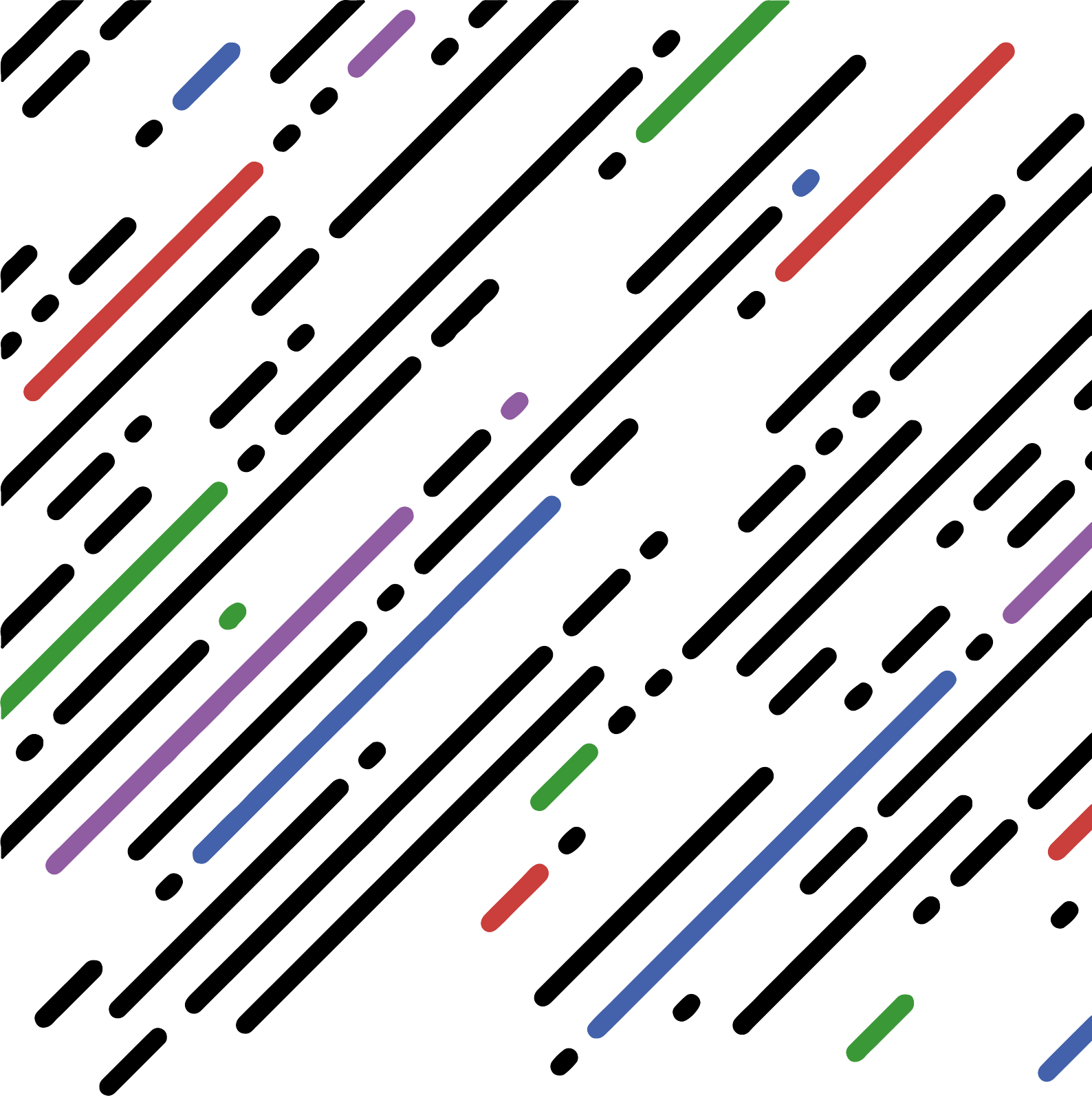

Introdução à Linguagem Julia

Paradigmas de Linguagens de Programação

Daniel Brito dos Santos

Ausberto S. Castro Vera

2 de novembro de 2021

UENF - UNIVERSIDADE ESTADUAL DO NORTE FLUMINENSE DARCY RIBEIRO

CCT - CENTRO DE CIÊNCIA E TECNOLOGIA
LCMAT - LABORATÓRIO DE MATEMÁTICAS
CC - CURSO DE CIÊNCIA DA COMPUTAÇÃO

Primeira edição, Maio 2019

Sumário

1. Introdução

Julia é, acima de tudo, uma linguagem ambiciosa. Surgiu com o objetivo de resolver o chamadado "problema das duas linguagens", em referência ao fato de muitos cientistas, engenheiros e matemáticos usarem uma linguagem flexível e amigável como Python, R, ou Matlab, que os permitia focar no problema sendo investigado ao invés de detalhes de implementação computacional. Mas logo em seguida ser necessário desenvolver uma nova implementação em uma linguagem performática como C para de fato computar os dados, pois do contrário enfrentariam um tempo de processamento ordens de grandeza maior e portanto muitas vezes impeditivo. [Lau19]

Mas a ambição de seus criadores, em suas próprias palavras, foi além:

> Queremos uma linguagem open source com uma licença liberal. Queremos a velocidade de C com o dinamismo de Ruby. Uma linguagem homoiconica com macros verdadeiras como Lisp, mas com notação matemática obvia e familiar como Matlab. Quermos algo tão geral quanto Python, tão fácil para estatística quanto R, tão natural para processamento de strings quanto Perl, tão poderoso para algebra linear quanto Matlab e tão bom como cola de programas quanto shell. Algo estupidamente simples de aprender, e ainda assim deixe o mais sério dos hakers satisfeito. Queremos algo interativo e que seja compilado.
>
> Mencionamos que deve ser tão rápida quanto C? [Why We Created Julia, 2012]

Assim foi lançada em 2012 uma linguagem gratuita, de código aberto, alta performace, tipagem dinâmica e paradigma multimétodos (despacho múltiplo) que permite expressar procedural, funcional, metaprogramação e orientação a objetos. Com suporte nativo e de alto nível ao paralelismo, alta integração com as extensas bilbiotcas já disponíveis para as liguagens que a inspiraram, variáveis unicode com sintaxe de LaTex permitindo letras gregas, subscritos e decoradores, especialmente pensada na computação numérica mas plenamente geral e sim, praticamente tão rápida quanto C. [Lob19, BEKS17]

1.1 Aspectos históricos da linguagem Julia

Figura 1.1: Criadores da linguagem Julia

Fonte: Julia Computing

Na Fig.1.1 vemos seus criadores da esquerda para a direita: Keno Fischer, Viral Shah, Stefan Karpinski, Alan Edelman, e Jeff Bezanson.

Assim como outras linguagens e construtos computacionais, muito de sua história tem pouco registro formal sendo a maior fonte seu site, blog e documentação oficiais, além de palestras em eventos técnicos e artigos jornalísticos, o que torna mais difícil uma recontrução histórica precisa, não obstante, ressaltaremos os principais aspectos à seguir:

- O nome Julia não tem nenhum significado, foi apenas um nome sugerido por um amigo de Benzanson que os criadores acharam bonito. [1]
- O projeto da linguagem começou em 2009.
- Foi lançada para a comunidade open source em 2012, segundo os criadores com 90% das características que envisionaram, inclusive inicialmente chamaram de Julia 1.0, mas posteriormente adiaram esse marco.
- Julia 0.3 foi lançada em agosto de 2014 com melhorias de performace, e nas bibliotecas padrão, além de grande expanção no ecosistema de pacotes.
- Julia 0.4 em outubro de 2015 com refinamentos na linguagem e melhorias nas bibliotecas padrão, nesse momento haviam 700 pacotes registrados oficialmente.
- Nesse mesmo ano foi lançada a Julia Computing, Inc. considerando a necessidade de se ter uma companhia que oferecesse suporte, treinamento e consultoria na adoção da linguagem por big players.
- Julia 0.5 no outubro de 2016, trouxe como principal novidade remover o custo de performace ao utilizar as funções anônimas, clausura, e funções de primeira ordem, conceitos fundamentais da programação funcional que a linguagem suporta desde o início. Trouxe também suporte arquiteturas ARM e Power, multi threading experimental, e simplificação no tipo string, dentre outras.
- Julia 0.6 em junho 2017 com diversas pequenas mudanças tendo em vista aumentar a estabilidade e semântica da linguagem.
- Julia 1.0 dia oito de agosto de 2018, finalmente é lançada versão 1.0. Quase uma década de trabalho, mais de 700 pessoas contribuiram no código fonte, e milhares nos pacotes. Apresentou como principal característica a consolidação da linguagem, com essa base solidificada o foco passa a ser em consruir sobre a fundação. Dentre as várias novidades podemos ressaltar o novo gerenciador de pacotes completamente refeito e representação canônica para missing values (valores faltantes) e tipo String seguro para dados arbitrarios, o

[1] https://docs.julialang.org/en/v1/manual/faq/

que permite trabalhar com dados do mundo real sem o risco de depois de horas ou dias de processamento um caractere inválido levar a uma falha geral.
- Julia 1.5 apresentou grande otimização na forma como seus Structs são alocados no heap, novas melhorias no multithreating, a possibilidade de selecionar o nível de otimização em cada módulo de modo a dimnuir a famigerada latência da primeira execução. nova macro para chamar funções de C, gerador de números aleatórios 6 vezes mais rápido, e padronização do protocolo pkg.
- Julia 1.6 em março de 2021 apresentou principalmente avanços na consolidação da linguagem, com diversas melhorias em diferentes aspectos que de modo geral trouxeram mais robustes e eficiência.

1.2 Áreas de aplicação da linguagem

Júlia é atualmente utilizada por mais de 10 000 empresas, e 1 500 universidades, com 29 milhões de downloads e 87% de crescimento anual. (Julia Computing)

Seus criadores ganharam os prestigiados prémios MIT James H Wilkinson 2018 para Software Numérico e o IEEE Sidney Fernbach 2019 por "avanços extraordinários na computação de alta performace, algebra linear, computação científica e contribuições para a linguagem de programação Julia".

Considerando ainda sua flexibilidade, simplicidade e performace, não é surpresa que tem crescido exonencialmente na computação científica, computação de alta performace e também no propósito geral. A figura 1.2 mostra alguns dos principais clientes da linguagem. [Klo21]

Figura 1.2: Clientes da linguagem Julia

Fonte: Julia Computing

1.2.1 Computação científica

Esse campo consiste principalmente na análise e modelagem exploratória de determinado problema, o que geralmente é feito por especialistas na área que necessitam de um ambiente que facilite a prototipação, e que disponha um ferramental técnico muito específico, e muitas vezes também necessitam da maior perfôrmace possível.

Júlia foi feita sob medida com esse propósito, suprindo todas essas necessidades e ainda oferecendo uma notação matemática clara e intuitiva, com suporte nativo a construtos como matrizes, vetores, funções anônimas e de primeira ordem além das bilbiotecas especializadas que implementam os métodos necessários de estatística, algebra linear, e equações diferenciais pra citar alguns exemplos. [Klo21]

Dessa forma temos visto um importante crescimento da linguagem nas áreas de Data Science, machine learning[2], economia [3], pesquisa operacional, e todas as ciências naturais como biologia, e física. [Per19]

1.2.2 Computação de alta performace

Outra área de igual destaque no uso da Julia é a computação de alta performace, que tem inúmeros exemplos notáveis de uso nas mais diversas industrias como a financeira, farmacéutica[4], médica, aeroespacial [5] dentre várias outras.

Podemos destacar na industria financeira a análise de séries temporais no fundo de investimentos BlackRock[6], cálculo de risco pelo banco britânico Aviva[7], modelos econômicos no Federal Reserve Bank of New York[8], e nosso próprio BND no manejo de quase um trilhão de reais com modelos de otimização estocástica multi-estágios[9], em todas essas implementações ouve um aumento em pelo menos 10x na velocidade de processamento, em boa parte dos casos reduzindo quase a metade o número de linhas de código, o que aumenta expenencialmente a legibilidade, diminui os erros e aumenta a produtividade dos desenvolvedores. Na Aviva chegou a reduzir 93% do código e aumentar em 1 000x a velocidade.

Ela também foi utilizada no projeto Celeste onde atingiu a performace de 1.54 petaFLOPS, e entrou para o seleto panteão das linguagens que alcançaram essa magnitude: Fortran, C, C++ e Julia.[10]

Não atoa foi selecionada pela Aliança de modelagem climática como a única linguagem na sua próxima geração de modelos climáticos. Além de ser utilizada pela NASA e pelo INPE brasileiro no planejamento de missão e simulação de satélites. [11]

1.2.3 Uso geral

Apesar de ser particularmente utilizada em computação técnica, a linguagem Julia é de propósito geral, e também é utilizada em uma variadade de outras aplicações, como por exemplo para gerar sites estáticos por meio da biblioteca Franklin.jl, cliente/servidor HTTP com HTTP.jl, desenvolver aplicações com Gtk.jl ou binários multiplataforma para diversas arquiteturas com a BinaryBuilder.jl

Também é interessante notar que ela saltou da 43ª para a 26ª posição no TIOBE index. Desse modo ela tem crescido de forma saudável, mesmo com uma proposta tão ambiciosa e sendo ainda tão jovem.

Finalmente, podemos esperar muitas novidades pro futuro, pois seu time original, agora expandido tanto em pessoas quanto em capital tem recebido cada vez mais atenção na comunidade open souece e incentivo das mais importantes entidades de programação numérica e industria. [12] [13]

[2] https://juliacomputing.com/case-studies/princeton/
[3] https://juliacomputing.com/case-studies/thomas-sargent/
[4] https://juliacomputing.com/case-studies/astra-zeneca/
[5] https://juliacomputing.com/case-studies/celeste/
[6] https://juliacomputing.com/case-studies/blackrock/
[7] https://juliacomputing.com/case-studies/aviva/
[8] https://juliacomputing.com/case-studies/ny-fed/
[9] https://juliacomputing.com/case-studies/bndb/
[10] https://juliacomputing.com/media/2017/09/julia-joins-petaflop-club/
[11] https://juliacomputing.com/case-studies/BrazilNationalinstituteforspaceResearch/
[12] https://www.fortuneita.com/2021/07/19/julia-computing-raises-24m-in-series-a-former-snowflake-ceo-bob-muglia-joins-board/
[13] https://www.hpcwire.com/off-the-wire/julia-computing-receives-darpa-award-to-accelerate-electronics-simulation-by-1000x/

2. Conceitos básicos da Linguagem Julia

Neste capítulo apresentamos os conceitos basais de sintaxe, design e ferramental necessários para se programar em Julia. Nos baseamos nos três principais livros-texto voltados ao estudo da linguagem, com destaque para [Lob19], seguido de [Lau19] e [Kwo20].

2.1 Preparação do ambiente

Para executar código Júlia, basta baixar e descompactar os arquivos binários no site oficial da linguagem. O arquivo executável vem com um console de interpretação (também conhecido como "REPL - Read, Eval, Print, Loop") por onde é possível executar Julia tanto por linha de comando quanto por meio de scripts .jl. Também é possível utilizar diversos ambientes de desenvolvimento integrado que serão abordados alguns capítulos adiante.

2.2 Considerações gerais sobre sintaxe

A seguir listamos alguns pontos importantes da sintaxe.

- Podemos adicionar ao código comentários de uma linha (#) ou múltiplas linhas (#= =#) que podem ser aninhados e colocados em qualquer lugar.
- Blocos não precisam de parênteses, para indicar o fim de um bloco utilizamos a palavra chave "end".
- O ponto e vírgula (;) é utilizado para suprimir o output de um comando ou acessar o shell (a partir do REPL).
- Espaços são sintaticamente significantes, identação, não.
- Nomes de variáveis aceitam um subconjunto dos simbolos Unicode como letras gregas e até emojis.
- Vetores iniciam com o índice 1.

2.3 Pacotes e módulos

Os criadores da lingua escolheram construir um núcleo leve complementado por uma biblioteca padrão (Standard Library) distribuída junto dos binários, e um poderoso gerenciador de pacotes capaz de baixar (muitas vezes direto de repositórios GitHub), pré compilar, atualizar e resolver dependências de pacotes a partir de comandos simples.

Para acessar o gerenciador de pacotes podemos importar o módulo de pacotes (using pkg) e executar os comandos no seguinte formato: pkg.(ARGS), mas no REPL também podemos apenas digitar] para entrar no modo especial de pacotes que tem algumas facilidades como autocomplete.

Listamos a seguir alguns dos principais comandos:

- **status:** retorna uma lista (nome e versão) dos pacotes instalados localmente.
- **update:** atualiza o índice local de pacotes e os próíos pacotes para a versão mais recente.
- **add nomePacote:** automaticamente baixa e instala um pacote.
- **rm nomePacote:** remove o pacote e todas as suas dependências.

Para utilizar um pacote devemos utilizar um dos comandos seguintes:

- **use:** permite acessar diretamente as funções de um pacote, basta adicionar o comando (using nomePacote) no inicio do script.
- **import:** tem a mesma funcionalidade do use com a diferença de precisarmos nos referir ao nome completo da função (nomePacote.nomeFunção), o que tem a vantagem de manter a organização dos nomes, via de regra se define apelidos para os pacotes como por exemplo chamar o pacote "Plots"de "pl"(const pl = Plots)

Da mesma forma podemos **incluir** ou **importar** qualquer arquivo Julia como módulo, que analogamente é executado ao ser invocado e qualquer símbolo definido estará disponível no escopo onde foi chamado.

2.4 Sistema de ajuda

Além do sistema de pacotes, a linguagem também oferece um sistema de ajuda que retorna informação sobre o uso da maior partes das funções que pode ser acessado no REPL digitando "?"ou através do comando "?termo de busca".

De acordo com a documentação do termo buscado o sistema de ajuda pode retornar uma lista de métodos, exemplo de uso, descrição, lista de argumentos ou termos relacionados.

2.5 Tipos e estruturas de dados

Julia nativamente oferece um sistema hierárquico e bastante completo de tipos predefinidos que são divididos entre escalares (como inteiros, floats, e chars) e containers que acomodam outros objetos (como vetores multidimencionais, dicionários, conjuntos, etc)

Podemos destacar as seguintes características:

- Cada valor (mesmo os primitivos) têm seu tipo único, por convenção iniciando com letra maiúscula como Int64 e Bool.
- Para containers e alguns escalares, o nome de seu tipo é seguido de chaves indicando sua dimenção e os tipos dos elementos contidos, como por exemplo ArrayInt,2. Na terminologia da linguagem são chamados tipos paramétricos.
- Em Julia não existe divisão entre objetos e não-objetos pois todos os valores são objetos tipados, enquanto variáveis são apenas nomes relacionados a valores, portanto sem tipo.
- Podemos converter objetos atraves da função convert(T,x).
- O operador :: pode ser utilizado para anexar anotações de tipo para expressões e variáveis. O que opcional, mas é utilizado em alguns casos para confirmar que um programa se comporta conforme o esperado e oferecer mais um pouco de informação ao compulador que pode em alguns casos melhorar a performace.

2.5.1 Tipos simples

- Caracteres individuais são do tipo Char, e representado por aspas simples.
- Booleanos são do tipo Bool, apenas com as intâncias True e False. Em um contexto de inteiros podem ser interpretados como 0 e 1, porém o oposto não ocorre (if 0 gera o erro de tipo "non-Boolean used in Boolean context"
- O tipo padrão de inteiro é o Int64 (existem outros 9 tipos) capaz de armazenar valores entre -2^{63} e 2^{63-1}.
- De modo análogo o padrão de ponto flutuante é o Float64.
- Números complexos são suportados pela variável global im que representa a raiz quadrada de -1.

2.5.2 Operações básicas

Além dos operadores padrões de soma, subtração, multiplicação e divisão (+, -, *, /) temos a potrenciação através do circunfléxo ("3^2"), o resto pelo operador %, e a divisão inteira pelo símbolo ÷ ("\div")

2.5.3 Strings

As strings em Julia são imutáveis, definidas com aspas duplas e podem ser vistas como um tipo especial de vetor pois suportam indexação e looping.

Algumas das operações típicas com strings são:

- split(s,) nesse exemplo utiliza o espaço como um separador e retorna um vetor com as sublistas.
- join([s1,s2],) que une strings utilizando, nesse caso, o espaço entre cada junção.
- replace(s, termo buscado => substituto) que substiui substrings compatíveis com a busca.
- parse(Int, "64") retorna a string convertida para um tipo numérico, no caso inteiro.
- string(123) retorna a string correspondente ao número.
- Existem três principais maneiras de concaternar strings:

```
> string("Hello, ","world")
> "Hello, World! The answer is $(43)"
> "Hello, " * "world"
```

2.5.4 Arrays

Vetores (Array{T, N}) são contâiners mutáveis de N dimensões que podem conter elementos de apenas um tipo (T), ou heterogêneos, nesse caso tendo o tipo genérico (Any) que em geral é consideravelmente menos performático ou tipo união de outros tipos (exemplo: Union{Int64,String}) que consegue ter performace próxima do monotipo a partir dessa restrição nos tipos.

Além das Strings que são vetores imutáveis de caractéres, também temos os tipos especiais Vector{T} e Matrix{T} que são respectivamente Arrays de uma e duas dimensões.

Criação de Array

Existem diversas formas de criarmos um Array do tipo T:

```
> a  = [1;2;3] #cria um vetor coluna de uma dimensao.
> a = [1 2 3] #cria um vetor linha de uma dimensao.
> a = [] #cria um Array{Any,1}
> a = Int64[] #constroi um vetor de Int64 com uma dimensao.
> a = Array{Int64,1}() #mesmo do anterior, usando construtor.
> a = zeros(n) #cria um Array{Float64,1}
```

```
> a = zeros(Int64,n) #cria um array do tipo Int64 com 10 zeros.
> a = fill(j,n) #vetor com n elementos j
> a = rand(n)) #vetor preenchido com n numeros aleatorios
```

Acessar elementos

Podemos acessar subvetores nos utilizando de chaves para selecionar um elemento (a[2]) ou uma fatia de intervalo fechado (a[de:passo:até]) da seguinte forma:

```
> a = [1 2 3 4 5 6 7]
> a[4] #retorna 4
> a[1:3] #retorna os elementos 1,2,3
> a[1:2:end] #retorna 1 3 5 7
> a[end:-1:1] #retorna 7 6 5 4 3 2 1
```

Principais funções

A seguir apresentamos as principais função para trabalharmos com Arrays, é interessante notar que normalmente as funções que modificam o primeiro argumento contém uma exclamação.

```
> push!(a,b) #adiciona o elemento b no fim de a.
> append!(a,b) #adiciona os elementos de b em a,
  #identico ao push! caso b seja escalar

> c = vcat(1,[2,3],[4,5]) #concatena arrays

> pop!(a) #remove o ultimo elemento do array,
> popfirst!(a) #remove o primeiro elemento do array.
> deletat!(a,pos) #remove o elemento da posicao pos

> pushfirst!(a,b) #b no inicio de a.

> length(a) #=ou=# if a in b end #retorna o comprimento do vetor

> sort!(a) #ordena o vetor a.
> sort(a) #retorna a ordenado sem modificar o vetor original
> reverse(a) #retorna elementos de a invertidos

> unique!(a) #remove duplicatas modificando a.
> unique(a) #retorna a sem duplicadas.
> in(b,a) #checa a existencia de b em a.

> a... #=operador "splat" coverte os elementos em
  parametros para uma funcao.=#

> maximum(a) #=ou=# max(a...) #retorna o maior valor.
> minimum(a) #=ou=# min(a...) #analogo ao anterior.
> sum(a) #retorna a soma dos elmentos de a.
> cumsum(a) #retorna um vetor com a soma cumulativa de a.

> empty!(a) #esvazia um vetor coluna.
> b = vec(a) #transforma vtores linha em vetores coluna.
> shuffle(a) #=ou=# shufle!(a) #=embaralha aleatoriamente
```

```
  os elementos de a.=#
  (requer o modulo Random).=#

> isempty(a) #checa se um array esta vazio.

> findall(x -> x == value, a) #=retorna o indice de todas
  as ocorrencias de x.=#
> deleteat!(a, findall(x -> x == value, a)) #=deleta todas as
  ocorrencias de x de a.=#

> enumerate(a) #retorna um iterador de pares (indice, elemento).
> zip(a,b) #retorna um iterador de pares (a_element,b_element).
```

Arrays aninhados e multidimensionias

Arrays multidimencionais são os objetos do tipo Array{T,N} sendo o número de dimenções N maior que um, enquanto Arrays Aninhados apresentam uma dimensão sendo pelo menos um de seus elementos outro Array, como por exemplo um vetor de vetores Array{Array{T,1},1}.

A principal diferença entre ambos é que em matrizes o número de elemento em cada coluna deve ser igual e as regras da Álgebra Linear são aplicáveis.

Os elementos de Arrays aninhados podem ser acessados por chaves duplas (a[2][3]), Já os elementos de multidimensionais podem ser acessados com os índices de cada dimensão separados por vírgula (a[lin,col]). Para vetores colunas tanto a[2] quanto a[1,2] retornam o segundo elemento.

Podemos contruir Arrays multidimensionais de modo análogo aos vetores, alias estes são um caso específico daqueles:

```
> a = [[1,2,3] [4,5,6]] #cria por colunas
> a = [1 4; 2 5; 3 6] #cria por linhas
> a = zeros(In64,n,m,g) #=cria uma matriz nxmxg
  preenchida com zeros.=#
> a = [3x + 2y + z for x in 1:2, y in 2:3, z in 1:2]
  #=tambem podemos usar list comprehension.=#
```

Também temos funções particularmente úteis para trabalharmos com Arrays multidimencionais:

```
> size(a) #retorna uma tupla com os damanhos de cada dimencao
> ndimns(a) #retorna o numero de dimensoes.
> reshape(a,nElementDim1, nElementsDim2,...,nElementDimN)
  #redimensiona o array.
> dropdimns(a, dimns=(dimRemov1, dimRemov2)
  #remove as dimensoes especificadas
> transpose(a) #=ou=# a' #transpoe vetores ou matrizes.
> hvat(col1,col2) #concatena horizontalmente
> vcat(row1, row2) # concatena verticalmente.
```

2.5.5 Tuplas

Tuplas (Tuple{T1, T2,...}) são listas imutáveis de elementos, que em contraste com Arrays não perdem performace ao conter elementos de tipos diversos porque suas assinaturas são mantidas. Podmos criar tuplas com parênteses ou sem conforme o exemplo:

```
> t = (1, 2.5, "a")
> t = 1, 2.5, "a"
```

```
#Podemos converter uma tupla em um array:
> a = [t...] #utilizando o operador splat
> a = [i[1] for i in t] #utilizando list comprehension
> a = collect(t) #utilizando o operador collect.
#De modo analogo podemos converter um array em uma tupla:
> t = (a...,)
```

2.5.6 Tuplas nomeadas

Já as Tuplas Nomeadas (NamedTuple) são coleções de itens que além do índice seus elementos podem ser identificados pelo nome da seguinte forma:

```
> nt = (a=1,b=2.5) #define uma tupla nomeada
> nt.a #acessa o elemento a da tupla nt.
> keys(nt) #returna uma tupla com os nomes: (a,b)
> values(nt) #retorna uma tupla com os valores: (1, 2.5).
> collect(nt) #retorna um array com os valores.
> pairs(nt) #retorna um iterador dos pares (nome,valor)
```

2.5.7 Dicionários

Dicionários (Dict{Tkey, Tvalue}) guardam mapas de chaves para valores com ordem aparentemente aleatória e diferente das Tuplas nomeadas, são mutáveis, tipo-instáveis. Os principais métodos para trabalharmos com dicionários são:

```
> d = Dict()
  #cria um dicionario vazio.
> d = Dict{String,Int64}()
  #cia um dicionario com tipagem definidapara chaves e valores.
> d = Dict('a'=>1,'b'=>2, 'c'=>3)
  #inicializa o dicionario com valores
> d[novaChave] = novoValor
  #adiciona um novo par key-value ao dicionario.
> delete!(d,"chave")
  #deleta o par correspondente a chave.
> map( (i,j) -> d[i]=j, ['a','b','c'],[1, 2 , 3])
  #adiciona pares de mapeamentos
> d["chave"]
  #=retorna o valor correspondente a chave,
  ou um erro caso ela nao exista.=#
> get(d, 'chave', "Chave inexistente")
  #=retorna o valor da chave ou um valor padrao
  caso ela nao exista.=#
> keys(d) #retorna um iterator com todas as chaves de d.
> values(d) #retorna um iterador com todos os valores de d.
> haskey(d,"chave")
  #retorna um booleano de acordo com a existencia da chave.
> in(('a'=>1), d)
  #=checa se o par existe e a chave corresponde
  ao valor especificado.=#
```

2.5.8 Sets

Usamos sets (Set{t}) para representar conjuntos mutáveis e sem ordem de valores únicos.

```
> s = Set() #=ou=# Set{T}()
#cria um set vazio
> s = Set([1,2,2,3,4])
#inicializa com valores desconsiderando o 2 duplicado
> push!(s,5)
#adiciona elementos
> delete!(s,1)
#deleta elementos
> intersect(s1,s2)
#intersessao de s1 e s2
> union(s1,s2)
#uniao de s1 e s2
> setdiff(s1,s2)
#diferenca entre s1 e s2
```

2.5.9 Números aleatórios

Julia também facilita a obtenção de números pseudo aleatórios:

```
> rand()
#retorna um float entre 0 e 1
> rand(a:b)
#retorna um inteiro no intervalo [a,b]
> rand(a:0.01:b)
#float aleatorio com precisao no segundo digito

#=Tambem podemos obter numeros de alguma distribuicao particular,
desde que usemos o pacote Distributions=#

> rand(nomeDistribuicao[parametros])
> rand(Uniform(a,b))#por exemplo

#=Igualmente importante e a definicao da semente pseudo aleatoria
que permite termos um script reprodutivel:=#
> Random.seed!(1234)
#define a semente pseudo-aleatoria para 1234
> Random.seed!()
#reseta a definicao da semente

> rand(Uniform(a,b),2,3)
#uma matriz 2x3 de numeros uniformemente aleatorios no
intervalo [a,b].
```

2.5.10 Valores Faltantes

Julia suporta diversos conceitos de falta:

- **nothing** (tipo Nothing) é o valor utilizado para blocos e funções que retornam nada. Conhecido como "null do engenheiro de software"

- **missing** (tipo Missing) representa um valor faltante no sentido estatístico, em que deveria haver um valor, apenas o desconhecemos. De modo que os containers e maioria das operações conseguem lidar eficientemente com esses casos. Também conhecido como "null do cientista de dados"
- **Nan** (tipo Float64) representa o resultado de uma operação que retorna é um "não-número". Similar ao missing no sentido de se propagar silentimente ao invés de gerar um erro. Analogamente Julia também oferece Inf (1/0) e -Inf (-1/0).

2.6 Controle de fluxo e funções

Em Julia nos temos as principais estruturas condicionais e repetitivas que encontramos nas linguagens mais populares.

2.6.1 Estrutura em bloco

A sintaxe do controle de fluxo tipicamente se dá atráves de blocos que se iniciam com uma palavra chave seguida de uma condição (com parêntesis opcionais), e finalizam com a palavra "end"da seguinte forma:

```
    <palavra chave> <condicao>
        #... conteudo do bloco....
    end
    #------------------------------#
    #Por exemplo:
    for i in 1:5
        print(i)
    end
```

2.6.2 Iteração repetida

For e while

As funções for e while em Julia são muito flexíveis, soportando os contrutos padrões como percorer vetores, break que imediatamente aborta uma sequência e continue que imediatamente pula para a próxima iteração. Além de suportar condições múltiplas como demonstrado no exemplo abaixo.

```
for i=1:2, j=2:5
        println("i:$i, j:$j")
end
#=temos condicoes multiplas que geram um
loop aninhado onde j percorre o intervalo 2:5
para cada elemento i.=#

for i in "string exemplo"
        println(i)
end
```

List comprehension

List comprehension é essencialmente uma forma compacta de escrever um loop:

```
a = [f(i) for i in [1 2 3]]
#cria um array com f aplicada a cada elemento de [1 2 3]
```

```
[x+2y for x in [10,20,30], y in [1,2,3]]

[d[i]=value for (i,value) in enumerate(lista)

[estudantes[nome] = idade for (nome,idade) in zip(nomes,idades)]
```

Map

Já a função Map aplica uma função a uma lista de argumentos.

```
map((nome,idade) -> estudantes[nome] = idade, nomes, idades)

a = map(f, [1,2,3])

a = map(x->f(x), [1,2,3])
```

2.6.3 Condicionais

Condicionais também são estruturados seguindo o design de simplicidade e similaridade com as linguagens dinámicas:

```
i = 5
if i == 1
        println("i = 1")
elseif i == 2
        println("i = 2")
else
        println("i nao e nem 1 nem 2")
end
```

As expressãos são avaliadas até que seja possível inferir seu resultado (short circuited).

Assim como list comprehension é uma forma concisa de escrever um loop temos no operador ternário uma forma consisa de escrever um condicional:

```
a? b: c
#se a eh verdadeiro execute b, do contrario execute c
```

2.6.4 Funções

Funções em Julia são muito flexíveis, podem ser definadas tanto em linha como em bloco quanto como funções lambda. Além disso, funções também são objetos, e portantanto podem ser atribuídas a novas variáveis, retornadas como valores ou aninhadas:

```
> f(x,y) = 2x+y
#definicao em uma linha
#-------------
> function f(x)
>       2x + y
> end
#definicao em bloco
#--------------
> x,y -> 2x + y
#funcao anonima
#--------------
```

```
> a = f
> a(5)
#funcao como objeto
```

A chamada de funções em Julia seguem uma convenção conhecida como chamada por compartilhamento, que seria entre chamada por referência (onde um ponteiro a memória da variável é passado) e chamada por valor (onde uma cópia da variável é passada e a função trabalha nessa cópia).

Assim, as funções em júlia trabalham nas novas variáveis locais, conhecidas apenas dentro da função, de modo que atribuir a variável a outro objeto não vai influenciar o valor original, mas se o objeto ligado a variável é mutável a mutação do objeto também será aplicada a variável original:

```
> function f(x,y)
>       x = 10
>       y[1] = 10
> end
> x = 1
> y = [1,1]
> f(x,y)
#Resultado: x = 1, y = [10, 1]
```

Na comunidade Julia se recomenda seguir duas regras em relação a funções:

- Que contenham todos os elementos necessários para sua lógica (sem acesso a nenhuma variável exceto seus parâmetros e constantes globais.)
- Que não alterem nenhuma outra parte do programa, ou seja, que não produza nenhum efeito colaterial além da eventual modificação de algum de seus argumentos.

Argumentos

- Normalmente os argumentos são posicionais, mas temos o operador ponto e vírgula (;) para estabelecer que após o mesmo todos os argumentos sejam especificados por nome (keywords arguments)
- Os últimos argumentos podem ser especificados junto com valores padrão.
- Funções podem receber um número variável de argumentos.
- Também temos a opção de especificar os tipos dos argumentos. Como mencionado, a principal razão para limitar os tipos dos parâmetros é para evidenciar possíveis bugs logo no início caso uma função seja acidentalmente chamada com um tipo diferente do planejado, retornando um erro de tipo ao invés de silentemente continuar a execução.
- Retornar um valor é opcional e normalmente as funções retornam o último valor computado, podendo ser inclusive uma tupla.

```
> f(a,b=1;c=2) = (a+b+c)
> f(1,c=3)
#(1+1+3)

> function f(args...)
>       for arg in args
>               println(arg)
>       end
> end

> f(a::In64, b::Int64, c::Int64) = (a+b+c)
```

```
> f(a,b) = a*2,b+2
> x,y = f(1,2)
# x = 2, y = 4
```

3. Alma da linguagem

A maior vantagem de Julia é a posssibilidade de se ter código simultaneamente abstrato e eficiente. Essa característica definidora da linguagem emerge de aspectos fundamentais porém mais avançados de seu funcionamento, dos quais abordaremos os dois principais a seguir, baseado principalte em [Bal16] [Kwo20].

3.1 Compilação

O segredo de sua velocidade reside na habilidade de gerar código especializado para diferentes tipos de inputs, aliada a capacidade ddo seu compilador inferir esses tipos.

Isso porque Julia não tem um passo estático de compilação. O código de máquina é gerado em tempo de execução (JIT) por uma Máquina Virtual de de Baixo Nível (LLVM). Juntos esse sistema e o design da linguagem permitem que ela atinja máxima performace na computacão cienifica, técnica e numérica.

A chave dessa performace é a informação de tipo, que é feita por uma engine de inferência de tipos inteligente e totalmente automática que deduz os tipos baseada nos dados contidos nas variáveis (modelo *DataFlow*). Tanto que a declaração de tipos é opcional, msa pode ser feito para documentar o código, e dar pistas ao compilador para encontrar o caminho ótimo.

Assim, na primeira vez executamos uma fução Julia, ela é parseada para inferência de tipos, a seguir o JIT gera código LLVM que em seguida é otimizado e compilado para código de máquina. A partir da segunda execução, ela é executada diretamente em código de máquina. Podemos inspicionar ambos com as respectivas funções:

```
code_llvm(f,(Int64))

code_native(f,(Int64))
```

3.2 Multimétodo

A partir de sua poderosa inferência de tipos, Julia tem como paradigma principal o chamado Dispacho Múltiplo (Multiple Dispatch) ou Multimétodo. Onde um sistema chamado *dynamic multiple dispatch* eficientemente seleciona o método ótimo para cada um dos argumentos de função dentre os vários métodos definidos.

Assim, de acordo com o tipo é selecionada ou gerada uma implementação específica e extramemente eficiente em código nativo.

Julia, portanto, leva a programação genérica e funções polimórficas ao limite, ao escrever o algorito uma vez e aplica-lo a uma amplo espectro de tipos, oferecendo funcionalidade comum a tipos drasticamente diferentes. Exemplo disso é a função genérica *size* que contem 50 implementações de métodos concretos.

4. Aplicações da linguagem

Neste capítulo vamos apresentar cinco exemplos de aplicações da linguagem Julia.

4.1 Quicksort

4.2

Referências Bibliográficas

[Bal16] Ivo Balbaert. *Julia: high performance programming: learning path.* Packt Publishing, Birmingham, UK, 2016. Citado na página 21.

[BEKS17] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017. Citado na página 5.

[Klo21] Hayden Klok. *Statistics with Julia*. Springer International Publishing, Berlin Heidelberg New York, 1 edition, 2021. Citado na página 7.

[Kwo20] Tom Kwong. *Hands-on design patterns and best practices with Julia*. Packt Publishing, Birmingham, UK, 1 edition, 2020. Citado 2 vezes nas páginas 9 e 21.

[Lau19] Ben Lauwens. *Think Julia*. O'Reilly Media, Sebastopol, CA, 1 edition, 2019. Citado 2 vezes nas páginas 5 e 9.

[Lob19] Antonello Lobianco. *Julia quick syntax reference*. Apress, Berkeley, CA, 1 edition, 2019. Citado 2 vezes nas páginas 5 e 9.

[Per19] Jeffrey M. Perkel. Julia: come for the syntax, stay for the speed. *Nature*, 572(7767):141–142, Jul 2019. Citado na página 8.