor. A definição de uma constante não armazena um valor em memória, o que ocorre é a definição de um rótulo de identificação associado a um valor, tanto o valor como o rótulo representam em si a mesma definição. No entanto, na linguagem Julia a definição de uma constante é armazenada "fisicamente" em memória como são as variáveis, tendo como diferença que após a definição de um valor na constante este não poderá sofrer alteração como ocorre com as variáveis. No paradigma da programação funcional este efeito é comum, sendo estranhamento chamado de *variável imutável*. As variáveis convencionais são então chamadas de *variáveis de estado*. Mas em Julia os termos são *constantes* e *variáveis* simplesmente.

Veja a seguir uma sequência de definição de valor inicial em uma variável e sua alteração:

```
julia> var = 1
1

julia> var = var + 1
2
```

Atente agora para a definição de uma constante e resultado da operação de alteração de seu valor:

```
julia> const con = 1
1

julia> con = con + 1
WARNING: redefinition .... This may fail, cause incorrect answers, or produce other errors.
```

Veja que em relação ao uso de variáveis não ocorreu nenhuma observação do ambiente em relação a alteração de seu valor. No entanto, com a definição da constante uma mensagem de advertência foi apresentada informando que o valor da constante foi redefinido e que isso poderá ocasionar falhas, respostas incorretas ou até produzir outros erros. Uma vez definida uma posição de memória como constante esta não deve ser alterada por qualquer motivo que seja.

Observe que em Julia a definição de constante, diferentemente de outras linguagens de programação, ocupa espaço de memória e se comporta, se "forçada a barra", como se comportam as variáveis, o que pode ser inconveniente. Uma maneira alternativa de tornar esta ação mais segura é fazer a definição de uma estrutura imutável de constantes e a partir dessa estrutura definir um objeto constante de operação, evitando-se o uso da funcionalidade "**@set**" do pacote "**Setfield**" apresentado no capítulo três.

Como exemplo considere a definição de campos para a definição de constantes matemáticas similares as existentes na linguagem C++. Desta forma, observe as próximas instruções:

```
julia> struct MyMath
           M_E::Float64
           M_LOG2E::Float64
           M_LOG10E::Float64
           M_LN2::Float64
           M_LN10::Float64
           M_PI::Float64
           M_PI_2::Float64
           M_PI_4::Float64
           M_1_PI::Float64
           M_2_PI::Float64
           M_2_SQRTPI::Float64
           M_SQRT2::Float64
           M_SQRT1_2::Float64
       end
```

A partir da definição da estrutura imutável cabe preparar a constante com a definição dos valores de cada campo:

```
julia> const math = MyMath(
                      2.7182818284590452354,
                      1.4426950408889634074,
                      0.43429448190325182765,
                      0.69314718055994530942,
                      2.30258509299404568402,
                      3.14159265358979323846,
                      1.57079632679489661923,
                      0.78539816339744830962,
                      0.31830988618379067154,
                      0.63661977236758134308,
                      1.12837916709551257390,
                      1.41421356237309504880,
                      0.70710678118654752440
                  )
```

A partir desta definição tem-se em memória um conjunto de constantes para operação. Para deixar este recurso disponibilizado basta criar a partir dele um módulo ou um pacote. Veja exemplos de uso das constantes definidas:

```
julia> math.M_PI
3.141592653589793

julia> math.M_2_PI
0.6366197723675814
```

Observe que se tem agora um mecanismo bem mais próximo da definição de constantes encontrados em outras linguagens de programação.

REFERÊNCIAS BIBLIOGRÁFICAS

AKITA, F. **Qual a real diferença entre arquivos binário e texto?**. YouTube, 20 jan. 2021. Disponível em: <https://www.youtube.com/watch?v=oSCVb4Ts-G4>. Acesso em: 10 ago.2022.

ANDRADE, E. L. **Introdução à pesquisa operacional**: *métodos e modelos para análise de decisões*. 4. ed. Rio de Janeiro: LTC, 2009.

BACKES, A. R. **Linguagem C**: *Descomplicada*. Minas Gerais: Edição do autor, 2014. Disponível em: <http://www.facom.ufu.br/~backes/apostilaC.pdf>. Acesso em: 3 jul. 2014.

BALBAERT, I. & SALCEANU, A. **Julia**: *High performance programming*. New York: Apress, 2016.

BALBAERT, I. **Getting starded with Julia programming**. New York: Apress, 2015.

BALBAERT, I.; SENGUPTA, A. & SHERRINGTON, M. **Julia 1.0 programming complete reference guide**: *Discover Julia, a high-performance language for technical computing*. New York: Apress, 2019.

CHRISTIANSON, A. **Object Orientation and Polymorphism in Julia**. Github, 2020. Disponível em: <https://github.com/ninjaaron/oo-and-polymorphism-in-julia>. Acesso em: 3, jul. 2022.

DANISCH, S. **A Brief Introduction to Julia Metaprogramming, Macros, and Generated Functions**. Nextjournal, 2019. Disponível em: <https://nextjournal.com/a/KpqWNKDvNLnkBrgiasA35?change-id=CQRuZrWB1XaT71H92x8Y2Q>. Acesso em: 29, jun. 2022.

FARIAS, C. **Using Julia for Data Science**. https://cleytonfar.github.io/, 2019. Disponível em: <https://cleytonfar.github.io/posts/using-julia-for-data-science/>. Acesso em: 23 jul. 2022.

HEWLETT PACKARD. **HP Data Entry and Forms Management System (VPLUS): Reference Manual – HP 3000 MPE/iX Computer System**. 6 ed. California: HP, 2000.

HIMITE, B. **What Is Currying in Programming?**: *Spice up your code with currying*. Towards Data Science, 2021. Disponível em: <https://towardsdatascience.com/what-is-currying-in-programming-56fd57103431>. Acesso em: 11 set. 2022.

JOSHI, A. & LAKHANPAL, R. **Learning Julia**: *Build high-performance applications for scientific computing*. Birminghan: Packt Publishing Ltd., 2017.

JOSHI, A. **Julia for data science**: *Explore the world of data science from scratch with Julia by your side*. Birminghan: Packt Publishing Ltd., 2016.

KALICHARAN, N. **Julia - bit a bit**: *Programmer for beginners*. Switzerland: Springer, 2021.

KÖLLER, J. **Mathematische Basteleien**: *Römische Ziffern*. jkoller@gmax.de, 2003. Disponível em: <http://www.mathematische-basteleien.de/roemisch.htm>. Acesso em: 15 jun. 2022.

LAUWENS, B. & DOWNEY, A. B. **Think Julia**: *How to think like a computer scientist*. California: O'Reilly Media, 2019.

LOBIANCO, A. **Julia quick syntax reference**: *A pocket guide for data science programmin*. New York: Apress, 2019.

MAGALHÃES, J. W. de S.; ÁVALOS, J. B.; SANTOS, L. J. A. dos. **Dossiê Linguagem Julia**. Universidade Federal de Viçosa - Campus Florestal: MG, 2021. Disponível em: <https://scholar.google.com/citations?view_op=view_citation&hl=pt-BR&user=JwKrVb8AAAAJ&citation_for_view=JwKrVb8AAAAJ:u-x6o8ySG0sC>. Acesso em: 25, mar. 2022.

MUKAKA, M. M. **Statistics Corner**: *A Guide to Appropriate Use of Correlation Coefficient in Medical Research*. Malawi Medical Journal, 24, 69-71. 2012.

MENESES, P. B. **Matemática discreta para computação e informática**. 4. ed. Porto Alegre: Bookman, 2013.

MOURA, H. A. S. **Programação funcional profissional em Scala**. Quixada: HM Software, 2019. Disponível em: <http://hmsoftware.com.br/>. Acesso em: 28 ago. 2019.

NAGAR, S. **Beginning Julia programming**: *For Engineers and scientists*. New York: Apress, 2017.

NAZARATHY, Y. & KLOK, H. **Statistics with Julia**: *Fundamentals for data science, machine learning and artificial intelligence*. Switzerland: Springer, 2021.

ODIFREDDI, P.; COOPER, S. B. **Recursive Functions**. Stanford Encyclopedia of Philosophy, 2012. Disponível em: <https://plato.stanford.edu/entries/recursive-functions/>. Acesso em: 5 jul. 2022.

PRIYA, H. & RANJEET, R. **Programming and Problem Solving Through "C" Language**. New Deli: FireWall Media / Laxmi Publications, 2006.

RIP Tutorial. **Install an unregistered package**. EUA, 2022. Disponível em: <https://riptutorial.com/julia-lang/example/20490/install-an-unregistered-package>. Acesso em: Acesso em: 10 ago. 2022.

ROHIT, J. R. **Julia cookbook**. Birminghan: Packt Publishing Ltd., 2016.

SINGH, S. **O livro dos códigos: A ciência do sigilo, do antigo Egito à criptografia quântica**. São Paulo: Record, 2011.

TATE, A. B. **Programmer passport**: *Julia*. North Carolina: Pragmatic Bookshelf, 2020.

VICKI, V. **Arte e Cultura: A cifra de Vigenère**. 2006. Disponível em: <http://www.numaboa.com.br/index.php?option=com_content&view=article&id=506&Itemid=134>. Acesso em: Acesso em: 6 jul. 2018.

WAGNER, N. R. & PUTER, P. **Error Detecting Decimal Digits**. CACM Publisher ACM, Nova York, v. 32, issue 1, p. 106-110, jan. 1989.

A linguagem Julia é diferente de outras linguagens de programação em diversos aspectos. Neste sentido, convido você a fazer uma viajem de aprendizado inicial, dinâmico e aplicado com este livro. Este texto, apesar de introdutório, é sem sombra de dúvidas o mais completo material publicado no Brasil em língua portuguesa até o momento.

O estilo de apresentação de seu conteúdo é dinâmico e dirigido, principalmente quando relacionado ao uso interativo do ambiente de programação a partir da orientação de uso de diversas funcionalidades contextualizadas. Para seu estudo parte-se do pressuposto de que você possui bons conhecimentos de lógica de programação, além da devida destreza de uso de seu sistema operacional no tocante aos procedimentos de gerenciamento de arquivos e diretórios (pastas).

Por possuir um ambiente interativo de operação, Julia oferece a seu usuário um mecanismo rápido de acesso a seus recursos. Isso permite experimentar muitas de suas operações sem a necessidade de escrever longos programas. E é neste sentido que este livro segue sua apresentação: grande parte da abordagem deste texto segue o caminho interativo de uso da linguagem sem deixar de lado a parte programada.

Ao final deste estudo o leitor estará capacitado a entender e aplicar as técnicas básicas de programação no desenvolvimento de aplicações Julia. Este trabalho é dividido em oito capítulos mais apêndices e deverá ser estudado em sua totalidade.

ISBN 978-65-00-59016-6
9 786500 590166