1. Como instalar e começar a usar?

https://docs.julialang.org/en/v1/manual/getting-started/

- No terminal, usando a linha de comando interna Julia usando os binários fornecidos;
- Usando Docker imagens de Docker Hub mantido pela Comunidade Docker;
- JuliaPro da Julia Computing inclui Julia e o IDE Juno , junto com acesso a um conjunto de pacotes selecionados para plotagem, otimização, aprendizado de máquina, bancos de dados e muito mais (requer registro).

2. Quais são os processos de tradução utilizados?

Compilação *just-in-time* (JIT) (a execução do código de computador envolve a compilação durante a execução de um programa - em tempo de execução - em vez de antes da execução), implementada usando LLVM.

3. Em que paradigmas se encaixa?

Multiparadigma, combinando recursos de programação imperativa, funcional e orientada a objetos.

4. Os nomes são sensíveis à capitalização?

Os nomes das variáveis diferenciam maiúsculas de minúsculas e não têm significado semântico (ou seja, a linguagem não tratará as variáveis de maneira diferente com base em seus nomes).

5. Quais os caracteres aceitos em um nome?

https://docs.julialang.org/en/v1/manual/variables/

Nomes Unicode (em codificação UTF-8).

No Julia REPL e em vários outros ambientes de edição Julia, você pode digitar muitos símbolos matemáticos Unicode digitando o nome do símbolo LaTeX com barra invertida seguido por tab. Por exemplo, o nome da variável δ pode ser inserido digitando `\delta-tab`. (Se você encontrar um símbolo em algum lugar, por exemplo, no código de outra pessoa, que você não sabe digitar, a ajuda do REPL lhe dirá: basta digitar ? e colar o símbolo.)

6. Existe alguma restrição de tamanho para nomes?

Os nomes de variáveis devem começar com uma letra (AZ ou az), sublinhado ou um subconjunto de pontos de código Unicode maior que 00A0.

7. Como é a questão das palavras-chave x palavras reservadas?

https://docs.julialang.org/en/v1/base/base/#Keywords

Esta é a lista de palavras-chave reservadas em Julia: `baremodule`, `begin`, `break`, `catch`, `const`, `continue`, `do`, `else`, `elseif`, `end`, `export`, `false`, `finally`, `for`, `function`, `global`, `if`, `import`, `let`, `local`, `macro`, `module`, `quote`, `return`, `struct`, `true`, `try`, `using`, `while`. Essas palavras-chave não podem ser usadas como nomes de variáveis.
As sequências de duas palavras seguintes são reservados: `abstract type`, `mutable struct`, `primitive type`. No entanto, você pode criar variáveis com nomes: `abstract`, `mutable`, `primitive type`.
Finalmente, `where` é analisado como um operador infixo para escrever métodos paramétricos e definições de tipo. Além disso, `in` e `isa` são analisados como operadores infixos. A criação de uma variável chamada `where`, `in` ou `isa` é permitida.

8. É possível definir uma variável anônima? Mostre exemplo.
https://docs.julialang.org/en/v1/base/libc/
```
malloc(size::Integer) -> Ptr{Cvoid}
```

Call `malloc` from the C standard library

9. A vinculação de tipos (tipagem) é estática ou dinâmica?
https://docs.julialang.org/en/v1/manual/types/
O sistema de tipos de Julia é dinâmico. O comportamento padrão em Julia quando os tipos são omitidos é permitir que os valores sejam de qualquer tipo. Se necessário, o operador `::` pode ser usado para anexar anotações de tipo a expressões e variáveis em programas.

10. Quais categorias de variável (Sebesta, Seção 5.4.3) apresenta? Mostre exemplos.
https://discourse.julialang.org/t/why-is-heap-memory-bad/33290
https://discourse.julialang.org/t/how-to-know-if-object-memory-resides-on-stack-or-heap/4927/14
Semanticamente, os objetos mutáveis devem estar no monte, enquanto os objetos imutáveis devem estar na pilha. No entanto, isso não é regra, pois o compilador é livre.
Em `x = "string"`, o compilador pode usar a memória da pilha para isso.
Já se redefinir `x = "newstring"`, não significa que `x` seja mutável, pois apenas alterou-se a ligação de `x`. No entanto, em algo como `x = Array{mytype}`, provavelmente usará memória do monte para isso.

11. Permite ocultamento de nomes (variáveis) em blocos aninhados? Mostre exemplo.
https://docs.julialang.org/en/v1/manual/variables-and-scoping/#scope-of-variables
Julia usa escopo léxico, o que significa que o escopo de uma função não herda do escopo de seu chamador, mas do escopo no qual a função foi definida. Por exemplo, no código a seguir, o x interior `foo` refere-se a x no escopo global de seu módulo `Bar`:
```
julia> module Bar
           x = 1
           foo() = x
       end;
```
e não x no escopo onde `foo` é usado:

```
julia> import .Bar
```

```
julia> x = -1;

julia> Bar.foo()
1
```

12. Permite definir constantes? Vinculação estática ou dinâmica? Mostre exemplos.
https://docs.julialang.org/en/v1/base/base/#const
Objetos imutáveis são normalmente alocados dinamicamente na pilha.
`const x = 5`

13. Quais os tipos oferecidos? Mostre exemplos de definição de variáveis de cada tipo.
https://docs.julialang.org/en/v1/manual/types/

- Tipos abstratos: não podem ser instanciados e servem apenas como nós no grafo de tipo, descrevendo assim conjuntos de tipos concretos relacionados, i.e., aqueles tipos concretos que são seus descendentes.
```
abstract type Number end
abstract type Real     <: Number end
abstract type AbstractFloat <: Real end
abstract type Integer  <: Real end
abstract type Signed   <: Integer end
abstract type Unsigned <: Integer end
```

- Tipos primitivos: tipo concreto cujos dados consistem em bits antigos simples. Os exemplos clássicos de tipos primitivos são números inteiros e valores de ponto flutuante.
```
primitive type Float16 <: AbstractFloat 16 end
primitive type Float32 <: AbstractFloat 32 end
primitive type Float64 <: AbstractFloat 64 end
```

- Tipos Compostos: chamados de registros, estruturas ou objetos em várias linguagens. Objetos compostos declarados com `struct` são imutáveis; eles não podem ser modificados após a construção.
```
julia> struct Foo
           bar
           baz::Int
           qux::Float64
       end
```

- Tipos compostos mutáveis: um tipo composto em que as instâncias dele podem ser modificadas
```
julia> mutable struct Bar
           baz
           qux::Float64
       end
```

- Tipos Declarados
```
julia> typeof(Real)
```

```
DataType
```

- Tipo Uniões
```
julia> IntOrString = Union{Int,AbstractString}
Union{Int64, AbstractString}

julia> 1 :: IntOrString
1
```

- Tipos compostos paramétricos
```
julia> struct Point{T}
           x::T
           y::T
       end
```

- Tipos abstratos paramétricos
```
julia> abstract type Pointy{T} end
```

- Tipos de tupla
```
struct Tuple2{A,B}
    a::A
    b::B
end
```

- Tipos de tupla Vararg
```
julia> mytupletype = Tuple{AbstractString,Vararg{Int}}
Tuple{AbstractString,Vararg{Int64,N} where N}
```

- Tipos de tupla nomeados
```
julia> typeof((a=1,b="hello"))
NamedTuple{(:a, :b),Tuple{Int64,String}}
```

- Tipos *Singleton*
```
julia> isa(Float64, Type{Float64})
true
```

- Tipos primitivos paramétricos
```
# 64-bit system:
primitive type Ptr{T} 64 end
```

- *UnionAll* Types
```
julia> const T1 = Array{Array{T,1} where T, 1}
Array{Array{T,1} where T,1}
```

- "Tipos de valor"
```
julia> firstlast(::Val{true}) = "First"
firstlast (generic function with 1 method)

julia> firstlast(::Val{false}) = "Last"
```

```
firstlast (generic function with 2 methods)

julia> firstlast(Val(true))
"First"

julia> firstlast(Val(false))
"Last"
```

14. Existe o tipo função? São cidadãs de primeira classe? Mostre exemplo.

https://docs.julialang.org/en/v1/manual/functions/

As funções em Julia são objetos de primeira classe: podem ser atribuídas a variáveis e chamados usando a sintaxe de chamada de função padrão da variável à qual foram atribuídos. Eles podem ser usados como argumentos e podem ser retornados como valores. Eles também podem ser criados anonimamente, sem receber um nome.

15. Possui ponteiros ou referências? Permite aritmética de ponteiros?

https://stackoverflow.com/questions/59125601/can-i-define-a-pointer-in-julia

Não se pode ter um ponteiro para uma variável. Ao contrário de C/C ++, Julia não funciona assim: variáveis não têm locais de memória. Entretanto, pode-se ter um ponteiro para objetos alocados no monte usando a função `pointer_from_objref(x)`:

Obtenha o endereço de memória de um objeto Julia como o `Ptr`. A existência do resultante `Ptr` não protegerá o objeto da coleta de lixo, portanto, você deve garantir que o objeto permaneça referenciado durante todo o tempo em que `Ptr` será usado.

Esta função não pode ser chamada em objetos imutáveis, uma vez que eles não possuem endereços de memória estáveis.

16. Oferece coletor de lixo? Se sim, qual a técnica utilizada?

https://discourse.julialang.org/t/on-the-garbage-collection/35695

https://github.com/JuliaLang/julia/blob/master/src/gc.c

Possui o coletor de lixo marcar-e-varrer, que marca um conjunto de objetos com raiz em GC e outros objetos rastreáveis (ou alcançáveis) a partir deles e, em seguida, varre os objetos não marcados.

17. É possível quebrar seu sistema de tipos (forçar erro de tipo)? Mostre exemplo.

```
julia> (1+2)::AbstractFloat
ERROR: TypeError: in typeassert, expected AbstractFloat, got a value of
type Int64

julia> (1+2)::Int
3
```

18. Quais os operadores oferecidos? Mostre exemplo de uso de cada operador.

https://docs.julialang.org/en/v1/manual/functions/

Operadores são funções. A maioria dos operadores são apenas funções com suporte para sintaxe especial.

```
julia> 1 + 2 + 3
6
```

```
julia> +(1,2,3)
6
```

Operadores com nomes especiais. Algumas expressões especiais correspondem a chamadas para funções com nomes não óbvios.

Expressão	Ligações
`[A B C ...]`	`hcat`
`[A; B; C; ...]`	`vcat`
`[A B; C D; ...]`	`hvcat`
`A'`	`adjoint`
`A[i]`	`getindex`
`A[i] = x`	`setindex!`
`A.n`	`getproperty`
`A.n = x`	

19. Permite sobrecarga de operadores? Mostre exemplo.

https://discourse.julialang.org/t/overloading-operators/6773/2

Os operadores podem ser definidos como funções normais.

```
import Base.+

struct TestType end

(+)(::TestType, ::Void) = nothing
```

20. Quais operadores funcionam com avaliação em curto-circuito?

https://docs.julialang.org/en/v1/manual/control-flow/#Short-Circuit-Evaluation

A avaliação de curto-circuito é bastante semelhante à avaliação condicional. O comportamento é encontrado na maioria das linguagens de programação imperativas que possuem os operadores booleanos && e `||`: em uma série de expressões booleanas conectadas por esses operadores, apenas o número mínimo de expressões é avaliado conforme necessário para determinar o valor booleano final de toda a cadeia.

Explicitamente, isso significa que:

Na expressão a `&&` b, a sub expressão b só é avaliada se a for avaliada como `true`.

Na expressão a `||` b, a sub expressão b só é avaliada se a for avaliada como `false`.

21. O operador de atribuição funciona como uma expressão?

https://www.juliabloggers.com/returned-value-of-assignment-in-julia/

```
julia> if (t = true)
           println("vai!")
```

```
    end
vai!
```

22. Quais as estruturas de controle (seleção, iteração) oferecidas? Mostre exemplos.
https://docs.julialang.org/en/v1/manual/control-flow/

- Expressões compostas
```
julia> z = begin
           x = 1
           y = 2
           x + y
       end
julia> z = (x = 1; y = 2; x + y)
```

- Avaliação condicional
```
if x < y
    println("x is less than y")
elseif x > y
    println("x is greater than y")
else
    println("x is equal to y")
end

julia> 1 < 2 ? v("yes") : v("no")
yes
```

- Curto-circuito de avaliação

- Avaliação repetida
```
julia> while i <= 5
           println(i)
           global i += 1
       end

julia> for i = 1:5
           println(i)
       end
```

- Manipulação de exceção

- Tarefas (aka co-rotinas)
  Tarefas são um recurso de fluxo de controle que permite que os cálculos sejam suspensos e retomados de maneira flexível. Nós os mencionamos aqui apenas para integridade

23. Quais sentenças de desvio incondicional oferecidas? Mostre exemplos.
https://github.com/JuliaLang/julia/blob/master/test/goto.jl
```
    function goto_test1()
        @goto a
        return false
```

```
    @label a
    return true
end
@test goto_test1()
```

24. Quais os métodos de passagem de parâmetros oferecidos? Mostre exemplos.
Em Julia, todos os argumentos para funções são passados por referência.

25. Permite sobrecarga de subprogramas? Mostre exemplo.
```
import Base.+
struct TestType end
(+)(::TestType, ::Void) = nothing
```

26. Permite subprogramas genéricos? Mostre exemplo.
https://docs.julialang.org/en/v1/manual/types/
```
julia> struct Point{T}
           x::T
           y::T
       end

julia> Point{Float64}
Point{Float64}

function norm(p::Point{<:Real})
    sqrt(p.x^2 + p.y^2)
end
```

27. Como é o suporte para definição de Tipos Abstratos de Dados? Mostre exemplo.
https://docs.julialang.org/en/v1/manual/types/#Composite-Types
https://discourse.julialang.org/t/classes-in-julia/24521/9
Em Julia, é possível ter tipos compostos e os métodos equivalentes para definição de um TAD
```
julia> struct Foo
           bar
           baz::Int
           qux::Float64
       end

julia> foo = Foo("Hello, world.", 23, 1.5)
Foo("Hello, world.", 23, 1.5)

julia> typeof(foo)
Foo
```

28. Permite TADs genéricos/parametrizáveis? Mostre exemplo.
https://docs.julialang.org/en/v1/manual/types/#Mutable-Composite-Types
https://discourse.julialang.org/t/classes-in-julia/24521/2
```
julia> mutable struct Bar
           baz
```

```
            qux::Float64
        end

julia> bar = Bar("Hello", 1.5);

julia> bar.qux = 2.0
2.0

julia> bar.baz = 1//2
1//2
```

29. Quais as construções de encapsulamento oferecidas? Mostre exemplos.

https://stackoverflow.com/questions/40310787/julia-code-encapsulation-is-this-a-generally-good-idea
https://docs.julialang.org/en/v1/manual/modules/

```
module MyModule

export x

x() = "x"
p() = "p"

end
```

30. Quais tipos de polimorfismo suporta? Mostre exemplos.

Ad-hoc	Coerção	Não possui. Só existe conversão explícita de tipos.

	Sobrecarga	`function g(x::Float64, y::Float64)`
		`    2x + 2y`
		`end function`
		
		`function g(x::Int64, y::Int64)`
		`    2x + y`
		`end function`

Universal	Paramétrico	`function same_type{T}(x::T, y::T)`
		`    true`
		`end`
		
		`function f{T}(x::T)::T`
		`    return x`
		`end`

	Inclusão	`abstract type Number end`
		`abstract type Real <: Number end`
		`abstract type AbstractFloat <: Real end`
		`abstract type Integer <: Real end`
		`abstract type Signed <: Integer end`

		`abstract type Unsigned <: Integer end`

31. Permite herança de tipos? Herança múltipla? Mostre exemplo.
https://discourse.julialang.org/t/why-doesnt-julia-allow-multiple-inheritance/14342/3
https://github.com/JuliaLang/julia/issues/2345#issuecomment-54537633
Herança simples, sim. Múltipla, não.

32. Permite sobrescrita de subprogramas? Mostre exemplo.
https://discourse.julialang.org/t/overwriting-functions/404
https://stackoverflow.com/questions/50583861/overloading-vs-overriding-in-julia

33. Permite a definição de subprogramas abstratos? Mostre exemplo.
https://stackoverflow.com/questions/40204956/abstract-types-and-inheritance-in-julia
https://github.com/JuliaLang/julia/issues/6975
https://discourse.julialang.org/t/overriding-a-method-in-a-different-module/5805/6

34. Oferece mecanismo de controle de exceções? Mostre exemplo.
https://docs.julialang.org/en/v1/manual/control-flow/#Exception-Handling

```
julia> try
           sqrt("ten")
       catch e
           println("You should have entered a numeric value")
       end
You should have entered a numeric value
```

35. Possui hierarquia de exceções controlada, como em Java? Qual a raiz?
https://docs.julialang.org/en/v1/manual/control-flow/#Exception-Handling

```
julia> supertype(ArgumentError)
     Exception
```

36. Categoriza as exceções em checadas e não-checadas? Como?
https://discourse.julialang.org/t/is-there-a-better-way-to-do-error-handling-on-julia-than-try-catch/21716/20
https://www.juliabloggers.com/managing-exceptions-with-resulttypes/
https://scls.gitbooks.io/ljthw/content/_chapters/11-ex8.html

37. Obriga a declaração de exceções lançadas para fora de um subprograma?
https://docs.julialang.org/en/v1/manual/control-flow/#Exception-Handling

```
julia> sqrt_second(x) = try
           sqrt(x[2])
       catch y
           if isa(y, DomainError)
               sqrt(complex(x[2], 0))
           elseif isa(y, BoundsError)
               sqrt(x)
           end
```

```
end
```

38. Como você avalia a LP usando os critérios do Sebesta (Seção 1.3)?

Simplicidade	+
Ortogonalidade	+
Tipos de dados	+-
Projeto de sintaxe	+
Suporte para abstração	+-
Expressividade	+
Verificação de tipos	+
Tratamento de exceções	+
Apelidos restritos	+- + = 2

39. Como você avalia a LP usando os critérios do Varejão?

Legibilidade	+
Redigibilidade	+
Confiabilidade	-
Eficiência	+-
Facilidade de aprendizado	+
Ortogonalidade	+
Reusabilidade	+
Modificabilidade	+
Portabilidade	+