

Introdução à Linguagem Julia

Alfredo Goldman

Lucas de Sousa Rosa

2024-10-16

Índice

Boas Vindas	4
1 Visão Geral do Curso	5
2 Visão Geral da Evolução de Hardware e Linguagens	6
3 Usando o Interpretador (REPL) como Calculadora	7
3.1 Começando com o modo interativo do Julia	7
3.2 Variáveis e seus tipos	12
3.2.1 Tipagem dinâmica	12
3.3 Saída de dados	14
4 Introdução às Funções	16
4.1 O uso de funções é uma abstração natural	16
5 Comparações, o comando if e recursão	20
5.1 Agora sim: Funções que se chamam	23
7 Testes automatizados e um pouco mais de código	33
7.1 Funções caóticas	36
8 Uma outra forma de se fazer laços	37
9 Aula de exercícios	40
9.1 Revisitando o cálculo do fatorial, recursivo e interativo	40
9.2 Aproximação da raiz quadrada	41
9.3 Verificar se um número é primo	42
9.4 Verificar se um número é palíndromo	43
10 Revisitando a aula passada	44
10.1 Aleatoriedade	45
11 Entrada de dados e o começo de listas	49
11.1 O comando input	49
11.2 Lendo através da linha de comando	51
11.3 Listas	52
11.3.1 Álgebra linear e Listas	54

11.3.2	Exercício de permutação	55
--------	-----------------------------------	----

Boas Vindas

This is a Quarto book.

To learn more about Quarto books visit <https://quarto.org/docs/books>.

1 Visão Geral do Curso

2 Visão Geral da Evolução de Hardware e Linguagens

3 Usando o Interpretador (REPL) como Calculadora

Objetivo: Ver o interpretador de Julia como uma calculadora poderosa, introduzir a noção de variáveis.

3.1 Começando com o modo interativo do Julia

Quem quiser já pode instalar o ambiente de programação, usem esse [link](#). Há também alguns ambientes que permitem o uso da linguagem no seu navegador, sugiro a busca pelas palavras chave `Julia Language online`.

Dentro do Julia (após chamar `julia` na linha de comando), vamos começar com contas com números inteiros:

```
1 + 2
```

```
3
```

```
40 * 4
```

```
160
```

Sim, como era de se esperar, podemos em Julia usar os operandos: `+`, `-` e `*`, o resultado será como o esperado. Vejamos a seguir que com a divisão fica um pouco diferente:

```
a = 84
b = 2

# As variáveis a e b são do tipo Int64

resultado = a / b
println(resultado)
```

42.0

Notem que nesse caso, houve uma mudança de tipos, pois 84 e 2 são inteiros e o resultado é um número em ponto flutuante (float), podemos ver isso, pois ao invés de 42, tivemos como resultado 42.0.

Também é possível pedir o resultado inteiro usando o operador `div`:

```
div(84,2)
```

42

Ou de forma equivalente usando o operador `\div` (para conseguir ver o símbolo da divisão é necessário digitar `\div` seguido da tecla `<tab>`).

Além das contas básicas, também dá para fazer a exponenciação:

```
2^31
```

2147483648

Expressões mais complexas também podem ser calculadas:

```
23 + 2 * 2 + 3 * 4
```

39

Sim, a precedência de operadores usual também é válida em Julia. Mas, segue a primeira lição de programação: *Escreva para humanos, não para máquinas.*

```
23 + (2 * 2) + (3 * 4)
```

39

Em Julia também podemos fazer operações com números em ponto flutuante:

```
23.5 * 3.14
```

73.79

ou


```
12.5 / 2.0
```

6.25

Acima temos mais um exemplo de código escrito para pessoas, ao se escrever 2.0 estamos deixando claro que o segundo parâmetro é um número float.

É importante saber que números em ponto flutuante tem precisão limitada, logo não se espante com resultados inesperados como abaixo:

```
1.2 - 1.0
```

0.19999999999999996

Erros como esse são bastante raros, tanto que usualmente confiamos plenamente nas contas feitas com computadores e calculadoras. Mas, é bom saber que existem limitações.

```
2.6 - 0.7 - 1.9
```

2.220446049250313e-16

ou

```
0.1 + 0.2
```

0.30000000000000004

ou ainda

```
10e15 + 1 - 10e15
```

0.0

Esses problemas de precisão estão ligados a limitação de como os números são representados no computador. De maneira simplificada, os valores no computador são codificados em palavras, formadas por bits. Nos computadores modernos as palavras tem 64 bits, ou 8 bytes. Logo, uma outra limitação está ligada aos números inteiros muito grandes

```
2^63
```

```
-9223372036854775808
```

Mas, para um curso introdutório basta saber que existem essas limitações. Como lidar com elas é parte de um curso mais avançado.

É importante notar que o erro acima é um *erro silencioso*, ou seja quanto estamos usando números inteiros, pode ocorrer que o número a ser representado não caiba no número de bits disponível, o que faz com que ocorra um erro.

Voltando para as contas. Um outro operador interessante é o % que faz o resto da divisão

```
4 % 3
```

```
1
```

Até agora vimos como trabalhar com um único valor, ou seja, como se fosse no visor de uma calculadora. Mas, é possível ir além. Ao invés de termos teclas de memória, o computador nos oferece variáveis. Elas são como nomes para valores que queremos guardar e usar mais tarde.

Além das operações básicas também temos as operações matemáticas (funções), como por exemplo o seno, sin em inglês. Para saber como uma função funciona podemos pedir ajuda ao ambiente, usando uma ? ou o macro @doc, e em seguida digitando o que queremos saber, como por exemplo em:

```
@doc sin
```

A saída desse comando indica a operação que a função realiza e ainda apresenta alguns exemplos:

```
sin(x)
```

```
Compute sine of x, where x is in radians.
```

```
See also sind, sinpi, sincos, cis, asin.
```

```
Examples
```

```
julia> round.(sin.(range(0, 2pi, length=9)'), digits=3)
1×9 Matrix{Float64}:
 0.0  0.707  1.0  0.707  0.0 -0.707 -1.0 -0.707 -0.0
```

Ambos os comandos `? sin` e `@doc sin` possuem a mesma saída.

Notem que nem tudo que foi apresentado faz sentido no momento, mas já dá para entender o uso de uma função como `sin`. Vejamos agora a raiz quadrada:

```
@doc sqrt
```

Nela vemos que é possível calcular a raiz como em:

```
sqrt(4)
```

2.0

```
sqrt(4.0)
```

2.0

Mas, observamos também na documentação a função `big()`, que tem a seguinte ajuda:

```
@doc BigInt
```

A função `big()` em Julia é usada para criar números inteiros grandes, representados pelo tipo `BigInt`. Essa função é especialmente útil quando você precisa lidar com números muito grandes que excedem o limite dos tipos inteiros padrão, como `Int64` ou `Int32`.

Com números `BigInt`, já não há problemas de estouro, como podemos ver abaixo:

```
big(2) ^ 1002
```

42860344287450692837937001962400072422456192468221344297750015534814042044997444899727935152

Podemos ainda carregar funções de outros arquivos em nosso arquivo Julia ou no próprio terminal, para isso basta utilizar o comando `include("caminho/do/arquivo.jl")`, Julia lê o arquivo especificado e executa todo o seu conteúdo no contexto atual. Isso significa que todas as funções, variáveis e definições no arquivo tornam-se disponíveis no ambiente onde `include` foi chamado.

Como por exemplo no primeiro caso tenho um arquivo chamado `funcoes.jl` que possui a função `soma`:

```
function ola(nome)
    println("Olá", nome)
end
```

ola (generic function with 1 method)

Podemos incluir essa função em um segundo arquivo utilizando o `include("funcoes.jl")`, e utilizar a função definida no arquivo `funcoes.jl`

```
include("funcoes.jl")
println(ola("Alfredo"))
```

Cuja saída deverá ser Olá Alfredo.

3.2 Variáveis e seus tipos

Em Julia também temos o conceito de variáveis, que servem para armazenar os diferentes conteúdos de dados possíveis.

```
a = 7
2 + a
```

9

3.2.1 Tipagem dinâmica

É importante notar que as variáveis em Julia podem receber novos valores e o tipo da variável depende do que foi atribuído por último.

```
a = 3
typeof(a)
```

Int64

```
a = a + 1
typeof(a)
```

Int64

Neste próximo exemplo, a variável `b` é inicializada com um valor de tipo inteiro, contudo, após a operação de multiplicação, seu valor é do tipo ponto flutuante:

```
b = 3
b = b * 0.5
typeof(b)
```

Float64

A tipagem dinâmica apresenta diversas vantagens, entre elas a flexibilidade, pois é possível reutilizar variáveis para armazenar diferentes tipos de dados ao longo do tempo; e menos verbosidade, pois não é necessário especificar o tipo de cada variável, o que melhora a legibilidade do código.

Aproveitando o momento, podemos ver que há vários tipos primitivos em Julia, sendo os principais:

```
typeof(1)
```

Int64

```
typeof(1.1)
```

Float64

```
typeof("Bom dia")
```

String

Falando em Strings, elas são definidas por conjuntos de caracteres entre aspas como:

```
s1 = "Olha que legal"
s2 = "Outra String"
```

"Outra String"

Dá também para fazer operações com strings como concatenação:

```
s1 = "Tenha um"
s2 = " Bom dia"
s3 = s1 * s2
```

"Tenha um Bom dia"

Ou potência:

```
s = "Nao vou mais fazer coisas que possam desagradar os meus colegas "
s ^ 10
```

"Nao vou mais fazer coisas que possam desagradar os meus colegas Nao vou mais fazer coisas q

Ainda sobre variáveis, há umas regras com relação aos seus nomes, tem que começar com uma letra (ou com `_`), pode ter dígitos e não pode ser uma palavra reservada. É bom notar que Julia por ser uma linguagem moderna, aceita nomes de caracteres em unicode, por exemplo o Δ (`\Delta`):

```
 $\Delta$  = 2
```

2

Mas, a linguagem vai bem além com caracteres de animais e símbolos:

```
= 5 # \:cat: <tab>
= 3 # \:dog: <tab>
= 20 # \:house: <tab>
```

20

Isso não adiciona nada do lado de algoritmos, mas é possível ter variáveis bem bonitinhas. A lista de figuras pode ser encontrada [aqui](#).

3.3 Saída de dados

Para fazer saídas usam-se dois comandos, `print()` e o `println()`, sendo que o primeiro não pula linha e o segundo pula.

```
print("Hello ")
println("World!")
println("Ola, mundo!")
```

Hello World!
Ola, mundo!

Para evitar que se digitem muitos caracteres, por vezes podemos usar “açucares sintáticos”.

```
x = 1
x = x + 1
x += 1 # forma equivalente a acima, o mesmo vale para os operadores *, - e /
```

3

Acima, vimos a forma de se inserir comentários em Julia (sim esses serão ignorados pelo computador).

Exercício: Faça o passo a passo para encontrar as raízes da equação de segundo grau $x^2 - 5x + 6$, usando as variáveis `a`, `b`, `c`, `\Delta`, `x1` e `x2`. Após isso, compare com a solução a seguir:

```
# Definição dos coeficientes
a = 1
b = -5
c = 6

# Cálculo do discriminante
delta = b^2 - 4 * a * c

# Cálculo das raízes
if delta >= 0
    x1 = (-b + sqrt(delta)) / (2 * a)
    x2 = (-b - sqrt(delta)) / (2 * a)
    println("As raízes são: x1 = $x1 e x2 = $x2")
else
    println("A equação não possui raízes reais.")
end
```

As raízes são: `x1 = 3.0` e `x2 = 2.0`

4 Introdução às Funções

Objetivo: Começar a entender como funcionam as funções em uma linguagem de programação

4.1 O uso de funções é uma abstração natural

Na aula passada já vimos umas funções e isso foi bem natural, foram elas:

- `typeof()` - Dado um parâmetro devolve o seu tipo. Variáveis estão associadas a tipos;
- `div()` - Dados dois parâmetros devolve a divisão inteira do primeiro pelo segundo;
- `print()` e `println()` - Dados diversos parâmetros os imprime, sem devolver nada.

Inclusive, aqui vale a pena ver que podemos pedir ajuda ao Julia para saber o que fazem as funções. Para isso, se usa o `?` antes da função:

```
?typeof()  
?div()  
?print()
```

Ao fazer isso, inclusive descobrimos que o `div()` pode ser usado também como \div .

Uma outra função bem útil é a que permite transformar um tipo de valor em outro.

```
parse(Float64, "32")
```

Para conversão de valores em ponto flutuante para inteiros, temos a função `trunc`.

```
trunc{Int64}(2.25)
```

De forma inversa temos o `float`.

```
float{Float64}(2)
```


Finalmente, podemos transformar um valor em uma string, como em:

```
string(3)
```

ou

```
string(3.57)
```

Também tem muitas funções matemáticas prontas como

Função	Descrição
<code>sin(x)</code>	Calcula o seno de (x) em radianos
<code>cos(x)</code>	Calcula o cosseno de (x) em radianos
<code>tan(x)</code>	Calcula a tangente de (x) em radianos
<code>deg2rad(x)</code>	Converte (x) de graus em radianos
<code>rad2deg(x)</code>	Converte (x) de radianos em graus
<code>log(x)</code>	Calcula o logaritmo natural de (x)
<code>log(b, x)</code>	Calcula o logaritmo de (x) na base (b)
<code>log2(x)</code>	Calcula o logaritmo de (x) na base 2
<code>log10(x)</code>	Calcula o logaritmo de (x) na base 10
<code>exp(x)</code>	Calcula o expoente da base natural de (x)
<code>abs(x)</code>	Calcula o módulo de (x)
<code>sqrt(x)</code>	Calcula a raiz quadrada de (x)
<code>cbrt(x)</code>	Calcula a raiz cúbica de (x)
<code>factorial(x)</code>	Calcula o fatorial de (x)

A melhor forma de se acostumar a usar as funções é fazendo contas e verificando os resultados. Uma dica importante é que para funções mais complexas, pode ser que já existam funções prontas em Julia. Para isso uma busca com as palavras chave. Um exemplo a seguir para procurar a função para o cálculo de seno hiperbólico: “julia lang hiperbolic sin”. A busca pelo termo em inglês é uma boa dica para buscas em geral.

Em julia também é possível criar funções conforme as suas necessidades, como abaixo:

```
function mensagemDeBomDia()  
    println("Tenha um bom dia!")  
end
```

`mensagemDeBomDia` (generic function with 1 method)

Para usar uma função, basta chamá-la:

```
MensagemDeBomDia()
```

Funções, podem receber um ou mais parâmetros:

```
function imprime(a)
    println(" Vou imprimir ", a)
end
imprime(42)
```

```
Vou imprimir 42
```

Também é possível que uma função chame outra função como em:

```
function imprimeduasvezes(a)
    imprime(a)
    imprime(a)
end
imprimeduasvezes(13)
```

```
Vou imprimir 13
```

```
Vou imprimir 13
```

Mais ainda, também é possível diferenciar funções por meio da quantidade de parâmetros.

```
function recebe(a)
    println("Recebi um parametro: ", a)
end
function recebe(a, b)
    println("Recebi dois parametros: ", a, " ", b)
end
```

```
recebe (generic function with 2 methods)
```

Conforme a chamada, a função chamada será diferente:

```
recebe(1)
```

```
recebe(1, 2)
```

Também dá para chamar funções com variáveis e com operações, como em:

```
a = 10
recebe(a)
recebe(a, a + 1)
```

As funções que vimos até agora imprimem mensagens, mas não devolvem nada. O `typeof()` delas é `nothing`, ou seja, algo que não pode ser atribuído.

Mas, também é possível fazer funções que devolvem valores, como:

```
function soma1(a)
  return a + 1
end
```

Nesse caso, se for passado um parâmetro numérico, a função devolverá o valor incrementado (adicionado de 1).

Claro que isso pode ser usado com fórmulas mais complicadas como:

```
function hipotenusa(a, b)
  hip = a * a + b * b
  return hip
end
```

`hipotenusa` (generic function with 1 method)

Exercício: Faça uma função para encontrar o Δ de uma equação de segundo grau

5 Comparações, o comando if e recursão

Antes de falar em desvio (if), vamos ver um novo tipo de variável que foi introduzido de forma natural. O tipo booleando, ou seja uma variável que pode valer true (verdadeiro) ou false (falso). O seu uso está intimamente ligado ao if.

Observem os seguintes exemplos:

```
2 + 2 == 4
```

```
true
```

```
3 != 8
```

```
true
```

```
23 < 24
```

```
true
```

```
42 <= 44
```

```
true
```

```
42 < 2
```

```
false
```

Vale chamar a atenção, como em linguagens de programação o = é usado para atribuições, para comparações se usa o ==. Da mesma forma o != é usado como diferente. Esses operadores, em conjunto com o <, <=, > e >= nos permitem comparar valores.

Sobre as variáveis booleanas vale também observar o seu tipo. Uma explicação mais aprofundada sobre como essas variáveis funcionam será fornecida quando abordarmos os operadores condicionais:

```
typeof(2 == 3)
```

Bool

Finalmente, também podemos negar variáveis booleanas para inverter o seu valor:

```
!true  
!false
```

true

Nessa aula, vamos aprender um novo comando. O desvio condicional, através dele é possível alterar o fluxo de execução de um programa. Até o momento não tínhamos comentado isso explicitamente, mas a ordem de execução de instruções segue a ordem em que elas estão. Vejamos o exemplo abaixo:

```
println("Oi")  
println("um")  
println("dois")
```

Oi
um
dois

A ordem de impressão será Oi, um e dois.

Da mesma forma não temos problema ao executar o código abaixo.

```
denominador = 0  
denominador += 2  
30 / denominador
```

15.0

Apesar da variável denominador começar inicialmente com 0, antes de se fazer a divisão, ela estará valendo 2.

Como é de se esperar nem sempre queremos que essa ordem seja respeitada. Observe o seguinte exemplo:

```
pandemia = true
println("Vou sair de casa?")
if pandemia == true
    println("Só vou sair de casa se for essencial")
end
```

```
Vou sair de casa?
Só vou sair de casa se for essencial
```

O exemplo acima é claro, se uma condição for verdadeira, o código que está no escopo do if (isso é entre a condição e o end) será executado.

Um outro exemplo:

```
denominador = 1
if denominador != 0
    println("sei fazer a divisão se não for por zero")
    println("o resultado da divisão de 30 por ", denominador, " é igual a ", 30/denominador)
end
```

```
sei fazer a divisão se não for por zero
o resultado da divisão de 30 por 1 é igual a 30.0
```

Situações muito comuns em computação devem ser favorecidas pela linguagem, nesse caso do if, é muito comum termos duas ou mais situações. Nesse sentido em Julia podemos também ter alternativas como abaixo:

```
pandemia = true
println("Vou sair de casa?")
if pandemia == true
    println("Só vou sair de casa se for essencial")
else
    println("Balada liberada!!")
end
```

```
Vou sair de casa?
Só vou sair de casa se for essencial
```

No caso de termos mais de uma alternativa, não basta termos só uma condição, nesse caso temos que usar elseif.

```

pandemia = true
tenhoqueestudar = true
println("Vou sair de casa?")
if pandemia == true
    println("Só vou sair de casa se for essencial")
elseif tenhoqueestudar == true
    println("Melhor ficar em casa")
else
    println("Balada liberada")
end

```

```

Vou sair de casa?
Só vou sair de casa se for essencial

```

Conhecendo o if, agora, escreva uma função que recebe os coeficientes, a, b e c de uma equação de segundo grau e imprime as suas raízes reais.

Sim, a forma de se aprender a programar é programando.

Vamos agora a parte mais importante do curso, lembrando que até o momento aprendemos:

- valores - variáveis e alguns dos seus tipos - alguma funções já prontas como `div()`, `typeof()`, `parse()`, `string()`, `println()`, `sin()`, etc - como fazer as nossas funções com a palavra reservada `function` e que termina por `end` - lembrando que a função pode ou não devolver algo através do `return` - lembrando também que uma função pode chamar outra função - como mudar o fluxo de execução normal com o `if`, `elseif`

5.1 Agora sim: Funções que se chamam

Agora podemos, ir ao tópico principal da aula.

Observe a seguinte função `imprime()`.

```

function imprime()
    println("Mensagem positiva")
    imprime()
end

```

```

imprime (generic function with 1 method)

```

Ao ser chamada, o que acontece? O computador fará chamadas seguidas a função, imprimindo a mensagem, até o momento que ocorra uma limitação de memória. Logo, fazer chamadas onde uma função se chama, sem controle não é uma boa ideia.

Por outro lado, podemos pensar em uma forma de chamada controlada, onde a própria função decide o momento de parar de se chamar. Para isso, vamos pegar uma função matemática bem conhecida, o fatorial.

Sabemos que $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$. Mais, ainda dado um número n , sabemos que $n! = n \cdot (n - 1)!$. Continuando, temos que $(n - 1)! = (n - 1) \cdot (n - 2)!$ e assim por diante. Para reproduzir isso no computador precisamos saber quando parar. Para isso, podemos usar que o fatorial de zero é 1, ou $0! = 1$. Logo já temos a primeira parte da função:

```
function fatorial(n)
  if n == 0
    return 1
  else
    # o que vamos colocar aqui?
  end
```

No código acima, temos o critério de parada, ou seja quando n for igual a zero, a resposta será 1. Mas, e se n não for zero. Nesse caso, temos que seguir a fórmula da recursão ou seja $n \cdot (n - 1)!$. Como $(n - 1)!$ pode ser escrito como `fatorial(n - 1)`. Ficamos com a expressão `n * fatorial(n - 1)`.

```
function fatorial(n)
  # Critério de parada: quando n é igual a 0, a recursão termina
  if n == 0
    return 1
  else
    return n * fatorial(n - 1) # Chamada recursiva
  end
end
fatorial(5)
```

120

Vamos a um segundo exemplo, a contagem regressiva. Mais uma vez, quando se chega a zero, podemos considerar que a contagem terminou. Além disso, a cada número, o próximo passo é o número menos 1.


```
function contagem(n)
  if n < 0
    println("Bum!")
  else
    print(n, " ")
    contagem(n - 1)
  end
end
contagem(5)
```

5 4 3 2 1 0 Bum!

Essa estrutura é bem poderosa, pois permite que operações sejam executadas um número controlado de vezes. Voltando ao countdown, imagine que ao invés de imprimir uma mensagem quiséssemos fazer uma conta com o que será devolvido.

```
function soma(n)
  if n > 0
    return n + soma(n - 1)
  else
    return 1
  end
end
soma(11)
```

67

Essa estrutura é bastante poderosa e pode ser usada para o cálculo de produto, nesse caso, a mudança é bem pequena.

Da mesma forma segue um exemplo para o cálculo dos n primeiros elementos da soma harmônica.

```
function somaharmonica(atual, n)
  # Caso base: se 'atual' é maior ou igual a 'n'
  if atual >= n
    # Retorna o recíproco de 'atual' (último termo da soma)
    return 1.0 / atual
  else
    # Caso recursivo: soma o recíproco de 'atual' e chama a função para o próximo número
```

```
    return 1.0 / atual + somaharmonica(atual + 1, n)
  end
end

somaharmonica(1, 10)
```

2.9289682539682538

6

Nessa aula, vamos ver algoritmos um pouco mais elaborados. Mas, sabendo que vamos usar algo com um maior grau de sofisticação, que tal pensar em testes?

De uma forma geral, para verificar o funcionamento de um programa, podemos escrever testes que verificam o funcionamento em algumas situações específicas.

Dado que o primeiro problema que queremos resolver é um algoritmo que encontra o n -ésimo número de Fibonacci. Por que não começar com testes?

Uma forma de se fazer testes, e de forma manual, mas isso não é reproduzível. A melhor maneira de se fazer testes, é de forma automatizada, ou seja criar código que teste código. Isso pode parecer complicado, mas vamos ver abaixo que não é.

Em uma busca rápida, podemos ver que a sequência de Fibonacci é definida da seguinte forma, os dois primeiros elementos F_1 e F_2 valem 1, em seguida temos a fórmula $F_n = F_{n-1} + F_{n-2}$. Mas, antes de pensar em resolver o problema vamos pensar em como testar.

Já sabemos os primeiros valores, além disso, através de uma busca rápida, podemos descobrir alguns valores da sequência como $F_5 = 5$ e $F_{12} = 144$. Supondo que a função para o cálculo do n -ésimo número de Fibonacci chamará `fibonacci()`. Podemos escrever o seguinte trecho de código:

```
function testafibo_versao1()
  if fibo(1) == 1
    println("Deu certo para 1")
  end
  if fibo(2) == 1
    println("Deu certo para 2")
  end
  if fibo(5) == 5
    println("Deu certo para 5")
  end
  if fibo(12) == 144
    println("Deu certo para 12")
  end
  println("Final dos testes")
end
```

testafibo_versao1 (generic function with 1 method)

A função de testes acima verifica se a função fibo() devolve o resultado correto para três casos. Mas, ela tem um defeito, ela imprime mensagens demais, o que pode ser ruim. Considerando isso, vamos ver o primeiro fundamento importante com relação a testes automatizados.

Se o teste passou, ele deve indicar apenas que deu certo!

Levando em conta o que foi escrito acima, podemos mudar o nosso teste para:

```
function testafibo()
  if fibo(1) != 1
    println("Não deu certo para 1")
  end
  if fibo(2) != 1
    println("Não deu certo para 2")
  end
  if fibo(5) != 5
    println("Não eu certo para 5")
  end
  if fibo(12) != 144
    println("Não deu certo para 12")
  end
  println("Final dos testes")
end
```

testafibo (generic function with 1 method)

Agora de posse da nossa função de testes, podemos pensar em escrever a nossa função de Fibonacci. Vamos ao caso fácil de n for menor que 2, a resposta é 1. Como vemos abaixo:

```
function fibo(n)
  if n <= 2
    return 1
  else
    # ainda não sabemos o que colocar aqui...
  end
end
```

fibo (generic function with 1 method)

Mas, a resposta está na própria definição da função, ou seja: $F_n = F_{n-1} + F_{n-2}$. Se o n for maior do que 2, temos que fazer a soma dos valores de Fibonacci de $n - 1$ e de $n - 2$. Ou seja:

```
function fibo(n)
    if n <= 2
        return 1
    else
        return fibo(n - 1) + fibo(n - 2)
    end
end

fibo(10)
```

55

É interessante notar que apesar de ser um dos exemplos clássicos de uso de recursão, o algoritmo acima é extremamente ineficiente. A razão é simples, cada vez que é feita a chamada, todos os valores de Fibonacci são recalculados para os valores de n e $n - 1$.

Como Julia é uma linguagem moderna podemos usar o conceito de Memoização, que evita calcular o que já foi calculado. O Memoize tem que ser instalado no Julia com os comandos `import Pkg` e `Pkg.add("Memoize")`.

```
using Memoize
@memoize function fibo(n)
    if n <= 2
        return 1
    else
        return fibo(n - 1) + fibo(n - 2)
    end
end

fibo(10)
```

55

As diferenças de tempo das duas versões podem ser verificada com o comando `@time`. Da seguinte forma:

```
@time fibo(10)
```

0.000003 seconds

55

Esse tipo de comando, que começa com @ é conhecido como anotação, e tem o poder de mudar o comportamento de partes do código.

Vamos ao segundo algoritmo da aula, o MDC (Máximo Divisor Comum). A ideia é usar o algoritmo de Euclides.

Basicamente ele diz que o MDC de dois números a e b, é igual ao MDC de b e r, onde $r = a \% b$. Quando esse resto for zero, chegamos a solução, que é b.

Vamos começar com os testes para alguns valores bem conhecidos. Por sinal começar pelos testes antes de escrever o código é uma boa prática de programação conhecida por TDD (Test Driven Design).

```
function testaMDC()
  if MDC(3298, 2031) != 1
    println("deu erro, para 3298 e 2031")
  end
  if MDC(120, 36) != 12
    println("deu erro, para 120 e 36")
  end
  if MDC(36, 120) != 12
    println("deu erro, para 36 e 120")
  end
  println("Acabaram os testes")
end
```

testaMDC (generic function with 1 method)

Vamos pensar na função agora. Dessa vez, se o resto for 0, temos que devolver o segundo termo. Caso contrário temos que continuar com a regra

```
function MDC(a, b)
  r = a % b
  if r == 0
    return b
  end
end
```

```

        else
            return MDC(b, r)
        end
    end
end

testaMDC()

```

Acabaram os testes

Até agora usamos o modo interativo do Julia para fazer os nossos códigos. Mas, existe outra forma bem mais reutilizável, ou seja, escrever o texto em arquivos. Isso é relativamente simples, basta usar um editor de texto (puro) da sua preferência, como o notepad, nano, juno, atom, vscode ou outro e salvar um arquivo com a extensão .jl.

Mas, para que algo seja executado é importante colocar uma chamada ao final. Veja abaixo um possível arquivo mdc.jl.

```

function testeMDC()
    if mdc(70, 5) != 5
        println("Não funcionou para 70 e 5")
    end
    if mdc(13, 7) != 1
        println("Não funcionou para 13 e 7")
    end
    if mdc(127, 15) != 1
        println("Não funcionou para 127 e 15")
    end
    if mdc(20, 15) != 5
        println("Não funcionou para 20 e 15")
    end
    if mdc(42, 3) != 3
        println("Não funcionou para 42 e 3")
    end
    if mdc(42, 8) != 2
        println("Não funcionou para 42 e 8")
    end
    println("Final dos testes")
end

function mdc(a, b)
    r = a % b
    if r == 0

```

```
        return b
    else
        mdc(b, r)
    end
end

testeMDC()
println("O mdc entre 1227 e 321 é ", mdc(1227, 321))
```

Final dos testes

O mdc entre 1227 e 321 é 3

7 Testes automatizados e um pouco mais de código

Vamos começar o capítulo vendo uma forma mais simples de se rodar testes. Nos testes que vimos até agora sempre havia o teste de uma condição booleana associado a uma mensagem de erro quando não funcionasse. Mas, observando que a mensagem de erro geralmente está ligada à condição, por vezes a condição pode ser auto-explicativa.

Logo, uma forma elegante de expressar as condições pode ser útil na escrita dos testes. Para isso, vamos usar o módulo de testes. Em linguagens modernas, várias das situações repetitivas que enfrentamos podem ser evitadas usando alguma técnica mais moderna.

```
using Test
@testset "Modelo de testes" begin
    @test 2 == 1 + 1
    @test true
    @test !false
end
```

```
Test Summary:      | Pass  Total  Time
Modelo de testes |     3      3  0.0s
```

```
Test.DefaultTestSet("Modelo de testes", Any[], 3, false, false, true, 1.729115739073162e9, 1
```

No trecho acima primeiro indicamos que queremos fazer testes. Em seguida usamos o *test* que espera uma condição ou valor booleano. Finalmente todos os testes são reunidos em um *testset*.

Claro que o teste dá informações relevantes quando falha:

```
using Test
@test 2 + 2 != 4
```

```
Test Failed at REPL[2]:1
Expression: 2 + 2 != 4
Evaluated: 4 != 4
```

Agora sim, vamos pensar em problemas algorítmicos novos. Que tal fazer a soma dos dígitos de um número inteiro. Ou seja, pensar em um número dígito à dígito. Vamos aos testes primeiro:

```
using Test
@testset "Teste da Soma de Dígitos" begin
    @test somaDig(0) == 0
    @test somaDig(1) == 1
    @test somaDig(100) == 1
    @test somaDig(123) == 6
    @test somaDig(321) == 6
    @test somaDig(99) == 18
end
```

Vamos agora tentar pensar em como “descascar” um número, dado o número 123, uma forma seria pegar o resto por 10 (ou seja 3) e depois dividir por 10 (ou seja 12), e assim por diante. Ou seja.

```
function somaDig(n)
    if n <= 0 return 0
    else
        return n % 10 + somaDig(n ÷ 10)
    end
end

println(somaDig(1234))
```

10

Vamos agora a um outro problema clássico, a verificação se um número é ou não é primo. Na prática para fazer isso, temos a definição, um número n é primo apenas se for divisível apenas por 1 e por ele mesmo. Ou seja, nenhum número entre 2 e $n - 1$ pode ser divisor de um número primo.

A forma de se fazer isso é relativamente simples. Vamos pensar em uma função que tenta dividir um número recursivamente, se conseguir devolve falso, se não conseguir devolve verdadeiro.

Vamos aos código:

```
function divide(n, i)
    if n % i == 0
        return false
    end
end
```

```

elseif i == n - 1
    return true
else
    return divide(n, i + 1)
end
end
end

```

divide (generic function with 1 method)

Que pode ser chamada por:

```

function éPrimo(n)
    return divide(n, 2)
end

```

éPrimo (generic function with 1 method)

Mais um exemplo, o método de Newton para o cálculo de raiz quadrada. Para achar a raiz de x , a partir de um chute inicial (por exemplos $y = x/2$), chegamos a um novo chute que é a média de y e x/y .

Mas, sim, vamos começar com os testes. Como estamos usando números do tipo *double* é bom sempre ter uma tolerância, por isso vamos usar uma comparação aproximada. Também poderíamos ter usado a função *isapprox* da linguagem Julia.

```

using Test
function quaseIgual(a, b)
    if abs(a - b) <= 1e-10
        return true
    else
        return false
    end
end

@testset "Teste da raiz pelo método de Newton" begin
    @test quaseIgual(3.0, raiz(3.0 * 3.0))
    @test quaseIgual(33.7, raiz(33.7 * 33.7))
    @test quaseIgual(223.7, raiz(223.7 * 223.7))
    @test quaseIgual(0.7, raiz(0.7 * 0.7))
    @test quaseIgual(1.0, raiz(1.0 * 1.0))
end

```

Note que como estamos comparando números em ponto flutuante, não usamos a comparação exata.

A solução final é:

```
function newton(c, n)
    q = n / c
    if quaseIgual(q, c)
        return q
    else
        return newton( (c + q) / 2.0, n)
    end
end

function raiz(n)
    a = newton(n / 2.0, n)
    println("a raiz de ", n, " é ", a)
    return a
end
```

raiz (generic function with 1 method)

7.1 Funções caóticas

Vamos brincar um pouco agora com funções caóticas :), isso é, funções, que conforme o comportamento de uma constante k , apresentam resultados que podem convergir ou não. Isso é, a cada passo, quero saber o valor do próximo ponto aplicando a função novamente, isso é:

$$x_1 = f(x_0), x_2 = f(x_1), \dots, x_n = f(x_{n-1})$$

As funções caóticas desempenham um papel significativo em diversas áreas da matemática e da física, com aplicações que vão desde a modelagem de crescimento populacional até a previsão de padrões climáticos. Elas também são fundamentais na análise de circuitos elétricos não lineares, onde pequenas variações nas condições iniciais podem levar a resultados drasticamente diferentes.

Para o nosso teste, a função f é extremamente simples: $x_{i+1} = x_i * (1 - x_i) * k$.

Implemente a função e imprima os 30 primeiros resultados. Comece com um valor de x entre 0 e 1, como 0.2. Use constantes $k = 2.1, 2.5, 2.8$ e 3.1 o que ocorre com $k = 3.7$?

Entregue o código e um pequeno relatório sobre o que acontece.

8 Uma outra forma de se fazer laços

Até o momento vimos que o computador é muito bom para fazer contas e repetições. Fizemos isso até agora com funções recursivas. Mas, existe um outro comando para isso, o `while`. A motivação é que enquanto alguma condição for válida, o computador continua repetindo os comandos.

O formato básico é o seguinte:

```
while condição
  # execute obloco
end
```

Enquanto a condição continuar verdadeira, o computador vai seguir repetindo o bloco que pode ser formado por várias instruções. Logo, para que a repetição, ou laço, não seja repetido indefinidamente, é essencial que algo ligado a condição seja atualizado no corpo do `while`.

Vejamos o exemplo simples da contagem regressiva:

```
n = 5
while n > 0
  println(n)
  n = n - 1
end
println("Acabou")
```

```
5
4
3
2
1
Acabou
```

Mas, vamos ver abaixo um caso onde o uso de `while` deixa o código mais Claro que com a recursão (onde é ruim fazer uma com vários parâmetros). Veja a resolução da série de Taylor abaixo:

```

function sinTaylor2(x)
    i = 1
    termo = x
    soma = 0.0
    while i <= 15
        soma = soma + termo
        termo = -1 * termo * x * x / ((2 * i) * (2 * i + 1))
        i = i + 1
    end
    return soma
end

```

sinTaylor2 (generic function with 1 method)

Nela são calculados os 15 primeiros termos.

Observem a versão recursiva:

```

function sinTaylor(x)
    return sinTaylorRec(1, 15, x, 1, x)
end

function sinTaylorRec(i, n, x, sinal, termo)
    if n == i
        return 0.0
    else
        return sinal * termo +
            sinTaylorRec(i + 1, n, x, -1 * sinal, termo * x * x / (2*i * (2*i+1)))
    end
end

```

sinTaylorRec (generic function with 1 method)

Podemos também fazer operações com os dígitos de um número inteiro, para isso operações como o resto da divisão por 10 e a divisão inteira por 10 são bastante úteis. Abaixo temos as duas versões que fazem a soma dos dígitos de um número inteiro.

```

using Test
function testaSD()
    @test sd(123) == 6
    @test sd(321) == 6
end

```

```

@test sd(0) == 0
@test sd(1001) == 2
@test sd(3279) == 21
println("Fim dos testes")
end

function sd(x)
    if x == 0
        return 0
    else
        d = x % 10
        return d + sd(div(x, 10))
    end
end

function sd1(x)
    soma = 0
    while x != 0
        d = x % 10
        soma = soma + d
        x = div(x, 10)
    end
    return soma
end
testaSD()

```

Fim dos testes

9 Aula de exercícios

9.1 Revisitando o cálculo do fatorial, recursivo e iterativo

Agora que aprendemos a fazer também repetições com o comando while, sempre é bom pensar em qual o comando mais adequado. Vejamos o exemplo abaixo com duas versões da função para o cálculo do Fatorial.

```
function fatorial_recursivo(n::Int64) # Com o ::Int64 estamos definindo que o parâmetro da f
    # Caso base do fatorial: 0! e 1! são iguais a 1
    if n == 0 || n == 1
        return 1
    # Chamada recursiva: n! = n * (n-1)!
    else
        return n * fatorial_recursivo(n - 1)
    end
end

function fatorial_iterativo(n::Int64)
    # Inicializa o resultado como 1 (já que o fatorial de 0 é 1)
    resultado = 1

    # No loop estamos fazendo a multiplicação: n * (n-1) * ... * 2
    while n > 1
        # Multiplica o resultado pelo valor atual de n
        resultado *= n

        # Decrementa n em 1 para continuar o cálculo do fatorial
        n -= 1
    end
    return resultado
end

println(fatorial_recursivo(3))
```


No código acima temos uma novidade, nos parâmetros da função, o tipo está sendo declarado explicitamente. No caso, estamos dizendo que o valor n que a função vai receber é de um tipo específico. Ou seja um Inteiro de 64 bits.

O estilo de código está um pouco diferente do que antes, pois foi escrito por outra pessoa. A monitora. Vemos que ela tem o hábito de usar nomes de variáveis maiores além do que usar contrações como $+=$ e $*=$.

9.2 Aproximação da raiz quadrada

Para o próximo exemplo, vamos ver o método de Newthon-Raphson para o cálculo da raiz quadrada. É um método recursivo no qual o próximo valor é baseado no valor anterior. Quanto mais chamadas forem feitas, mais próximo do valor final vai se chegar.

Mais informações sobre o método podem ser encontradas em [aqui](#). Mas para o momento temos que pensar na seguinte implementação. Para se calcular a raiz, podemos usar a seguinte fórmula, a partir de um palpite inicial r , para o valor da raiz de x .

$$r_{n+1} = 0.5 * (r + x/r)$$

Como o código abaixo é mais complicado, foram usados comentários.

```
function aproxima_raiz(x::Float64, epsilon::Float64)::Float64
    if x < 0
        return nothing
    end

    # Chute inicial
    aproximacao = x/2
    melhor_aproximacao = aproximacao

    while true
        # Fórmula para aproximação de raiz quadrada utilizando o método de Newthon-Raphson
        melhor_aproximacao = 0.5 * (aproximacao + x/aproximacao)

        # Se a distância absoluta entre os dois pontos é menor do que epsilon, então podemos
        if abs(aproximacao - melhor_aproximacao) <= epsilon
            break
        end

        # Se a aproximação ainda não for boa o suficiente, então atualizamos a aproximação p
        aproximacao = melhor_aproximacao
    end
end
```

```

    end

    return melhor_aproximacao
end

```

aproxima_raiz (generic function with 1 method)

Notem que foi introduzido um comando novo, o break, esse comando apenas interrompe a execução do while. Ou seja, força a saída do laço.

9.3 Verificar se um número é primo

No próximo exemplo, vamos verificar se um número é primo, ou seja, se os seus únicos divisores são 1 e o próprio. A forma mais simples de se fazer isso é procurando dividir o número por outros. Se algum dividir, o número não é primo.

```

function verifica_primo(num :: Int64)
    if num <= 1
        return false
    end
    i=2
    # pode ser melhorado com i<=num/2
    # ou também com i<= sqrt(num): baseado no fato que um número composto deve ter um fator n
    while i<num
        if num % i == 0
            return false
        end
        i+=1
    end
    return true
end

```

verifica_primo (generic function with 1 method)

Assim, como o comando break é usado para interromper a execução de um laço, o comando return, pode ser usado para terminar a execução de uma função, a qualquer momento.

9.4 Verificar se um número é palíndromo

Um número palíndromo é um número que é simétrico. Ou seja, a leitura dos dígitos da esquerda para a direita é igual a leitura dos dígitos na ordem inversa. Por exemplo, o número 121 é palíndromo, assim como o 11 e o 25677652. Os números de um dígito também são.

```
function e_palindromo(n::Int64)
    #=
        Guarda os dígitos de n que ainda devem ser invertidos
        A variável auxiliar é necessária para que o valor de n não seja, perdido, e possamos
    =#
    aux = n
    # Guarda a inversão do número n
    n_inv = 0

    #=
        Continuamos o while enquanto ainda há números a serem invertidos,
        ou seja, enquanto aux for maior que 0.
    =#
    while aux > 0
        # Coloca o último dígito de aux na variável que guarda a inversão
        resto = aux % 10
        n_inv = n_inv * 10 + resto

        # Retira o último dígito de aux
        aux = div(aux, 10)
    end

    if n == n_inv
        println("O número $n é palíndromo")
    else
        println("O número $n não é palíndromo")
    end
end

e_palindromo(2002)
e_palindromo(1234)
```

O número 2002 é palíndromo

O número 1234 não é palíndromo

10 Revisitando a aula passada

Além de discutirmos o que vimos na aula passada. Nessa aula, vimos uma nova solução para o problema de verificar de um número é palíndromo.

Para isso usamos uma técnica um pouco diferente, ou seja, ao invés de inverter o número e compará-lo com o original. Verificamos se os seus extremos são iguais.

Observe o número 234432, o primeiro passo seria verificar que nos extremos, mais significativo e menos significativo, temos os números 2. Em seguida, podemos continuar com a verificação para o número 3443. Se em algum momento a verificação falhar o número não é palíndromo.

Seguem os testes e o código abaixo.

```
using Test

function testaPal()
    @test testaPal(1)
    @test testaPal(131)
    @test testaPal(22)
    @test testaPal(53877835)
    @test !testaPal(123)
    @test !testaPal(23452)
    println("Final dos testes")
end

function testaPal(n::Int64)
    # o primeiro passo é encontrar um número com o mesmo número de dígitos de n
    pot10 = 1
    while pot10 < n
        pot10 = pot10 * 10
    end
    pot10 = div(pot10, 10)

    while n > 9
        d1 = n % 10
        d2 = div(n, pot10)
```

```

    if d1 != d2
        return false
    end
    n = div(n % pot10, 10)
    pot10 = div(pot10, 100)
end
return true
end

```

testaPal (generic function with 2 methods)

10.1 Aleatoriedade

Em julia temos a função `rand()` que devolve um número em ponto flutuante entre 0 e 1. Conforme os parâmetros, podemos ter outros tipos de número como:

```

rand{Int}() # devolve um inteiro
rand{Int}(1:10) # devolve um número entre 1 e 10
rand{Bool}() # devolve verdadeiro ou falso

```

false

Mas, antes de ver um código com `rand()`. Vamos pensar em um problema da vida real. Imagine que temos que fazer um sorteio justo, e o único instrumento que possuímos para o sorteio é uma moeda viciada. Que tem como resultado muito mais faces do que coroas. Dá para usar essa moeda em um sorteio justo?

A ideia para resolver o problema é olhar para pares de sorteios. Ou seja, vamos ignorar sorteios onde tenhamos duas faces ou duas coroas. Nos outros, teremos uma coroa e uma face ou vice versa. As chances das duas serão de 50%. Logo podemos assim, corrigir a moeda viciada.

Para simplificar o exercício, a moeda pode devolver 0, ou 1, correspondentes a cara ou a coroa. Observe a seguinte função que simula uma moeda viciada.

```

function sorteio()
    if rand() > 0.90
        return 1
    else
        return 0
    end
end

```

sorteio (generic function with 1 method)

Pode se observar que a função devolve 0 na maior parte das vezes. Podemos inclusive ver isso, fazendo mil sorteios:

```
function verificaSorteio()
  cara = 0
  coroa = 0
  i = 0
  while i < 1000
    if sorteio() == 0
      cara = cara + 1
    else
      coroa = coroa + 1
    end
    i = i + 1
  end
  println("O número de caras foi: ", cara," e de coroas foi :", coroa)
end
```

verificaSorteio (generic function with 1 method)

Mas, podemos corrigir o sorteio da seguinte forma:

```
function sorteioBom()
  sorteio1 = sorteio()
  sorteio2 = sorteio()
  while sorteio1 == sorteio2 # se forem iguais, tente novamente
    sorteio1 = sorteio()
    sorteio2 = sorteio()
  end
  return sorteio1 # ao termos um diferente, podemos devolver o primeiro sorteio
end
```

sorteioBom (generic function with 1 method)

Podemos usar o verificaSorteio para ver a diferença.

```

function verificaSorteio()
    cara = 0
    coroa = 0
    i = 0
    while i < 1000
        if sorteioBom() == 0
            cara = cara + 1
        else
            coroa = coroa + 1
        end
        i = i + 1
    end
    println("O número de caras foi: ", cara," e de coroas foi :", coroa)
end

```

verificaSorteio (generic function with 1 method)

Podemos ainda aproximar o número de Euler (e), constante matemática que é a base dos logaritmos naturais, usando uma simulação probabilística. A ideia por trás desse código é que o número médio de tentativas necessárias para que a soma de números aleatórios entre 0 e 1 ultrapasse 1 se aproxima do valor de e . Isso é baseado em uma relação matemática que conecta essa situação ao número e .

```

function calculaEuler(total)
    soma_tentativas = 0
    for i in 1:total
        soma = 0.0
        tentativas = 0
        while soma <= 1 # Continue gerando números até a soma ultrapassar 1
            soma += rand() # Gera número aleatório entre 0 e 1
            tentativas += 1
        end
        soma_tentativas += tentativas # Somar o número de tentativas necessárias
    end
    return soma_tentativas / total # A média do número de tentativas será uma estimativa
end

println("Estimativa de e (1000 iterações): ", calculaEuler(1000))
println("Estimativa de e (100000 iterações): ", calculaEuler(100000))
println("Estimativa de e (100000000 iterações): ", calculaEuler(100000000))

```

Estimativa de e (1000 iterações): 2.713
Estimativa de e (100000 iterações): 2.71589
Estimativa de e (100000000 iterações): 2.71811446

Para terminar a aula vamos aplicar o método de Monte Carlo para o cálculo de Pi. Imaginem o primeiro quadrante, onde temos um semi-círculo de raio 1, dentro de um quadrado de lado 1. Podemos sortear valores, os que saírem dentro do círculo podem contar para a área desse. Mais informações podem ser vistas aqui (https://pt.wikipedia.org/wiki/M%C3%A9todo_de_Monte_Carlo)

```
function calculaPi(total)
    noAlvo = 0
    i = 0
    while i < total
        x = rand() / 2.0 # gera um número entre 0 e 0.5
        y = rand() / 2.0
        if sqrt(x * x + y * y) <= 0.5
            noAlvo = noAlvo + 1
        end
        i = i + 1
    end
    return 4 * (noAlvo / total) # precisamos multiplicar para ter a área de 4 quadrantes
end

println(calculaPi(100))
println(calculaPi(100000))
println(calculaPi(100000000))
```

3.04
3.139624
3.141516436

11 Entrada de dados e o começo de listas

Nessa aula, temos dois tópicos principais, como fazer a entrada de dados, através de comandos de entrada e com argumentos na linha de comando. Além disso também veremos como tratar de um tipo especial de variável, onde é possível, guardar mais de um valor.

11.1 O comando input

Quando queremos inserir dados, em Julia, basta colocar dados. Mas, como podemos fazer para entrar dados em um programa comum?

Para isso temos o comando `readline()`, que interrompe a execução do programa e espera pela entrada de uma `String`, o que ocorre quando a tecla “enter” é pressionada.

```
println("Digite o seu nome")
resposta = readline()
println("O seu nome é: ", resposta)
```

Caso, ao rodar o programa, você digitar `Maria`, e pressionar a tecla `enter`, a resposta final do seu programa será `O seu nome é: Maria`.

Como o `readline()` lê `Strings`, se quisermos ler números, é necessário usar o comando `parse`. O comando `parse` de forma simples possui dois parâmetros, o primeiro corresponde ao tipo que se quer transformar, e o segundo o valor original.

```
println("Digite um inteiro")
valor = parse{Int64, readline()}
println("O numero digitado foi ", valor)
```

Sabendo ler números do teclado, vamos a um exercício simples, ler uma sequência de números inteiros terminada por zero e devolver a sua soma.

```

function somaVarios()
    soma = 0.0
    println("Digite um número")
    n = parse(Float64, readline())
    while n!=0
        soma = soma + n
        println("Digite um número")
        n = parse(Float64, readline())
    end
    println("A soma é: ", soma)
end

```

Observe o seguinte exemplo que calcula os quadrados dos números de uma lista terminada por zero.

```

function leQ()
    x = readline()
    n = parse(Float64, x)
    while n != 0
        println("$n ao quadrado é ", n * n)
        x = readline()
        n = parse(Float64, x)
    end
end

```

Notem que o `readline` também pode receber uma variável de arquivo para que dados sejam lidos diretamente. Mas, nesse caso temos que tomar cuidado para abrir (`open()`) e fechar (`close()`) o arquivo. Como abaixo:

```

function leQ()
    println("Digite um número")
    f = open("numeros.txt", "r+")
    x = readline(f)
    n = parse(Float64, x)
    while n != 0
        println("$n ao quadrado é ", n * n)
        println("Digite outro número")
        x = readline(f)
        n = parse(Float64, x)
    end
    close(f)
end

```

11.2 Lendo através da linha de comando

A outra forma de ler comandos é através da constante `ARGS` que é preparada na chamada de um programa. Para entender melhor isso, vamos ver o seguinte programa.

```
println(ARGS)
```

Se a linha acima está no arquivo `args.jl`, ao chamar `julia args.jl` com diversos parâmetros, teremos diversos resultados diferentes.

Por exemplo ao chamar:

```
julia args.jl 1 2 3 abc
```

Teremos como resposta

```
["1", "2", "3", "abc"]
```

Vamos analisar um pouco melhor essa resposta observando que cada parâmetro está em uma posição.

```
tam = length(ARGS)
println("O tamanho dos argumentos é: ", tam)
for i in 1:tam
    println(ARGS[i])
end
```

Olhando o código acima, podemos ver que a função `length()` devolve o número de argumentos, ou seja, o tamanho da lista `ARGS`. Além disso com os colchetes é possível acessar a cada posição da lista de forma individual.

O exemplo abaixo soma os parâmetros inteiros dados como argumentos. Ele também ilustra uma boa prática que é, sempre colocar o código em módulos, no caso abaixo em funções:

```
function SomaEntrada()
    tam = length(ARGS)
    s = 0
    i = 1
    while i <= tam
        valor = parse{Int, ARGS[i]}
        println(valor)
        s = s + valor
        i = i + 1
    end
end
```

```

    end
    println("A soma foi: ", s)
end
SomaEntrada()

```

A flexibilidade que temos ao usar listas é enorme! Por isso, listas ou vetores, merecem um tópico próprio.

11.3 Listas

Vamos primeiro brincar um pouco no console.

```

vetor = [1, 2, 3]
println(vetor[1])
println(length(vetor))
vetor[2] = vetor[2] + 1
vetor[1] = 2 * vetor[3]
println(vetor)

```

```

1
3
[6, 3, 3]

```

Como disse antes, o for foi feito para manipular vetores, vamos ver umas funções, a primeira que imprime os elementos de um vetor um por linha.

```

function imprimeVetor(v)
    for el in v
        println(el)
    end
end

```

Isso também pode ser feito por meio dos índices do vetor:

```

function imprimeVetor(v)
    for i in 1:length(v)
        println(v[i])
    end
end

```

Como cada posição é independente, podemos calcular a soma dos elementos ímpares de um vetor

```
function somaImpVetor(v)
    soma = 0
    for i in 1:length(v)
        if v[i] % 2 == 1
            soma = soma + v[i]
        end
    end
    return soma
end
```

Também vimos em aula alguns outros exemplos, como calcular a média dos elementos em um vetor.

```
function mediaV(v)
    soma = 0.0
    for i in v
        soma = soma + i
    end
    return soma / length(v)
end
```

Devolver a soma dos elementos ímpares de um vetor

```
function somaImpar(v)
    soma = 0
    for i in v
        if i % 2 == 1
            soma = soma + i
        end
    end
    return soma
end
```

Imprimir os números divisíveis por 5 de um vetor.

```
function imprimeDivisivelPor5(v)
    for i in v
        if i % 5 == 0
            println(i)
        end
    end
end
```

```

        end
    end
end

```

Com uma pequena variação e usando o comando `push!()` podemos ver como devolver um vetor com os números divisíveis por 5.

```

function devolveDivisivelPor5(v)
    x = [] # começa com um vetor vazio
    for i in v
        if i % 5 == 0
            push!(x, i) # adiciona um elemento ao vetor x
        end
    end
    return x
end

```

11.3.1 Álgebra linear e Listas

A manipulação de listas é uma parte fundamental da álgebra linear, que estuda vetores e matrizes. Funções como o produto escalar de dois vetores são exemplos clássicos. Abaixo temos dois exemplos de produto escalar de dois vetores. lembrado esse é definido como a soma dos produtos de elementos em posições iguais.

```

function dotProduct(a, b)
    soma = 0
    if length(a) != length(b)
        return soma # o produto não está definido se os tamanhos são diferentes
    end
    for i in 1:length(a)
        soma = soma + a[i] * b[i]
    end
    return soma
end

```

Acima vimos que um caso especial do uso do `for`, consiste em fazer `for` varias entre 1 e um tamanho (`1:length(a)`)

Observem a diferença na versão abaixo:

```
function dotProduct(a, b)
    soma = 0
    if length(a) != length(b)
        return soma    # o produto não está definido se os tamanhos são diferentes
    end
    i = 1
    for x in a
        soma = soma + x * b[i]
        i = i + 1
    end
    return soma
end
```

11.3.2 Exercício de permutação

Para terminar, vamos fazer uma função onde dado um vetor de inteiros de tamanho n , verifica se esse vetor é uma permutação dos números de 1 a n . Para isso, veremos se cada número de 1 a n está no vetor.

Mas, sem esquecer dos testes:

```
@testset "Verifica Permutação" begin
    @test permuta([1,2,3])
    @test permuta([3, 2, 1])
    @test permuta([1])
    @test permuta([2, 1])
    @test permuta([4, 2, 3, 1])
    @test !permuta([1, 1])
    @test !permuta([1, 3])
    @test permuta([])
end
```

e o código:

```
function permuta(v)
    tam = length(v)
    for i in 1:tam
        if !(i in v)
            return false
        end
    end
    end
```

```
    return true  
end
```

Foi usado o comando `in` de Julia que verifica se um elemento está no vetor.