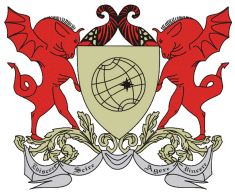

Universidade Federal de Viçosa
Campus Florestal

Dossiê Linguagem Julia

José Wesley de Souza Magalhães

Josué Barbosa Ávalos

Leonardo Júnio Alves dos Santos

Florestal - MG
2017

Sumário

1 Introdução

Julia é uma linguagem de programação com tipagem dinâmica de alto nível com foco em programação científica e numérica. Atualmente, muito da programação matemática, física e da engenharia é desenvolvida em ambientes como MATLAB® e Octave devido à sua facilidade de uso. Porém o C e o Fortran ainda permanecem como mais eficientes, dessa maneira, atingir os dois objetivos faz com que os programadores tenham muitas vezes que utilizar um esquema híbrido abordando os dois tipos de linguagem, fazendo o parte de alto nível numa linguagem dinâmica e o processamento mais complexo é feito em C ou Fortran. Embora vantajoso para algumas aplicações, esse tipo de abordagem pode diminuir de forma acentuada o desempenho e trazer desvantagens, em especial no caso de algoritmos paralelos.

Uma alternativa para contornar tais problemas é aumentar a eficiência das linguagens de tipagem dinâmica existentes, para isso a linguagem Julia foi desenvolvida, tirando proveito de técnicas modernas para executar linguagens dinâmicas de uma forma eficiente.

A linguagem Julia começou a ser desenvolvida por quatro cientistas da computação em 2009: Jeff Bezanson, Stefan Karpinski, Alan Edelman e Viral B. Shah, através do Massachusetts Institute of Technology (MIT). Foi lançada como um projeto *open source* em fevereiro de 2012 no artigo [1]. Os ciradores também lançaram o site oficial da linguagem [2]. Desde então tem sido muito difundida e popularizada entre os programadores devido ao fato de que ela mantém a facilidade de escrita das linguagems de alto nível porém apresenta desempenho melhor.

Sua primeira versão lançada em 2012 foi a 0.1.2, já em novembro de 2013 a versão 0.3 foi lançada. Atualmente se encontra na versão 0.6.0 que foi lançada em novembro de 2016. Também há uma versão beta chamada 0.5.0-dev e é atualizada constantemente.

Figura 1: Logo da linguagem Julia.

Desde seu lançamento a linguagem Julia vem atraindo clientes de peso como grandes empresas do mercado [3]. Um exemplo é a BlackRock, Inc, uma empresa americana

considerada a maior do mundo em gesrão de ativos utiliza Julia desde 2014 para fazer análises de dados temporais e aplicações em big data e foi usada para desenvolver o seu produto Alladin, que é um sistema eletrônico para gerenciadores de investimento com capacidade para processar dados em tempo real. A partir de 2015 o Federal Reserve Bank of New York (FRBNY) passou a usar Julia para fazer modelos da economia dos Estados Unidos, apresentando um desempenho 10 vezes mais rápido que o MATLAB® (que era usado anteriormente). O Banco Nacional de Desenvolvimento Econômico e Social do Brasil (BNDES) também utiliza Julia num modelo matemático programado para realizar cálculos financeiros que incluíam acumulações, dividendos e transações de classificação. Os resultados alcançados pelo BNDES mostraram um ganho de velocidade de 10x, explorando as posibilidades de paralelismo oferecidas pela linguagem. Alguns outros projetos com utilização da linguagem Julia são:

- Sistema de Prevenção de Colisão Aérea, da Administração Federal de Aviação dos Estados Unidos (FAA).
- Pesquisadores do Reino Unido utilizam Julia para um modelo de simulação de crescimento de tumores.
- Pesquisadores da Universidade da California em Berkeley utilizam um modelo de otimização desenolvido em Julia para controle de carros de corrida autônomos.
- Uma das maiores empresas de seguros da europa, a Aviva passou a utilizar Julia em 2016 em seu sistema de modelagem de riscos para a economia global.

Todas as empresas e universidades envolvidas nesses projetos já declaram publicamente ganhos diretos do uso de Julia, principalmente ganhos de velocidade em desempenho.

Em 2017 foi anunciada a participação da Julia no Celest Project, e a sua utilização nos testes de um novo método de computação paralela para processar um conjunto de dados coletados por um outro projeto de 1998 chamado Sloan Digital Sky Survey. Esse método de computação foi capaz de produzir um catálogo contendo 188 milhões de objetos astronômicos, incluindo estrelas e galáxias, em apenas 14,8 minutos, com baixas taxas de incerteza.

A linguagem Julia possui uma conferência que ocorre anualmente no mês de junho, a JuliaCon, [4]. Sua primeira edição ocorreu em 2014 no Gleacher Center, em Chicago; em 2015 e 2016 aconteceu no próprio Massachusetts Institute of Technology em Cambridge e em 2017 foi realizada na University of California, em Berkeley. A JuliaCon 2018 irá acontecer especialmente em agosto, mais precisamente do dia 7 ao dia 11, e

será a primeira edição europeia da conferência, que acontecerá na University College London, em Londres.

Figura 2: Pôster oficial da JuliaCon 2017.

Julia apresenta diversas funções matématicas e com alta precisão dos valores numéricos, recursos para computação paralela e distribuída, além de facilidade para integrar com funções em C sem necessidade de nenhuma API específica e em Python através do pacote PyCall. Funciona nos sistemas operacionais Windows, Linux e Mac OS.

2 Propriedades Desejáveis

Existe uma série de propriedades que são desejáveis em uma linguagem de programação, sendo determinantes na escolha da mesma para o desenvolvimento de algum software. Aqui abordaremos estas propriedades no contexto da linguagem Julia.

2.1 Legibilidade

Julia possui muitos mecanismos de forma a facilitar o entendimento de um código. No exemplo abaixo podemos citar o fechamento de blocos com o termo reservado *end*, e uso do termo *function*, indicando que está sendo criado uma função.

```
function Positivo(x::Array{Int64})
    for i in x
        if(i>=0)
            println("Positivo")
        else
            println("Negativo")
        end
    end
end
```

2.2 Regibilidade

A linguagem Julia também possui elementos pró redigibilidade, facilitando a vida do programador que pode expressar uma idéia utilizando poucas linhas de código. No exemplo abaixo vemos como é fácil escrever na tela, criar uma função, e criar um vetor de 10 posições preenchido com valores entre 100 e 110, em que todos os exemplos são feitos em uma linha de código.

```
print("Trigonometria\n");
f(x) = mod(x,360) > 0 && mod(x,360) < 180 ? print("sen(x) > 0"): print
r = rand(100:110, 10)
```

2.3 Confiabilidade

A linguagem Julia possui muitos mecanismos para garantir a confiabilidade de um código. Dentre os mesmo podemos destacar no exemplo abaixo a verificação dos índices de vetores em tempo de compilação, além do tratamento de excessões.

```
vetor = [1,2,3,4,5]
try (x = vetor[6])
    print(x)
catch e
    print(e)
end
```

Resultado de execução: BoundsError([1, 2, 3, 4, 5], (6,))

2.4 Eficiência

A linguagem Julia mesmo sendo uma linguagem dinâmica preza muito pela eficiência, podendo em alguns cenários se equiparar a linguagem C. Abaixo é mostrado uma tabela comparativa da execução de um Bechmark em diferentes linguagens, onde é possível perceber a eficiência que a linguagem Julia possui. O tempo gasto é relativo a linguagem C (Tempo em C = 1).

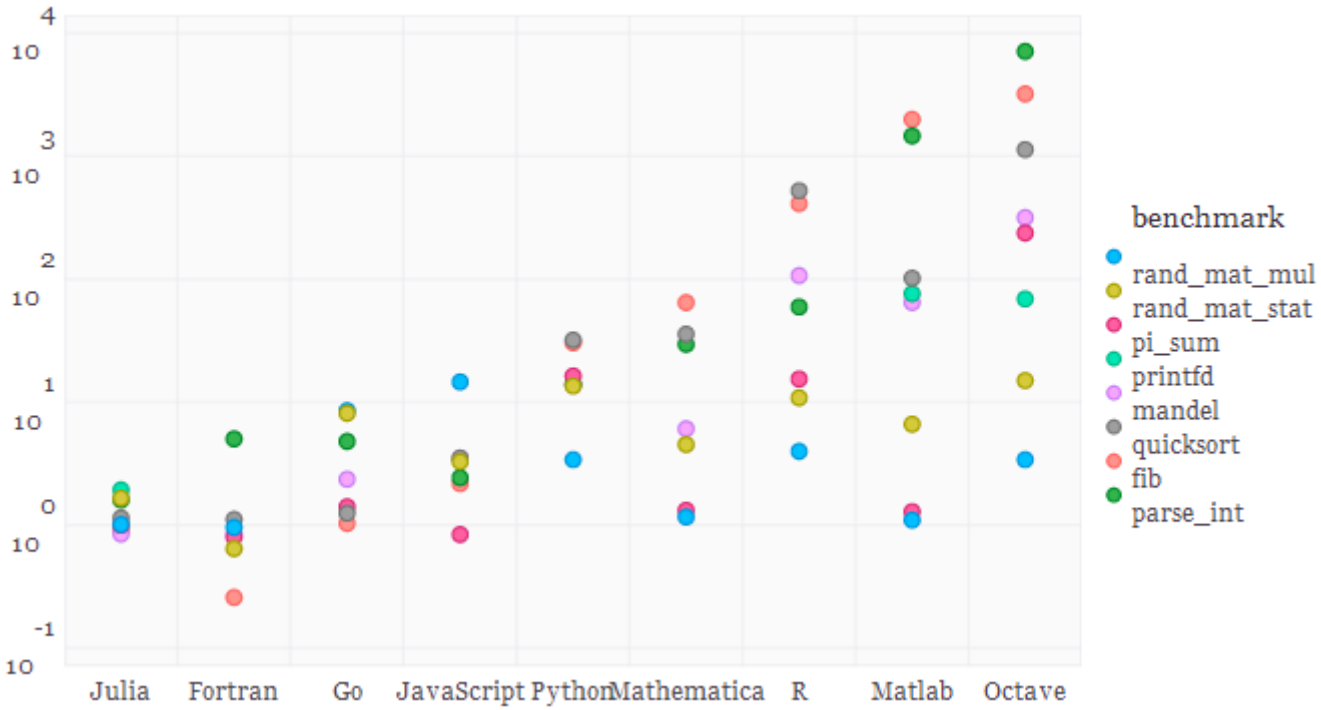

Figura 3: Comparação de desempenho da linguagem Julia com outras linguagens.

2.5 Facilidade de Aprendizado

A linguagem Julia, não possui muitas formas de se realizar o mesmo procedimento, facilitando assim o aprendizado do programador. Durante a pesquisa, foi possível perceber apenas duas formas de se incrementar uma variável, sendo elas:

```
julia> i = 2
2
julia> i = i + 1
3
julia> i += 1
4
```

2.6 Ortogonalidade

A ortogonalidade é a propriedade que uma linguagem possui de combinar seus conceitos sem produzir efeitos anômalos. Durante a pesquisa aos livros e documentação de Julia não encontramos falta de ortogonalidade para ser abordada.

2.7 Modificabilidade

A modificabilidade é a propriedade que permite ao programador adequar um código a um novo requisito sem fazer muitas alterações. Em Julia esse conceito se faz presente através do uso de constantes e identificadores, como por exemplo:

```
pi = 3.1415
function areaCirculo(raio)
    2*pi*raio
end

function perimetroCirculo(raio)
    pi*(raio^2)
end

function areaEsfera(raio)
    (4/3)*pi*(raio^3)
end
```

Assim, se fosse necessário alterar a precisão do número pi, bastaria alterar a linha 1 que as funções areaCirculo, perimetroCirculo e areaEsfera calculariam seus respectivos retornos utilizando o valor atualizado de pi.

2.8 Reusabilidade

A reusabilidade é desejável em qualquer linguagem de programação, pois começar um programa do zero não é desejado. Na linguagem Julia a reusabilidade pode ser conseguida através da definição de subprogramas. A linguagem Julia também possui várias bibliotecas e pacotes com métodos implementados que podem ser utilizados.

2.9 Portabilidade

Por ser uma linguagem que utiliza uma máquina virtual(LLVM), a portabilidade de Julia é muito bem favorecida. Assim, só é necessário que a máquina virtual esteja instalada para garantir a correta execução de um código.

3 Método de Implementação

A linguagem Julia foi projetada para ser uma linguagem de alta performance desde o início. A maior parte de sua biblioteca padrão foi escrita na própria linguagem. Há certa discussão na literatura sobre o método de implementaçaõ de Julia, e embora em [5] os autores definam Julia como compilada; por utilizar uma máquina virtual e um conceito similar ao de código intermediário, o mais correto é classificar a linguagem como híbrida.

Julia utiliza um compilador *Just-in-Time* (JIT). Esse é um método no qual o código fonte é traduzido para código de máquina em tempo de execução na CPU. Esse método apresenta facilidade para a linguagem possuir tipagem dinâmica, pois não é necessário o compilador ficar realizando verificação de tipos em tempo de execução.

Esse compilador *Just-in-Time* é baseado e utilizado por uma Low Level Virtual Machine (LLVM). O compilador LLVM foi desenvolvido iriginalmente na Universidade de Illinois, mas atualmente recebe contribuições de diversos pesquisadores e programadores inclusive independentes.

O fluxo da compilação de um programa em Julia funciona da seguinte forma. Em tempo de execução, Julia gera o que os autores chamam de Representação Intermediária (IR) da LLVM, o que seria análogo aos códigos intermediários de outras linguagens híbiras como JAVA, por exemplo. Essa IR é então repassada a LLVM, que por sua vez utiliza o compilador JIT para gerar código de máquina em tempo de execução.

Figura 4: Fluxo de compilação de Julia

O compilador de Julia tenta deduzir os tipos de dados utilizados em um programa e compila versões diferentes de funções com parâmetros e retornos de tipos diferentes. Por exemplo, a função **hypot(x,y)**, que calcula a hipotenusa de um triângulo dados os valores dos catetos, pode receber tanto valores do tipo inteiro como ponto flutuante. Nesse caso, Julia irá compilar duas versões do código, uma para cada tipo de parâmetro. Dessa maneira não será necessário fazer verificação de tipos em tempo de execução.

Baseando-se em algoritmos de dataflow, o compilador de Julia usa para realizar a inferência de tipos fatores de design da própria linguagem. O fato de um programador

definir o tipo da variável não fará com que haja ganho de performance, o fato de o compilador fazer essa inferência de tipos mostra que ele é capaz de descobrir o tipo de dados quando necessário,

4 Paradigma

Julia, por ser uma linguagem voltada para explorações algébricas, combina muitas características das linguagens de programação matemática. Entretanto, a mesma vai além da exploração matemática e apresenta características presentes em outras linguagens como LISP, Lua, Ruby, Python e Perl. Dessa forma, Julia se apresenta como uma linguagem multiparadigma tendo características presentes em paradigmas imperativos, funcionais e orientado a objetos, e pode ser considerada uma linguagem de programação de propósito geral.

Quanto a tipagem, Julia possui tipagem dinâmica, onde os tipos só são amarrados aos identificadores em tempo de execução.

5 Amarrações

Amarração define-se na associação entre qualquer elemento (entidade) relacionado na execução/construção de um programa.[6]

5.1 Identificadores

Identificadores, um dos elementos de amarração, são os nomes dados pelos programadores para a referência de outras entidades de computação. Em Julia é possível executar operações somente com constantes sem se fazer necessário o uso de amarrações com identificadores como, por exemplo, é mostrado abaixo.

```
julia> (36/6 + 4)%8
2.0
```

Os identificadores podem ser criados começando com os caracteres {A..Z + Ç, a..z + ç} e conter os números de 0..9 e ainda os caracteres podem ser acentuados, um exemplo disso é apresentado a seguir.

```
julia> água = 100
100
```

Além disso, Julia é *case sensitive*, ou seja, há distinção entre caracteres maiúsculo e minúsculo e como é uma linguagem dinâmica não é preciso fazer a amarração dos identificadores a um tipo, isso é feito de acordo com o tipo de dado.

5.2 Tipo de Escopo e Estrutura de Blocos

Cada amarração possui uma região de visibilidade, essa região denomina-se escopo. Julia possui escopo estático, ou seja, em tempo de compilação que as entidades são amarradas.

O escopo de uma amarração é delimitado por um bloco. A maneira como os blocos estão aninhados textualmente define o ambiente de amarração. Os blocos podem ser um subprograma ou um trecho de código delimitado pela própia definição e um *end*, como mostrado abaixo.

```
julia> function soma1(x)
           x+=1
       end
soma1 (generic function with 1 method)
```

5.3 Definições e Declarações

Definição e Declaração são frases utilizadas para criar amarrações. A diferença entre elas é que a definição produz amarrações entre entidades que estão sendo criadas

na própria definição, já a declaração produz amarrações entre entidades já existentes ou que ainda serão criadas.

5.3.1 Declarações de Constantes

Uma declaração de constante amarra um identificador a um valor pré definido. Em Julia é possível utilizar o modificador *const* para declarar uma constante, mas esse valor pode sofre alterações durante o código, neste caso um *warning* é gerado.

```
julia> const i = 10
10
julia> const i = 8
WARNING: redefining constant i
8
```

Alterações em constante só são aceitas se o novo valor for igual o tipo original, caso contrário um erro é gerado.

```
julia> const i = 10
10
julia> const i = 8.7
ERROR: invalid redefinition of constant i
8
```

5.3.2 Definições e Declarações de Tipos

As definições de tipos compostos são feitos utilizando a palavra-chave *type*.

```
julia> type Item
         chave :: Int
         valor :: Float32
         qualquer
       end
```

O tipo Item possui três variáveis uma delas é a qualquer que por definição da linguagem possui o tipo Any.

Outra forma de tipo composto é o *immutable* que não permite alteração das instâncias.

```
julia> type Documento
         CPF :: Int
         RG :: Int
       end
julia> doc = Documento(1,2)
Documento(1,2)
julia> doc.CPF
1
julia> doc.CPF = 123
ERROR: type Documento is immutable
```

Observe que foi realizada uma declaração ao criar a variável doc.

Julia também possui o tipo união. Diferente de compiladores de outras linguagem o de Julia mostra como a união é organizada.

```
julia> uniao = Union{Int, Float64}
Union{Float64, Int32}
```

5.3.3 Definições e Declarações de Variáveis

As definições são feitas somente passando o identificador e o valor da variável sem necessariamente o tipo.

```
julia> a = Int
Int32
```

Mas geralmente são feitas definições com inicialização sem passa o tipo.

```
julia> a = 8
8
```

É possível também fazer a inicialização com uso de expressão dinâmica.

```
julia> a = 3
3
julia> b = a*5
15
```

Variáveis compostas também podem ser inicializadas em sua definição.

```
julia> v = [1,2,3]
3-element Array{Int32,1}:
1
2
3
```

5.3.4 Definições e Declarações de Subprogramas

Subprogramas são compostos de cabeçalho (especificações) e corpo (algoritmo). Definição de subprograma é quando possui cabeçalho e corpo, já declaração quando possui somente cabeçalho. A seguir é apresentado um exemplo de definição:

```
julia> function potencia(a,b)
           a^b
       end
potencia (generic function with 1 method)
```

5.3.5 Composição de Definições

As definições podem ser compostas utilizando elas mesmas (Definição Recursiva) ou outras definições (Definição Sequencial).

Definição Sequencial

Quando uma definição utiliza outra definição já criada. Pode ser feito utilizando variáveis, como a seguir:

```
julia> g = 5
5
julia> k = g
5
```

Outro exemplo é a composição de tipos:

```
julia> type Item
         chave :: Int
         valor :: Float32
         qualquer
       end
julia> type Dados
         item :: Item
         n :: Int
       end
```

Definição Recursiva

Quando uma definição utiliza a própria amarração que produz.

```
julia> function fibonacci(n::Int32)::Int32
         if n < 2
           return n
         else
           return fibonacci(n-1) + fibonacci(n-2)
         end
       end
fibonacci (generic function with 1 method)
```

No caso, no corpo da função fibonacci são feitas duas chamadas para a própria função fibonacci. Essa definição recursiva também pode ser utilizada com tipos compostos.

6 Tipos Primitivos

Tipos primitivos são aqueles que não podem ser subdividos em tipos mais simples e são definidos no momento em que a linguagem é projetada. Em Julia, os tipos de dados são dinâmicos, ou seja, os mesmos são amarrados aos identificadores em tempo de execução. No entanto, Julia também permite explicitar que um identificador é de um tipo em tempo de compilação, resultando em um código mais eficiente. Esse fato é mostrado abaixo, onde os identificadores x e y são amarrados ao tipo inteiro:

```
julia> soma(x::Int64, y::Int64) = x+y
soma (generic function with 1 method)
```

6.1 Tipo Inteiro

O tipo inteiro em Julia depende da arquitetura, podendo ter uma representação padrão de 32 ou 64 bits. Logicamente a quantidade de bits utilizados para representação possibilita um *range* maior de valores. Julia, no entanto, apresenta subtipos do tipo inteiro que podem ser usados para aumentar ou reduzir esse *range*, sendo eles:

- Int8: inteiros de 8 bits com sinal variando de -2^7 a $2^7 - 1$
- UInt8: inteiros de 8 bits sem sinal variando de 0 a $2^8 - 1$
- Int16: inteiros de 16 bits com sinal variando de -2^{15} a $2^{15} - 1$
- UInt16: inteiros de 16 bits sem sinal variando de 0 a $2^{16} - 1$
- Int32: inteiros de 32 bits com sinal variando de -2^{31} a $2^{31} - 1$
- UInt32: inteiros de 32 bits sem sinal variando de 0 a $2^{32} - 1$
- Int64: inteiros de 64 bits com sinal variando de -2^{63} a $2^{63} - 1$
- UInt64: inteiros de 64 bits sem sinal variando de 0 a $2^{64} - 1$
- Int128: inteiros de 128 bits com sinal variando de -2^{127} a $2^{127} - 1$
- UInt64: inteiros de 64 bits sem sinal variando de 0 a $2^{128} - 1$

As funções *typemax()* e *typemin()* podem ser usadas para verificar os valores aceitos por um subtipo:

```
julia> typemax(Int64)
9223372036854775807

julia> typemin(Int64)
-9223372036854775808
```

6.2 Tipo Ponto flutuante

O tipo ponto flutuante em Julia segue o padrão IEEE754 , podendo ser:

- Float32: Números de ponto flutuante de 32 bits (1bit de sinal; 8 bits de expoente; 23 bits de representação)
- Float64: Números de ponto flutuante de 64 bits (1bit de sinal; 11 bits de expoente; 52 bits de representação)
- Inf: Infinito Positivo. Um valor maior do que todos os valores de ponto flutuante.
- -Inf: Infinito Negativo. Um valor menor do que todos os valores de ponto flutuante.
- NaN: Não é número. Valor utilizado quando não se tem um resultado.

Os três últimos valores não são reais, sendo utilizados apenas para resultados de operações que causariam algum problema, como por exemplo:

- 1/0 = Inf
- 0/0 = NaN
- -1/0 = -Inf

6.3 Tipo Caractere

O tipo caractere em Julia é representado por 32 bits que correspondem a algum caractere na tabela *Unicode*. Abaixo é mostrado a representação do caractere *a*.

```
julia> 'a'
'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)

julia> bits('a')
"00000000000000000000000001100001"
```

6.4 Tipo Booleano

O tipo booleano em Julia é representado por 8 bits. Embora para representar um valor booleano seje necessário apenas 1 bit, a linguagem Julia só permite a representação de uma quantidade multipla de 8 bits. Dessa forma apenas o bit menos significativo é usado. A função bits é usado abaixo para mostrar a sequência de bits usada na representação.

```
julia> bits(true)
"00000001"

julia> bits(false)
"00000000"
```

6.5 Tipos Abstratos X Tipos Concretos

Em Julia são adotados os conceitos de tipos abstratos (AbstractType) e tipos concretos (ConcreteTypes), sendo que aqueles não podem ser instanciados. Assim os tipos abstratos são usados apenas para diferenciar possíveis tipos concretos que seriam idênticos sem a presença dos mesmos. Dessa forma se tem uma hierarquia (Árvore) de tipos, onde os tipos concretos são subtipos dos tipos abstratos e estão nas folhas da árvore. A árvore de tipos em Julia tem como raiz o tipo Any, asssim qualquer objeto que não tenha tipo é do tipo Any. Para saber se um tipo é subtipo de outro pode ser usado o operador <: que retorna true se verdadeiro ou false caso contrário.

```
julia> bits(true)
"00000001"

julia> bits(Int8(1))
"00000001"

julia> Int8 <: Signed
true

julia> Bool <: Signed
false
```

No exemplo acima é mostrado a importância dos tipos abstratos, pois como pode ser visto tanto 1 como true possuem a mesma representação em bits, sendo diferenciados apenas pelos seus respectivos supertipos.

7 Tipos Compostos

7.1 Produtos Cartesianos

Em Julia, produtos cartesianos são utilizados criando um novo tipo que contém tipos primitivos ou concretos. Como por exemplo:

```
julia> type Ponto
       x :: Int
       y :: Int
       end
```

No exemplo, a cardinalidade do tipo Ponto será $(\#Intx\#Int) = (\#Int)^2$.

7.2 Uniões

O tipo composto união consiste na junção de tipos distintos para formar um novo. Em Julia para definir uma união é utilizada a palavra-chava *Union*.

```
julia> distancia = Union{Int64, Float64}
Union{Float64, Int64}
```

A cardinalidade da união distancia é #Int64 + #Float64 = (#Int64 + #Float64).

7.3 Mapeamentos

Mapeamentos são os conjuntos de valores mapeados de um tipo a outro.

7.3.1 Vetores Dinâmicos

Vetores são mapeamentos do tipo inteiro para qualquer outro tipo. A seguir é apresentado um exemplo de vetor dinâmico com mapeamento [Inteiro → Inteiro]:

```
julia> vetor = [1,2,3,4,5]
5-element Array{Int32, 1}:
1
2
3
4
5
```

7.3.2 Vetores Multidimensionais

Um exemplo de vetores multidimensionais é apresentado a seguir. Esse mapeamento é do tipo [{1,2,3,4,5} X {1} → Inteiro] U [{1,2,3,4,5} X {2} → Float] U [{1,2,3,4,5} X {3} → Caractere].

```
julia> matriz = [(1,3.5,'j'),(1,5.8,'u'),(0,5.4,'l'),
(4,9.0,'i'),(5,2.4,'a')]
6-element Array{Tuple{Int64,Float64,Char}
1  3.5  'j'
1  5.8  'u'
0  5.4  'l'
4  9.0  'i'
5  2.4  'a'
```

7.3.3 Dicionário

Dicionários são mapeamento de um tipo qualquer para outro tipo qualquer. Abaixo são mostrados mapeamentos [String → String] e [Inteiro → String]

```
julia> d1 = Dict( "key1" => "Julia", "key2" => "Linguagem")
Dict{String,String} with 2 entries:
  "key2" => "Linguagem"
  "key1" => "Julia"

julia> d1["key1"]
"Julia"

julia> d2 = Dict( 1 => "valor1", 2 => "valor2")
Dict{Int64,String} with 2 entries:
  2 => "valor2"
  1 => "valor1"

julia> d2[2]
"valor2"
```

7.4 Conjuntos Potência

Considerando um tipo base, conjunto potência seria todas as combinações dos subconjuntos possíveis. Julia não apresenta suporte para conjunto potência.

7.5 Tipos Recursivos

Tipo recursivo é aquele composto por um valor do mesmo tipo. Ou seja, um tipo que usa para se definir sua própria definição, como uma espécie de função recursiva.

Durante as pesquisas não foram encontrados detalhes específicos ou informações detalhadas e confiáveis de que Julia dá suporte a esse tipo.

8 Ponteiros e Referências

Julia trata a alocação dinâmica de forma automâtica, como a maioria das linguages orientadas a objetos. Assim, a necessidade do uso de ponteiros explicitamente acontece, na maioria das vezes, quando há uma integração com uma linguagem estruturada, como, por exemplo, C.

Para utilizar ponteiros é necessário usar a palavra-chave *Ptr*.

```
julia> ptr = Ptr{Int32}
Ptr{Int32}
```

Uma forma de se utilizar ponteiros é na manipulação de arquivos. Para isso, é necessário usar o tipo IOStream. IOStream é um tipo do supertipo IO. O identificador de arquivo é um ponteiro do tipo *Ptr*, que é uma referência ao objeto de arquivo. Porém, não é necessário colocar *Ptr* na declaração como mostrado na seção 11.1.

Ponteiros podem ser utilizados também para acessar objetos Julia no espaço de memória. Isso é possível utilizando a estrutura *jl_value_t* pertecente ao *Julia Garbage Collector*.

```
typedef struct jl_value_t* jl_pvalue_t;
```

Na verdade, no livro *Julia: High Performance Programming*[5] percebe-se que o uso de ponteiros em Julia está fortemente ligado na sua conexão com outras linguagens.

Esta linguagem não possui o tipo referência como C++, por exemplo.

9 Tratamento de Strings

Julia define o tipo de dados String e adota tanto a tabela ASCII como a tabela Unicode, utilizando a codificação UTF-8. Os criadores da linguagem optaram por usar ambas as tabelas devido à rapidez de se manipular caracteres ASCII e adota também a tabela Unicode dada a limitação de outras tabelas, inclusive a própria ASCII. Os criadores pretendiam que a linguagem Julia desse suporte a inúmeros tipos de caracteres, não só os de línguas derivadas do latim. Este tipo de dados trata uma string como sendo um vetor de caracteres, sendo possível indexar uma string e obter o caractere correspondente ao índice.

Todas as strings em Julia são subtipos do tipo abstrato AbstractString, ou seja, quaisquer métodos definidos pelo programador que desejam receber strings como argumento devem declarar o método com o parâmetro AbstractString. Assim como JAVA, o valor de uma AbstractString não pode ser mudado.

9.1 Strings Básicas

As strings básicas em Julia podem ser criadas por aspas duplas ()- assim como várias linguagens de programação consolidadas- ou aspas dupas triplas ("..."). Esse último tipo de criação já inicia a String com uma quebra de linha e traz também a vantagem de que o nível de identação da String pode ser definido com base no fechamento (").

```
julia> string = """
                Nova
                estrela
                no horizonte.
       """
"         Nova\n         estrela\n         no horizonte.\n"
```

Como dito anteriormente, os caracteres podem ser indexados em uma string. Em Julia os índices de vetores começam de 1, mas isso não implica que o último caractere da string está necessáriamente no índice correspondente ao tamanho da string. Para acessar o último caractere utiliza-se o índice especial end.

Julia permite a criação de Strings usando a representação de caracteres Unicode através do uso dos identificadores **u** ou **U**.

```
julia> string = "\u004a u \u006C i \u0061"
"J u l i a"
```

Contudo é importante ressaltar que a codificação UTF-8 é uma codificação de tamanho variável, ou seja, nem todos os caracteres possuem o mesmo número de bytes (outras codificações podem ser usadas adicionando pacotes externos). Isso pode levar a

erros de indexação em strings desse tipo, dado que não necessariamente cada caractere vai ocupar apenas uma posição do vetor.

```
julia> formula_delta = "\u0394 = b\u00B2+4ac"
"Δ = b²+4ac"
```

Nesse exemplo, indexando a posição 1 da string, obtém-se o caractere Δ, porém indexar a posição dois irá gerar um erro porque o caractere Δ ocupa mais de um byte (no caso 2 bytes), fazendo necessário indexar a posição 3 da string para obter o próximo caractere.

```
julia> formula_delta[1]
'Δ': Unicode U+0394 (category Lu: Letter, uppercase)

julia> formula_delta[2]
ERROR: UnicodeError: invalid character index
Stacktrace:
 [1] slow_utf8_next(::Ptr{UInt8}, ::UInt8, ::Int64, ::Int64) at ./str
 [2] next at ./strings/string.jl:204 [inlined]
 [3] getindex(::String, ::Int64) at ./strings/basic.jl:32

julia> formula_delta[3]
' ': ASCII/Unicode U+0020 (category Zs: Separator, space)
```

Obviamente, o tamanho da string dado por .length() não é sempre correspondente ao último índice.

9.2 Operações com Strings

Dentre as operações que são feitas com strings, umas das mais comuns e utilizadas são a concatenação e a interpolação. Em Julia essas duas operações são bem definidas.

Há três maneiras de se realizar uma concatenação de strings em Julia. A primeira é utilizando o método string(); onde uma string pode ser instanciada passando como parâmetro várias substrings que serão concatenadas formando uma só.

```
julia> string_aleatoria1 = "A vida"
"A vida"

julia> string_aleatoria2 = "campo"
"campo"

julia> string(string_aleatoria1, " no ", string_aleatoria2)
"A vida no campo"
```

A segunda maneira é dada com o operador *. Os autores da linguagem revelam que apesar da contrariedade com a maioria das outras linguagens de programação e ao fato de que normalmente o símbolo + é usado para concatenação, a escolha do operador *

vem da álgebra abstrata. Na matemática o símbolo + denota operações comutativas, ao contrário de *, que geralmente denota operações não comutativas, como é o caso de concatenação de strings.

```
julia> string_aleatoria = "A vida"
"A vida "

julia> string_aleatoria * "no campo"
"A vida no campo"
```

Por fim é possível fazer concatenação através da função join(), que recebe um vetor de strings e o transforma em uma string só.

Entretanto, a contrução de strings através de concatenação pode se tornar um pouco complexa. Pensando nisso, Julia também permite interpolação em strings básicas utilizando o operador **$**, (que também é utilizado em Perl).

```
julia> produto = "45*2= $(45*2)"
"45*2= 90"
```

Utilizando o operador **$**, qualquer objeto - incluindo tipos compostos- é convertido automaticamente em uma String; o compilador imediatamente realiza uma chamada ao método string()

Julia também traz outras operações e funções que facilitam o uso e manipulação de strings. Abaixo segue uma pequena lista:

- search(string, caractere) - procura um caractere em uma string. Pode também receber como parâmetro um índice para começar a procura a partir de tal. Muito útil em casos do caractere se repetir na string.
- contains(string, substring) - verifica se uma string contém uma substring.
- repeat(string, x) - repete a string um número x de vezes.
- endof(string) - retorna o índice máximo (em byte) que pode ser indexado numa string.
- lenght(string) - retorna a quantidade de caracteres numa string.
- start(string) - retorna o primeiro índice válido de uma string.
- next(string, i) - retorna o próximo caractere na string a partir do índice i.

Também é possível fazer comparações quanto à ordem lexicográfica de maneira bem simples. Comparar duas strings usando os operadores "==", ">", "<" ou "!=" são comandos válidos em Julia e retornam valores booleanos correspondentes

ao resultado da operação. As comparações são feitas utilizando o valor inteiro dos caracteres correspondentes de acordo com a tabela Unicode .

```
julia> "federal" > "universidade"
false

julia> "federal" < "universidade"
true

julia> "federal" == "universidade"
false

julia> "federal" != "universidade"
true
```

9.3 Outros tipos de Strings

Julia fornece também as chamadas "Strings Fora do Padrão". Esse tipo de string é preceida por um caractere especial e se comporta de forma diferente das strings básicas. Os tipos de strings fora do padrão apresentados na linguagem são as expressões regulares, vetores de bytes, números de versão e strings "brutas".

As expressões regulares em Julia funcionam de maneira similar as da linguagem Perl. Podem ser utilizadas tanto para encontrar padrões em strings como entrada para máquinas de estados finitos. São precedidas pelo caractere **r**. A partir da criação de uma expressão regular, existem funções definidas que realizam operações usuais com esse tipo de dado, o que deixa a sua utilização muito mais prática e simples. A mais significativo delas é a função match().

A função match() recebe como parâmetro uma expressão regular e uma string e verifica se aquela string segue o padrão da expressão, ou seja, verifica se a string faz parte da linguagem denotada por aquela expressão regular. Essa função retorna um objeto do tipo RegexMatch. Esse tipo de objeto traz facilidades para o programador pois a partir dele é trivial recuperar informações relevantes como a string inteira que corresponde ao padrão (.match, o índice onde começa a correspondência (.offset, é possível também colocar toda as substrings que correspondem ao padrão em um vetor através da função .offsets. Caso não houver correspondência, essa função retorna nothing, um valor que não escreve nada no terminal.

```
julia> resultado = match(r"(ele|triun)(fante)", "elefante")
RegexMatch("elefante", 1="ele", 2="fante")

julia> resultado.match
"elefante"

julia> resultado.offset
1

julia> resultado.offsets
2-element Array{Int64,1}:
 1
 4
```

Os vetores de bytes são precedidos pelo caractere **b**. Esse tipo de string é útil para representar vetores de bytes na forma de uma string, sendo que os caracteres ASCII correspondem a um byte; os Unicode correspondem às sequências de bytes correspondentes a sua codificação na tabela (que como já dito podem ser mais de um byte); e \x e sequências de oito espaços equivalem ao byte de espaço. Mesmo que mesmo tendo a representação de uma string, as operações de uma string básica não são válidas para esse tipo.

```
julia> b"Ano=\u0032\u0030\u0031\u0037"
8-element Array{UInt8,1}:
 0x41
 0x6e
 0x6f
 0x3d
 0x32
 0x30
 0x31
 0x37
```

Em Julia também é possível expressar em forma de string e trabalhar facilmente com números de versão utilizando o objeto VersionNumber, para isso deve se preceder a string com o caractere **v**. Esse tipo de objeto segue as espicificações de padrão de versionamento e, portanto, são compostos de um número principal, números menores e patch, seguidos de pré-lançamento e compilação de anotações alfa-numéricas. É possível fazer, por exemplo, comparação de números de versão, inclusive comparando com a própria versão da linguagem utilizada. Esse dado fica armazenado na constante VERSION.

```
julia> v"0.2" < VERSION
true
```

Por fim as strings "brutas"são criadas precedendo uma string com **raw**. Isto irá criar objetos String que contêm o conteúdo incluído exatamente como entrou sem interpolação ou desestruturação. Esse recurso pode ser útil para strings que contém caracteres que são considerados especiais para linguagens de programação em geral, como \ ou $.

10 Gerenciamento de Memória para Variáveis e Constantes

A linguagem Julia faz uso de diversos tipos de alocação de memoria. O principal artifício da linguagem nesse quesito é o coletor de lixo que aloca os objetos no heap os objetos que não estão no pool (alocação fixa de memória) ou, para objetos maiores, através de chamadas de funções como malloc ou mmap. Todo objeto em Julia possui um endereço de memória, e a maioria desses objetos podem ter seus endereços acessados pelo programador. Todo objeto o qual é permitido ter acesso ao endereço é alocado no heap . Também são alocados no heap as variáveis globais definidas e objetos amarrados à essas variáveis globais.

Porém existem objetos os quais não é permitido ao programador acessar esses endereço de todos os tipos de objetos. Objetos do tipo **isbtis** não permitem que o endereço seja acessado por questões de segurança nas quais os criadores da linguagem não entraram em muitos detalhes. Um objeto do tipo isbits é um objeto imutável e não contem referências para outros valores, ou seja, não necessitam de definições de outros tipos para o objeto ser manipulado seja na execução ou na memória. Exemplos desse tipo de objetos são tipos como os tipos primitivos numéricos Float64. A função booleana isbits() indica se um objeto é desse tipo ou não.

Os demais objetos podem ter seus endereços acessados por meio de ponteiros através do comando pointer_from_objref(object). É possível também fazer o caminho contrário, dado um ponteiro obter o valor por ele apontado. O comando para tal unsafe_pointer_to_objref(ptr). Esse último método porém não é recomendável pois Julia aloca os objetos no heap automaticamente.

O compilador então fica responsável por fazer as demais otimizações e alocações de memória, podendo usar também fazer alocações na pilha caso necessário. O compilador também é capaz de remover da memória um objeto que não está sendo usado.

Os arrays na linguagem Julia são alocados no heap, porém há diferença principalmente em desempenho de acesso entre vetores de tipos concretos e vetores de tipos abstratos. Em vetores de tipos concretos, o compilador pode alocar memória de forma a acessar sequencialmente os valores do vetor. Já em vetores de tipos abstratos isso não ocorre, devido ao fato de que os objetos no vetor podem ter tamanhos variados; sendo assim quando o array é criado são armazenados na memória apenas ponteiros para os valores do vetor. Os valores são alocados de forma automática e aleatória no heap. Isso provoca uma carga extra de memória e um tempo maior para ler os valores, o que pode prejudicar eventuais pipelines e compatibilidade com cache.

11 Armazenamento de Variáveis e Constantes em Memória Secundária

A persistência dos dados é de extrema importância, até o momento nossos exemplos foram com variáveis que só existem em tempo de execução. Julia apresenta as duas formas comuns de se fazer persistência: através de arquivos e de conexão com banco de dados.

11.1 Persistência com Arquivos

A linguagem permite a persistência em arquivos, um exemplo disso é mostrado abaixo:

```
julia> match = [("Nome","Com"), ("Amaral", "IV"),
("Chaubert","Pink Floyd"),
("Josué","Eduarda"),("Leonardo","Luciana")]
5-element Array{Tuple{String,String},1}:
 ("Nome", "Com")
 ("Amaral", "IV")
 ("Chaubert" , "Pink Floyd")
 ("Josué", "Eduarda")
 ("Leonardo", "Luciana")

julia> a = writecsv("/home/jw/Área de Trabalho/match.csv", match)

julia> r = readcsv("/home/jw/Área de Trabalho/match.csv")
5x2 Array{Any,2}:
 "Nome"       "Com"
 "Amaral"     "IV"
 "Chaubert"   "Pink Floyd"
 "Josué"      "Eduarda"
 "Leonardo"   "Luciana"
```

O exemplo acima foi baseado no site *Julia By Example*[7]. No caso, uma matriz foi criada com duas posições e passada para um arquivo csv com o uso da função *writecsv()* e e a leitura com o uso da função *readcsv()*.

11.2 Persistência com Banco de Dados

Julia possui um pacote ODBC(*Open Database Connectivity*) que permite conexão com banco de dados. Para isso, é indispensável a instalação do pacote ODBC através do comando:

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

Assim, a conexão através do DSN (*Data Source Name*) fica da seguinte forma:

```
julia> conexao = ODBC.connect("exemploODBC")
```

Dessa forma, um comando SQL seria simples, como por exemplo:

```
julia> results = query("select * from tabela")
```

11.3 Persistência Ortogonal

A persistência ortogonal permite que variáveis persistentes sejam tratadas como transiente sem nenhum tipo de distinção. Julia não se qualifica como ortogonalmente persistente.

12 Expressões

Como a grande maioria das linguagens de programação, Julia dá suporte a diversos tipos de expressões, simples e compostas; várias pré-existentes na linguagem e oferecendo a possibilidade do programador criar novas utilizando o conceito de funções; e operadores de diversas aridades. Os operadores em Julia podem ser tanto infixados, como por exemplos os operadores aritméticos, e prefixados, como o operador de negação (!). Julia não contém operadores posfixados.

```
julia> a=5
5

julia> b=3
3

julia> a+b
8
```

```
julia> c=true
true

julia> !c
false
```

Detalharemos abaixo os tipos de expressão fornecidos pela linguagem Julia.

12.1 Expressões Literais

As expressões literais são o tipo mais simples que pode ser encontrado numa linguagem e produzem valores que não são mudados. Julia dá suporte a esse tipo de expressão, podendo utilizar notação natural para os valores ao de alguma codificação, como por exemplo a da tabela Unicode.

```
julia> 14
14

julia> 'J'
'J': ASCII/Unicode U+004a (category Lu: Letter, uppercase)

julia> '\u004a'
'J': ASCII/Unicode U+004a (category Lu: Letter, uppercase)
```

12.2 Expressões de Agregação

Em Julia existem expressões de agregação e, semelhante a outras linguagens como C, são frequentemente usadas para atribuir valores para valores compostos. Mas diferente da linguagem C, em Julia essas expressões podem ser usadas tanto para inicializar quanto para atualizar esses valores. Para separar as subexpressões é utilizado o caractere ','. No caso de inicialização de tipos cartesianos, as subexpressões ficam entre

'()' - o que faz mais sentido devido à caracteristica de paradigma orienteado a objetos de Julia. Nesse caso a inicialização funciona como uma espécie de construtor. Já no caso de outros tipos como vetores, as subexpressões ficam entre os caracteres '[]'. As agregações podem ser tanto estáticas quanto dinâmicas.

```
julia> struct data
            dia :: Int
            mes :: Int
            ano :: Int
       end

julia> d = data(14,2,1996)
data(14, 2, 1996)

julia> vetor = [10,20,30,40,50]
 5-element Array{Int32, 1}:
 10
 20
 30
 40
 50

julia>i=2
2

julia> vetor2=[2*i,5,3*i]
3-element Array{Int64,1}:
 4
 5
 6
```

12.3 Expresões Aritméticas

Por se tratar de uma linguagem desenvolvida para programação científica e numérica, é esperado que Julia ofereça um grande número de expressões aritméticas. E de fato isso é verdade. Os operadores das expressões aritméticas são em sua grande maioria infixados e possuem aridade 2. Na tabela abaixo estão listadas a principais expressões aritméticas oferecidos pela linguagem.

Tabela 1: Algumas expresões aritméticas fornecidas por Julia

Expressão	Descrição
+x	Valor literal
-x	Valor inverso
x + y	Soma
x - y	Subtração
x * y	Multiplicação
x / y	Divisão
x \ y	Divisão inversa (equivalente a y/x)
x ^ y	Exponenciação
x % y	Resto da divisão inteira

```
julia> 10/5
2.0

julia> 10\5
0.5
```

12.4 Expressões Relacionais

Expressões Relacionais são aquelas que comparam valores de mesmo tipo. Em Julia essas expressões retornam valor boolano e são definidas para todos os tipos primitivos da linguagem. Os operadores que compõem essas expressões são infizados e possuem aridade 2.

Tabela 2: Operadores relacionais fornecidos por Julia

Operadores	Descrição
==	Igualdade
!=, ≠	Desigualdade
<	Menor
<=, ≤,	Menor ou igual
>	Maior
>=, ≥,	Maior ou igual

Em Julia existe ainda o operador '===' para comparar objetos. Primeiramente ele compara se os tipos dos objetos são iguais e em seguida compara se os endereços em memória também são os mesmos. Ou seja, essa expressão só retorna verdadeiro quando um objeto é comparado com ele mesmo.

Alguns exemplos de uso de operadores relacionais:

```
julia> 1==2
false

julia> 'a'=='a'
true

julia> "capitalismo"!="comunismo"
true

julia> 30<40
true

julia> 12>15
false

julia> 25>=25
true
```

12.5 Expressões Booleanas

As oexpressões booleanas realizam as operações definidas na álgebra de George Boole (1815 — 1864). São elas negação, disjunção e conjunção. Julia oferece todos esses tipos de expressões através dos operadores listados na tabela abaixo.

Tabela 3: Operadores booleanos fornecidos por Julia

Operadores	Descrição
!x	Negação
||	Disjunção Lógica
&&	Conjunção Lógica

```
julia> v=true
true

julia> f=false
false

julia> !v
false

julia> !f
true

julia> v || f
true
```

```
julia> v && f
false
```

Os operadores de conjunção e disjunção lógica utilizam curto circuito em Julia. Isto será discutido com mais detalhes posteriormente.

12.6 Expressões Binárias

Em Julia as expressões binárias são definidas para os tipos primitivos inteiros e manipulam bits, dessa maneira economixando memória e sendo mais eficiente por manipular espaços menores. A tabela abaixo lista os operadores binários *bitwise* (Bit-a-Bit) fornecidos pela linguagem Julia.

Tabela 4: Operadores binários fornecidos por Julia

Operadores	Descrição
x	Negação
x | y	Or Bit-a-Bit
x & y	And Bit-a-Bit
x $\veebar$ y	Ou exclusivo (xor)
x >>> y	Deslocamento lógico a direita
x >> y	Deslocamento aritmético a direita
x << y	Deslocamento aritmético/lógico a esquerda

```
julia> 2 << 2
8

julia> 2 >> 3
0

julia> 256 >> 2
64

julia> 1 & 1
1

julia> 1 & 0
0

julia> 0 | 1
1

julia> 0 | 0
0
```

Para cada operador binário, existe na linguagem Julia um operador correspondente ".operador". Esse tipo é definido para realizar a operação indicada com cada elemente de um vetor.

```
julia> vetor = [true,false,true,false] .& true
4-element BitArray{1}:
true
false
true
false
```

12.7 Expressões Condicionais

A linguagem Julia permite o uso de expressões condicionais com a seguinte sintaxe:

```
x ? y : z
```

onde x é uma subexpressão codicional. O operador ternário '?' avalia essa subexpressão e caso seja verdadeira é executado y, caso contrário é executado z. Esse tipo de expressão é equivalente a um comando condicional com seleção de caminho duplo.

```
julia> 22>11? print("verdadeiro") : print("falso")
verdadeiro

julia> 22<11? print("verdadeiro") : print("falso")
falso
```

12.8 Chamadas de Função

Em Julia as funções são consideradas objetos que manipulam valores para retornar um outro. Sua definição segue as regras das maiorias das linguagens de programação imperativas, possuindo cabeçalho e corpo. Como Julia é uma linguagem de tipagem dinâmica, não é necessário informar o tipo do parâmetro na definição/declaração da função, embora isso seja possível e benéfico para entendimento do código.

```
julia> function funcao(x,y)
           print(x,y)
       end
funcao (generic function with 1 method)

julia> function funcao(x::Int,y::Int)
           print(x,y)
       end
funcao (generic function with 1 method)
```

Mesmo com foco em computação numérica, as funções em Julia não são somente matemáticas, podendo o programador em si definir as próprias funções com seu próprio número de parâmetro. Uma característica interessante e que pode ser de grande utilidade para os programadores é que pode ser retornado mais de um valor numa mesma função.

```
julia> function subtrai_e_divide(a,b)
           a-b, a/b
       end
subtrai_e_divide (generic function with 1 method)
```

Existe também a forma de "definição de atribuição"de funções em Julia. Nessa forma a função é definida como um objeto e é incializada com o corpo da função. As palavras chave function e end são omitidas, aumentando a redigibilidade.

Em Julia as chamadas de função são posicionais.

```
julia> funcao(x,y) = print(x,y)
funcao (generic function with 1 method)
```

Por fim, como Julia trata funções como objetos, é possível fazer cópias de funções através de atribuições simples.

```
julia> funcao2 = funcao
funcao (generic function with 1 method)

julia> funcao2(2,5)
2 5
```

12.9 Expressões com Efeito Colateral

Algumas expressões em Julia podem ter efeitos colaterais, embora não seja comum o efeito de atualização de valores em variáveis. Uma expressão aritmética por exemplo só altera o valor do resultado e não dos operandos envolvidos.

Entretanto, toda chamada de função em Julia possui um efeito colateral. Para cada chamada de função o programa em Julia faz uma aproximação do número de campos (tipos simples são considerados como um campo e nessa aproximação entram campos de tipos compostos) que foram alterados durante a execução da função e dos métodos que forem sendo chamados caso haja recursão.

12.10 Expressões de Referenciamento

Como dito anteriormente, Julia não possui o tipo referência como algumas linguagens, principalmente orientadas a objeto, possuem. Mas é possível acessar o endereço

de um objeto alocado na memória (desde que não seja do tipo isbits()) através do comando pointer_from_objref(object), que retorna um ponteiro para o endereço de memória do objeto.

Semelhante a linguagem C, o operador [] serve para acessar um elemento de vetor, mas não retorna referência.

12.11 Expressões Categóricas

Expressões categóricas são usadas para obter informações sobre os tipos de dados. Em Julia isto pode se mostrar interessante para o programador principalmente por ser um linguagem de tipagem dinâmica.

O tipo de um objeto/variável em Julia pode ser obtido através da expressão typeof().

```
julia> x = 2.3
2.3

julia> typeof(x)
Float64

julia> typeof("frase aleatoria")
String
```

Expressões categóricas também podem ser usadas para realizar conversões de tipos. Em Julia isso pode ser feito usando convert(x,T). Essa função irá converter o objeto x para um valor do tipo T. Caso x não possa ser representado no tipo T , será reportado um erro do tipo InexactError. Tal situação pode acontecer, por exemplo, se x está fora dos limites suportados por T.

```
julia> convert(Int, 40.5)
ERROR: InexactError()
Stacktrace:
 [1] convert(::Type{Int64}, ::Float64) at ./float.jl:679

julia> convert(Float64, 7)
7.0
```

No exemplo acima, não foi possível converter o valor 40,5 para inteiro, pois a parte decimal não pôde ser convertida, está fora dos limites do tipo Int.

Por fim as expressões categóricas podem ser usadas para obter informações sobre o tamanho das variáveis. Em Julia existem três maneiras de fazer isso. A primeira é utilizando a função whos(). Essa função exibe informação do tamanho de todos os objetos alocados na memória dentro do módulo na qual for chamada. Esse cálculo é feito com base no tamanho das estruturas internas de cada objeto.

Para informações de objetos individuais podem ser usadas as funções Base.summarysize() e sizeof(). A primeira calcula a quantidade total de memória ocupada pelos objetos passados no argumento (essa função aceita mais de um objeto como parâmetro). A segunda funciona para objetos de forma individual e retorna a quantidade em bytes da representação binária do tipo de dado em memória.

```
julia> type aluno
           matricula :: Int
           nome :: String
       end

julia> Wesley = aluno(2267,"Wesley")
aluno(2267, "Wesley")

julia> Base.summarysize(Wesley)
30

julia> sizeof(Wesley)
16
```

12.12 Expressões Compostas

A linguagem Julia oferece ao programador o recurso de criar suas prórprias expressões compostas. Este tipo de expressão é formado de várias subexpressões e retorna o valor da última a ser avaliada. Podem ser criadas de duas formas: utilizando blocos com BEGIN ou separando as subexpressões com o caractere ';'.

```
julia> raiz = begin
           j = 64
           w = 36
           sqrt(j+w)
       end
10.0

julia> raiz = (j = 64; w = 36; sqrt(j+w))
10.0
```

No contexto de expressões compostas, é importante avaliarmos a precedência de operandos e operadores. Em se tratando de operandos, Julia, assim como C, não tem ordem definida, deixando esta tarefa a cargo do compilador para que este faças da forma mais otimizada. Já pra operadores, Julia assume a seguinte ordem de precedência; da linha mais acima para a mais abaixo são a ordem de precedência dos operadores da mais alta para a mais baixa:

Tabela 5: Ordem de precedência de operadores em Julia

Categoria	Operador
Sintaxe	. seguido de ::
Exponenciação	^
Divisões	//
Multiplicações	*, /, %, &, \
Deslocamento de bits	<<, >>, >>>
Adição	+, -, |, ⊻
Sintaxe	: .. seguido de | >
Comparações	>, <, >=, <=, ==, ===, ! =, ! ==, <:
Fluxo de controle	&& seguido de || seguido de ?
Atribuições	=, +=, -=, *=, /=, //=, \=, %=

Em relação a associatividade, a linguagem Julia assume a postura de determinar que os operadores sejam avaliados da esquerda pra direita.

Como a maioria das linguagens imperativas, os operadores && e || utilizam curto circuito em Julia. Dessa forma, dado uma quantidade de expressões booleanas interligadas, o mínimo de expressões possível é avaliado para determinar o valor final.

```
julia> x=true
true

julia> y=false
false

julia> x && y
false

julia> y && x
false
```

No exemplo acima, a subexpressão y só foi avaliada porque x é verdadeira (visto que de acordo com a álgebra booleana esta é a única possibilidade de não ser verdadeira). Análogamente, em x&&y x não foi avaliada porque y é falsa.

Muitas vezes curto circuitos em Julia são usados para substituir comandos if.

```
julia> function raiz_quadrada(num)
           num>=0 || print("impossivel calcular raiz de numero negati
           num ==0 &&  return 0
           sqrt(num)
       end
raiz_quadrada (generic function with 1 method)
```

```
julia> raiz_quadrada(64)
8.0

julia> raiz_quadrada(0)
0

julia> raiz_quadrada(-1)
impossivel calcular raiz de numero negativo
```

É possível implementar estas expressões sem utilizar curto circuito usando os operadores equivalentes Bit-a-Bit. Esses operadores só estão definidos para valores booleanos, mas graças a utilização de curto circuito é possível que a subexpressão final seja uma não-booleana.

```
julia> true && 10+90
100

julia> false && 10+90
false
```

13 Comandos

Como definido em [6], comandos são frases de código características das linguagens do paradigma imperativo que têm por objetivo atualizar valores de varáveis e controlar o fluxo de execução do problema. Assim como as expressões, comandos podem ser simples ou compostos. Nas seções abaixo descreveremos em mais detalhes quais os tipos de comandos são fornecidos pela linguagem Julia.

13.1 Atribuições

Como esperado em uma linguagem que possui várias características do paradigma imperativo, o comando de atrbuição é fornecido por Julia. O comando de atribuição simples é executado utilizando o caractere "="e através dele é possível amarrar valores aos identificadores das variáveis/objetos, atualizando assim os valores das variáveis. Este operador funciona para todos os tipos primitivos definidos pela linguagem, e como dito na seção de expressões de agregação, também pode ser inicializado para tipos compostos.

```
julia> a = 5
5

julia> v = true
true

julia> cumprimento = "bom dia"
"bom dia"
```

Além da atribuição simples, em Julia é existem atribuições múltipla, composta e por expressão. No exemplo abaixo, x recebe como valor o retorno da função typeof(), no caso o tipo da variável a.

```
julia> a=b=0
0

julia> a
0

julia> b
0

julia> b+=3
3

julia> x = typeof(a)
Int64
```

```
julia> x
Int64
```

Todavia, não é possível fazer atribuições unárias (Julia não permite que por exemplo os operadores '+' e '-' sejam unários); tampouco atribuições condicionais.

```
julia> ++a
ERROR: syntax: "++" is not a unary operator
```

13.2 Comandos Sequenciais

Assim como na maioria das linguagens de programação imperativas, em Julia a maneira de implementar os comandos sequenciais é através de blocos. Os marcadores de bloco inicial e final são ,respectivamente, o comando em si (for, if, function) e a palavra chave end. Esses blocos formam ambientes de amarração.

```
julia> function (alguem_viu)
           ap = 3
           if(alguem_viu)
               ap = 7
               alguem_viu = false
           end
       end
(::#1) (generic function with 1 method)
```

13.3 Comandos Condicionais

Os comandos condicionais são de grande importância para a programação como um todo pois eles influenciam o fluxo de execução, decidindo qual bloco será executado na sequência depois de avaliada uma condição. A maioria das linguagens de programação oferecem três tipos de comandos condicionais: seleção de caminho condicional, seleção de caminho duplo e seleção de caminhos múltiplos. A linguagem Julia em sua versão atual não oferece um comando de seleção de caminhos múltiplos, mas os autores já declararam que existe o desejo de incluí-lo na biblioteca da linguagem em versões posteriores.

Em Julia o comando condicional é o if, sendo que para seleção de caminho duplo é utilizado também else e elseif. Para simular o comando de seleção de caminhos múltiplos é preciso fazer um aninhamento de comandos desse tipo pode . Uma postura adotada por Julia que é semelhante à adotada por JAVA consiste em restringir que os valores avaliados na condição do comando condicional seja do tipo booleano, caso contrário irá ocorrer um erro de execução informando que o valor avaliado não é desse tipo.

```
julia> if nota>=60
           print("Aprovado")
       elseif nota>=39 && nota<60
           print("Exame final")
       else
           print("Reprovado direto")
       end
```

Ao contrário de muitas linguagens de programação já consolidadas, em Julia blocos de comando if não definem escopos locais (por isso não é preciso utilizar end para cada else ou elseif). Dessa forma é possível que variáveis declaradas dentro desses blocos sejam utilizadas posteriormente no código sem necessidade de terem sido definidas anteriormente.

Importante destacar que o comando if em Julia também é uma expressão pois ele retorna um valor. O valor a ser retornado pelo comando é o valor da última declaração que foi executada no caminho tomado pelo comando. Abaixo vemos um código que faz exatamente a mesma função do exemplo anterior, só que retornado os valores das strings ao invés de utilizar a função print().

```
julia> nota=39
       if nota>=60
           "Aprovado"
       elseif nota>=39 && nota<60
           "Exame final"
       else
           "Reprovado direto"
       end
"Exame final"
```

13.4 Comandos Iterativos

Comandos iterativos avaliam uma condição mais de uma vez no mesmo código, isto pode ser de grande utilidade caso o programador queira realizar um mesmo conjunto de comandos/expressões mais de uma vez ou submeter valores diferentes a tal conjunto. Basicamente dois tipos de comandos iterativos - também comumente chamados de loops - em linguagens de programação: os com número definido e indefinidos de repetições. A linguagem Julia fornece ambos os dois tipos através dos comandos for e while respectivamente.

A variável de controle no comando for na linguagem Julia pode iterar em intervalos estabelecidos no comando, que na prática são objetos do tipo Range, que equivale a um intervalo de números, separando limite inferior e superior do intervalo pelo caractere ':'; e pode iterar também num vetor ou qualquer conjunto de dados. Para essa última

possibilidade basta utilizar o caractere '∈' ao invés do '='.

```
julia> for i = 10:15
           println(i*2)
       end
20
22
24
26
28
30

julia> for i ∈ ["no final","vai","dar","certo"]
           println(i)
       end
no final
vai
dar
certo
```

Importante destacar que o escopo de visibilidade da variável de controle no comando for é o próprio comando. Isto significa que, a menos que a variável tenha sido definida em outro escopo, ela não pode ser acessada, não é visível fora do escopo do comando.

Também é possível fazer combinação de comandos iterativos em Julia, com o resultado de cada iteração sendo algum elemento do produto cartesiano entre os valores assumidos pelas variáveis de controle dos comandos.

```
julia> for i = 1:2, j = 9:10
           println((i, j))
       end
(1, 9)
(1, 10)
(2, 9)
(2, 10)
```

Já o comando while funciona da maneira tradicional a das outras linguagens de programação: avaliando uma expressão e executando uma sequência de comandos enquanto essa expressão for verdadeira. A varíavel de controle nesse caso é visível fora do bloco do comando, visto que ela deve ser previamente declara e inicializada.

```
julia> i = 10
10
```

```
julia> while i <= 14
           println(i*2)
           i+=1
       end
20
22
24
26
28
```

13.5 Comandos de Desvio Incondicional

Como o nome já sugere, comandos de desvio incondicionais são comandos de desvio que não precisam ter uma condição satisfeita para mudar o fluxo de execução. Esse tipo de comando se divide em basicamente doi subtipos: comandos de desvio irrestritos e escapes. Julia não possui o primeiro tipo de comando citado, fornecendo para os programadores apenas os escapes.

A linguagem Julia oferece os comandos de escape para interromper comandos iterativos através da palavra chave break e para continuar o comando a partir de determinado ponto em cada iteração, sem executar os comandos que estão na sequência. Isso é possível utilizando a palavra chave continue. Julia permite que tais comandos sejam utilizados juntamente com rótulos.

```
julia> for i = 0:100
           print(i,' ')
           if(i==4)
               break
           end
       end
0 1 2 3 4

julia> for i = 1:10
           if(i%2!=0)
               continue
           end
           print(i,' ')
       end

2 4 6 8 10
```

Os outros escapes fornecidos por Julia têm a função de finalizar a execução de um subprograma ou de um programa imediatamente. Esses comandos são respectivamente return, e exit(). O comando exit() inclusive também pode ser usado para fechar o executável de Julia caso o programador esteja utilizando o próprio terminal para escrever o código de seu programa.

14 Orientação a Expressão

Como vistos nos tópicos anteriores uma expressão tem por objetivo gerar um valor, ao passo que um comando tem por objetivo atualizar valores de uma variável ou controlar o fluxo de execução de um programa. Porém existem linguagens que ao avaliar uma expressão também podem atualizar uma variável, e ao executar comandos podem retornar valores, essas linguagens são conhecidas como linguagens orientadas a expressões. Julia é uma dessas linguagens, suprimindo as distinções entre comandos e expressões. Como pode ser visto no exemplo abaixo comandos também retornam valores, sendo este a ultima expressão avaliada.

```
julia> z=2
2

julia> x = if(z>0)
       z = z+2
       end
4

julia> print(x)
4
```

Uma expressão ao ser avaliada também pode ter como efeito colateral atualizar uma variável:

```
julia> x = y = z = 3*5+1
16

julia> print(x)
16
julia> print(y)
16
julia> print(z)
16
```

No exemplo acima ao ser avaliada a expressão $3 * 5 + 1$, a mesma tem como efeito colateral alterar o valor das variáveis x, y e z.

15 Parâmetros

No estudo de Linguagens de Programação uma das definições utilizadas é a de parâmetros reais e parâmetros formais. Os parâmetros formais são o aos identificadores que estão listados no cabeçalho do subprograma e que são usados no seu corpo. Já os parâmetros reais se referem aos identificadores, valores ou expressões utilizados na chamada de um subprograma.

15.1 Correspondência entre Parâmetros Reais e Parâmetros Formais

A correspondência entre as listas de parâmetros reais e parâmetros reais pode ser a posicional. Nesse tipo de correspondência o posicionamento (sequência) dos parâmetros é que relaciona cada parâmetro real com um parâmetro formal. Abaixo é mostrado um exemplo disso:

```
julia> function soma(x :: Float64, y :: Int64)
         x+y
         end
soma (generic function with 1 method)
julia> soma(2.5,5)
7.5
julia> soma(5,2.5)
ERROR: MethodError: no method matching soma(::Int64, ::Float64)
```

A chamada dessa função *soma* passando os dois valores na ordem correta funciona normalmente como é possível visualizar na primeira chamada. Mas, ao passar os valores com a sequência trocada um erro ocorre, pois a correspondência é sequêncial.

Julia também ofere a correspondência com o uso de palavras chave. Isso é muito útil pois um subprograma com uma lista muito grande de parâmetros é mais díficil de utilizá-lo, já que o programador deve lembrar a sequência das correspondências entre parâmetros reais e formais. Com o uso de palavras chave este problema é resolvido. Um exmplo disso são os parâmetros *size* e color da função *h*. Note que a função com parâmetros é definida com um caracter ponto e vírgula na assinatura e note também que as palavras chaves possuem um valor *default*.

```
function h(x, y; size=100, color="blue")
    ###
end
```

Dessa maneira, a função *h* pode ser chamada de várias formas, algumas delas são:

Utilizando palavra-chave.

```
julia> h(1,2, size=10,color="black")
```

Utilizando palavra-chave, mas omitindo algumas delas.

```
julia> h(1,2,size=200)
```

Omitindo todas as palavra-chave.

```
julia> h(1,2)
```

15.2 Direção da Passagem dos Parâmetros

Julia oferece passagem unidirecional de entrada. Dessa forma, quando os parâmetros formais sofrem alterações no corpo do subprograma os valores dos parâmetros reais não sofreram alterações. Abaixo é apresentado um exemplo disso:

```
julia> function igual12(i::Int32)
                  i = 12
         end
igual12 (generic function with 1 method)

julia> a = 1
1

julia> igual12(a)
12

julia> a
1
```

No exemplo é possível ver que a variável *a* não sofre alteração após a chamada da função *igual12*.

15.3 Mecanismos e os Momentos de Passagem de Parâmetros

O mecanismo para implementação de passagem de parâmetros que Julia utiliza é o de referência. Isso significa que os valores dos parâmetros reais não são copiados para os parâmetros formais. Os parâmetros formais funcionam como novas amarrações para o mesmo espaço de memória alocado para os parâmetros reais.

Julia avalia a passagem de parâmetros reais no momemnto da chamada do subprograma, modo normal (*eager*).

16 Tipos Abstratos de Dados

Tipos Abstratos de Dados (TADs) é uma representação utilizada pelos programadores para abstrair informações. Os TADs são compostos pela Estruturas de Dados (EDs) e as operações que podem ser executadas sobre esses dados.

Em Julia não é possivél implementar efetivamente TADs, da mesma forma como a linguagem C, Julia não tem um mecanismo que faz a junção obrigatoriamente entre os EDs e as operações. Diferente de outras linguagens Orientadas a Objetos, C++ e Java, por exemplo, Julia não possui o concentido da entidade de computação classe. Dessa forma, uma maneira de aplicar o conceito de TAD é criar uma estrutura e criar funções que receba o tipo criado, um exemplo disso é mostrado a seguir.

```
julia> struct Valores
        x :: Int
        y :: Int
        end

julia> function adicao(v :: Valores)
        v.A+v.B
        end
adicao (generic function with 1 method)

julia> function subtracao(v :: Valores)
        v.A-v.B
        end
subtracao (generic function with 1 method)

julia> function multiplicacao(v :: Valores)
        v.A*v.B
        end
multiplicacao (generic function with 1 method)

julia> function divisao(v :: Valores)
        v.A/v.B
        end
divisao (generic function with 1 method)

julia> function potencia(v :: Valores)
        v.A^v.B
        end
potencia (generic function with 1 method)

julia> function maior(v :: Valores)
        v.A > v.B ? v.A : v.B
        end
maior (generic function with 1 method)
```

No exemplo acima é possível ver que a ED *Valores* possui as operações: *adicao*, *subtracao*, *multiplicacao* e *divisao*, essa implementação não obriga necessariamente que estas operações só sejam realizadas por instâncias de *Valores* e também que seja seguida uma ordem de execução.

17 Pacotes e Compilação Separada de Arquivos

17.1 Pacotes

Julia possui um gerenciador de pacotes próprio, que pode ser usado para adicionar os vários pacotes registrados na linguagem Julia ou pacotes criados pelo usuário.

Os pacotes instalados podem ser vistos utilizando o comando Pkg.status():

```
julia> Pkg.status()
2 required packages:
 - Atom                          0.6.2
 - Gadfly                        0.6.4                  master
70 additional packages:
 - AxisAlgorithms                0.2.0
 - BinDeps                       0.7.0
 - Blink                         0.5.3
 - Calculus                      0.2.2
 - CodeTools                     0.4.7
 - Codecs                        0.3.0
 - ColorTypes                    0.6.3
 - Colors                        0.8.1
 - CommonSubexpressions          0.0.1
 - Compat                        0.32.0
 - Compose                       0.5.4
```

Novos pacotes podem ser instalados e removidos utilizando os seguintes comandos: Pkg.add("nome_do_pacote") e Pkg.rm("nome_do_pacote"). Para se atualizar os pacotes disponíveis é utilizando o seguinte comando: Pkg.update().

Pacotes registrados na linguagem julia são incluidos apenas com o nome do mesmo, porém para pacotes não inclusos deve ser usado o seguinte comando: Pkg.clone("Url_do_pacote").

Para criar um novo deve ser utilizado o comando Pkg.generate("nome_do_pacote", "licença"), sendo que o campo licença é uma licença conhecida pelo gerenciador de pacotes.

17.2 Módulos

Módulos são espaços de trabalho separados, que permitem agrupar funções e variáveis com própositos semelhantes. Em Julia, os módulos são delimitados pelas palavras-chaves **module** e **end**. Além disso é usado a palavra chave **export** para definir quais funções e variáveis poderão ser importadas quando algum subprograma fizer uso do mesmo.

```
julia> module MeuModulo
       export x,y
       x() = "4"
```

```
y() = "y"
p() = "p"
end
```

A importação de um módulo pode ser feita de duas formas, utilizando a palavra-chave **import** ou utilizando **using**. Ao se utilizar **import**, os métodos só podem ser acessados utilizando o nome do módulo seguido pelo nome do método, o que não acontece quando se utiliza **using**.

```
julia> import MeuModulo

julia> print(x())
ERROR: UndefVarError: x not defined
Stacktrace:
 [1] macro expansion at .\REPL.jl:97 [inlined]
 [2] (::Base.REPL.##1#2{Base.REPL.REPLBackend})() at .\event.jl:73

julia> print(MeuModulo.x())
4
```

Quando se utiliza **using** as variáveis e funções exportadas podem ser acessadas diretamente:

```
julia> using MeuModulo

julia> print(x())
4

julia> print(p())
ERROR: UndefVarError: p not defined
Stacktrace:
 [1] macro expansion at .\REPL.jl:97 [inlined]
 [2] (::Base.REPL.##1#2{Base.REPL.REPLBackend})() at .\event.jl:73


julia> print(MeuModulo.p())
p
```

Como pôde ser percebido, as funções não exportadas ainda não podem ser acessadas sem o uso do nome do módulo.

Julia permite que um módulo seja divido em vários arquivos, bastando usar a palavra-chave **include** para incluir os arquivos pertencentes ao módulo. Assim as funções e variáveis pertencentes aos arquivos incluídos passam a fazer parte do módulo.

```
module Foo

include("arquivo1.jl")
include("arquivo2.jl")
```

```
end
```

Grandes módulos podem demorar vários segundos para carregar porque a execução de todas as instruções em um módulo envolve a compilação de uma grande quantidade de código. Julia oferece a capacidade de criar versões pré-compiladas de módulos para reduzir esse tempo. Isso é feito usando a diretiva __precompile__(). Como exemplo foi definido dois modulos que executam a mesma função, um contendo a diretiva e outro não.

```
module teste
        export n
        n=0
        for i in 1:10^8
        n+=i
        end
end

__precompile()__
module testePre
        export n
        n=0
        for i in 1:10^8
        n+=i
        end
end

julia> @time using teste
  2.151297 seconds (200.00 M allocations: 2.980 GB, 8.12% gc time)

julia> workspace()

julia> @time using teste
  2.018412 seconds (200.00 M allocations: 2.980 GB, 2.90% gc time)
```

Sem se usar o __precompile__, o tempo de carregamento do módulo é o mesmo para execuções diferentes. O comando workspace() é utilizando para se limpar o ambiente de trabalho.

```
julia> @time using testePre
INFO: Precompiling module TestPackage.
  2.696702 seconds (222.21 k allocations: 9.293 MB)

julia> workspace()

julia> @time using testePre
  0.000206 seconds (340 allocations: 16.180 KB)
```

Já se utilizando a diretiva, podemos perceber que o tempo de carregamento na segunda vez é muito menor, uma vez que o mesmo já foi pré-compilado na primeira execução.

18 Tipagem

O sistema de tipos em Julia é dinâmico, dessa forma a linguagem não pode ser classificada como fortemente tipada. Embora esse tipo de sistema seja menos eficiente, visto que os tipos tem que ser verificados em tempo de execução, ele ganha do sistema estático de tipos no aspecto de poder identificar se um dado valor é de determinado tipo. Tal aspecto permite que o método de despacho aplicado nos argumentos de funções seja melhor integrado com a linguagem, tornando o sistema de tipos mais eficiente e ainda assim intuitivo e de fácil entendimento.

Caso o tipo seja omitido na definição, o compilador de Julia irá permitir que qualquer tipo de dado seja amarrado à variável, permitindo que o programador escreva códigos eficientes sem declaração explícita de tipos; aumentando também a redigibilidade. Graças a alta modificabilidade da linguagem, é fácil explicitar os tipos das variáveis posteriormente se necessário. Como dito anteriormente, o tipo do parâmetro também pode ser omitido na declaração de funções.

Além de dinâmico, o sistema de tipos de Julia também pode ser classificado como parametrizado e nominativo. Tipos genéricos podem ser passados como parâmetros e seus relacionamenos entre tipos são explicitamente declarados. Especialmente em Julia, os tipos concretos podem não se "subtiparem", ou seja, todos os tipos concretos são finais e podem ter apenas tipos abstratos como seus supertipos. Essa abordagem traz vantagens a nível de execução, pois herdar o comportamento do tipo ao invés da estrutura do tipo é mais relevante para aumentar a eficiência do código.

Ainda é importante destacar que em Julia apenas valores tem tipos, não variáveis. Todos os valores em Julia são objetos e possuem um tipo, obviamente, este tipo pode não necessariamente o mesmo durante a execução do programa. Esse tipo amarrado a um valor é o único tipo amarrado ao valor no programa, é chamado seu tipo real. Não existe o conceito de um tipo em tempo de compilação, apenas o tipo em tempo de execução.

Para garantir a execução correta do programa, a linguagem Julia trabalha com um algoritmo baseado em dataflow MPF (ponto fixo máximo) de inferência de tipos. Esse algoritmo já é aplicado em linguagens como Self, LISP e JavaScript. Não entraremos em detalhes de implementção desse algoritmo pois esse não é o intuito desse estudo, mas sua idéia básica é acompanhar o estado (os tipos de todos os valores) em cada ponto do programa, determinar o efeito de cada declaração nesse estado e garantir que as informações de tipo de cada declaração sejam espalhadas para todas as outras declarações acessíveis pelo fluxo de controle.

19 Coersão

Coersão é a conversão implícita de tipos. Julia possui coersão e abaixo é mostrado um exemplo disso.

```
julia> function adiciona2(a :: Int)
        a+2
        end
adiciona2 (generic function with 1 method)

julia> adiciona2(2)
4

julia> adiciona2(2.5)
2.5

julia> adiciona2('a')
'c' : ASCII/Unicode U+0063 (category L1: Letter, lowercase)

julia> adiciona2('abc')
ERROR: syntax: invalid character literal
```

No exemplo, a funcão *adiciona2* foi criada com o parâmetro formal *a* do tipo *Int*. Na primeira chamada é passado o inteiro dois e como esperado não ocorre nenhum problema e o resultado quatro é apresentado.

Na segunda chamada, um número *Float64* é passado e o valor de retorno foi 2.5, ou seja, o valor foi convertido para *Float64*.

A terceira chamada é com o caracter 'a', como no caso anterior houve uma conversão de tipos, onde o caracter foi convertido para inteiro, somado duas unidades e por fim reconvertido para caracter, no caso o caracter 'c'.

Já na quarta chamada de *adiciona2* ocorre um erro. Isso porque, a string 'abc' passada não pode ser convertida para um número inteiro e incrementada duas unidades.

Outro exemplo de coersão em Julia é a utilização de uma função que possui como parâmetro formal um tipo ponto flutuante, um exemplo disso é mostrado a seguir:

```
julia> x = 10
10
julia> y = 3.5
3.5
julia> function adicionaMeio(f :: Float64)
        f+0.5
        end
adicionaMeio (generic function with 1 method)
julia> adicionaMeio(y)
4
```

```
julia> adicionaMeio(x)
ERROR: MethodError: no method matching adicionaMeio(::Int32)
```

Neste segundo exemplo é possível perceber que não ocorre a coersão do tipo inteiro para o ponto flutuante. A função possui o parêmetro formal f do tipo *Float*, o parâmetro real y, mesmo tipo de f, é utilizado na chamada da função e a operação de soma ocorre normalmente. Porém, quando o parâmetro real é um inteiro, no caso x, um erro ocorre pois não ocorre a conversão de tipos. A única maneira seria realizar a conversão explicitamente, como mostrado abaixo:

```
julia> adicionaMeio(Float64(x))
10.5
```

No caso, a variável x foi convertida para *Float64*.

20 Polimorfismo

Polimorfismo é uma capacidade que uma linguagem possui de se criar código capaz de operar ou aparentar operar sobre valores de tipos distintos. Existem diferentes tipos de polimorfismo, que são classificados como Ad-hoc ou Universal. Os tipos de polimorfismo são detalhados abaixo, onde também é feito uma análise no contexto da linguagem Julia.

20.1 Polimorfismo por Coersão

Nesse tipo de polimorfismo, tem-se a impressão de um método poder operar sobre diferentes tipos. Porém o que ocorre na prática é que o tipo é convertido implicitamente para o mesmo tipo requerido pela função e então a mesma é chamada com o tipo correto. Julia não têm suporte a esse tipo de polimorfismo, uma vez que não faz a conversão implicita de tipos.

```
julia> function printa_float( f:: Float64)
       print(f)
       end
printa_float (generic function with 1 method)

julia> printa_float(2)
ERROR: MethodError: no method matching printa_float(::Int64)
Closest candidates are:
  printa_float(::Float64) at REPL[3]:2
Stacktrace:
 [1] macro expansion at .\REPL.jl:97 [inlined]
 [2] (::Base.REPL.##1#2{Base.REPL.REPLBackend})() at .\event.jl:73
```

20.2 Polimorfismo por Sobrecarga

Nesse tipo de polimorfismo, também não se tem o polimorfismo em sua forma plena. Aqui têm-se a impressão de um método poder operar sobre diferentes tipos. Porém o que ocorre na prática é que o mesmo método é definido várias vezes recebendo diferentes tipos de parâmetros. Assim quando o método é chamado para determinados parâmetros, o método chamado será aquele que possuir em sua assinatura os mesmos tipos dos parâmetros. Julia tem suporte a esse tipo de polimorfismo.

```
julia> function soma(f1:: Float64, f2:: Float64)
       f1+f2
       end
soma (generic function with 1 method)

julia> function soma(i1:: Int64, i2:: Int64)
       i1+i2
       end
soma (generic function with 2 methods)

julia> soma(2.0,2.3)
4.3

julia> soma(3,6)
9
```

20.3 Polimorfismo Paramétrico

Nesse tipo de polimorfismo os métodos recebem um parâmetro implícito genérico sobre o qual o mesmo irá operar. E assim, no momento da passagem do parâmetro real, esse tipo genérico assume o mesmo tipo do parâmetro real. Dessa forma se têm realmente o polimorfismo em sua forma plena. Julia tem suporte a esse tipo de polimorfismo, bastando se definir o parâmetro parâmétrico sendo do tipo Any.

```
julia> function parametrico(x::Any)
       x
       end
parametrico (generic function with 1 method)

julia> struct Point
       x ::Int
       y:: Int
       end

julia> p = Point(1,2)
Point(1, 2)
```

```
julia> parametrico(p)
Point(1, 2)

julia> parametrico(3)
3

julia> parametrico(3.2)
3.2
```

20.4 Polimorfismo por Inclusão

O polimorfismo por inclusão é característico de linguagens orientadas a objetos. Ele se dá através do uso de uma hierarquia de tipos para criar abstrações de dados e controle polimórficas. A idéia fundamental do polimorfismo por inclusão é que elementos dos subtipos são também elementos do supertipo. Quando um subtipo também possui um método com a mesma assinatura de um método presente no supertipo, o que define o método a ser chamado é a associação tardia de tipos. Julia não tem suporte para esse tipo de polimorfismo, uma vez que além de não possuir suporte para classes, também não apresenta o conceito de herança.

Referências

[1] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman, "Julia: A fast dynamic language for technical computing," *arXiv preprint arXiv:1209.5145*, 2012.

[2] "The julia language," https://julialang.org/, 2012. [Online]. Available: https://julialang.org/

[3] "Julia computing," https://juliacomputing.com/, 2015. [Online]. Available: https://juliacomputing.com/

[4] "Juliacon 2018," http://juliacon.org/2018/, 2017. [Online]. Available: http://juliacon.org/2018/

[5] I. Balbaert, A. Sengupta, and M. Sherrington, *Julia: High Performance Programming*, 1st ed., 2016.

[6] F. Varejão, *Linguagens de Programação - Conceitos e Técnicas*. Elsevier, 2004.

[7] "Julia by example," https://juliabyexample.helpmanual.io/, 2017. [Online]. Available: https://juliabyexample.helpmanual.io/