



Бесплатная электронная книга

УЧУСЬ

Julia Language

Free unaffiliated eBook created from
Stack Overflow contributors.

#julia-lang

.....	1
1: Julia Language	2
.....	2
Examples	2
, !	2
2: @goto @label	4
.....	4
.....	4
Examples	4
.....	4
.....	5
3: Conditionals	7
.....	7
.....	7
Examples	7
if ... else	7
if ... else statement	8
.....	8
.....	9
: && 	9
.....	9
.....	10
.....	11
ifelse	11
4: JSON	12
.....	12
.....	12
Examples	12
JSON.jl	12
JSON	12
JSON	13

5: sub2ind	15
.....	15
.....	15
.....	15
Examples	15
.....	15
.....	15
6:	17
.....	17
Examples	17
.....	17
.....	17
.....	18
.....	18
.....	19
.....	19
7:	21
.....	21
.....	21
Examples	21
Collatz	21
.....	22
.....	22
8:	25
.....	25
Examples	25
.....	25
9:	27
.....	27
.....	27
Examples	27
.....	

.....	29
.....	31
.....	31
.....	32
10:	33
Examples	33
.....	33
.....	33
.....	36
.....	37
.....	38
11:	39
.....	39
.....	39
Examples	39
Fizz Buzz	39
.....	40
.....	40
.....	41
12:	42
.....	42
.....	42
Examples	42
.....	42
.....	43
.....	44
13:	47
.....	47
.....	47
Examples	47
.....	47

.....	49
.....	49
.....	50
.....	51
-	51
14:	53
.....	53
Examples.....	53
Y Z.....	53
SKI.....	54
.....	54
SKI.....	55
15:	58
.....	58
.....	58
Examples.....	58
.....	58
.....	60
.....	62
.....	63
16:	64
.....	64
.....	64
Examples.....	64
.....	64
.....	65
-	66
.....	67
.....	67
.....	69
.....	69
.....	70

17:	72
.....	72
.....	72
Examples.....	72
@show.....	72
.....	73
QuoteNode, Meta.quot Expr (: quote).....	74
Meta.quot QuoteNode ,	75
Expr (: quote)?.....	80
.....	80
& bobs	80
.....	81
Expr (AST).....	82
Expr s quote.....	83
quote quote.....	83
\$: (...) - ?.....	84
\$ foo , eval(foo) ?.....	84
macro	84
@show :.....	84
expand Expr.....	85
esc().....	86
: swap esc().....	86
: until	88
assert	89
{}	89
ADVANCED	90
:.....	91
/	92
/	92
eval(Symbol("@M")) ?.....	93
code_typed ?.....	93

???	95
Gotcha	95
Python `dict` / JSON `Dict`	96
	96
	97
	97
Misusage	98
18:	99
	99
Examples	99
	99
	100
19:	102
	102
	102
Examples	102
	102
-	103
20:	104
	104
	104
Examples	104
,	104
	105
	106
21:	107
Examples	107
	107
@	107
@spawn @spawnat	109
@parallel vmap	111
@async @sync	113

.....	117
22:	118
.....	118
.....	118
Examples.....	118
.....	118
.....	120
23:	122
Examples.....	122
.....	122
.....	122
.....	122
.....	123
.....	124
.....	124
24:	125
.....	125
.....	125
Examples.....	125
.....	125
.....	126
.....	126
25:	128
.....	128
.....	128
Examples.....	128
REPL.....	128
Unix-.....	128
Windows.....	128
REPL	128
.....	131
REPL.....	131

.....	131
.....	132
26:	134
.....	134
Examples.....	134
REPL.....	134
.....	134
27:	136
Examples.....	136
.....	136
28:	137
.....	137
.....	137
Examples.....	137
.....	137
Compat.jl.....	138
.....	138
.....	139
29:	141
.....	141
.....	141
Examples.....	141
.....	141
.....	143
.....	144
==, === isequal.....	145
==.....	145
===.....	146
isequal.....	147
30:	149
.....	149

Examples.....	149
.....	149
31:	150
.....	150
.....	150
Examples.....	150
.....	150
@b_str.....	151
@big_str.....	151
@doc_str.....	151
@html_str.....	152
@ip_str.....	152
@r_str.....	153
@s_str.....	153
@text_str.....	153
@v_str.....	153
@MIME_str.....	153
,	153
.....	154
.....	155
.....	155
.....	155
32:	157
.....	157
.....	157
Examples.....	157
, !.....	157
.....	158
.....	159
(,).....	160
sprint -.....	161

33:	163
.....	163
.....	163
Examples.....	163
.....	163
.....	164
.....	165
.....	168
.....	168
34:	170
.....	170
.....	170
Examples.....	170
.....	170
?	171
?	172
.....	172
.....	172
Singleton.....	172
.....	174
.....	174
35:	176
.....	176
.....	176
Examples.....	176
.....	176
.....	177
.....	177
.....	177
.....	177
.....	178

.....	179
.....	180
.....	181
.....	182
.....	182
.....	183
.....	183
36:	185
.....	185
.....	185
Examples.....	185
.....	185
,	186
37: DataFrame	188
Examples.....	188
.....	188
.....	188
.....	189

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: [julia-language](#)

It is an unofficial and free Julia Language ebook created for educational purposes. All the content is extracted from [Stack Overflow Documentation](#), which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Julia Language.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Версии

Версия	Дата выхода
0.6.0-DEV	2017-06-01
0.5.0	2016-09-19
0.4.0	2015-10-08
0.3.0	2014-08-21
0.2.0	2013-11-17
0.1.0	2013-02-14

Examples

Привет, мир!

```
println("Hello, World!")
```

Чтобы запустить Julia, сначала получите переводчика со страницы [загрузки веб-сайта](#). Текущий стабильный выпуск - v0.5.0, и эта версия рекомендуется для большинства пользователей. Некоторые разработчики пакетов или опытные пользователи могут использовать ночную сборку, которая намного менее стабильна.

Когда у вас есть интерпретатор, напишите свою программу в файле с именем `hello.jl`. Затем он может быть запущен с системного терминала следующим образом:

```
$ julia hello.jl
Hello, World!
```

Джулия также может запускаться в интерактивном режиме, запустив программу `julia`. Вы должны увидеть заголовок и запрос, как показано ниже:

```

      _
     _ _(_) _ | A fresh approach to technical computing
  ( _ ) | ( _ ) | Documentation: http://docs.julialang.org
    _ _ _ | | _ _ _ | Type "?help" for help.
    | | | | | | | / _ ` | |
    | | | _ | | | | ( _ | | | Version 0.4.2 (2015-12-06 21:47 UTC)
  _/ | \ _ ' _ | _ | \ _ ' _ | | Official http://julialang.org/ release

```

```
|__/_ | x86_64-w64-mingw32  
julia>
```

Вы можете запустить любой код Юлии в этом [REPL](#) , поэтому попробуйте:

```
julia> println("Hello, World!")  
Hello, World!
```

В этом примере используется [строка](#) "Hello, World!" , а [функция](#) `println` одна из многих в стандартной библиотеке. Для получения дополнительной информации или справки попробуйте следующие источники:

- REPL имеет интегрированный [режим справки](#) для доступа к документации.
- Официальная [документация](#) достаточно полная.
- У Stack Overflow есть небольшая, но растущая коллекция примеров.
- Пользователи на [Gitter](#) рады помочь с небольшими вопросами.
- Главная онлайн-площадка для Джулии - форум Дискурса на discourse.julialang.org .
Здесь должны быть размещены более сложные вопросы.
- Коллекция учебных пособий и книг , можно найти [здесь](#) .

Прочитайте [Начало работы с Julia Language](#) онлайн: <https://riptutorial.com/ru/julia-lang/topic/485/начало-работы-с-julia-language>

глава 2: @goto и @label

Синтаксис

- @goto label
- метка @label

замечания

Чрезмерное использование или неправильное использование расширенного потока управления делает код трудным для чтения. @goto или его эквиваленты на других языках при неправильном использовании приводит к нечитаемому коду спагетти.

Подобно языкам типа C, нельзя переключаться между функциями в Julia. Это также означает, что @goto невозможно на верхнем уровне; он будет работать только внутри функции. Кроме того, нельзя переходить от внутренней функции к ее внешней функции или от внешней функции к внутренней функции.

Examples

Проверка ввода

Хотя это не традиционно считается циклом, макросы @goto и @label могут использоваться для более продвинутого потока управления. Один случай использования - когда отказ одной части должен привести к повторению целой функции, часто полезной при проверке ввода:

```
function getsequence()
    local a, b

    @label start
    print("Input an integer: ")
    try
        a = parse{Int, readline()}
    catch
        println("Sorry, that's not an integer.")
        @goto start
    end

    print("Input a decimal: ")
    try
        b = parse{Float64, readline()}
    catch
        println("Sorry, that doesn't look numeric.")
        @goto start
    end
end
```



```
a, b
end
```

Однако этот вариант использования часто более понятен с помощью рекурсии:

```
function getsequence()
    local a, b

    print("Input an integer: ")
    try
        a = parse{Int, readline()}
    catch
        println("Sorry, that's not an integer.")
        return getsequence()
    end

    print("Input a decimal: ")
    try
        b = parse{Float64, readline()}
    catch
        println("Sorry, that doesn't look numeric.")
        return getsequence()
    end

    a, b
end
```

Хотя оба примера делают то же самое, второе легче понять. Однако первый из них более эффективен (поскольку он позволяет избежать рекурсивного вызова). В большинстве случаев стоимость звонка не имеет значения; но в ограниченных ситуациях первая форма приемлема.

Очистка ошибок

В таких языках, как C, оператор `@goto` часто используется, чтобы гарантировать, что функция очищает необходимые ресурсы, даже в случае ошибки. Это менее важно в Julia, потому что вместо этого часто используются блоки исключений и `try finally`.

Однако код Julia может взаимодействовать с C-кодом и API-интерфейсом C, поэтому иногда функции должны быть написаны как C-код. Приведенный ниже пример надуман, но демонстрирует общий прецедент. Код Julia вызовет `Libc.malloc` для выделения некоторой памяти (это имитирует вызов API C). Если не все распределения успешны, то функция должна освобождать ресурсы, полученные до сих пор; в противном случае возвращается выделенная память.

```
using Base.Libc
function allocate_some_memory()
    mem1 = malloc(100)
    mem1 == C_NULL && @goto fail
    mem2 = malloc(200)
    mem2 == C_NULL && @goto fail
    mem3 = malloc(300)
```

```
    mem3 == C_NULL && @goto fail
    return mem1, mem2, mem3

@label fail
    free(mem1)
    free(mem2)
    free(mem3)
end
```

Прочитайте @goto и @label онлайн: <https://riptutorial.com/ru/julia-lang/topic/5564/-goto-и--label>

глава 3: Conditionals

Синтаксис

- если `cond`; тело; конец
- если `cond`; тело; еще; тело; конец
- если `cond`; тело; `elseif cond`; тело; еще; конец
- если `cond`; тело; `elseif cond`; тело; конец
- `cond ? iftrue : iffalse`
- `cond && iftrue`
- `cond || iffalse`
- `ifelse (cond, iftrue, iffalse)`

замечания

Все условные операторы и функции включают в себя использование булевых условий (`true` или `false`). В Julia тип булевых элементов - `Bool` . В отличие от некоторых других языков, другие типы чисел (например, `1` или `0`), строки, массивы и т. Д. Не могут использоваться непосредственно в условных выражениях.

Как правило, используются предикатные функции (функции, возвращающие `Bool`) или [операторы сравнения](#) в условии условного оператора или функции.

Examples

if ... else выражение

Наиболее распространенным условным в Юлии является выражение `if ... else` . Например, ниже мы реализуем [евклидовой алгоритм](#) вычисления [наибольшего общего делителя](#) , используя условное выражение для обработки базового случая:

```
mygcd(a, b) = if a == 0
    abs(b)
else
    mygcd(b % a, a)
end
```

Форма `if ... else` в Julia на самом деле является выражением и имеет значение; значение представляет собой выражение в позиции хвоста (то есть последнее выражение) на ветке, которая берется. Рассмотрим следующий пример ввода:

```
julia> mygcd(0, -10)
10
```

Здесь `a` равно 0 а `b` равно -10 . Условие `a == 0` `true` , поэтому берется первая ветвь. Возвращаемое значение - `abs(b)` которое равно 10 .

```
julia> mygcd(2, 3)
1
```

Здесь `a` равно 2 и `b` равно 3 . Условие `a == 0` ложно, поэтому берется вторая ветвь, и мы вычисляем `mygcd(b % a, a)` , который является `mygcd(3 % 2, 2)` . Оператор `%` возвращает остаток, когда 3 делится на 2 , в этом случае 1 . Таким образом, мы вычисляем `mygcd(1, 2)` , и на этот раз `a` равно 1 и `b` равно 2 . Еще раз, `a == 0` является ложным, поэтому берется вторая ветвь, и мы вычисляем `mygcd(b % a, a)` , который является `mygcd(0, 1)` . На этот раз возвращается `a == 0` и так возвращается `abs(b)` , что дает результат 1 .

if ... else statement

```
name = readline()
if startswith(name, "A")
    println("Your name begins with A.")
else
    println("Your name does not begin with A.")
end
```

Любое выражение, такое как выражение `if ... else` , может быть помещено в позицию оператора. Это игнорирует его значение, но все же выполняет выражение для его побочных эффектов.

если утверждение

Как и любое другое выражение, возвращаемое значение выражения `if ... else` можно игнорировать (и, следовательно, отбрасывать). Это обычно полезно только тогда, когда тело выражения имеет побочные эффекты, такие как запись в файл, изменение переменных или печать на экране.

Кроме того, `else` филиал в `if ... else` выражение не является обязательным. Например, мы можем написать следующий код для вывода на экран только в том случае, если выполнено конкретное условие:

```
second = Dates.second(now())
if iseven(second)
    println("The current second, $second, is even.")
end
```

В приведенном выше примере мы используем функции [времени и даты](#), чтобы получить текущую секунду; например, если в настоящее время 10:55:27, переменная `second` будет удерживать 27 . Если это число четное, тогда строка будет напечатана на экране. В противном случае ничего не будет сделано.

Тернарный условный оператор

```
pushunique!(A, x) = x in A ? A : push!(A, x)
```

Тернарный условный оператор является менее многословным, `if ... else` выражение.

Синтаксис:

```
[condition] ? [execute if true] : [execute if false]
```

В этом примере мы добавляем `x` в коллекцию `A` только если `x` уже не находится в `A`. В противном случае мы оставим `A` без изменений.

Тернарный оператор Ссылки:

- [Документация Джулии](#)
- [Wikibooks](#)

Операторы короткого замыкания: `&&` и `||`

Для ветвления

Короткозамкнутые условные операторы `&&` и `||` могут использоваться как легкие замены для следующих конструкций:

- `x && y` эквивалентно `x ? y : x`
- `x || y` эквивалентно `x ? x : y`

Одно использование для операторов короткого замыкания - это более сжатый способ проверить состояние и выполнить определенное действие в зависимости от этого условия. Например, следующий код использует оператор `&&` для выдачи ошибки, если аргумент `x` отрицателен:

```
function mysqrt(x)
    x < 0 && throw(DomainError("x is negative"))
    x ^ 0.5
end
```

`||` оператор также может использоваться для проверки ошибок, за исключением того, что он вызывает ошибку, *если* условие не выполняется, а не *если* условие выполнено:

```
function halve(x::Integer)
    iseven(x) || throw(DomainError("cannot halve an odd number"))
    x ÷ 2
end
```

Другим полезным приложением является предоставление значения по умолчанию для объекта, только если оно не определено ранее:

```
isdefined(:x) || (x = NEW_VALUE)
```

Здесь это проверяет, определен ли символ `x` (т.е. если есть значение, назначенное объекту `x`). Если так, то ничего не происходит. Но, если нет, то `x` будет назначен `NEW_VALUE`. Обратите внимание, что этот пример будет работать только в области охвата.

В условиях

Операторы также полезны, поскольку их можно использовать для проверки двух условий, вторая из которых оценивается только в зависимости от результата первого условия. Из [документации Julia](#):

В выражении `a && b` подвыражение `b` оценивается только в том случае, если `a` оценивает значение `true`

В выражении `a || b`, Подвыражение `b` вычисляется только тогда, когда принимает значение `a false`

Таким образом, хотя оба `a & b` и `a && b` будут выдавать `true` если оба `a` и `b true`, их поведение, если `a false`, отличается.

Например, предположим, что мы хотим проверить, является ли объект положительным числом, где возможно, что он может даже не быть числом. Рассмотрим различия между этими двумя попытками:

```
CheckPositive1(x) = (typeof(x)<:Number) & (x > 0) ? true : false
CheckPositive2(x) = (typeof(x)<:Number) && (x > 0) ? true : false

CheckPositive1("a")
CheckPositive2("a")
```

`CheckPositive1()` приведет к ошибке, если в качестве аргумента будет предоставлен нечисловой тип. Это связано с тем, что он оценивает *оба* выражения независимо от результата первого, а второе выражение дает ошибку при попытке оценить его для нечислового типа.

`CheckPositive2()`, однако, даст `false` (а не ошибку), если к нему будет добавлен нечисловой тип, поскольку второе выражение оценивается только в том случае, если первое `true`.

Более одного оператора короткого замыкания можно натянуть вместе. Например:

```
1 > 0 && 2 > 0 && 3 > 5
```

если оператор с несколькими ветвями

```
d = Dates.dayofweek(now())
if d == 7
    println("It is Sunday!")
elseif d == 6
    println("It is Saturday!")
elseif d == 5
    println("Almost the weekend!")
else
    println("Not the weekend yet...")
end
```

Любое число ветвей `elseif` может использоваться с оператором `if`, возможно с или без последней ветви `else`. Последующие условия будут оцениваться только в том случае, если все предыдущие условия оказались `false`.

Функция `ifelse`

```
shift(x) = ifelse(x > 10, x + 1, x - 1)
```

Использование:

```
julia> shift(10)
9

julia> shift(11)
12

julia> shift(-1)
-2
```

Функция `ifelse` будет оценивать обе ветви, даже те, которые не выбраны. Это может быть полезно, когда ветви имеют побочные эффекты, которые необходимо оценить, или потому, что они могут быть быстрее, если обе ветви сами по себе дешевы.

Прочитайте **Conditionals** онлайн: <https://riptutorial.com/ru/julia-lang/topic/4356/conditionals>

глава 4: JSON

Синтаксис

- с использованием JSON
- JSON.parse (ул)
- JSON.json (OBJ)
- JSON.print (io, obj, indent)

замечания

Поскольку ни объекты Julia `Dict` ни объекты JSON по своей природе не упорядочены, лучше не полагаться на порядок пар ключ-значение в объекте JSON.

Examples

Установка JSON.jl

JSON - популярный формат обмена данными. Самой популярной библиотекой JSON для Julia является [JSON.jl](#). Чтобы установить этот пакет, используйте диспетчер пакетов:

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

Следующий шаг - проверить, работает ли пакет на вашем компьютере:

```
julia> Pkg.test("JSON")
```

Если все тесты пройдены, библиотека готова к использованию.

Разбор JSON

JSON, который был закодирован как строка, может быть легко проанализирован стандартным типом Julia:

```
julia> using JSON

julia> JSON.parse("""{
    "this": ["is", "json"],
    "numbers": [85, 16, 12.0],
    "and": [true, false, null]
}""")
Dict{String,Any} with 3 entries:
  "this" => Any["is", "json"]
  "numbers" => Any[85, 16, 12.0]
  "and" => Any[true, false, nothing]
```


Есть несколько непосредственных свойств `JSON.jl` примечания:

- Типы JSON сопоставляются с разумными типами в Julia: Object становится `Dict` , array становится `Vector` , число становится `Int64` или `Float64` , boolean становится `Bool` , а null становится `nothing::Void` .
- JSON - это нетипизированный формат контейнера: таким образом, возвращенные векторы Julia имеют тип `Vector{Any}` , а возвращаемые словари имеют тип `Dict{String, Any}` .
- Стандарт JSON не различает целые числа и десятичные числа, но `JSON.jl` делает. Число без десятичной точки или научной нотации анализируется в `Int64` , тогда как число с десятичной точкой анализируется на `Float64` . Это тесно связано с поведением парсеров JSON на многих других языках.

Сериализация JSON

Функция `JSON.json` сериализует объект Julia в `String` Julia, содержащую JSON:

```
julia> using JSON

julia> JSON.json(Dict{:a => :b, :c => [1, 2, 3.0], :d => nothing})
"{\"c\": [1.0, 2.0, 3.0], \"a\": \"b\", \"d\": null}"

julia> println(ans)
{"c": [1.0, 2.0, 3.0], "a": "b", "d": null}
```

Если строка не нужна, JSON можно печатать непосредственно в потоке ввода-вывода:

```
julia> JSON.print(STDOUT, [1, 2, true, false, "x"])
[1,2,true,false,"x"]
```

Обратите внимание, что `STDOUT` является значением по умолчанию и может быть опущен в вышеуказанном вызове.

Более мелкая печать может быть достигнута путем передачи необязательного параметра `indent` :

```
julia> JSON.print(STDOUT, Dict{:a => :b, :c => :d}, 4)
{
    "c": "d",
    "a": "b"
}
```

Существует стандартная сериализация по умолчанию для сложных типов Julia:

```
julia> immutable Point3D
    x::Float64
    y::Float64
    z::Float64
end
```

```
julia> JSON.print(Point3D(1.0, 2.0, 3.0), 4)
{
  "y": 2.0,
  "z": 3.0,
  "x": 1.0
}
```

Прочитайте JSON онлайн: <https://riptutorial.com/ru/julia-lang/topic/5468/json>

глава 5: sub2ind

Синтаксис

- `sub2ind(dims :: Tuple {Vararg {Integer}}, I :: Integer ...)`
- `sub2ind{T <: Integer}(dims :: Tuple {Vararg {Integer}}, I :: AbstractArray{T <: Integer, 1} ...)`

параметры

параметр	подробности
<code>затемняет :: Кортеж {Vararg {Integer}}</code>	размер массива
<code>I :: Integer ...</code>	индексы (скалярные) массива
<code>I :: AbstractArray{T <: Integer, 1} ...</code>	индексы (вектор) массива

замечания

Второй пример показывает, что результат `sub2ind` может быть очень `sub2ind` в некоторых конкретных случаях.

Examples

Преобразование индексов в линейные индексы

```
julia> sub2ind((3,3), 1, 1)
1

julia> sub2ind((3,3), 1, 2)
4

julia> sub2ind((3,3), 2, 1)
2

julia> sub2ind((3,3), [1,1,2], [1,2,1])
3-element Array{Int64,1}:
 1
 4
 2
```

Питы и водопады

```
# no error, even the subscript is out of range.
julia> sub2ind((3,3), 3, 4)
```

Нельзя определить, находится ли индекс в диапазоне массива, сравнивая его индекс:

```
julia> sub2ind((3,3), -1, 2)
2

julia> 0 < sub2ind((3,3), -1, 2) <= 9
true
```

Прочитайте `sub2ind` онлайн: <https://riptutorial.com/ru/julia-lang/topic/1914/sub2ind>

глава 6: арифметика

Синтаксис

- $+ x$
- $-x$
- $a + b$
- $a - b$
- $a * b$
- a / b
- $a ^ b$
- $a \% b$
- $4a$
- $\text{SQRT}(a)$

Examples

Квадратичная формула

Джулия использует аналогичные двоичные операторы для основных арифметических операций, как и математика или другие языки программирования. Большинство операторов могут быть записаны в инфиксной нотации (т. Е. Помещены между вычисленными значениями). У Джулии есть порядок операций, соответствующий общей конвенции в математике.

Например, приведенный ниже код реализует [квадратичную формулу](#) , которая демонстрирует операции $+$, $-$, $*$ и $/$ для сложения, вычитания, умножения и деления соответственно. Также показано *неявное умножение* , где число может быть помещено непосредственно перед символом, чтобы означать умножение; то есть $4a$ означает то же, что и $4*a$.

```
function solvequadratic(a, b, c)
    d = sqrt(b^2 - 4a*c)
    (-b - d) / 2a, (-b + d) / 2a
end
```

Использование:

```
julia> solvequadratic(1, -2, -3)
(-1.0, 3.0)
```

Сито Эратосфена

Оператор остатка в Юлии является оператором `%`. Этот оператор ведет себя аналогично `%` в таких языках, как C и C++. `a % b` - остаток, оставшийся после подписания, после деления `a` на `b`.

Этот оператор очень полезен для реализации определенных алгоритмов, таких как следующая реализация [Сита Эратосфена](#).

```
iscoprime(P, i) = !any(x -> i % x == 0, P)

function sieve(n)
    P = Int[]
    for i in 2:n
        if iscoprime(P, i)
            push!(P, i)
        end
    end
    P
end
```

Использование:

```
julia> sieve(20)
8-element Array{Int64,1}:
 2
 3
 5
 7
11
13
17
19
```

Матричная арифметика

Джулия использует стандартные математические значения арифметических операций при применении к матрицам. Иногда вместо этого требуются элементарные операции. Они помечены полной остановкой (`.`), Предшествующей оператору, выполняемой по-разному. (Обратите внимание, что элементарные операции часто не так эффективны, как циклы).

Суммы

Оператор `+` на матрицах является матричной суммой. Он похож на элементную сумму, но не передает форму. То есть, если `A` и `B` имеют одинаковую форму, то `A + B` совпадает с `A .+ B`; в противном случае `A + B` является ошибкой, тогда как `A .+ B` не обязательно может быть.

```
julia> A = [1 2
            3 4]
2×2 Array{Int64,2}:
 1  2
 3  4
```

```

julia> B = [5 6
            7 8]
2×2 Array{Int64,2}:
 5  6
 7  8

julia> A + B
2×2 Array{Int64,2}:
 6  8
10 12

julia> A .+ B
2×2 Array{Int64,2}:
 6  8
10 12

julia> C = [9, 10]
2-element Array{Int64,1}:
 9
10

julia> A + C
ERROR: DimensionMismatch("dimensions must match")
 in promote_shape(::Tuple{Base.OneTo{Int64},Base.OneTo{Int64}}, ::Tuple{Base.OneTo{Int64}}) at
 ./operators.jl:396
 in promote_shape(::Array{Int64,2}, ::Array{Int64,1}) at ./operators.jl:382
 in _elementwise(::Base.#+, ::Array{Int64,2}, ::Array{Int64,1}, ::Type{Int64}) at
 ./arraymath.jl:61
 in +(::Array{Int64,2}, ::Array{Int64,1}) at ./arraymath.jl:53

julia> A .+ C
2×2 Array{Int64,2}:
10 11
13 14

```

Аналогично, – вычисляет разность матриц. Оба + и – также могут использоваться как унарные операторы.

Товары

Оператором * на матрицах является **матричное произведение** (а не элементное произведение). Для элементарного произведения используйте оператор. .* . Сравните (используя те же матрицы, что и выше):

```

julia> A * B
2×2 Array{Int64,2}:
19 22
43 50

julia> A .* B
2×2 Array{Int64,2}:
 5 12
21 32

```

ПОЛНОМОЧИЯ

Оператор `^` вычисляет [экспонентуцию матрицы](#). Выражение матрицы может быть полезно для быстрого вычисления значений определенных повторений. Например, [числа Фибоначчи](#) могут быть сгенерированы [матричным выражением](#)

```
fib(n) = (BigInt[1 1; 1 0]^n)[2]
```

Как обычно, оператор `.``^` Можно использовать, когда поэтапное возведение в степень является желаемой операцией.

Прочитайте [арифметика онлайн](#): <https://riptutorial.com/ru/julia-lang/topic/3848/арифметика>

глава 7: в то время как циклы

Синтаксис

- в то время как cond; тело; конец
- перерыв
- Продолжить

замечания

В `while` цикл не имеет значения; хотя его можно использовать в позиции выражения, его тип - `Void` и полученное значение будет `nothing`.

Examples

Последовательность Collatz

В `while` цикл выполняется его тело, пока имеет место условие. Например, следующий код вычисляет и печатает [последовательность Collatz](#) с заданного числа:

```
function collatz(n)
    while n ≠ 1
        println(n)
        n = iseven(n) ? n ÷ 2 : 3n + 1
    end
    println("1... and 4, 2, 1, 4, 2, 1 and so on")
end
```

Использование:

```
julia> collatz(10)
10
5
16
8
4
2
1... and 4, 2, 1, 4, 2, 1 and so on
```

Можно написать любой цикл рекурсивно, так и для комплекса, `while` петли, иногда рекурсивный вариант более ясно. Однако в Julia циклы имеют определенные преимущества перед рекурсией:

- Julia не гарантирует устранения хвостового вызова, поэтому рекурсия использует дополнительную память и может вызвать ошибки переполнения стека.
- И далее, по той же причине, цикл может уменьшиться накладные расходы и работать

быстрее.

Запуск один раз перед тестированием

Иногда перед запуском какого-либо условия требуется запустить некоторый код инициализации. В некоторых других языках этот вид цикла имеет специальный синтаксис `do while`. Тем не менее, этот синтаксис может быть заменен постоянным `while` цикла и `break` заявлением, поэтому Джулия не специализировалась `do - в while` синтаксис. Вместо этого вы пишете:

```
local name

# continue asking for input until satisfied
while true
    # read user input
    println("Type your name, without lowercase letters:")
    name = readline()

    # if there are no lowercase letters, we have our result!
    !any(islower, name) && break
end
```

Обратите внимание, что в некоторых ситуациях такие циклы могут быть более ясными с рекурсией:

```
function getname()
    println("Type your name, without lowercase letters:")
    name = readline()
    if any(islower, name)
        getname() # this name is unacceptable; try again
    else
        name      # this name is good, return it
    end
end
```

Поиск по ширине

0.5.0

(Хотя этот пример написан с использованием синтаксиса, представленного в версии v0.5, он может работать и с небольшими изменениями в старых версиях.)

Эта реализация **поиска по ширине** (BFS) на графике, представленном списками смежности, использует циклы `while` и оператор `return`. Задача, которую мы решаем, заключается в следующем: у нас есть последовательность людей, и последовательность дружеских отношений (дружба взаимна). Мы хотим определить степень связи между двумя людьми. То есть, если два человека являются друзьями, мы вернем `1`; если кто-то друг друга другого, мы вернем `2` и т. д.

Сначала предположим, что у нас уже есть список смежности: `Dict` отображающий `T` в

`Array{T, 1}`, где ключи - это люди, а значения - все друзья этого человека. Здесь мы можем представлять людей с любым типом `T` мы выбираем; в этом примере мы будем использовать `Symbol`. В алгоритме BFS мы сохраняем очередь людей для «посещения» и отмечаем их расстояние от узла происхождения.

```
function degree(adjlist, source, dest)
    distances = Dict{source => 0}
    queue = [source]

    # until the queue is empty, get elements and inspect their neighbours
    while !isempty(queue)
        # shift the first element off the queue
        current = shift!(queue)

        # base case: if this is the destination, just return the distance
        if current == dest
            return distances[dest]
        end

        # go through all the neighbours
        for neighbour in adjlist[current]
            # if their distance is not already known...
            if !haskey(distances, neighbour)
                # then set the distance
                distances[neighbour] = distances[current] + 1

                # and put into queue for later inspection
                push!(queue, neighbour)
            end
        end
    end

    # we could not find a valid path
    error("$source and $dest are not connected.")
end
```

Теперь мы напишем функцию для построения списка смежности с учетом последовательности людей и последовательности `(person, person)` кортежей:

```
function makeadjlist(people, friendships)
    # dictionary comprehension (with generator expression)
    result = Dict{p => eltype(people)[] for p in people}

    # deconstructing for; friendship is mutual
    for (a, b) in friendships
        push!(result[a], b)
        push!(result[b], a)
    end

    result
end
```

Теперь мы можем определить исходную функцию:

```
degree(people, friendships, source, dest) =
    degree(makeadjlist(people, friendships), source, dest)
```

Теперь давайте проверим нашу функцию на некоторых данных.

```
const people = [:jean, :javert, :cosette, :gavroche, :éponine, :marius]
const friendships = [
  (:jean, :cosette),
  (:jean, :marius),
  (:cosette, :éponine),
  (:cosette, :marius),
  (:gavroche, :éponine)
]
```

Жан связан с собой в 0 шагах:

```
julia> degree(people, friendships, :jean, :jean)
0
```

Джин и Козетт - друзья, и у них есть степень 1 :

```
julia> degree(people, friendships, :jean, :cosette)
1
```

Жан и Гаврох связаны косвенно через Косет, а затем Мариус, поэтому их степень составляет 3 :

```
julia> degree(people, friendships, :jean, :gavroche)
3
```

Javert и Marius не связаны ни с одной цепью, поэтому возникает ошибка:

```
julia> degree(people, friendships, :javert, :marius)
ERROR: javert and marius are not connected.
 in degree(::Dict{Symbol,Array{Symbol,1}}, ::Symbol, ::Symbol) at ./REPL[28]:27
 in degree(::Array{Symbol,1}, ::Array{Tuple{Symbol,Symbol},1}, ::Symbol, ::Symbol) at
 ./REPL[30]:1
```

Прочитайте в то время как циклы онлайн: <https://riptutorial.com/ru/julia-lang/topic/5565/в-то-время-как-циклы>

глава 8: Время

Синтаксис

- `сейчас()`
- `Dates.today ()`
- `Dates.year (τ)`
- `Dates.month (τ)`
- `Dates.day (τ)`
- `Dates.hour (τ)`
- `Dates.minute (τ)`
- `Dates.second (τ)`
- `Dates.millisecond (τ)`
- `Dates.format (t, s)`

Examples

Текущее время

Чтобы получить текущую дату и время, используйте функцию `now` :

```
julia> now()
2016-09-04T00:16:58.122
```

Это местное время, которое включает настраиваемый часовой пояс машины. Чтобы получить время в часовом поясе [скоординированного универсального времени \(UTC\)](#) , используйте `now(Dates.UTC)` :

```
julia> now(Dates.UTC)
2016-09-04T04:16:58.122
```

Чтобы получить текущую дату, без времени, используйте `today()` :

```
julia> Dates.today()
2016-10-30
```

Возвращаемое значение `now` является объектом `DateTime` . Существуют функции для получения отдельных компонентов `DateTime` :

```
julia> t = now()
2016-09-04T00:16:58.122

julia> Dates.year(t)
2016
```

```
julia> Dates.month(t)
9

julia> Dates.day(t)
4

julia> Dates.hour(t)
0

julia> Dates.minute(t)
16

julia> Dates.second(t)
58

julia> Dates.millisecond(t)
122
```

Можно форматировать `DateTime` с помощью форматированной строки:

```
julia> Dates.format(t, "yyyy-mm-dd at HH:MM:SS")
"2016-09-04 at 00:16:58"
```

Поскольку многие функции `Dates` экспортируются из модуля `Base.Dates`, он может сохранить некоторую типизацию для записи

```
using Base.Dates
```

который затем позволяет получить доступ к квалифицированным функциям выше без `Dates.` квалификация.

Прочитайте Время онлайн: <https://riptutorial.com/ru/julia-lang/topic/5812/время>

глава 9: вход

Синтаксис

- Readline ()
- readlines ()
- ReadString (STDIN)
- грызть (ул)
- открыть (f, файл)
- eachline (ИО)
- ReadString (файл)
- чтения (файл)
- readcsv (файл)
- readdlm (файл)

параметры

параметр	подробности
<code>chomp(str)</code>	Удалите одну строку из новой строки строки.
<code>str</code>	Строка для переноса конечной новой строки. Обратите внимание, что строки являются неизменными по соглашению. Эта функция возвращает новую строку.
<code>open(f, file)</code>	Откройте файл, вызовите функцию и закройте файл позже.
<code>f</code>	Генерируется функция вызова потока ввода-вывода, открывающего файл.
<code>file</code>	Путь файла для открытия.

Examples

Чтение строки из стандартного ввода

Поток `STDIN` в Julia относится к [стандартному вводу](#) . Это может представлять собой пользовательский ввод, для интерактивных программ командной строки или вход из файла или [конвейера](#) , который был перенаправлен в программу.

Функция `readline` , если не предоставлена никаких аргументов, будет считывать данные из `STDIN`

до тех пор, пока не встретится `STDIN`, или поток `STDIN` войдет в состояние конца файла. Эти два случая можно отличить от того, был ли символ `\n` прочитан как последний символ:

```
julia> readline()
some stuff
"some stuff\n"

julia> readline() # Ctrl-D pressed to send EOF signal here
""
```

Часто для интерактивных программ мы не заботимся о состоянии EOF и просто хотим строку. Например, мы можем запросить пользователя для ввода:

```
function askname()
    print("Enter your name: ")
    readline()
end
```

Однако это не совсем удовлетворительно из-за дополнительной новой строки:

```
julia> askname()
Enter your name: Julia
"Julia\n"
```

Функция `chomp` доступна для удаления одной строки с новой строки строки. Например:

```
julia> chomp("Hello, World!")
"Hello, World!"

julia> chomp("Hello, World!\n")
"Hello, World!"
```

Поэтому мы можем увеличить нашу функцию с помощью `chomp` чтобы результат был таким, как ожидалось:

```
function askname()
    print("Enter your name: ")
    chomp(readline())
end
```

который имеет более желательный результат:

```
julia> askname()
Enter your name: Julia
"Julia"
```

Иногда мы можем читать как можно больше строк (до тех пор, пока входной поток не войдет в состояние конца файла). Функция `readlines` обеспечивает эту возможность.

```
julia> readlines() # note Ctrl-D is pressed after the last line
```



```
A, B, C, D, E, F, G
H, I, J, K, LMNO, P
Q, R, S
T, U, V
W, X
Y, Z
6-element Array{String,1}:
"A, B, C, D, E, F, G\n"
"H, I, J, K, LMNO, P\n"
"Q, R, S\n"
"T, U, V\n"
"W, X\n"
"Y, Z\n"
```

0.5.0

Еще раз, если нам не нравятся строки новой строки в конце строк, читаемые `readlines`, мы можем использовать функцию `chomp` для их удаления. На этот раз мы [передали](#) функцию `chomp` по всему массиву:

```
julia> chomp.(readlines())
A, B, C, D, E, F, G
H, I, J, K, LMNO, P
Q, R, S
T, U, V
W, X
Y, Z
6-element Array{String,1}:
"A, B, C, D, E, F, G"
"H, I, J, K, LMNO, P"
"Q, R, S"
"T, U, V"
"W, X "
"Y, Z"
```

В других случаях нам могут не нравиться строки, и они просто хотят читать как можно больше, как одну строку. Функция `readstring` выполняет следующее:

```
julia> readstring(STDIN)
If music be the food of love, play on,
Give me excess of it; that surfeiting,
The appetite may sicken, and so die. # [END OF INPUT]
"If music be the food of love, play on,\nGive me excess of it; that surfeiting,\nThe appetite
may sicken, and so die.\n"
```

(`# [END OF INPUT]` не является частью исходного ввода, он добавлен для ясности.)

Обратите внимание, что `readstring` должен быть передан аргумент `STDIN`.

Чтение номеров со стандартного ввода

Чтение чисел из стандартного ввода представляет собой комбинацию строк чтения и синтаксического анализа таких строк, как числа.

Функция `parse` используется для анализа строки в желаемом типе номера:

```
julia> parse{Int, "17"}
17

julia> parse{Float32, "-3e6"}
-3.0f6
```

Формат, ожидаемый в результате `parse(T, x)`, похож на, но не совсем то же, что и формат, который Джулия ожидает от [числовых литералов](#) :

```
julia> -00000023
-23

julia> parse{Int, "-00000023"}
-23

julia> 0x23 |> Int
35

julia> parse{Int, "0x23"}
35

julia> 1_000_000
1000000

julia> parse{Int, "1_000_000"}
ERROR: ArgumentError: invalid base 10 digit '_' in "1_000_000"
 in tryparse_internal{::Type{Int64}, ::String, ::Int64, ::Int64, ::Int64, ::Bool} at
 ./parse.jl:88
 in parse{::Type{Int64}, ::String} at ./parse.jl:152
```

Сочетание `parse` и `readline` функции позволяет читать одно число из строки:

```
function asknumber()
    print("Enter a number: ")
    parse{Float64, readline()}
end
```

которая работает так, как ожидалось:

```
julia> asknumber()
Enter a number: 78.3
78.3
```

Применяются обычные оговорки о [точности с плавающей точкой](#) . Обратите внимание, что `parse` может использоваться с `BigInt` и `BigFloat` для удаления или минимизации потери точности.

Иногда полезно читать более одного номера из той же строки. Как правило, линия может быть разделена пробелом:

```
function askints()
```

```
print("Enter some integers, separated by spaces: ")
[parse{Int, x} for x in split(readline())]
end
```

который может быть использован следующим образом:

```
julia> askints()
Enter some integers, separated by spaces: 1 2 3 4
4-element Array{Int64,1}:
 1
 2
 3
 4
```

Чтение данных из файла

Чтение строк или байтов

Файлы можно открывать для чтения с помощью `open` функции, которая часто используется вместе с [синтаксисом do block](#) :

```
open("myfile") do f
    for (i, line) in enumerate(eachline(f))
        print("Line $i: $line")
    end
end
```

Предположим, что `myfile` существует, и его содержимое

```
What's in a name? That which we call a rose
By any other name would smell as sweet.
```

Затем этот код приведет к следующему результату:

```
Line 1: What's in a name? That which we call a rose
Line 2: By any other name would smell as sweet.
```

Обратите внимание, что `eachline` является ленивой [итерируемой](#) по строкам файла. Предпочтительно `readlines` по соображениям производительности.

Поскольку `do` блок синтаксис просто синтаксический сахар для анонимных функций, мы можем передать по имени функции для `open` тоже:

```
julia> open(readstring, "myfile")
"What's in a name? That which we call a rose\nBy any other name would smell as sweet.\n"

julia> open(read, "myfile")
84-element Array{UInt8,1}:
 0x57
 0x68
```

```
0x61
0x74
0x27
0x73
0x20
0x69
0x6e
0x20
:
0x73
0x20
0x73
0x77
0x65
0x65
0x74
0x2e
0x0a
```

Функции `read` и `readstring` предоставляют удобные методы, которые автоматически открывают файл:

```
julia> readstring("myfile")
"What's in a name? That which we call a rose\nBy any other name would smell as sweet.\n"
```

Чтение структурированных данных

Предположим, у нас был [CSV-файл](#) со следующим содержимым в файле с именем `file.csv`:

```
Make,Model,Price
Foo,2015A,8000
Foo,2015B,14000
Foo,2016A,10000
Foo,2016B,16000
Bar,2016Q,20000
```

Затем мы можем использовать функцию `readcsv` для чтения этих данных в `Matrix`:

```
julia> readcsv("file.csv")
6×3 Array{Any,2}:
 "Make"  "Model"  "Price"
 "Foo"   "2015A"   8000
 "Foo"   "2015B"  14000
 "Foo"   "2016A"  10000
 "Foo"   "2016B"  16000
 "Bar"   "2016Q"  20000
```

Если файл был разделен на закладки, в файле с именем `file.tsv`, `readdlm` можно использовать функцию `delim` аргумент `delim = '\t'`. Более сложные рабочие нагрузки должны использовать [пакет CSV.jl](#).

Прочитайте вход онлайн: <https://riptutorial.com/ru/julia-lang/topic/7201/вход>

глава 10: Выражения

Examples

Введение в выражения

Выражения - это особый тип объекта в Джулии. Вы можете представить выражение как часть кода Юлии, которая еще не была оценена (т.е. выполнена). Существуют конкретные функции и операции, такие как `eval()` которые будут оценивать выражение.

Например, мы могли бы написать скрипт или ввести в интерпретатор следующее: `julia> 1 + 1 2`

Один из способов создания выражения - использовать синтаксис `:()`. Например:

```
julia> MyExpression = :(1+1)
:(1 + 1)
julia> typeof(MyExpression)
Expr
```

Теперь у нас есть `Expr` типа `Expr`. Только что сформировавшись, он ничего не делает - он просто сидит, как любой другой объект, пока он не будет действовать. В этом случае мы можем *оценить* это выражение, используя функцию `eval()`:

```
julia> eval(MyExpression)
2
```

Таким образом, мы видим, что следующие два эквивалентны:

```
1+1
eval(:(1+1))
```

Почему мы хотим пройти гораздо более сложный синтаксис в `eval(:(1+1))` если мы просто хотим найти то, что `1 + 1` равно? Основная причина заключается в том, что мы можем определить выражение в одной точке нашего кода, потенциально модифицировать его позже, а затем оценить его в более поздней точке. Это потенциально может открыть новые мощные возможности для программиста Julia. Выражения являются ключевым компонентом [метапрограммирования](#) в Джулии.

Создание выражений

Существует несколько различных методов, которые можно использовать для создания одного и того же типа выражения. В [выражении intro](#) упоминается синтаксис `:()`. Возможно, лучшее место для начала, однако, связано со строками. Это помогает выявить

некоторые фундаментальные сходства между выражениями и строками в Джулии.

Создать выражение из строки

Из [документации](#) Julia:

Каждая программа Джулии начинает жизнь как строка

Другими словами, любой скрипт Джулии просто написан в текстовом файле, который представляет собой ни что иное, как строку символов. Аналогично, любая команда Julia, введенная в интерпретатор, представляет собой всего лишь строку символов. Роль Джулии или любого другого языка программирования заключается в том, чтобы интерпретировать и оценивать строки символов логичным, предсказуемым образом, чтобы эти строки символов могли использоваться для описания того, что программист хочет выполнить для компьютера.

Таким образом, одним из способов создания выражения является использование функции `parse()` применительно к строке. Следующее выражение, после его вычисления, присваивает значение 2 символу `x`.

```
MyStr = "x = 2"
MyExpr = parse(MyStr)
julia> x
ERROR: UndefVarError: x not defined
eval(MyExpr)
julia> x
2
```

Создать выражение с помощью `:` Синтаксис

```
MyExpr2 = :(x = 2)
julia> MyExpr == MyExpr2
true
```

Обратите внимание, что с помощью этого синтаксиса Julia будет автоматически обрабатывать имена объектов как относящиеся к символам. Мы можем видеть это, если мы посмотрим на `args` выражения. (См. «[Поля объектов выражения](#)» для более подробной информации о поле `args` в выражении.)

```
julia> MyExpr2.args
2-element Array{Any,1}:
 :x
 2
```

Создать выражение с помощью функции `Expr()`

```
MyExpr3 = Expr(:(=), :x, 2)
MyExpr3 == MyExpr
```

Этот синтаксис основан на [нотации префикса](#). Другими словами, первый аргумент, указанный для функции `Expr()` является `head` или префиксом. Остальные `arguments` выражения. `head` определяет, какие операции будут выполняться по аргументам.

Дополнительные сведения об этом см. В разделе «[Поля объектов выражения](#)»

При использовании этого синтаксиса важно различать использование объектов и символов для объектов. Например, в приведенном выше примере выражение присваивает значению `2` символу `:x` - совершенно разумная операция. Если бы мы использовали сам `x` в таком выражении, мы получили бы бессмысленный результат:

```
julia> Expr(:(=), x, 5)
:(2 = 5)
```

Аналогично, если мы рассмотрим `args` мы видим:

```
julia> Expr(:(=), x, 5).args
2-element Array{Any,1}:
 2
 5
```

Таким образом, `Expr()` не выполняет одно и то же автоматическое преобразование в символы как синтаксис `:()` для создания выражений.

Создание многострочных выражений с использованием `quote...end`

```
MyQuote =
quote
    x = 2
    y = 3
end
julia> typeof(MyQuote)
Expr
```

Обратите внимание, что с `quote...end` мы можем создавать выражения, содержащие другие выражения в поле `args`:

```
julia> typeof(MyQuote.args[2])
Expr
```

Дополнительные сведения об этом поле `args` см. В разделе «[Поля объектов выражения](#)».

Подробнее о создании выражений

Этот пример просто дает основы для создания выражений. См. Также, например, [Интерполяция и выражения](#) и [поля объектов выражения](#) для получения дополнительной информации о создании более сложных и расширенных выражений.

Поля выражений

Как упоминалось в выражениях [Intro to Expressions](#) , это особый тип объекта в Julia. Таким образом, у них есть поля. Два наиболее используемых поля выражения - его `head` и его `args` . Например, рассмотрим выражение

```
MyExpr3 = Expr(:(=), :x, 2)
```

обсуждается в [разделе «Создание выражений»](#) . Мы можем видеть `head` и `args` следующим образом:

```
julia> MyExpr3.head
:(=)

julia> MyExpr3.args
2-element Array{Any,1}:
 :x
 2
```

Выражения основаны на [префиксной нотации](#) . Таким образом, `head` обычно указывает операцию, которая должна выполняться на `args` . Голова должна быть `Symbol` типа Юлии.

Когда выражение должно назначать значение (когда оно оценивается), оно обычно использует заголовок `:(=)` . Конечно, существуют очевидные вариации, которые могут быть применены, например:

```
ex1 = Expr(:(+=), :x, 2)
```

: вызов для заголовков выражений

Еще одна общая `head` для выражений `:call` . Например

```
ex2 = Expr(:call, :(*), 2, 3)
eval(ex2) ## 6
```

Следуя соглашениям префиксной нотации, операторы оцениваются слева направо. Таким образом, это выражение означает, что мы будем называть функцию, указанную в первом элементе `args` для последующих элементов. Аналогичным образом мы могли бы:

```
julia> ex2a = Expr(:call, :(-), 1, 2, 3)
:(1 - 2 - 3)
```

Или другие, потенциально более интересные функции, например

```
julia> ex2b = Expr(:call, :rand, 2,2)
:(rand(2,2))

julia> eval(ex2b)
```



```
2x2 Array{Float64,2}:
 0.429397  0.164478
 0.104994  0.675745
```

Автоматическое определение `head` при использовании `:` обозначения создания выражения

Обратите внимание, что `:call` неявно используется как глава в некоторых конструкциях выражений, например

```
julia> :(x + 2).head
:call
```

Таким образом, с синтаксисом `:` для создания выражений, Julia будет стремиться автоматически определять правильную голову для использования. Так же:

```
julia> :(x = 2).head
:(=)
```

На самом деле, если вы не уверены, какую правильную голову использовать для выражения, которое вы формируете с помощью, например, `Expr()` это может быть полезным инструментом для получения советов и идей для использования.

Интерполяция и выражения

Создание выражений означает, что выражения тесно связаны со строками. Таким образом, принципы интерполяции в строках также актуальны для выражений. Например, в базовой интерполяции строк мы можем иметь что-то вроде:

```
n = 2
julia> MyString = "there are $n ducks"
"there are 2 ducks"
```

Мы используем знак `$` чтобы вставить значение `n` в строку. Мы можем использовать ту же технику с выражениями. Например

```
a = 2
ex1 = :(x = 2*$a) ##      :(x = 2 * 2)
a = 3
eval(ex1)
x # 4
```

Сравните это с этим:

```
a = 2
ex2 = :(x = 2*a) # :(x = 2a)
a = 3
eval(ex2)
x # 6
```

Таким образом, в первом примере мы заранее устанавливаем значение `a` которое будет использоваться во время вычисления выражения. Однако со вторым примером компилятор Julia будет смотреть только на `a` чтобы найти его ценность *во время оценки* для нашего выражения.

Внешние ссылки на выражения

Существует целый ряд полезных веб-ресурсов, которые могут помочь вам в получении знаний о выражениях в Джулии. Они включают:

- [Julia Docs - Метaprogramмирование](#)
- [Викиучебники - Юлиа Метaprogramмирование](#)
- [Макросы Юлии, выражения и т. Д. Для и смущенными, Сером Калхоном](#)
- [Месяц Джулии - Метaprogramмирование, Эндрю Коллье](#)
- [Символическая дифференциация в Джулии, Джон Майлс Уайт](#)

SO Сообщений:

- [Что такое «символ» в Джулии? Ответа на этот вопрос Stefan Karpinski](#)
- [Почему джулия выражает это выражение таким сложным образом?](#)
- [Объяснение примера интерполяции выражения Юлии](#)

Прочитайте [Выражения онлайн](#): <https://riptutorial.com/ru/julia-lang/topic/5805/выражения>

глава 11: для циклов

Синтаксис

- для `i` в `iter`; ...; конец
- в то время как `cond`; ...; конец
- перерыв
- Продолжить
- `@parallel (op)` для `i` в `iter`; ...; конец
- `@parallel` для `i` в `iter`; ...; конец
- `@goto label`
- метка `@label`

замечания

Всякий раз, когда он делает код короче и легче читать, рассмотрите возможность использования более высоких функций, таких как `map` или `filter`, вместо циклов.

Examples

Fizz Buzz

Обычный прецедент для цикла `for` состоит в том, чтобы перебирать predetermined диапазон или коллекцию и выполнять одну и ту же задачу для всех своих элементов.

Например, здесь мы объединяем цикл `for` с условным **выражением** `if elseif else`:

```
for i in 1:100
  if i % 15 == 0
    println("FizzBuzz")
  elseif i % 3 == 0
    println("Fizz")
  elseif i % 5 == 0
    println("Buzz")
  else
    println(i)
  end
end
```

Это классический вопрос интервью **Fizz Buzz**. Выход (усеченный):

```
1
2
Fizz
4
Buzz
Fizz
```

```
7
8
```

Найти наименьший первичный коэффициент

В некоторых ситуациях можно вернуться из функции до завершения всего цикла. Для этого можно использовать оператор `return`.

```
function primefactor(n)
    for i in 2:n
        if n % i == 0
            return i
        end
    end
    @assert false # unreachable
end
```

Использование:

```
julia> primefactor(100)
2

julia> primefactor(97)
97
```

Циклы также могут быть завершены на ранней стадии с помощью инструкции `break`, которая завершает только замкнутый цикл вместо целой функции.

Многомерная итерация

В Julia цикл `for` может содержать запятую (,) для указания итерации по нескольким измерениям. Это действует аналогично вложению петли в другую, но может быть более компактным. Например, приведенная ниже функция порождает элементы [декартова произведения](#) двух итераций:

```
function cartesian(xs, ys)
    for x in xs, y in ys
        produce(x, y)
    end
end
```

Использование:

```
julia> collect(@task cartesian(1:2, 1:4))
8-element Array{Tuple{Int64,Int64},1}:
 (1,1)
 (1,2)
 (1,3)
 (1,4)
 (2,1)
 (2,2)
```

```
(2,3)
(2,4)
```

Однако индексирование по массивам любого измерения должно выполняться с `eachindex`, а не с многомерным циклом (если возможно):

```
s = zero(eltype(A))
for ind in eachindex(A)
    s += A[ind]
end
```

Редукционные и параллельные петли

Julia предоставляет макросы для упрощения распределения вычислений на нескольких машинах или рабочих. Например, следующее вычисляет сумму некоторого числа квадратов, возможно, параллельно.

```
function sumofsquares(A)
    @parallel (+) for i in A
        i ^ 2
    end
end
```

Использование:

```
julia> sumofsquares(1:10)
385
```

Более подробная информация по этой теме, см [пример](#) на `@parallel` в рамках параллельной Processing [теме](#).

Прочитайте для циклов онлайн: <https://riptutorial.com/ru/julia-lang/topic/4355/для-циклов>

глава 12: Затворы

Синтаксис

- $x \rightarrow [\text{body}]$
- $(x, y) \rightarrow [\text{body}]$
- $(xs \dots) \rightarrow [\text{body}]$

замечания

0.4.0

В более старых версиях Julia закрытие и анонимные функции имели ограничение производительности во время выполнения. Это наказание было отменено в 0,5.

Examples

Состав функции

Мы можем определить функцию для выполнения [композиции функции](#) с использованием [анонимного синтаксиса функции](#) :

```
f ∘ g = x -> f(g(x))
```

Обратите внимание, что это определение эквивалентно каждому из следующих определений:

```
∘(f, g) = x -> f(g(x))
```

или же

```
function ∘(f, g)
    x -> f(g(x))
end
```

напомним, что в Julia $f \circ g$ является просто синтаксическим сахаром для $\circ(f, g)$.

Мы видим, что эта функция правильно составлена:

```
julia> double(x) = 2x
double (generic function with 1 method)

julia> triple(x) = 3x
triple (generic function with 1 method)
```

```
julia> const sextuple = double ∘ triple
(::#17) (generic function with 1 method)

julia> sextuple(1.5)
9.0
```

0.5.0

В версии v0.5 это определение очень показательно. Мы можем просмотреть созданный код LLVM:

```
julia> @code_llvm sextuple(1)

define i64 @"julia_#17_71238"(i64) #0 {
top:
    %1 = mul i64 %0, 6
    ret i64 %1
}
```

Ясно, что эти два умножения были сведены в одно умножение и что эта функция настолько эффективна, насколько это возможно.

Как работает эта функция более высокого порядка? Он создает так называемое **замыкание**, которое состоит не только из его кода, но и отслеживает определенные переменные из его области. Все функции в Julia, которые не созданы на уровне верхнего уровня, являются закрытием.

0.5.0

Можно проверить переменные, закрытые через поля замыкания. Например, мы видим, что:

```
julia> (sin ∘ cos).f
sin (generic function with 10 methods)

julia> (sin ∘ cos).g
cos (generic function with 10 methods)
```

Реализация каррирования

Одно применение закрытий - частично применить функцию; то есть предоставить некоторые аргументы и создать функцию, которая принимает остальные аргументы.

Каррирование - это конкретная форма частичного применения.

Начнем с простой функции `curry(f, x)`, которая предоставит первый аргумент функции и ожидает дополнительных аргументов позже. Определение довольно просто:

```
curry(f, x) = (xs...) -> f(x, xs...)
```

Еще раз, мы используем **синтаксис анонимных функций**, на этот раз в сочетании с

синтаксисом вариационных аргументов.

Мы можем реализовать некоторые базовые функции в **молчаливом** (или без точек) стиле, используя эту функцию `curry`.

```
julia> const double = curry(*, 2)
(::#19) (generic function with 1 method)

julia> double(10)
20

julia> const simon_says = curry println, "Simon: "
(::#19) (generic function with 1 method)

julia> simon_says("How are you?")
Simon: How are you?
```

Функции поддерживают ожидаемый генералитет:

```
julia> simon_says("I have ", 3, " arguments.")
Simon: I have 3 arguments.

julia> double([1, 2, 3])
3-element Array{Int64,1}:
 2
 4
 6
```

Введение в закрытие

Функции являются важной частью программирования Julia. Они могут быть определены непосредственно внутри модулей, и в этом случае функции называются *верхним уровнем*. Но функции также могут быть определены в рамках других функций. Такие функции называются « **замыканиями** ».

Замыкания фиксируют переменные в их внешней функции. Функция верхнего уровня может использовать только глобальные переменные из своего модуля, параметров функции или локальных переменных:

```
x = 0 # global
function toplevel(y)
    println("x = ", x, " is a global variable")
    println("y = ", y, " is a parameter")
    z = 2
    println("z = ", z, " is a local variable")
end
```

С другой стороны, закрытие может использовать все это в дополнение к переменным от внешних функций, которые он захватывает:

```
x = 0 # global
```



```

function toplevel(y)
    println("x = ", x, " is a global variable")
    println("y = ", y, " is a parameter")
    z = 2
    println("z = ", z, " is a local variable")

    function closure(v)
        println("v = ", v, " is a parameter")
        w = 3
        println("w = ", w, " is a local variable")
        println("x = ", x, " is a global variable")
        println("y = ", y, " is a closed variable (a parameter of the outer function)")
        println("z = ", z, " is a closed variable (a local of the outer function)")
    end
end

```

Если мы запустим `c = toplevel(10)` , мы увидим, что результат

```

julia> c = toplevel(10)
x = 0 is a global variable
y = 10 is a parameter
z = 2 is a local variable
(::closure) (generic function with 1 method)

```

Обратите внимание, что хвостовое выражение этой функции является самой функцией; то есть замыкание. Мы можем назвать замыкание `c` как и любую другую функцию:

```

julia> c(11)
v = 11 is a parameter
w = 3 is a local variable
x = 0 is a global variable
y = 10 is a closed variable (a parameter of the outer function)
z = 2 is a closed variable (a local of the outer function)

```

Обратите внимание, что `c` все еще имеет доступ к переменным `y` и `z` из вызова `toplevel` - даже если `toplevel` уже вернулся! Каждое закрытие, даже те, которые возвращаются одной и той же функцией, закрываются по разным переменным. Мы снова можем вызвать `toplevel`

```

julia> d = toplevel(20)
x = 0 is a global variable
y = 20 is a parameter
z = 2 is a local variable
(::closure) (generic function with 1 method)

julia> d(22)
v = 22 is a parameter
w = 3 is a local variable
x = 0 is a global variable
y = 20 is a closed variable (a parameter of the outer function)
z = 2 is a closed variable (a local of the outer function)

julia> c(22)
v = 22 is a parameter
w = 3 is a local variable
x = 0 is a global variable

```

```
y = 10 is a closed variable (a parameter of the outer function)
z = 2 is a closed variable (a local of the outer function)
```

Обратите внимание, что, несмотря на то, что `d` и `c` имеют одинаковый код и передаются одни и те же аргументы, их вывод отличается. Они являются отличными закрытиями.

Прочитайте Затворы онлайн: <https://riptutorial.com/ru/julia-lang/topic/5724/затворы>

глава 13: итерируемыми

Синтаксис

- начать (ITR)
- next (itr, s)
- done (itr, s)
- take (itr, n)
- drop (itr, n)
- цикл (ITR)
- Base.product (xs, ys)

параметры

параметр	подробности
За	Все функции
itr	Итерируемый для работы.
За	next И done
s	Состояние итератора, описывающее текущую позицию итерации.
За	take И drop
n	Количество элементов, которые нужно взять или удалить.
За	Base.product
xs	Итерируемое, чтобы взять первые элементы пар из.
ys	Итерабельность для принятия вторых элементов пар из.
...	(Обратите внимание, что product принимает любое количество аргументов, если предусмотрено более двух, он построит кортежи длиной более двух.)

Examples

Новый тип итерации

В Julia, когда цикл через итерируемый объект `ι` выполняется с синтаксисом `for` :

```
for i = I    # or "for i in I"
    # body
end
```

За кулисами это переводится на:

```
state = start(I)
while !done(I, state)
    (i, state) = next(I, state)
    # body
end
```

Поэтому, если вы хотите, `I` быть итерацией, вам нужно определить `start`, `next` и `done` методы его типа. Предположим, вы определяете **тип** `Foo` содержащий **массив** как одно из полей:

```
type Foo
    bar::Array{Int,1}
end
```

Мы создаем объект `Foo`, выполняя:

```
julia> I = Foo([1,2,3])
Foo{Array{Int64,1}}

julia> I.bar
3-element Array{Int64,1}:
 1
 2
 3
```

Если мы хотим итерации через `Foo`, каждая `bar` элементов возвращается каждой итерацией, мы определяем методы:

```
import Base: start, next, done

start(I::Foo) = 1

next(I::Foo, state) = (I.bar[state], state+1)

function done(I::Foo, state)
    if state == length(I.bar)
        return true
    end
    return false
end
```

Обратите внимание: поскольку эти **функции** принадлежат модулю `Base`, мы должны сначала `import` их имена, прежде чем добавлять к ним новые методы.

После определения методов `Foo` совместим с интерфейсом итератора:

```
julia> for i in I
        println(i)
    end
```

```
1
2
3
```

Объединение ленивых итераций

Стандартная библиотека поставляется с богатой коллекцией ленивых итераций (и библиотеки, такие как [Iterators.jl](#), обеспечивают еще больше). Lazy iterables может быть составлен для создания более мощных итераций в постоянное время. Самые важные ленивые итерации - это выбор и удаление, из которых можно создать множество других функций.

Ленько нарежьте итерируемый

Массивы могут быть нарезаны нотной записью. Например, следующее возвращает 10-15 элементов массива, включая:

```
A[10:15]
```

Однако нотация среза не работает со всеми повторами. Например, мы не можем разрезать выражение генератора:

```
julia> (i^2 for i in 1:10)[3:5]
ERROR: MethodError: no method matching getindex(::Base.Generator{UnitRange{Int64},##1#2},
::UnitRange{Int64})
```

Строки для нарезки могут не иметь ожидаемого поведения в Юникоде:

```
julia> "aaaa"[2:3]
ERROR: UnicodeError: invalid character index
in getindex(::String, ::UnitRange{Int64}) at ./strings/string.jl:130

julia> "aaaa"[3:4]
"a"
```

Мы можем определить функцию `lazysub(itr, range::UnitRange)` чтобы сделать этот вид нарезки на произвольных итерациях. Это определяется в терминах `take` and `drop` :

```
lazysub(itr, r::UnitRange) = take(drop(itr, first(r) - 1), last(r) - first(r) + 1)
```

Реализация здесь работает, потому что для значения `UnitRange a:b` выполняются следующие шаги:

- сбрасывает первые $a-1$ элементы

- берет a й элемент, $a+1$ й элемент и т. д., пока $a + (ba) = b$ й элемент

Всего берутся элементы ba . Мы можем подтвердить, что наша реализация верна в каждом случае выше:

```
julia> collect(lazysub("aaaa", 2:3))
2-element Array{Char,1}:
 'a'
 'a'

julia> collect(lazysub((i^2 for i in 1:10), 3:5))
3-element Array{Int64,1}:
 9
16
25
```

Ленько сдвиньте итерируемый кругооборот

Операция `circshift` на массивах будет перемещать массив, как если бы это был круг, а затем переиздавать его. Например,

```
julia> circshift(1:10, 3)
10-element Array{Int64,1}:
 8
 9
10
 1
 2
 3
 4
 5
 6
 7
```

Можем ли мы сделать это лениво для всех повторений? Мы можем использовать `cycle`, `drop` и `take` итерации для реализации этой функции.

```
lazycircshift(itr, n) = take(drop(cycle(itr), length(itr) - n), length(itr))
```

Наряду с ленивыми типами, которые более `circshift` во многих ситуациях, это позволяет нам `circshift` функцию `circshift` для типов, которые в противном случае не поддерживали бы ее:

```
julia> circshift("Hello, World!", 3)
ERROR: MethodError: no method matching circshift(::String, ::Int64)
Closest candidates are:
  circshift(::AbstractArray{T,N}, ::Real) at abstractarraymath.jl:162
  circshift(::AbstractArray{T,N}, ::Any) at abstractarraymath.jl:195

julia> String(collect(lazycircshift("Hello, World!", 3)))
"ld!Hello, Wor"
```

Создание таблицы умножения

Давайте создадим [таблицу умножения](#), используя ленивые итерационные функции для создания матрицы.

Ключевыми функциями для использования здесь являются:

- `Base.product`, который вычисляет [декартово произведение](#).
- `prod`, который вычисляет регулярное произведение (как при умножении)
- `:`, который создает диапазон
- `map`, которая является функцией более высокого порядка, применяющей функцию к каждому элементу коллекции

Решение:

```
julia> map(prod, Base.product(1:10, 1:10))
10×10 Array{Int64,2}:
 1  2  3  4  5  6  7  8  9 10
 2  4  6  8 10 12 14 16 18 20
 3  6  9 12 15 18 21 24 27 30
 4  8 12 16 20 24 28 32 36 40
 5 10 15 20 25 30 35 40 45 50
 6 12 18 24 30 36 42 48 54 60
 7 14 21 28 35 42 49 56 63 70
 8 16 24 32 40 48 56 64 72 80
 9 18 27 36 45 54 63 72 81 90
10 20 30 40 50 60 70 80 90 100
```

Лениво-оцененные списки

Можно сделать простой лениво-оцененный список, используя изменяемые типы и [замыкания](#). Лениво-оцененный список - это список, чьи элементы не оцениваются при его построении, а скорее при его доступе. Преимущества лениво оцениваемых списков включают возможность бесконечности.

```
import Base: getindex
type Lazy
    thunk
    value
    Lazy(thunk) = new(thunk)
end

evaluate!(lazy::Lazy) = (lazy.value = lazy.thunk(); lazy.value)
getindex(lazy::Lazy) = isdefined(lazy, :value) ? lazy.value : evaluate!(lazy)

import Base: first, tail, start, next, done, iteratorsize, HasLength, SizeUnknown
abstract List
immutable Cons <: List
    head
    tail::Lazy
end
immutable Nil <: List end
```

```

macro cons(x, y)
    quote
        Cons($(esc(x)), Lazy(() -> $(esc(y))))
    end
end

first(xs::Cons) = xs.head
tail(xs::Cons) = xs.tail[]
start(xs::Cons) = xs
next(::Cons, xs) = first(xs), tail(xs)
done(::List, ::Cons) = false
done(::List, ::Nil) = true
iteratorsize(::Nil) = HasLength()
iteratorsize(::Cons) = SizeUnknown()

```

Что действительно работает так, как на языке [Haskell](#) , где все списки лениво оцениваются:

```

julia> xs = @cons(1, ys)
Cons{1, Lazy{false, #3, #undef}}

julia> ys = @cons(2, xs)
Cons{2, Lazy{false, #5, #undef}}

julia> [take(xs, 5)...]
5-element Array{Int64,1}:
 1
 2
 1
 2
 1

```

На практике лучше использовать пакет [Lazy.jl](#). Тем не менее, реализация ленивого списка выше проливает свет на важные детали о том, как построить свой собственный итерируемый тип.

Прочитайте итерируемые онлайн: <https://riptutorial.com/ru/julia-lang/topic/5466/>
итерируемые

глава 14: Комбинаторы

замечания

Хотя комбинаторы имеют ограниченное практическое применение, они являются полезным инструментом в образовании, чтобы понять, как программирование в корне связано с логикой, и как очень простые строительные блоки могут сочетаться для создания очень сложного поведения. В контексте Джулии, изучение того, как создавать и использовать комбинаторы, укрепит понимание того, как программировать в функциональном стиле в Джулии.

Examples

Комбинатор Y или Z

Хотя Julia не является чисто функциональным языком, она полностью поддерживает многие из краеугольных камней функционального программирования: первоклассные [функции](#), лексический охват и [закрывание](#).

[Комбинатор с фиксированной запятой](#) является ключевым комбинатором в функциональном программировании. Поскольку у Джулии [очень интересная](#) семантика [оценки](#) (как и многие функциональные языки, включая Scheme, в которой Джулия сильно вдохновлена), оригинальный Y-combinator от Curry не будет работать из коробки:

```
Y(f) = (x -> f(x(x))) (x -> f(x(x)))
```

Однако близкий родственник Y-комбинатора, Z-комбинатор, действительно будет работать:

```
Z(f) = x -> f(Z(f), x)
```

Этот комбинатор принимает функцию и возвращает функцию, которая при вызове с аргументом `x` передается сама и `x`. Почему было бы полезно использовать функцию для себя? Это позволяет рекурсии, фактически не ссылаясь на имя функции вообще!

```
fact(f, x) = x == 0 ? 1 : x * f(x)
```

Следовательно, `Z(fact)` становится рекурсивной реализацией факториальной функции, несмотря на то, что в определении этой функции не видно рекурсии. (Конечно, рекурсия очевидна в определении комбинатора `Z`, но это неизбежно на нетерпеливом языке.) Мы можем убедиться, что наша функция действительно работает:

```
julia> Z(fact)(10)
3628800
```

Не только это, но и так же быстро, как мы можем ожидать от рекурсивной реализации. Код LLVM демонстрирует, что результат скомпилирован в обычную ветвь, вычитает, вызывает и умножает:

```
julia> @code_llvm Z(fact)(10)

define i64 @"julia_#1_70252"(i64) #0 {
top:
    %1 = icmp eq i64 %0, 0
    br i1 %1, label %L11, label %L8

L8:                                     ; preds = %top
    %2 = add i64 %0, -1
    %3 = call i64 @"julia_#1_70060"(i64 %2) #0
    %4 = mul i64 %3, %0
    br label %L11

L11:                                   ; preds = %top, %L8
    %"#temp#.0" = phi i64 [ %4, %L8 ], [ 1, %top ]
    ret i64 %"#temp#.0"
}
```

Комбинированная система SKI

Система [комбинаторов SKI](#) достаточна для представления любых членов лямбда-исчисления. (На практике, конечно, абстракции лямбда взрываются до экспоненциального размера, когда они переводятся в SKI.) Из-за простоты системы реализация комбинаторов S, K и I необычайно проста:

Прямой перевод из исчисления Лямбды

```
const S = f -> g -> z -> f(z) (g(z))
const K = x -> y -> x
const I = x -> x
```

Мы можем подтвердить, используя [модульную систему тестирования](#), что каждый комбинатор имеет ожидаемое поведение.

Комбинатор I проще всего проверять; он должен возвращать заданное значение без изменений:

```
using Base.Test
@test I(1) === 1
@test I(I) === I
@test I(S) === S
```

Комбинатор K также довольно прост: он должен отказаться от своего второго аргумента.

```
@test K(1) (2) === 1
@test K(S) (I) === S
```

Комбинатор S является самым сложным; его поведение можно обобщить как применение первых двух аргументов к третьему аргументу, применение первого результата ко второму. Мы можем наиболее легко проверить комбинатор S , проверив некоторые его карри-формы. $S(K)$, например, должен просто вернуть свой второй аргумент и отбросить его первый, как мы видим, происходит:

```
@test S(K) (S) (K) === K
@test S(K) (S) (I) === I
```

$S(I)$ должен применить свой аргумент к себе:

```
@test S(I) (I) (I) === I
@test S(I) (I) (K) === K(K)
@test S(I) (I) (S(I)) === S(I) (S(I))
```

$S(K(S(I)))$ применяет свой второй аргумент к первому:

```
@test S(K(S(I))) (K) (I) (I) === I
@test S(K(S(I))) (K) (K) (S(K)) === S(K) (K)
```

Комбинатор I , описанный выше, имеет имя в стандартном `Base Julia`: `identity`. Таким образом, мы могли бы переписать приведенные выше определения со следующим альтернативным определением I :

```
const I = identity
```

Отображение комбайнов SKI

Одна из недостатков подхода, описанного выше, заключается в том, что наши функции не проявляются так хорошо, как хотелось бы. Можем ли мы заменить

```
julia> S
(::#3) (generic function with 1 method)

julia> K
(::#9) (generic function with 1 method)

julia> I
(::#13) (generic function with 1 method)
```

с некоторыми более информативными дисплеями? Ответ - да! Давайте перезапустим REPL, и на этот раз определим, как каждая функция должна быть показана:

```
const S = f -> g -> z -> f(z) (g(z));
```

```

const K = x -> y -> x;
const I = x -> x;
for f in (:S, :K, :I)
    @eval Base.show(io::IO, ::typeof($f)) = print(io, $(string(f)))
    @eval Base.show(io::IO, ::MIME"text/plain", ::typeof($f)) = show(io, $f)
end

```

Важно не показывать ничего, пока мы не закончим определение функций. В противном случае мы рискуем аннулировать кеш метода, и наши новые методы, похоже, не сразу вступят в силу. Вот почему мы поставили точки с запятой в приведенные выше определения. Точки с запятой подавляют вывод REPL.

Благодаря этому функции отображаются хорошо:

```

julia> S
S

julia> K
K

julia> I
I

```

Тем не менее, мы все еще сталкиваемся с проблемами, когда пытаемся отобразить закрытие:

```

julia> S(K)
(::#2) (generic function with 1 method)

```

Было бы лучше показать это как $S(K)$. Чтобы сделать это, мы должны использовать, что у закрытий есть свои индивидуальные типы. Мы можем получить доступ к этим типам и добавить методы к ним посредством отражения, используя `typeof` и `primary` поле `name` поля типа. Перезапустите REPL снова; мы внесем дальнейшие изменения:

```

const S = f -> g -> z -> f(z) (g(z));
const K = x -> y -> x;
const I = x -> x;
for f in (:S, :K, :I)
    @eval Base.show(io::IO, ::typeof($f)) = print(io, $(string(f)))
    @eval Base.show(io::IO, ::MIME"text/plain", ::typeof($f)) = show(io, $f)
end
Base.show(io::IO, s::typeof(S(I)).name.primary) = print(io, "S(", s.f, ')')
Base.show(io::IO, s::typeof(S(I)(I)).name.primary) =
    print(io, "S(", s.f, ')', '(', s.g, ')')
Base.show(io::IO, k::typeof(K(I)).name.primary) = print(io, "K(", k.x, ')')
Base.show(io::IO, ::MIME"text/plain", f::Union{
    typeof(S(I)).name.primary,
    typeof(S(I)(I)).name.primary,
    typeof(K(I)).name.primary
}) = show(io, f)

```

И теперь, наконец, все выглядит так, как мы хотели бы, чтобы они:

```
julia> S(K)
S(K)

julia> S(K)(I)
S(K)(I)

julia> K
K

julia> K(I)
K(I)

julia> K(I)(K)
I
```

Прочитайте Комбинаторы онлайн: <https://riptutorial.com/ru/julia-lang/topic/5758/комбинаторы>

глава 15: Кортеж

Синтаксис

- `a`,
- `a, b`
- `a, b = xs`
- `()`
- `(A,)`
- `(a, b)`
- `(a, b ...)`
- `Tuple {T, U, V}`
- `NTuple {N, T}`
- `Tuple {T, U, Vararg {V}}`

замечания

Кортежи имеют гораздо лучшую производительность во время выполнения, чем [массивы](#), по двум причинам: их типы более точны, а их неизменность позволяет им выделяться в стеке вместо кучи. Тем не менее, этот более точный ввод печатает с более сложными накладными расходами и большими трудностями при достижении [стабильности типа](#).

Examples

Введение в кортежи

`Tuple` s являются неизменяемыми упорядоченными наборами произвольных отдельных объектов, одного или того же типа или разных [типов](#). Как правило, кортежи строятся с использованием синтаксиса `(x, y)`.

```
julia> tup = (1, 1.0, "Hello, World!")
(1,1.0,"Hello, World!")
```

Отдельные объекты кортежа можно получить с помощью синтаксиса индексирования:

```
julia> tup[1]
1

julia> tup[2]
1.0

julia> tup[3]
"Hello, World!"
```

Они реализуют **итерируемый интерфейс** и поэтому могут быть повторены при использовании **for циклов** :

```
julia> for item in tup
    println(item)
end
1
1.0
Hello, World!
```

Кортежи также поддерживают множество общих функций коллекций, таких как `reverse` или `length` :

```
julia> reverse(tup)
("Hello, World!", 1.0, 1)

julia> length(tup)
3
```

Кроме того, кортежи поддерживают множество операций коллекций **более высокого порядка** , включая `any` , `all` , `map` или `broadcast` :

```
julia> map(typeof, tup)
(Int64, Float64, String)

julia> all(x -> x < 2, (1, 2, 3))
false

julia> all(x -> x < 4, (1, 2, 3))
true

julia> any(x -> x < 2, (1, 2, 3))
true
```

Пустой кортеж можно построить с помощью `()` :

```
julia> ()
()

julia> isempty(ans)
true
```

Однако для построения кортежа одного элемента требуется конечная запятая. Это связано с тем, что скобки `(и)` в противном случае рассматривались бы как операции группировки, а не для построения кортежа.

```
julia> (1)
1

julia> (1,)
(1,)
```

Для согласованности конечная запятая также допускается для кортежей с несколькими элементами.

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

Типы кортежей

`typeof` кортеж является подтипом `Tuple` :

```
julia> typeof((1, 2, 3))
Tuple{Int64, Int64, Int64}

julia> typeof((1.0, :x, (1, 2)))
Tuple{Float64, Symbol, Tuple{Int64, Int64}}
```

В отличие от других типов данных, типы `Tuple` являются **ковариантными** . Другие типы данных в Юлии обычно являются инвариантными. Таким образом,

```
julia> Tuple{Int, Int} <: Tuple{Number, Number}
true

julia> Vector{Int} <: Vector{Number}
false
```

Это так, потому что везде принят `Tuple{Number, Number}` , так же будет `Tuple{Int, Int}` , так как он также имеет два элемента, оба из которых являются числами. Это не относится к `Vector{Int}` и `Vector{Number}` , так как функция, принимающая `Vector{Number}` может захотеть сохранить плавающую точку (например, `1.0`) или комплексное число (например, `1+3im`) в таких вектор.

Ковариация типов кортежей означает, что `Tuple{Number}` (опять же, в отличие от `Vector{Number}`) фактически является абстрактным типом:

```
julia> isleatype(Tuple{Number})
false

julia> isleatype(Vector{Number})
true
```

Конкретные подтипы `Tuple{Number}` включают в себя `Tuple{Int}` , `Tuple{Float64}` , `Tuple{Rational{BigInt}}` и т. Д.

Типы `Tuple` могут содержать завершающий `Vararg` качестве последнего параметра для указания неопределенного количества объектов. Например, `Tuple{Vararg{Int}}` является типом всех кортежей, содержащих любое число `Int` s, возможно, ноль:

```
julia> isa((), Tuple{Vararg{Int}})
```



```

true

julia> isa((1,), Tuple{Vararg{Int}})
true

julia> isa((1,2,3,4,5), Tuple{Vararg{Int}})
true

julia> isa((1.0,), Tuple{Vararg{Int}})
false

```

тогда как `Tuple{String, Vararg{Int}}` принимает кортежи, состоящие из строки, за которой следует любое число (возможно, ноль) `Int` `s`.

```

julia> isa(("x", 1, 2), Tuple{String, Vararg{Int}})
true

julia> isa((1, 2), Tuple{String, Vararg{Int}})
false

```

В сочетании с совпадением это означает, что `Tuple{Vararg{Any}}` описывает любой кортеж. Действительно, `Tuple{Vararg{Any}}` - это еще один способ сказать `Tuple` :

```

julia> Tuple{Vararg{Any}} == Tuple
true

```

`Vararg` принимает второй параметр числового типа, указывающий, сколько раз должен быть точно его первый параметр типа. (По умолчанию, если этот параметр не задан, этот второй тип-тип является `typevar`, который может принимать любое значение, поэтому любое количество `Int` `s` принимается в `Vararg`.) Типы `Tuple` заканчивающиеся на заданный `Vararg`, автоматически будут расширены до запрошенное количество элементов:

```

julia> Tuple{String,Vararg{Int, 3}}
Tuple{String,Int64,Int64,Int64}

```

Обозначение существует для однородных кортежей с заданным `Vararg : NTuple{N, T}`. В этих обозначениях `N` обозначает количество элементов в кортеже, а `T` обозначает принятый тип. Например,

```

julia> NTuple{3, Int}
Tuple{Int64,Int64,Int64}

julia> NTuple{10, Int}
NTuple{10,Int64}

julia> ans.types
svec{Int64,Int64,Int64,Int64,Int64,Int64,Int64,Int64,Int64,Int64}

```

Обратите внимание, что `NTuple` за пределами определенного размера показаны просто как `NTuple{N, T}` вместо расширенной формы `Tuple`, но они все те же:

```
julia> Tuple{Int,Int,Int,Int,Int,Int,Int,Int,Int,Int,Int}
NTuple{10,Int64}
```

Отправка по типам кортежей

Поскольку списки параметров функции Julia сами являются кортежами, [диспетчеризация](#) различных типов кортежей часто проще выполнять самими параметрами метода, часто с либеральным использованием для оператора «splating» ... Например, рассмотрим реализацию `reverse` для кортежей, начиная с `Base` :

```
revargs() = ()
revargs(x, r...) = (revargs(r...)..., x)

reverse(t::Tuple) = revargs(t...)
```

Таким образом, реализация методов на кортежах сохраняет [стабильность типов](#) , что имеет решающее значение для производительности. Мы видим, что для этого подхода нет накладных расходов с использованием макроса `@code_warntype` :

```
julia> @code_warntype reverse((1, 2, 3))
Variables:
  #self#::Base.#reverse
  t::Tuple{Int64,Int64,Int64}

Body:
  begin
    SSAValue(1) = (Core.getfield)(t::Tuple{Int64,Int64,Int64},2)::Int64
    SSAValue(2) = (Core.getfield)(t::Tuple{Int64,Int64,Int64},3)::Int64
    return
  (Core.tuple)(SSAValue(2),SSAValue(1),(Core.getfield)(t::Tuple{Int64,Int64,Int64},1)::Int64)::Tuple{Int64,Int64,Int64}

  end::Tuple{Int64,Int64,Int64}
```

Хотя это довольно сложно прочитать, здесь просто создается новый кортеж со значениями 3-го, 2-го и 1-го элементов исходного кортежа соответственно. На многих машинах это сводится к чрезвычайно эффективному LLVM-коду, который состоит из загрузок и магазинов.

```
julia> @code_llvm reverse((1, 2, 3))

define void @julia_reverse_71456([3 x i64]* noalias sret, [3 x i64]*) #0 {
top:
  %2 = getelementptr inbounds [3 x i64], [3 x i64]* %1, i64 0, i64 1
  %3 = getelementptr inbounds [3 x i64], [3 x i64]* %1, i64 0, i64 2
  %4 = load i64, i64* %3, align 1
  %5 = load i64, i64* %2, align 1
  %6 = getelementptr inbounds [3 x i64], [3 x i64]* %1, i64 0, i64 0
  %7 = load i64, i64* %6, align 1
  %.sroa.0.0..sroa_idx = getelementptr inbounds [3 x i64], [3 x i64]* %0, i64 0, i64 0
  store i64 %4, i64* %.sroa.0.0..sroa_idx, align 8
  %.sroa.2.0..sroa_idx1 = getelementptr inbounds [3 x i64], [3 x i64]* %0, i64 0, i64 1
  store i64 %5, i64* %.sroa.2.0..sroa_idx1, align 8
```

```
%.sroa.3.0..sroa_idx2 = getelementptr @inbounds [3 x i64], [3 x i64]* %0, i64 0, i64 2
store i64 %7, i64* %.sroa.3.0..sroa_idx2, align 8
ret void
}
```

Множественные возвращаемые значения

Кортежи часто используются для множественных возвращаемых значений. Большая часть стандартной библиотеки, включая две функции **итерируемого интерфейса** (`next` и `done`), возвращает кортежи, содержащие два связанных, но разных значения.

Скобки вокруг кортежей могут быть опущены в определенных ситуациях, что упрощает реализацию нескольких возвращаемых значений. Например, мы можем создать функцию, возвращающую как положительные, так и отрицательные квадратные корни действительного числа:

```
julia> pmsqrt(x::Real) = sqrt(x), -sqrt(x)
pmsqrt (generic function with 1 method)

julia> pmsqrt(4)
(2.0, -2.0)
```

Назначение **Destructuring** можно использовать для распаковки множественных возвращаемых значений. Чтобы сохранить квадратные корни в переменных `a` и `b`, достаточно написать:

```
julia> a, b = pmsqrt(9.0)
(3.0, -3.0)

julia> a
3.0

julia> b
-3.0
```

Другим примером этого являются функции `divrem` и `fldmod`, которые выполняют одно **целое (усечение или перекрытие, соответственно) деления** и операции `divrem` в `divrem` и то же время:

```
julia> q, r = divrem(10, 3)
(3, 1)

julia> q
3

julia> r
1
```

Прочитайте Кортеж онлайн: <https://riptutorial.com/ru/julia-lang/topic/6675/кортеж>

глава 16: Массивы

Синтаксис

- [1,2,3]
- [1 2 3]
- [1 2 3; 4 5 6; 7 8 9]
- Массив (тип, dims ...)
- (тип, тускнеет ...)
- нули (тип, тускне ...)
- истины (тип, тускнеет ...)
- falses (type, dims ...)
- push! (A, x)
- поп! (A)
- unshift! (A, x)
- сдвиг! (A)

параметры

параметры	замечания
За	push! (A, x) , unshift! (A, x)
A	Массив для добавления.
x	Элемент для добавления в массив.

Examples

Ручная конструкция простого массива

Можно инициализировать массив Julia вручную, используя синтаксис квадратных скобок:

```
julia> x = [1, 2, 3]
3-element Array{Int64,1}:
 1
 2
 3
```

Первая строка после команды показывает размер созданного массива. Он также показывает тип его элементов и его размерность (в этом случае `Int64` и `1` , соответственно). Для двумерного массива можно использовать пробелы и точки с запятой:

```
julia> x = [1 2 3; 4 5 6]
2x3 Array{Int64,2}:
 1  2  3
 4  5  6
```

Чтобы создать неинициализированный массив, вы можете использовать метод `Array(type, dims...)` :

```
julia> Array{Int64, 3, 3}
3x3 Array{Int64,2}:
 0  0  0
 0  0  0
 0  0  0
```

Функции `zeros` , `ones` , `true`s , `false`s имеют методы, которые ведут себя точно так же, но создают массивы, полные `0.0` , `1.0` , `True` или `False` соответственно.

Типы массивов

В Julia массивы имеют типы, параметризованные двумя переменными: тип `T` и размерность `D` (`Array{T, D}`). Для одномерного массива целых чисел тип:

```
julia> x = [1, 2, 3];
julia> typeof(x)
Array{Int64, 1}
```

Если массив является двумерной матрицей, `D` равно 2:

```
julia> x = [1 2 3; 4 5 6; 7 8 9]
julia> typeof(x)
Array{Int64, 2}
```

Тип элемента также может быть абстрактным:

```
julia> x = [1 2 3; 4 5 "6"; 7 8 9]
3x3 Array{Any,2}:
 1  2  3
 4  5  "6"
 7  8  9
```

Здесь `Any` (абстрактный тип) является типом результирующего массива.

Указание типов при создании массивов

Когда мы создаем `Array` так, как описано выше, Julia сделает все возможное, чтобы вывести нужный тип, который нам может понадобиться. В начальных примерах выше мы вводили входы, которые выглядели как целые числа, и поэтому Джулия по умолчанию `Int64` тип `Int64` по умолчанию. Время от времени, однако, мы могли бы быть более конкретными. В следующем примере мы указываем, что мы хотим, чтобы тип был вместо `Int8` :

```
x1 = Int8[1 2 3; 4 5 6; 7 8 9]
typeof(x1)  ## Array{Int8,2}
```

Мы могли бы даже указать тип как-то наподобие `Float64` , даже если мы записываем входы таким образом, который по умолчанию мог бы интерпретироваться как целые числа (например, писать `1` вместо `1.0`). например

```
x2 = Float64[1 2 3; 4 5 6; 7 8 9]
```

Массивы массивов - свойства и конструкция

В Julia вы можете иметь массив, содержащий другие объекты типа `Array`. Рассмотрим следующие примеры инициализации различных типов массивов:

```
A = Array{Float64}(10,10)  # A single Array, dimensions 10 by 10, of Float64 type objects

B = Array{Array}(10,10,10) # A 10 by 10 by 10 Array. Each element is an Array of unspecified
                             type and dimension.

C = Array{Array{Float64}}(10)  ## A length 10, one-dimensional Array. Each element is an
                                Array of Float64 type objects but unspecified dimensions

D = Array{Array{Float64, 2}}(10)  ## A length 10, one-dimensional Array. Each element of is
                                an 2 dimensional array of Float 64 objects
```

Рассмотрим, например, различия между `C` и `D` здесь:

```
julia> C[1] = rand(3)
3-element Array{Float64,1}:
 0.604771
 0.985604
 0.166444

julia> D[1] = rand(3)
ERROR: MethodError:
```

`rand(3)` создает объект типа `Array{Float64,1}` . Поскольку единственная спецификация для элементов `C` состоит в том, что они являются массивами с элементами типа `Float64`, это соответствует определению `C` Но для `D` мы указали, что элементы должны быть 2 мерными массивами. Таким образом, поскольку `rand(3)` не создает двумерный массив, мы не можем использовать его для назначения значения определенному элементу `D`

Укажите конкретные размеры массивов в массиве

Хотя мы можем указать, что `Array` будет содержать элементы, которые имеют тип `Array`, и мы можем указать, что, например, эти элементы должны быть двумерными массивами, мы не можем напрямую указывать размеры этих элементов. Например, мы не можем напрямую указать, что мы хотим, чтобы `Array` держал 10 массивов, каждый из которых составлял 5,5. Мы можем видеть это из синтаксиса для функции `Array()` используемой для построения

массива:

Массив {T} (тускнеет)

строит неинициализированный плотный массив с типом элемента T. dims может быть кортежем или серией целочисленных аргументов. Синтаксис Array (T, dims) также доступен, но устарел.

Тип массива в Джулии охватывает количество измерений, но не размер этих размеров. Таким образом, в этом синтаксисе нет места для указания точных измерений. Тем не менее аналогичный эффект может быть достигнут с использованием понимания Array:

```
E = [Array{Float64}(5,5) for idx in 1:10]
```

Примечание: эта документация отражает следующий [SO-ответ](#)

Инициализировать пустой массив

Мы можем использовать [] для создания пустого массива в Джулии. Самый простой пример:

```
A = [] # 0-element Array{Any,1}
```

Массивы типа Any, как правило, не выполняются так же, как те, которые имеют заданный тип. Так, например, мы можем использовать:

```
B = Float64[] ## 0-element Array{Float64,1}
C = Array{Float64}[] ## 0-element Array{Array{Float64,N},1}
D = Tuple{Int, Int}[] ## 0-element Array{Tuple{Int64,Int64},1}
```

См. [Инициализация пустого массива кортежей в Джулии](#) для источника последнего примера.

векторы

Векторы представляют собой одномерные массивы и поддерживают в основном тот же интерфейс, что и их многомерные копии. Однако векторы также поддерживают дополнительные операции.

Во-первых, заметим, что Vector{T} где T - некоторый тип, означает то же самое, что и Array{T,1}.

```
julia> Vector{Int}
Array{Int64,1}

julia> Vector{Float64}
Array{Float64,1}
```

Один читает `Array{Int64,1}` как «одномерный массив `Int64` ».

В отличие от многомерных массивов, векторы могут быть изменены. Элементы могут быть добавлены или удалены из передней или задней части вектора. Эти операции являются **постоянным временем амортизации** .

```
julia> A = [1, 2, 3]
3-element Array{Int64,1}:
 1
 2
 3
```

```
julia> push!(A, 4)
4-element Array{Int64,1}:
 1
 2
 3
 4
```

```
julia> A
4-element Array{Int64,1}:
 1
 2
 3
 4
```

```
julia> pop!(A)
4
```

```
julia> A
3-element Array{Int64,1}:
 1
 2
 3
```

```
julia> unshift!(A, 0)
4-element Array{Int64,1}:
 0
 1
 2
 3
```

```
julia> A
4-element Array{Int64,1}:
 0
 1
 2
 3
```

```
julia> shift!(A)
0
```

```
julia> A
3-element Array{Int64,1}:
 1
 2
 3
```

Как принято, каждая из этих функций `push!` , `pop!` , `unshift!` , и `shift!` заканчивается

восклицательным знаком, чтобы указать, что они мутируют их аргументы. Функции `push!` и `unshift!` верните массив, тогда как `pop!` и `shift!` вернуть элемент.

конкатенация

Часто бывает полезно создавать матрицы из меньших матриц.

Горизонтальная конкатенация

Матрицы (и векторы, которые рассматриваются как векторы столбцов) могут быть объединены по горизонтали с использованием функции `hcat`.

```
julia> hcat([1 2; 3 4], [5 6 7; 8 9 10], [11, 12])
2×6 Array{Int64,2}:
 1  2  5  6  7 11
 3  4  8  9 10 12
```

Существует удобный синтаксис, использующий квадратные обозначения и пробелы:

```
julia> [[1 2; 3 4] [5 6 7; 8 9 10] [11, 12]]
2×6 Array{Int64,2}:
 1  2  5  6  7 11
 3  4  8  9 10 12
```

Это обозначение может точно соответствовать обозначениям для матричных матриц, используемых в линейной алгебре:

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> B = [5 6; 7 8]
2×2 Array{Int64,2}:
 5  6
 7  8

julia> [A B]
2×4 Array{Int64,2}:
 1  2  5  6
 3  4  7  8
```

Обратите внимание, что вы не можете горизонтально конкатенировать одну матрицу с использованием синтаксиса `[]`, поскольку вместо этого будет создан одноэлементный вектор матриц:

```
julia> [A]
1-element Array{Array{Int64,2},1}:
 [1 2; 3 4]
```

Вертикальная конкатенация

Вертикальная конкатенация похожа на горизонтальную конкатенацию, но в вертикальном направлении. Функция вертикальной конкатенации - `vcat` .

```
julia> vcat([1 2; 3 4], [5 6; 7 8; 9 10], [11 12])
6×2 Array{Int64,2}:
 1  2
 3  4
 5  6
 7  8
 9 10
11 12
```

Альтернативно, квадратные скобки могут использоваться с точкой с запятой ; как разделитель:

```
julia> [[1 2; 3 4]; [5 6; 7 8; 9 10]; [11 12]]
6×2 Array{Int64,2}:
 1  2
 3  4
 5  6
 7  8
 9 10
11 12
```

Векторы также могут быть объединены по вертикали; результатом является вектор:

```
julia> A = [1, 2, 3]
3-element Array{Int64,1}:
 1
 2
 3

julia> B = [4, 5]
2-element Array{Int64,1}:
 4
 5

julia> [A; B]
5-element Array{Int64,1}:
 1
 2
 3
 4
 5
```

Горизонтальная и вертикальная конкатенация могут комбинироваться:

```
julia> A = [1 2
           3 4]
2×2 Array{Int64,2}:
 1  2
 3  4
```

```
julia> B = [5 6 7]
1×3 Array{Int64,2}:
 5  6  7

julia> C = [8, 9]
2-element Array{Int64,1}:
 8
 9

julia> [A C; B]
3×3 Array{Int64,2}:
 1  2  8
 3  4  9
 5  6  7
```

Прочитайте Массивы онлайн: <https://riptutorial.com/ru/julia-lang/topic/5437/массивы>

глава 17: Метaproграммирование

Синтаксис

- имя макроса (ex) ... end
- цитата ... конец
- : (...)
- \$ x
- Meta.quot (x)
- QuoteNode (x)
- ESC (x)

замечания

Функции метапрограммирования Джулии в значительной степени вдохновлены теми же Lisp-подобными языками и будут казаться знакомыми тем, у кого есть фон Лиспа. Метапрограммирование очень мощное. При правильном использовании это может привести к получению более сжатого и читаемого кода.

`quote ... end` - синтаксис квазикота. Вместо выражений внутри оценки они просто анализируются. Значение выражения `quote ... end` - это результирующее синтаксическое дерево (AST).

Синтаксис `: (...)` аналогичен синтаксису `quote ... end`, но он более легкий. Этот синтаксис более краток, чем `quote ... end`.

Внутри квазиквадрата оператор `$` является специальным и *интерполирует* его аргумент в AST. Ожидается, что аргумент будет выражением, которое сплайсируется непосредственно в AST.

Функция `Meta.quot (x)` цитирует свой аргумент. Это часто полезно в сочетании с использованием `$` for интерполяции, так как позволяет выражениям и символам спланировать буквально в AST.

Examples

Повторное выполнение макроса @show

В Julia макрос `@show` часто полезен для целей отладки. Он отображает как выражение, подлежащее оценке, так и его результат, и, наконец, возвращает значение результата:

```
julia> @show 1 + 1
```

```
1 + 1 = 2
2
```

Прямо создать собственную версию `@show` :

```
julia> macro myshow(expression)
    quote
        value = $expression
        println($(Meta.quot(expression)), " = ", value)
        value
    end
end
```

Чтобы использовать новую версию, просто используйте макрос `@myshow` :

```
julia> x = @myshow 1 + 1
1 + 1 = 2
2

julia> x
2
```

До цикла

Мы все привыкли к синтаксису `while` , который выполняет свое тело, а условие оценивается как `true` . Что делать , если мы хотим реализовать `until` цикла, который выполняет цикл , пока условие оценивается в `true` ?

В Julia мы можем сделать это, создав макрос `@until` , который перестает выполнять свое тело при выполнении условия:

```
macro until(condition, expression)
    quote
        while !($condition)
            $expression
        end
    end |> esc
end
```

Здесь мы использовали синтаксис цепочки функций `|>` , который эквивалентен вызову функции `esc` во всем кадре `quote` . Функция `esc` предотвращает применение макрогиды к содержимому макроса; без него переменные, скопированные в макрос, будут переименованы для предотвращения столкновений с внешними переменными. Для получения дополнительной информации см. Документацию Julia о [макрогигалии](#) .

Вы можете использовать более одного выражения в этом цикле, просто поместив все в `begin ... end` block:

```
julia> i = 0;
```

```
julia> @until i == 10 begin
    println(i)
    i += 1
end
0
1
2
3
4
5
6
7
8
9

julia> i
10
```

QuoteNode, Meta.quot и Expr (: quote)

Есть три способа процитировать что-то, используя функцию Julia:

```
julia> QuoteNode(:x)
:(:x)

julia> Meta.quot(:x)
:(:x)

julia> Expr(:quote, :x)
:(:x)
```

Что означает «цитирование», и для чего это полезно? Цитирование позволяет нам защищать выражения от интерпретации Джулией специальных форм. Обычный пример использования - это когда мы генерируем **выражения**, которые должны содержать вещи, которые оцениваются символами. (Например, **этот макрос** должен вернуть выражение, которое оценивает символ.) Это не работает просто для возврата символа:

```
julia> macro mysym(); :x; end
@mysym (macro with 1 method)

julia> @mysym
ERROR: UndefVarError: x not defined

julia> macroexpand(:(@mysym))
:x
```

Что тут происходит? `@mysym` расширяет до `:x`, который как выражение интерпретируется как переменная `x`. Но ничто не было назначено для `x` еще, поэтому мы получаем `x not defined` ошибку.

Чтобы обойти это, мы должны привести результат нашего макроса:

```
julia> macro mysym2(); Meta.quot(:x); end
```

```
@mysym2 (macro with 1 method)

julia> @mysym2
:x

julia> macroexpand(:(@mysym2))
:(:x)
```

Здесь мы использовали функцию `Meta.quot` чтобы превратить наш символ в `Meta.quot` символ, который мы хотим.

В чем разница между `Meta.quot` и `QuoteNode`, и какой я должен использовать? Почти во всех случаях разница не имеет большого значения. Иногда, возможно, немного безопаснее использовать `QuoteNode` вместо `Meta.quot`. Однако изучение различий является информативным в том, как работают выражения Julia и макросы.

Разница между `Meta.quot` и `QuoteNode`, объясняется

Вот эмпирическое правило:

- Если вам нужно или хотите поддерживать интерполяцию, используйте `Meta.quot` ;
- Если вы не можете или не хотите разрешать интерполяцию, используйте `QuoteNode` .

Короче говоря, разница в том, что `Meta.quot` допускает интерполяцию в цитируемой `QuoteNode`, а `QuoteNode` защищает ее аргумент от любой интерполяции. Чтобы понять интерполяцию, важно упомянуть выражение `$`. В Джулии есть выражение, называемое выражением `$`. Эти выражения позволяют ускользнуть. Например, рассмотрим следующее выражение:

```
julia> ex = :( x = 1; :($x + $x) )
quote
  x = 1
  $(Expr(:quote, :($ (Expr(:$, :x)) + $(Expr(:$, :x))))
end
```

При оценке это выражение будет оценивать `1` и назначать его `x`, а затем строить выражение вида `_ + _` где `_` будет заменено значением `x`. Таким образом, результатом этого должно быть *выражение* `1 + 1` (которое еще не оценено и поэтому отличается от *значения* `2`). Действительно, это так:

```
julia> eval(ex)
:(1 + 1)
```

Скажем, теперь мы пишем макрос для создания таких выражений. Наш макрос примет аргумент, который заменит `1` в `ex` выше. Разумеется, этот аргумент может быть любым выражением. Вот что-то, чего мы не хотим:

```
julia> macro makeex(arg)
    quote
        :( x = $(esc($arg)); :($x + $x) )
    end
end
@makeex (macro with 1 method)

julia> @makeex 1
quote
    x = $(Expr(:escape, 1))
    $(Expr(:quote, :($(Expr(:$, :x)) + $(Expr(:$, :x)))))
end

julia> @makeex 1 + 1
quote
    x = $(Expr(:escape, 2))
    $(Expr(:quote, :($(Expr(:$, :x)) + $(Expr(:$, :x)))))
end
```

Второй случай неверен, потому что мы должны оставить `1 + 1` необоснованным. Мы исправим это, указав аргумент с `Meta.quot` :

```
julia> macro makeex2(arg)
    quote
        :( x = $$ (Meta.quot (arg)); :($x + $x) )
    end
end
@makeex2 (macro with 1 method)

julia> @makeex2 1 + 1
quote
    x = 1 + 1
    $(Expr(:quote, :($(Expr(:$, :x)) + $(Expr(:$, :x)))))
end
```

Макро-гигиена не относится к содержанию цитаты, поэтому в этом случае экранирование не обязательно (и на самом деле не является законным).

Как упоминалось ранее, `Meta.quot` позволяет интерполяцию. Итак, давайте попробуем:

```
julia> @makeex2 1 + $(sin(1))
quote
    x = 1 + 0.8414709848078965
    $(Expr(:quote, :($(Expr(:$, :x)) + $(Expr(:$, :x)))))
end

julia> let q = 0.5
    @makeex2 1 + $q
end
quote
    x = 1 + 0.5
    $(Expr(:quote, :($(Expr(:$, :x)) + $(Expr(:$, :x)))))
end
```

Из первого примера мы видим, что интерполяция позволяет нам встраивать `sin(1)` вместо того, чтобы выражение было буквальным `sin(1)` . Второй пример показывает, что эта

интерполяция выполняется в области подсчета макросов, а не в собственной области макросов. Это потому, что наш макрос фактически не оценил какой-либо код; все, что он делает, это генерировать код. Оценка кода (который делает его путь в выражении) выполняется, когда выражение, которое генерирует макрос, фактически выполняется.

Что, если бы мы вместо этого использовали `QuoteNode` ? Как вы можете догадаться, поскольку `QuoteNode` предотвращает интерполяцию вообще, это означает, что это не сработает.

```
julia> macro makeex3(arg)
    quote
        :( x = $$ (QuoteNode(arg)); :($x + $x) )
    end
end
@makeex3 (macro with 1 method)

julia> @makeex3 1 + $(sin(1))
quote
    x = 1 + $(Expr(:$, :(sin(1))))
    $(Expr(:quote, :($ (Expr(:$, :x)) + $(Expr(:$, :x)))))
end

julia> let q = 0.5
    @makeex3 1 + $q
end
quote
    x = 1 + $(Expr(:$, :q))
    $(Expr(:quote, :($ (Expr(:$, :x)) + $(Expr(:$, :x)))))
end

julia> eval(@makeex3 $(sin(1)))
ERROR: unsupported or misplaced expression $
in eval(::Module, ::Any) at ./boot.jl:234
in eval(::Any) at ./boot.jl:233
```

В этом примере мы можем согласиться с тем, что `Meta.quot` дает большую гибкость, поскольку позволяет интерполяцию. Так почему мы можем когда-нибудь рассмотреть использование `QuoteNode` ? В некоторых случаях мы не можем на самом деле желать интерполяции и на самом деле хотим выражение буквального `$` . Когда это было бы желательно? Рассмотрим обобщение `@makeex` где мы можем передать дополнительные аргументы, определяющие, что происходит слева и справа от знака `+` :

```
julia> macro makeex4(expr, left, right)
    quote
        quote
            $$ (Meta.quot (expr) )
            : ($$(Meta.quot (left)) + $$$ (Meta.quot (right)))
        end
    end
end
@makeex4 (macro with 1 method)

julia> @makeex4 x=1 x x
quote # REPL[110], line 4:
```

```

x = 1 # REPL[110], line 5:
$(Expr(:quote, :($(Expr(:$, :x)) + $(Expr(:$, :x)))))
end

julia> eval(ans)
:(1 + 1)

```

Ограничением нашей реализации `@makeex4` является то, что мы не можем использовать выражения как левую и правую стороны выражения напрямую, потому что они интерполируются. Другими словами, выражения могут быть оценены для интерполяции, но мы можем захотеть их сохранить. (Поскольку здесь имеется много уровней цитирования и оценки, давайте выясним: наш макрос генерирует *код*, который строит *выражение*, которое при оценке производит другое *выражение*. Фу!)

```

julia> @makeex4 x=1 x/2 x
quote # REPL[110], line 4:
  x = 1 # REPL[110], line 5:
  $(Expr(:quote, :($(Expr(:$, : (x / 2))) + $(Expr(:$, :x)))))
end

julia> eval(ans)
:(0.5 + 1)

```

Мы должны разрешить пользователю указывать, когда произойдет интерполяция, и когда это не должно. Теоретически это легко исправить: мы можем просто удалить один из знаков `$` в нашем приложении и позволить пользователю внести свой вклад. Это означает, что мы интерполируем цитированную версию выражения, введенного пользователем (которое мы уже цитировали и интерполировали один раз). Это приводит к следующему коду, который может быть немного запутанным вначале из-за множественных вложенных уровней цитирования и нечеткости. Попробуйте прочитать и понять, для чего каждый побег.

```

julia> macro makeex5(expr, left, right)
    quote
        quote
            $$ (Meta.quot (expr))
            : ($$ (Meta.quot ($ (Meta.quot (left)))) + $$ (Meta.quot ($ (Meta.quot (right)))))
        end
    end
end

@makeex5 (macro with 1 method)

julia> @makeex5 x=1 1/2 1/4
quote # REPL[121], line 4:
  x = 1 # REPL[121], line 5:
  $(Expr(:quote, :($(Expr(:$, :($ (Expr(:quote, :(1 / 2)))))) + $(Expr(:$, :($ (Expr(:quote,
:(1 / 4))))))))))
end

julia> eval(ans)
:(1 / 2 + 1 / 4)

julia> @makeex5 y=1 $y $y

```

```
ERROR: UndefVarError: y not defined
```

Все началось хорошо, но что-то пошло не так. Созданный макросом код пытается интерполировать копию `y` в области вызова макроса; но *нет* копии `y` в области подсчета макросов. Наша ошибка позволяет интерполяцию со вторым и третьим аргументами в макросе. Чтобы исправить эту ошибку, мы должны использовать `QuoteNode`.

```
julia> macro makeex6(expr, left, right)
    quote
        quote
            $$ (Meta.quot (expr))
            : ($$ (Meta.quot ($ (QuoteNode (left)))) + $$ (Meta.quot ($ (QuoteNode (right))))
        end
    end
end
@makeex6 (macro with 1 method)

julia> @makeex6 y=1 1/2 1/4
quote # REPL[129], line 4:
    y = 1 # REPL[129], line 5:
        $(Expr(:quote, :($ (Expr(:$, :($ (Expr(:quote, :(1 / 2)))))) + $(Expr(:$, :($ (Expr(:quote,
:(1 / 4))))))))))
end

julia> eval(ans)
:(1 / 2 + 1 / 4)

julia> @makeex6 y=1 $y $y
quote # REPL[129], line 4:
    y = 1 # REPL[129], line 5:
        $(Expr(:quote, :($ (Expr(:$, :($ (Expr(:quote, :($ (Expr(:$, :y)))))))) + $(Expr(:$,
:($ (Expr(:quote, :($ (Expr(:$, :y))))))))))
end

julia> eval(ans)
:(1 + 1)

julia> @makeex6 y=1 1+$y $y
quote # REPL[129], line 4:
    y = 1 # REPL[129], line 5:
        $(Expr(:quote, :($ (Expr(:$, :($ (Expr(:quote, :(1 + $(Expr(:$, :y)))))))) + $(Expr(:$,
:($ (Expr(:quote, :($ (Expr(:$, :y))))))))))
end

julia> @makeex6 y=1 $y/2 $y
quote # REPL[129], line 4:
    y = 1 # REPL[129], line 5:
        $(Expr(:quote, :($ (Expr(:$, :($ (Expr(:quote, :($ (Expr(:$, :y)) / 2)))))) + $(Expr(:$,
:($ (Expr(:quote, :($ (Expr(:$, :y))))))))))
end

julia> eval(ans)
:(1 / 2 + 1)
```

Используя `QuoteNode`, мы защитили наши аргументы от интерполяции. Поскольку `QuoteNode` обладает только дополнительными защитами, никогда не вредно использовать `QuoteNode`, если вы не хотите интерполяции. Однако понимание разницы позволяет понять, где и

почему `Meta.quot` может быть лучшим выбором.

Это длительное упражнение представляет собой пример, который слишком сложный, чтобы проявляться в любом разумном приложении. Поэтому мы обосновали следующее эмпирическое правило, упомянутое ранее:

- Если вам нужно или хотите поддерживать интерполяцию, используйте `Meta.quot` ;
- Если вы не можете или не хотите разрешать интерполяцию, используйте `QuoteNode` .

Что относительно `Expr (: quote)`?

`Expr (: quote, x)` эквивалентен `Meta.quot (x)` . Однако последний более идиоматичен и является предпочтительным. Для кода, который сильно использует метапрограммирование, часто используется линия `using Base.Meta` , которая позволяет `Meta.quot` упоминаться как просто `quot` .

Руководство

Метапрограммирование метаданных & bobs

Цели:

- Учите минимальные целевые функциональные / полезные / не абстрактные примеры (например, `@swap` или `@assert`), которые вводят концепции в подходящих контекстах
- Предпочитаете, чтобы код иллюстрировал / демонстрировал концепции, а не параграфы объяснения
- Избегайте связывания «требуемого чтения» с другими страницами - это прерывает повествование
- Представьте вещи в разумном порядке, которые облегчат обучение

Ресурсы:

julialang.org

[wikibook \(@Cormullion\)](#)

[5 слоев \(Leah Hanson\)](#)

[Котировка SO-Doc \(@TotalVerb\)](#)

[SO-Doc - Символы, которые не являются юридическими идентификаторами \(@TotalVerb\)](#)

[SO: Что такое символ в Джулии \(@StefanKarpinski\)](#)

[Дискурсная нить \(@ pi-\) Метапрограммирование](#)

Большая часть материала пришла из канала дискурса, большая часть которого исходит от fcard ... пожалуйста, подтачивайте меня, если я забыл атрибуты.

Условное обозначение

```
julia> mySymbol = Symbol("myName") # or 'identifier'
:myName

julia> myName = 42
42

julia> mySymbol |> eval # 'foo |> bar' puts output of 'foo' into 'bar', so 'bar(foo)'
42

julia> :( $mySymbol = 1 ) |> eval
1

julia> myName
1
```

Передача флагов в функции:

```
function dothing(flag)
    if flag == :thing_one
        println("did thing one")
    elseif flag == :thing_two
        println("did thing two")
    end
end
julia> dothing(:thing_one)
did thing one

julia> dothing(:thing_two)
did thing two
```

Пример hashkey:

```
number_names = Dict{Symbol, Int}()
number_names[:one] = 1
number_names[:two] = 2
number_names[:six] = 6
```

(Дополнительно) (@fcard) `:foo` aka `:(foo)` дает символ, если `foo` является допустимым идентификатором, иначе выражение.

```
# NOTE: Different use of ':' is:
julia> :mySymbol = Symbol('hello world')

# (You can create a symbol with any name with Symbol("<name>"),
# which lets us create such gems as:
julia> one_plus_one = Symbol("1 + 1")
Symbol("1 + 1")

julia> eval(one_plus_one)
```

```

ERROR: UndefVarError: 1 + 1 not defined
...

julia> valid_math = :($one_plus_one = 3)
:(1 + 1 = 3)

julia> one_plus_one_plus_two = :($one_plus_one + 2)
:(1 + 1 + 2)

julia> eval(quote
    $valid_math
    @show($one_plus_one_plus_two)
end)
1 + 1 + 2 = 5
...

```

В основном вы можете рассматривать Символы как легкие строки. Это не то, за что они предназначены, но вы можете это сделать, так почему бы и нет. Сама Julia's Base делает это, `print_with_color(:red, "abc")` печатает красную абстракцию.

Expr (AST)

(Почти) все в Юлии - это выражение, то есть экземпляр `Expr`, который будет содержать [AST](#).

```

# when you type ...
julia> 1+1
2

# Julia is doing: eval(parse("1+1"))
# i.e. First it parses the string "1+1" into an `Expr` object ...
julia> ast = parse("1+1")
:(1 + 1)

# ... which it then evaluates:
julia> eval(ast)
2

# An Expr instance holds an AST (Abstract Syntax Tree). Let's look at it:
julia> dump(ast)
Expr
  head: Symbol call
  args: Array{Any}((3,))
    1: Symbol +
    2: Int64 1
    3: Int64 1
  typ: Any

# TRY: fieldnames(typeof(ast))

julia>      :(a + b*c + 1) ==
      parse("a + b*c + 1") ==
      Expr(:call, :+, :a, Expr(:call, :*, :b, :c), 1)
true

```

Expr :

```
julia> dump( :(1+2/3) )
Expr
  head: Symbol call
  args: Array{Any}((3,))
    1: Symbol +
    2: Int64 1
    3: Expr
      head: Symbol call
      args: Array{Any}((3,))
        1: Symbol /
        2: Int64 2
        3: Int64 3
      typ: Any
  typ: Any

# Tidier rep'n using s-expr
julia> Meta.show_sexpr( :(1+2/3) )
(:call, :+, 1, (:call, :/, 2, 3))
```

МНОГОСТРОЧНЫЙ `Expr` `s` с использованием `quote`

```
julia> blk = quote
      x=10
      x+1
    end
quote # REPL[121], line 2:
  x = 10 # REPL[121], line 3:
  x + 1
end

julia> blk == :( begin x=10; x+1 end )
true

# Note: contains debug info:
julia> Meta.show_sexpr(blk)
(:block,
 (:line, 2, Symbol("REPL[121]")),
 (:(=), :x, 10),
 (:line, 3, Symbol("REPL[121]")),
 (:call, :+, :x, 1)
)

# ... unlike:
julia> noDbg = :( x=10; x+1 )
quote
  x = 10
  x + 1
end
```

... так что `quote` функционально одинакова, но предоставляет дополнительную информацию об отладке.

(*) **СОВЕТ:** Используйте `let` держать `x` внутри блока

`quote` **ИЗ** `quote`

`Expr(:quote, x)` используется для представления котировок в кавычках.

```
Expr(:quote, :(x + y)) == :(:(x + y))

Expr(:quote, Expr(:$, :x)) == :(:($x))
```

`QuoteNode(x)` похож на `Expr(:quote, x)` но он предотвращает интерполяцию.

```
eval(Expr(:quote, Expr(:$, 1))) == 1

eval(QuoteNode(Expr(:$, 1))) == Expr(:$, 1)
```

([Разберитесь с различными механизмами цитирования в метапрограмме Джулии](#))

Существуют ли \$ и : (...) как-то обратные друг от друга?

`:(foo)` означает «не смотрите на значение, посмотрите на выражение» `$foo` означает «изменить выражение на его значение»

`:($(foo)) == foo . $(:(foo))` является ошибкой. `$(...)` не является операцией и ничего не делает сам по себе, это «интерполировать это!». что используется синтаксис цитирования. т.е. он существует только внутри цитаты.

Является ли \$ foo таким же, как eval(foo) ?

Нет! `$foo` обменивается на значение времени `compile`-времени `eval(foo)` означает сделать это во время выполнения

`eval` будет происходить в глобальной интерполяции.

`eval(:<expr>)` должен возвращать то же самое, что просто `<expr>` (предполагая, что `<expr>` является допустимым выражением в текущем глобальном пространстве)

```
eval(:(1 + 2)) == 1 + 2

eval(:(let x=1; x + 1 end)) == let x=1; x + 1 end
```

macro

Готовы? :)

```
# let's try to make this!
julia> x = 5; @show x;
x = 5
```

Давайте сделаем наш собственный макрос @show :


```

macro log(x)
  :(
    println( "Expression: ", $(string(x)), " has value: ", $x )
  )
end

u = 42
f = x -> x^2
@log(u)      # Expression: u has value: 42
@log(42)     # Expression: 42 has value: 42
@log(f(42))  # Expression: f(42) has value: 1764
@log(:u)     # Expression: :u has value: u

```

expand **чтобы уменьшить** Expr

5 слоев (Leah Hanson) <- объясняет, как Джулия берет исходный код в виде строки, маркирует его в Expr (AST), расширяет все макросы (все еще AST), **понижает** (понижает AST), затем преобразует в LLVM (и дальше - на данный момент нам не нужно беспокоиться о том, что лежит за пределами!)

Q: code_lowered действует на функции. Можно ли снизить Expr ? A: ууп!

```

# function -> lowered-AST
julia> code_lowered(*, (String,String))
1-element Array{LambdaInfo,1}:
LambdaInfo template for *(s1::AbstractString, ss::AbstractString...) at strings/basic.jl:84

# Expr(i.e. AST) -> lowered-AST
julia> expand(:(x ? y : z))
:(begin
    unless x goto 3
    return y
    3:
    return z
end)

julia> expand(:(y .= x.(i)))
:((Base.broadcast!)(x,y,i))

# 'Execute' AST or lowered-AST
julia> eval(ast)

```

Если вы хотите только развернуть макросы, вы можете использовать **macroexpand** :

```

# AST -> (still nonlowered-)AST but with macros expanded:
julia> macroexpand(:(@show x))
quote
    (Base.println)("x = ",(Base.repr)(begin # show.jl, line 229:
        #28#value = x
    end))
    #28#value
end

```

... который возвращает не пониженный АСТ, но со всеми макросами.

`esc()`

`esc(x)` возвращает `Expr`, в котором говорится: «Не применяйте к нему гигиенические `Expr(:escape, x)` », это то же самое, что и `Expr(:escape, x)` . Гигиена , что держит макрос самодостаточным, и вы `esc` вещи , если вы хотите, чтобы «утечки». например

Пример: макрос `swap` для иллюстрации `esc()`

```
macro swap(p, q)
  quote
    tmp = $(esc(p))
    $(esc(p)) = $(esc(q))
    $(esc(q)) = tmp
  end
end

x, y = 1, 2
@swap(x, y)
println(x, y)  # 2 1
```

`$` позволяет нам «убежать от» `quote` . Так почему бы не просто `$p` и `$q` ? т.е.

```
# FAIL!
tmp = $p
$p = $q
$q = tmp
```

Поскольку это будет выглядеть сначала в области `macro` для `p` , и он найдет локальный `p` то есть параметр `p` (да, если вы впоследствии получите доступ к `p` без `esc ing`, макрос считает параметр `p` в качестве локальной переменной).

Таким образом, `$p = ...` - это просто присвоение локальному `p` . это не влияет на передачу какой-либо переменной в вызывающий контекст.

Хорошо, а как насчет:

```
# Almost!
tmp = $p          # <-- you might think we don't
$(esc(p)) = $q    #      need to esc() the RHS
$(esc(q)) = tmp
```

Таким образом, `esc(p)` «утечка» `p` в вызывающий контекст. «То, что было передано в макрос, который **мы получаем как `p`** »,

```
julia> macro swap(p, q)
  quote
    tmp = $p
    $(esc(p)) = $q
    $(esc(q)) = tmp
  end
end
```

```

@swap (macro with 1 method)

julia> x, y = 1, 2
(1,2)

julia> @swap(x, y);

julia> @show(x, y);
x = 2
y = 1

julia> macroexpand(:(@swap(x, y)))
quote # REPL[34], line 3:
    #10#tmp = x # REPL[34], line 4:
    x = y # REPL[34], line 5:
    y = #10#tmp
end

```

Как вы можете видеть, `tmp` получает гигиеническую терапию `#10#tmp`, тогда как `x` и `y` нет. Джулия делает уникальный идентификатор для `tmp`, что вы можете вручную сделать с `gensym`, то есть:

```

julia> gensym(:tmp)
Symbol("#tmp#270")

```

Но: **есть gotcha:**

```

julia> module Swap
    export @swap

    macro swap(p, q)
        quote
            tmp = $p
            $(esc(p)) = $q
            $(esc(q)) = tmp
        end
    end
end

Swap

julia> using Swap

julia> x,y = 1,2
(1,2)

julia> @swap(x,y)
ERROR: UndefVarError: x not defined

```

Другое дело, что макро-гигиена `julia` заключается в том, что если макрос находится из другого модуля, он делает любые переменные (которые не были назначены внутри возвращаемого выражения макроса, например `tmp` в этом случае), глобальные значения текущего модуля, поэтому `$p` становится `Swap.$p`, также `$q` -> `Swap.$q`.

В общем случае, если вам нужна переменная, которая находится вне области макроса, вы должны ее использовать, поэтому вы должны использовать `esc(p)` и `esc(q)` независимо от

того, находятся ли они в LHS или RHS выражения или даже сами по себе.

люди уже упоминали `gensym` са несколько раз, и вскоре вас соблазнит темная сторона дефолта, чтобы избежать всего выражения с несколькими `gensym` s `reperered` здесь и там, но ... Удостоверьтесь, чтобы понять, как работает гигиена, прежде чем пытаться быть умнее этого! Это не очень сложный алгоритм, поэтому он не должен занять слишком много времени, но не спешите! Не используйте эту силу, пока не поймете все ее последствия ... (@fcard)

Пример: `until` макрос

(@ Исмаэль-VC)

```
"until loop"
macro until(condition, block)
  quote
    while ! $condition
      $block
    end
  end |> esc
end

julia> i=1; @until( i==5, begin; print(i); i+=1; end )
1234
```

(@fcard) `|>` является спорным. Я удивлен, что толпа еще не стала спорить. (может быть, все просто устали от этого). Существует рекомендация иметь большинство, если не все макросы просто являются вызовом функции, поэтому:

```
macro until(condition, block)
  esc(until(condition, block))
end

function until(condition, block)
  quote
    while !$condition
      $block
    end
  end
end
```

... является более безопасной альтернативой.

@ простой вызов макроса `fcard`

Задача: `swaps(1/2)` операнды, поэтому `swaps(1/2)` дают 2.00 т. 2/1

```
macro swaps(e)
  e.args[2:3] = e.args[3:-1:2]
  e
end
```

```
end
@swaps(1/2)
2.00
```

Еще больше макросов от @fcard [здесь](#)

Интерполировать и `assert` макрос

<http://docs.julialang.org/en/release-0.5/manual/metaprogramming/#building-an-advanced-macro>

```
macro assert(ex)
    return :( $ex ? nothing : throw(AssertionError($(string(ex)))) )
end
```

Q: Почему последний `$` ? А: Он интерполирует, то есть заставляет Джулию `eval` эту `string(ex)` поскольку выполнение проходит через вызов этого макроса. т. е. если вы просто запустите этот код, это не приведет к какой-либо оценке. Но в тот момент, когда вы `assert(foo)` Julia будет **вызывать** этот макрос, заменяя его «токен AST / Expr» на все, что он возвращает, а `$` будет действовать.

Интересный взлом для использования `{}` для блоков

(@fcard) Я не думаю, что есть что-то техническое сохранение `{}` от использования в качестве блоков, на самом деле можно даже каламбурить остаточный синтаксис `{}` чтобы заставить его работать:

```
julia> macro c(block)
    @assert block.head == :cellld
    esc(quote
        $(block.args...)
    end)
end
@c (macro with 1 method)

julia> @c {
    print(1)
    print(2)
    1+2
}

123
```

* (вряд ли все еще будет работать, если / когда синтаксис `{}` будет переназначен)

Итак, сначала Джулия видит токен макроса, поэтому он будет читать / разыгрывать токены до соответствующего `end` и создавать что? Expr `c.head=:macro` или что-то? Сохраняет ли она `"a+1"` в виде строки или разбивает ее на `:+(:a, 1)` ? Как посмотреть?

?

(@fcard) В этом случае из-за лексической области а не определено в области @M s, поэтому использует глобальную переменную ... Я фактически забыл избежать выражения flipplin в моем неом примере, но «*работает только в пределах тот же модуль*», который по-прежнему применяется.

```
julia> module M
    macro m()
        : (a+1)
    end
end
M

julia> a = 1
1

julia> M.@m
ERROR: UndefVarError: a not defined
```

Причина в том, что если макрос используется в любом модуле, отличном от того, в котором он был определен, любые переменные, не определенные в коде, подлежащем расширению, рассматриваются как глобальные модули модуля макроса.

```
julia> macroexpand(: (M.@m) )
: (M.a + 1)
```

ADVANCED

@ Исмаэль-VC

```
@eval begin
    "do-until loop"
    macro $(:do) (block, until::Symbol, condition)
        until ≠ :until &&
            error("@do expected `until` got `$until`")
        quote
            let
                $block
                @until $condition begin
                    $block
                end
            end
        end |> esc
    end
end

julia> i = 0
0

julia> @do begin
    @show i
    i += 1
end
```

```

        end until i == 5
i = 0
i = 1
i = 2
i = 3
i = 4

```

Макрос Скотта:

```

"""
Internal function to return captured line number information from AST

##Parameters
- a:      Expression in the julia type Expr

##Return
- Line number in the file where the calling macro was invoked
"""
_lin(a::Expr) = a.args[2].args[1].args[1]

"""
Internal function to return captured file name information from AST

##Parameters
- a:      Expression in the julia type Expr

##Return
- The name of the file where the macro was invoked
"""
_fil(a::Expr) = string(a.args[2].args[1].args[2])

"""
Internal function to determine if a symbol is a status code or variable
"""
function _is_status(sym::Symbol)
    sym in (:OK, :WARNING, :ERROR) && return true
    str = string(sym)
    length(str) > 4 && (str[1:4] == "ERR_" || str[1:5] == "WARN_" || str[1:5] == "INFO_")
end

"""
Internal function to return captured error code from AST

##Parameters
- a:      Expression in the julia type Expr

##Return
- Error code from the captured info in the AST from the calling macro
"""
_err(a::Expr) =
    (sym = a.args[2].args[2] ; _is_status(sym) ? Expr(:., :Status, QuoteNode(sym)) : sym)

"""
Internal function to produce a call to the log function based on the macro arguments and the
AST from the ()->ERRCODE anonymous function definition used to capture error code, file name
and line number where the macro is used

##Parameters

```

```

- level:      Loglevel which has to be logged with macro
- a:          Expression in the julia type Expr
- msgs:       Optional message

##Return
- Statuscode
"""
function _log(level, a, msgs)
    if isempty(msgs)
        :( log($level, $(esc(:Symbol))($_fil(a)), $_lin(a), $_err(a)) )
    else
        :( log($level, $(esc(:Symbol))($_fil(a)), $_lin(a), $_err(a)),
message=$(esc(msgs[1])) )
    end
end

macro warn(a, msgs...) ; _log(Warning, a, msgs) ; end

```

мусор / необработанный ...

просмотр / сброс макроса

(@ pi-) Предположим, что я просто делаю `macro m(); a+1; end` в новом REPL. При отсутствии `a` определено. Как я могу его просмотреть? например, есть ли способ «сбросить» макрос? Без его выполнения

(@fcard) Весь код в макросах фактически помещается в функции, поэтому вы можете просматривать только их пониженный или тип-вывод.

```

julia> macro m()  a+1  end
@m (macro with 1 method)

julia> @code_typed @m
LambdaInfo for @m()
:(begin
    return Main.a + 1
end)

julia> @code_lowered @m
CodeInfo(:(begin
    nothing
    return Main.a + 1
end))
# ^ or: code_lowered(eval(Symbol("@m")))[1] # ouf!

```

Другие способы получения функции макроса:

```

julia> macro getmacro(call) call.args[1] end
@getmacro (macro with 1 method)

julia> getmacro(name) = getfield(current_module(), name.args[1])
getmacro (generic function with 1 method)

```



```
julia> @getmacro @m
@m (macro with 1 method)

julia> getmacro(:@m)
@m (macro with 1 method)
```

```
julia> eval(Symbol("@M"))
@M (macro with 1 method)

julia> dump( eval(Symbol("@M")) )
@M (function of type #@M)

julia> code_typed( eval(Symbol("@M")) )
1-element Array{Any,1}:
LambdaInfo for @M()

julia> code_typed( eval(Symbol("@M")) )[1]
LambdaInfo for @M()
:(begin
    return $(Expr(:copyast, :($(QuoteNode(:(a + 1))))))
end::Expr)

julia> @code_typed @M
LambdaInfo for @M()
:(begin
    return $(Expr(:copyast, :($(QuoteNode(:(a + 1))))))
end::Expr)
```

^ похоже, я могу использовать `code_typed` вместо

Как понять `eval(Symbol("@M"))` ?

(@fcard) В настоящее время каждый макрос имеет связанную с ним функцию. Если у вас есть макрос, называемый `m`, то функция макроса называется `@m`. Как правило, вы можете получить значение функции, например, `eval(:print)` но с помощью функции макроса вам нужно сделать `Symbol("@M")`, так как просто `@M` становится `Expr(:macrocall, Symbol("@M"))` и оценка, которая вызывает макрорасширение.

Почему параметры `code_typed` не отображаются?

(@число Пи-)

```
julia> code_typed( x -> x^2 )[1]
LambdaInfo for (::##5#6)::Any
:(begin
    return x ^ 2
end)
```

^ здесь я вижу один `::Any` параметр, но он, похоже, не связан с токеном `x`.

```
julia> code_typed( print )[1]
LambdaInfo for print(::IO, ::Char)
```

```

: (begin
    (Base.write) (io, c)
    return Base.nothing
end)::Void)

```

Аналогично здесь; нет ничего, чтобы соединить `io` с `::IO`. Таким образом, это не может быть полной дамкой представления AST этого конкретного метода `print ...`?

(@fcard) `print (::IO, ::Char)` сообщает только, какой именно метод, это не часть AST. Он больше не присутствует в мастерстве:

```

julia> code_typed(print)[1]
CodeInfo(: (begin
    (Base.write) (io, c)
    return Base.nothing
end)) =>Void

```

(@pi-) Я не понимаю, что вы подразумеваете под этим. Кажется, это демпинг AST для тела этого метода, нет? Я думал, что `code_typed` дает AST для функции. Но, кажется, отсутствует первый шаг, то есть настройка токенов для параметров.

(@fcard) `code_typed` предназначен для отображения только AST, но на данный момент он дает полный AST метода в виде `LambdaInfo` (0.5) или `CodeInfo` (0.6), но большая часть информации опущена при печати на замену. Вам нужно будет проверить поле `LambdaInfo` по полю, чтобы получить все детали. `dump` собирается наполнить ваш реплик, чтобы вы могли попробовать:

```

macro method_info(call)
    quote
        method = @code_typed $(esc(call))
        print_info_fields(method)
    end
end

function print_info_fields(method)
    for field in fieldnames(typeof(method))
        if isdefined(method, field) && !(field in [Symbol(""), :code])
            println(" $field = ", getfield(method, field))
        end
    end
    display(method)
end

print_info_fields(x::Pair) = print_info_fields(x[1])

```

Что дает все значения названных полей метода AST:

```

julia> @method_info print(STDOUT, 'a')
rettype = Void
sparam_syms = svec()
sparam_vals = svec()
specTypes = Tuple{Base.#print, Base.TTY, Char}

```

```

slottypes = Any[Base.#print,Base.TTY,Char]
ssavaluetypes = Any[]
slotnames = Any[Symbol("#self#"),:io,:c]
slotflags = UInt8[0x00,0x00,0x00]
def = print(io::IO, c::Char) at char.jl:45
nargs = 3
isva = false
inferred = true
pure = false
inlineable = true
inInference = false
inCompile = false
jlcall_api = 0
fptr = Ptr{Void} @0x00007f7a7e96ce10
LambdaInfo for print(::Base.TTY, ::Char)
:(begin
    $(Expr(:invoke, LambdaInfo for write(::Base.TTY, ::Char), :(Base.write), :(io), :(c)))
    return Base.nothing
end::Void)

```

См. Lil ' def = print(io::IO, c::Char) ? Вот так! (также слова slotnames = [..., :io, :c])
 Также да, разница в выходе заключается в том, что я показывал результаты на master.

???

(@ Ismael-VC), вы имеете в виду вот так? [Общая отправка с символами](#)

Вы можете сделать это следующим образом:

```

julia> function dispatchtest{alg}(::Type{Val{alg}})
    println("This is the generic dispatch. The algorithm is $alg")
end
dispatchtest (generic function with 1 method)

julia> dispatchtest{alg::Symbol} = dispatchtest{Val{alg}}
dispatchtest (generic function with 2 methods)

julia> function dispatchtest(::Type{Val{:Euler}})
    println("This is for the Euler algorithm!")
end
dispatchtest (generic function with 3 methods)

julia> dispatchtest{Foo}
This is the generic dispatch. The algorithm is Foo

julia> dispatchtest{Euler}

```

Это для алгоритма Эйлера! Интересно, что @fcard думает об общей отправке символов! ---
 ^: angel:

Модуль Gotcha

```

@def m begin
    a+2

```

```
end

@m # replaces the macro at compile-time with the expression a+2
```

Точнее, только работает в пределах уровня модуля, в котором был определен макрос.

```
julia> module M
    macro m1()
        a+1
    end
end

M

julia> macro m2()
    a+1
end
@m2 (macro with 1 method)

julia> a = 1
1

julia> M.@m1
ERROR: UndefVarError: a not defined

julia> @m2
2

julia> let a = 20
    @m2
end
2
```

`esc` позволяет это происходить, но отказ от использования его всегда идет вразрез с языком. Хорошая защита для этого заключается в том, чтобы не использовать и вводить имена в макросах, что затрудняет их отслеживание для читателя.

Python `dict` / JSON как синтаксис для литералов `Dict`.

Вступление

Джулия использует следующий синтаксис для словарей:

```
Dict{K, V}({k1 => v1, k2 => v2, ..., kn-1 => vn-1, kn => vn})
```

Хотя Python и JSON выглядят так:

```
{k1: v1, k2: v2, ..., kn-1: vn-1, kn: vn}
```

Для **иллюстративных целей** мы могли бы также использовать этот синтаксис в Julia и добавить к нему новую семантику (синтаксис `Dict` является идиоматическим способом в Julia, который рекомендуется).

Сначала давайте посмотрим, что это за выражение:

```
julia> parse("{1:2 , 3: 4}") |> Meta.show_sexpr
(:cellld, (:(:), 1, 2), (:(:), 3, 4))
```

Это означает, что нам нужно принять это `:cellld` выражение `:cellld` и либо преобразовать его, либо вернуть новое выражение, которое должно выглядеть следующим образом:

```
julia> parse("Dict{1 => 2 , 3 => 4}") |> Meta.show_sexpr
(:call, :Dict, (:(>=), 1, 2), (:(>=), 3, 4))
```

Определение макроса

Следующий макрос, хотя и простой, позволяет продемонстрировать такое генерирование и преобразование кода:

```
macro dict(expr)
    # Check the expression has the correct form:
    if expr.head != :cellld || any(sub_expr.head != :(:) for sub_expr in expr.args)
        error("syntax: expected `{k₁: v₁, k₂: v₂, ..., kₙ: vₙ, kₙ: vₙ}`")
    end

    # Create empty `:Dict` expression which will be returned:
    block = Expr(:call, :Dict) # : (Dict())

    # Append `(key => value)` pairs to the block:
    for pair in expr.args
        k, v = pair.args
        push!(block.args, :($k => $v))
    end # : (Dict(k₁ => v₁, k₂ => v₂, ..., kₙ => vₙ, kₙ => vₙ))

    # Block is escaped so it can reach variables from it's calling scope:
    return esc(block)
end
```

Давайте посмотрим на полученное макроопределение:

```
julia> :(@dict {"a": :b, 'c': 1, :d: 2.0}) |> macroexpand
:(Dict{"a" => :b, 'c' => 1, :d => 2.0})
```

ИСПОЛЬЗОВАНИЕ

```
julia> @dict {"a": :b, 'c': 1, :d: 2.0}
Dict{Any,Any} with 3 entries:
  "a" => :b
  :d  => 2.0
  'c' => 1

julia> @dict {
    "string": :b,
    'c'      : 1,
```

```

        :symbol : π,
        Function: print,
        (1:10)   : range(1, 10)
    }
Dict{Any,Any} with 5 entries:
 1:10      => 1:10
Function   => print
"string"   => :b
:symbol    => π = 3.1415926535897...
'c'        => 1

```

Последний пример в точности эквивалентен:

```

Dict(
  "string" => :b,
  'c'      => 1,
  :symbol  => π,
  Function => print,
  (1:10)   => range(1, 10)
)

```

Misusage

```

julia> @dict {"one": 1, "two": 2, "three": 3, "four": 4, "five" => 5}
syntax: expected `{k₁: v₁, k₂: v₂, ..., kₙ₋₁: vₙ₋₁, kₙ: vₙ}`

julia> @dict ["one": 1, "two": 2, "three": 3, "four": 4, "five" => 5]
syntax: expected `{k₁: v₁, k₂: v₂, ..., kₙ₋₁: vₙ₋₁, kₙ: vₙ}`

```

Обратите внимание, что у Джулии есть другие способы использования двоеточия : как таковой вам нужно будет, например, обрезать литеральные выражения с круглыми скобками или использовать функцию `range` .

Прочитайте [Метапрограммирование онлайн](https://riptutorial.com/ru/julia-lang/topic/1945/метапрограммирование): <https://riptutorial.com/ru/julia-lang/topic/1945/метапрограммирование>

глава 18: Модули

Синтаксис

- модуль модуля; ...; конец
- использование модуля
- модуль импорта

Examples

Копировать код в модуль

Ключевое слово `module` можно использовать для создания модуля, который позволяет организовывать код и размещать имена. Модули могут определять внешний интерфейс, обычно состоящий из `export` символов. Для поддержки этого внешнего интерфейса модули могут иметь невыполненные внутренние [функции](#) и [типы](#), не предназначенные для общего пользования.

Некоторые модули в основном существуют для переноса типа и связанных функций. Такие модули, как правило, обычно называются с множественной формой имени типа. Например, если у нас есть модуль, который предоставляет тип `Building`, мы можем назвать такой модуль `Buildings`.

```
module Buildings

  immutable Building
    name::String
    stories::Int
    height::Int # in metres
  end

  name(b::Building) = b.name
  stories(b::Building) = b.stories
  height(b::Building) = b.height

  function Base.show(io::IO, b::Building)
    Base.print(stories(b), "-story ", name(b), " with height ", height(b), "m")
  end

  export Building, name, stories, height

end
```

Затем модуль можно использовать с `using` оператора:

```
julia> using Buildings

julia> Building("Burj Khalifa", 163, 830)
```

```
163-story Burj Khalifa with height 830m
```

```
julia> height(ans)  
830
```

Использование модулей для организации пакетов

Как правило, **пакеты** состоят из одного или нескольких модулей. По мере роста пакетов может оказаться полезным организовать основной модуль пакета на более мелкие модули. Общей идиомой является определение этих модулей как подмодулей основного модуля:

```
module RootModule  
  
    module SubModule1  
  
        ...  
  
    end  
  
    module SubModule2  
  
        ...  
  
    end  
  
end
```

Первоначально ни корневой модуль, ни подмодули не имеют доступа к экспортированным символам друг друга. Однако для решения этой проблемы поддерживается относительный импорт:

```
module RootModule  
  
    module SubModule1  
  
        const x = 10  
        export x  
  
    end  
  
    module SubModule2  
  
        # import submodule of parent module  
        using ..SubModule1  
        const y = 2x  
        export y  
  
    end  
  
    # import submodule of current module  
    using .SubModule1  
    using .SubModule2  
    const z = x + y  
  
end
```


В этом примере значение `RootModule.z` равно 30 .

Прочитайте Модули онлайн: <https://riptutorial.com/ru/julia-lang/topic/7368/модули>

глава 19: Нормализация строки

Синтаксис

- `normalize_string (s :: String, ...)`

параметры

параметр	подробности
<code>casefold=true</code>	Сложите строку в канонический регистр, основанный на стандарте Unicode .
<code>stripmark=true</code>	Разделите диакритические метки (например, акценты) от символов входной строки.

Examples

Сравнение строк без учета регистра

[Строки](#) можно сравнить с [оператором](#) `==` в Julia, но это чувствительно к различиям в случае. Например, `"Hello"` и `"hello"` считаются разными строками.

```
julia> "Hello" == "Hello"
true

julia> "Hello" == "hello"
false
```

Чтобы сравнить строки в нечувствительном к регистру образом, сначала нормализовать строки, складывая их. Например,

```
equals_ignore_case(s, t) =
    normalize_string(s, casefold=true) == normalize_string(t, casefold=true)
```

Этот подход также корректно обрабатывает Unicode Unicode:

```
julia> equals_ignore_case("Hello", "hello")
true

julia> equals_ignore_case("Weierstraß", "WEIERSTRASS")
true
```

Обратите внимание, что на немецком языке прописная форма ß-символа - SS.

Сравнение диакритико-нечувствительных строк

Иногда вам нужны строки, такие как "resume" и "résumé" для сравнения равных. То есть, **графемы**, которые имеют общий глиф, но, возможно, отличаются из-за добавления к этим основным глифам. Такое сравнение может быть выполнено путем снятия диакритических знаков.

```
equals_ignore_mark(s, t) =  
    normalize_string(s, stripmark=true) == normalize_string(t, stripmark=true)
```

Это позволяет приведенному выше примеру работать правильно. Кроме того, он хорошо работает даже с символами Unicode, отличными от ASCII.

```
julia> equals_ignore_mark("resume", "résumé")  
true  
  
julia> equals_ignore_mark("αβγ", "à β γ")  
true
```

Прочитайте Нормализация строки онлайн: <https://riptutorial.com/ru/julia-lang/topic/7612/нормализация-строки>

глава 20: пакеты

Синтаксис

- `Pkg.add` (пакет)
- `Pkg.checkout` (пакет, `branch = "master"`)
- `Pkg.clone` (URL)
- `Pkg.dir` (пакет)
- `Pkg.pin` (пакет, версия)
- `Pkg.rm` (пакет)

параметры

параметр	подробности
<code>Pkg.add(package)</code>	Загрузите и установите данный зарегистрированный пакет.
<code>Pkg.checkout(package , branch)</code>	Проверьте данную ветку для данного зарегистрированного пакета. <code>branch</code> является необязательной и по умолчанию используется <code>"master"</code> .
<code>Pkg.clone(url)</code>	Клонирование репозитория Git по указанному URL-адресу в виде пакета.
<code>Pkg.dir(package)</code>	Получите местоположение на диске для данного пакета.
<code>Pkg.pin(package , version)</code>	Заставить пакет оставаться в данной версии. <code>version</code> является необязательной и по умолчанию используется текущая версия пакета.
<code>Pkg.rm(package)</code>	Удалите данный пакет из списка необходимых пакетов.

Examples

Установка, использование и удаление зарегистрированного пакета

После нахождения официального пакета Julia, скачать и установить пакет просто. В-первых, рекомендуется обновить локальную копию METADATA:

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

Это обеспечит получение последних версий всех пакетов.

Предположим, что пакет, который мы хотим установить, называется `Currencies.jl`. Команда для запуска этого пакета:

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

Эта команда установит не только сам пакет, но и все его зависимости.

Если установка прошла успешно, вы можете [проверить правильность работы пакета](#) :

```
julia> Pkg.test("Currencies")
```

Затем, чтобы использовать пакет, используйте

```
julia> using Currencies
```

и действуйте так, как описано в документации пакета, обычно связанной с файлом `README.md` или включаемым в него.

Чтобы удалить пакет, который больше не нужен, используйте функцию `Pkg.rm` :

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

Обратите внимание, что это может фактически не удалить каталог пакета; вместо этого он просто помечает пакет как уже не требующийся. Часто это совершенно нормально - это экономит время, если вам понадобится пакет в будущем. Но если необходимо, чтобы физически удалить пакет, вызовите функцию `rm`, а затем вызовите `Pkg.resolve` :

```
julia> rm(Pkg.dir("Currencies"); recursive=true)
```

```
julia> Pkg.resolve()
```

Проверьте другой филиал или версию

Иногда последняя помеченная версия пакета является ошибкой или отсутствует некоторые необходимые функции. Продвинутые пользователи могут захотеть обновить последнюю версию версии пакета (иногда называемую «мастером», названной в честь обычного имени для [ветви](#) разработки в Git). Преимущества этого включают:

- Разработчики, вносящие вклад в пакет, должны вносить вклад в последнюю версию разработки.
- Последняя версия версии может иметь полезные функции, исправления или улучшения производительности.
- Пользователи, сообщившие об ошибке, могут захотеть проверить, произошла ли ошибка в последней версии разработки.

Тем не менее, существует множество недостатков для запуска последней версии

разработки:

- Последняя версия разработки может быть плохо протестирована и иметь серьезные ошибки.
- Последняя версия разработки может часто меняться, разбивая ваш код.

Например, чтобы проверить последнюю ветку разработки пакета с именем `JSON.jl`, используйте

```
Pkg.checkout("JSON")
```

Чтобы проверить другую ветку или тег (не названный «master»), используйте

```
Pkg.checkout("JSON", "v0.6.0")
```

Однако, если тег представляет собой версию, обычно лучше использовать

```
Pkg.pin("JSON", v"0.6.0")
```

Обратите внимание, что здесь используется литерал версии, а не простая строка. Версия `Pkg.pin` сообщает диспетчеру пакета ограничения версии, позволяя менеджеру пакетов предлагать обратную связь о том, какие проблемы могут возникнуть.

Чтобы вернуться к последней помеченной версии,

```
Pkg.free("JSON")
```

Установка незарегистрированного пакета

Некоторые экспериментальные пакеты не включены в репозиторий пакетов METADATA. Эти пакеты могут быть установлены путем прямого клонирования их репозитория в Git. Обратите внимание, что могут быть зависимости незарегистрированных пакетов, которые сами незарегистрированы; эти зависимости не могут быть разрешены диспетчером пакетов и должны быть разрешены вручную. Например, чтобы установить незарегистрированный пакет `OhMyREPL.jl`:

```
Pkg.clone("https://github.com/KristofferC/Tokenize.jl")  
Pkg.clone("https://github.com/KristofferC/OhMyREPL.jl")
```

Затем, как обычно, используйте `using` для использования пакета:

```
using OhMyREPL
```

Прочитайте пакеты онлайн: <https://riptutorial.com/ru/julia-lang/topic/5815/пакеты>

глава 21: Параллельная обработка

Examples

рmap

`рmap` принимает функцию (которую вы указываете) и применяет ее ко всем элементам массива. Эта работа разделена среди доступных работников. `рmap` затем возвращает результаты этой функции в другой массив.

```
addprocs(3)
sqrts = рmap(sqrt, 1:10)
```

если вы используете несколько аргументов, вы можете предоставить несколько векторов для `рmap`

```
dots = рmap(dot, 1:10, 11:20)
```

Однако, как и в случае `@parallel`, если функция, заданная для `рmap`, не находится в базе Julia (т.е. она определяется пользователем или определена в пакете), вы должны убедиться, что функция доступна всем сотрудникам:

```
@everywhere begin
    function rand_det(n)
        det(rand(n,n))
    end
end

determinants = рmap(rand_det, 1:10)
```

См. Также [этот вопрос Q & A](#).

@параллельно

`@parallel` можно использовать для параллелизации цикла, делящего шаги цикла на разных работников. В качестве очень простого примера:

```
addprocs(3)

a = collect(1:10)

for idx = 1:10
    println(a[idx])
end
```

Рассмотрим несколько более сложный пример:

```

@time begin
    @sync begin
        @parallel for idx in 1:length(a)
            sleep(a[idx])
        end
    end
end
27.023411 seconds (13.48 k allocations: 762.532 KB)
julia> sum(a)
55

```

Таким образом, мы видим, что если бы мы выполнили этот цикл без `@parallel`, для выполнения потребовалось бы 55 секунд, а не 27.

Мы также можем предоставить оператора сокращения для макроса `@parallel`. Предположим, что у нас есть массив, мы хотим суммировать каждый столбец массива, а затем умножить эти суммы друг на друга:

```

A = rand(100,100);

@parallel (*) for idx = 1:size(A,1)
    sum(A[:,idx])
end

```

Есть несколько важных вещей, которые следует иметь в виду при использовании `@parallel` чтобы избежать неожиданного поведения.

Во-первых: если вы хотите использовать любые функции в своих циклах, которые не находятся в базе Julia (например, любые функции, которые вы определяете в своем скрипте или импортируете из пакетов), вы должны сделать эти функции доступными для рабочих. Таким образом, например, следующее *не* будет работать:

```

myprint(x) = println(x)
for idx = 1:10
    myprint(a[idx])
end

```

Вместо этого нам нужно будет использовать:

```

@everywhere begin
    function myprint(x)
        println(x)
    end
end

@parallel for idx in 1:length(a)
    myprint(a[idx])
end

```

Во-вторых, хотя каждый рабочий сможет получить доступ к объектам в объеме контроллера, они *не* смогут их модифицировать. таким образом


```

a = collect(1:10)
@parallel for idx = 1:length(a)
    a[idx] += 1
end

julia> a'
1x10 Array{Int64,2}:
 1  2  3  4  5  6  7  8  9 10

```

Если бы мы выполнили цикл с `@parallel`, он бы успешно модифицировал массив `a`.

Чтобы сообщить `a` об этом, мы можем вместо этого создать `SharedArray` типа `SharedArray` чтобы каждый рабочий мог его получить и изменить:

```

a = convert(SharedArray{Float64,1}, collect(1:10))
@parallel for idx = 1:length(a)
    a[idx] += 1
end

julia> a'
1x10 Array{Float64,2}:
 2.0  3.0  4.0  5.0  6.0  7.0  8.0  9.0 10.0 11.0

```

@spawn и @spawnat

`@spawn` и `@spawnat` - два из инструментов, которые Джулия предоставляет для назначения задач рабочим. Вот пример:

```

julia> @spawnat 2 println("hello world")
RemoteRef{Channel{Any}}(2,1,3)

julia> From worker 2: hello world

```

Оба этих макроса будут оценивать **выражение** в рабочем процессе. Единственное различие между ними состоит в том, что `@spawnat` позволяет вам выбирать, какой рабочий будет оценивать выражение (в примере выше рабочий 2 указан), тогда как с `@spawn` рабочий будет автоматически выбран в зависимости от доступности.

В приведенном выше примере у нас просто был рабочий 2, выполняющий функцию `println`. Не было ничего интересного, чтобы вернуться или получить от этого. Часто, однако, выражение, которое мы отправили работнику, даст то, что мы хотим получить. Обратите внимание на приведенный выше пример, когда мы вызвали `@spawnat`, прежде чем мы получили распечатку с рабочего 2, мы увидели следующее:

```
RemoteRef{Channel{Any}}(2,1,3)
```

Это указывает на то, что `@spawnat` макрос будет возвращать `RemoteRef` объект типа. Этот объект, в свою очередь, будет содержать возвращаемые значения из нашего выражения, которое отправляется работнику. Если мы хотим получить эти значения, мы можем сначала

назначить `RemoteRef` который `@spawnat` вернется к объекту, а затем, а затем использовать функцию `fetch()` которая работает с `RemoteRef` типа `RemoteRef`, для получения результатов, сохраненных в результате оценки, выполненной на рабочий.

```
julia> result = @spawnat 2 2 + 5
RemoteRef{Channel{Any}}(2,1,26)

julia> fetch(result)
7
```

Ключом к эффективному использованию `@spawn` является понимание природы **выражений**, на которых он работает. Использование `@spawn` для отправки команд для работников немного сложнее, чем просто набирать непосредственно то, что вы набираете, если вы используете «интерпретатор» для одного из рабочих или выполняете код изначально на них. Например, предположим, что мы хотели использовать `@spawnat` для назначения значения переменной для рабочего. Мы могли бы попробовать:

```
@spawnat 2 a = 5
RemoteRef{Channel{Any}}(2,1,2)
```

Это сработало? Ну, давайте посмотрим, попробуем ли работника 2 напечатать `a`.

```
julia> @spawnat 2 println(a)
RemoteRef{Channel{Any}}(2,1,4)

julia>
```

Ничего не случилось. Зачем? Мы можем исследовать это, используя вышеприведенную функцию `fetch()`. `fetch()` может быть очень удобной, поскольку она будет извлекать не только успешные результаты, но и сообщения об ошибках. Без этого мы могли бы даже не знать, что что-то пошло не так.

```
julia> result = @spawnat 2 println(a)
RemoteRef{Channel{Any}}(2,1,5)

julia> fetch(result)
ERROR: On worker 2:
UndefVarError: a not defined
```

В сообщении об ошибке указано, что `a` не определено для рабочего 2. Но почему это? Причина в том, что нам нужно превратить нашу операцию назначения в выражение, которое мы затем используем `@spawn` чтобы сообщить работнику оценить. Ниже приведен пример, поясняющий следующее:

```
julia> @spawnat 2 eval(:(a = 2))
RemoteRef{Channel{Any}}(2,1,7)

julia> @spawnat 2 println(a)
RemoteRef{Channel{Any}}(2,1,8)
```

```
julia> From worker 2: 2
```

Синтаксис `: ()` - это то, что Джулия использует для обозначения [выражений](#). Затем мы используем функцию `eval()` в Julia, которая вычисляет выражение, и мы используем макрос `@spawnat` чтобы `@spawnat` что выражение будет оцениваться на рабочем 2.

Мы могли бы достичь такого же результата, как:

```
julia> @spawnat(2, eval(parse("c = 5")))
RemoteRef{Channel{Any}}(2,1,9)

julia> @spawnat 2 println(c)
RemoteRef{Channel{Any}}(2,1,10)

julia> From worker 2: 5
```

Этот пример демонстрирует два дополнительных понятия. Во-первых, мы видим, что мы также можем создать выражение, используя функцию `parse()` вызываемую в строке. Во-вторых, мы видим, что мы можем использовать круглые скобки при вызове `@spawnat`, в ситуациях, когда это может сделать наш синтаксис более понятным и управляемым.

Когда использовать `@parallel vmap`

Юлия [документация](#) сообщает, что

`vmap()` предназначен для случая, когда каждый вызов функции выполняет большой объем работы. Напротив, `@parallel for` может обрабатывать ситуации, когда каждая итерация крошечная, возможно, просто суммируя два числа.

На это есть несколько причин. Во-первых, `vmap` берет на себя большие затраты на запуск рабочих мест. Таким образом, если задания очень малы, эти затраты на запуск могут стать неэффективными. Напротив, однако, `vmap` делает «умную» работу по распределению рабочих мест среди работников. В частности, он создает очередь заданий и отправляет новое задание каждому работнику всякий раз, когда этот рабочий становится доступным. `@parallel`, напротив, разворачивает всю работу, которая должна выполняться среди рабочих, когда она называется. Таким образом, если некоторые работники занимают больше времени на своих рабочих местах, чем другие, вы можете столкнуться с ситуацией, когда большинство ваших работников закончили и простаивают, в то время как некоторые из них остаются активными в течение чрезмерного количества времени, заканчивая свою работу. Однако такая ситуация реже встречается с очень маленькими и простыми рабочими местами.

Ниже показано следующее: предположим, что у нас есть два рабочих, один из которых медленный, а другой из них в два раза быстрее. В идеале мы хотели бы дать быструю рабочую работу в два раза больше работы, чем медленный рабочий. (или мы могли бы иметь быструю и медленную работу, но главное - то же самое). `vmap` выполнит это, но

@parallel не будет.

Для каждого теста мы инициализируем следующее:

```
addprocs(2)

@everywhere begin
    function parallel_func(idx)
        workernum = myid() - 1
        sleep(workernum)
        println("job $idx")
    end
end
```

Теперь, для теста @parallel , мы запускаем следующее:

```
@parallel for idx = 1:12
    parallel_func(idx)
end
```

И верните распечатку:

```
julia>      From worker 2:      job 1
      From worker 3:      job 7
      From worker 2:      job 2
      From worker 2:      job 3
      From worker 3:      job 8
      From worker 2:      job 4
      From worker 2:      job 5
      From worker 3:      job 9
      From worker 2:      job 6
      From worker 3:      job 10
      From worker 3:      job 11
      From worker 3:      job 12
```

Это почти сладкое. Рабочие «разделили» работу равномерно. Обратите внимание, что каждый рабочий выполнил 6 заданий, хотя рабочий 2 в два раза быстрее, чем рабочий 3. Он может касаться, но он неэффективен.

Для теста pmap я запускаю следующее:

```
pmap(parallel_func, 1:12)
```

и получить результат:

```
From worker 2:      job 1
From worker 3:      job 2
From worker 2:      job 3
From worker 2:      job 5
From worker 3:      job 4
From worker 2:      job 6
From worker 2:      job 8
From worker 3:      job 7
```

```
From worker 2:    job 9
From worker 2:    job 11
From worker 3:    job 10
From worker 2:    job 12
```

Теперь обратите внимание, что работник 2 выполнил 8 заданий, а работник 3 выполнил 4. Это точно соответствует их скорости и тому, что мы хотим для оптимальной эффективности. `map` - это сложный мастер задачи - от каждого в зависимости от их способности.

@async и @sync

Согласно документации под `?@async`, «`@async` завершает выражение в задаче». Это означает, что для того, что попадает в сферу его охвата, Джулия начнет выполнение этой задачи, а затем перейдет к тому, что будет дальше в скрипте, не дожидаясь завершения задачи. Так, например, без макроса вы получите:

```
julia> @time sleep(2)
2.005766 seconds (13 allocations: 624 bytes)
```

Но с макросом вы получаете:

```
julia> @time @async sleep(2)
0.000021 seconds (7 allocations: 657 bytes)
Task (waiting) @0x0000000112a65ba0

julia>
```

Таким образом, Julia позволяет сценарию продолжить (и макрос `@time` полностью выполнить), не дожидаясь завершения задачи (в данном случае, спать в течение двух секунд).

Макрос `@sync`, напротив, будет «Подождать, пока все динамически закрытые приложения `@async`, `@spawn`, `@spawnat` и `@parallel` будут завершены». (в соответствии с документацией под `?@sync`). Таким образом, мы видим:

```
julia> @time @sync @async sleep(2)
2.002899 seconds (47 allocations: 2.986 KB)
Task (done) @0x0000000112bd2e00
```

В этом простом примере нет смысла включать один экземпляр `@async` и `@sync` вместе. Но, где `@sync` может быть полезным, вы можете использовать `@async` для нескольких операций, которые вы хотите разрешить всем, запускать сразу, не дожидаясь завершения каждого из них.

Например, предположим, что у нас есть несколько сотрудников, и мы бы хотели, чтобы каждый из них работал над задачей одновременно, а затем извлекал результаты из этих

задач. Первоначальная (но некорректная) попытка может быть:

```
addprocs(2)
@time begin
    a = cell(nworkers())
    for (idx, pid) in enumerate(workers())
        a[idx] = remotecall_fetch(pid, sleep, 2)
    end
end
## 4.011576 seconds (177 allocations: 9.734 KB)
```

Проблема заключается в том, что цикл ждет `remotecall_fetch()` каждой `remotecall_fetch()`, то есть для каждого процесса завершить свою работу (в этом случае спать в течение 2 секунд), прежде чем продолжить выполнение следующей `remotecall_fetch()`. Что касается практической ситуации, мы не получаем преимуществ параллелизма здесь, так как наши процессы не выполняют свою работу (т.е. спать) одновременно.

Однако мы можем исправить это, используя комбинацию макросов `@async` и `@sync`:

```
@time begin
    a = cell(nworkers())
    @sync for (idx, pid) in enumerate(workers())
        @async a[idx] = remotecall_fetch(pid, sleep, 2)
    end
end
## 2.009416 seconds (274 allocations: 25.592 KB)
```

Теперь, если мы посчитаем каждый шаг цикла как отдельную операцию, мы видим, что есть два отдельных операции, которым предшествует макрос `@async`. Макрос позволяет каждому из них запускаться, а код продолжать (в этом случае на следующий шаг цикла) до каждого завершения. Но использование макроса `@sync`, область видимости которого охватывает весь цикл, означает, что мы не допустим, чтобы сценарий прошел этот цикл до тех пор, пока все операции, предшествующие `@async` не будут завершены.

Можно получить еще более четкое представление о работе этих макросов путем дальнейшей настройки приведенного выше примера, чтобы увидеть, как он изменяется при определенных модификациях. Например, предположим, что мы просто имеем `@async` без `@sync`:

```
@time begin
    a = cell(nworkers())
    for (idx, pid) in enumerate(workers())
        println("sending work to $pid")
        @async a[idx] = remotecall_fetch(pid, sleep, 2)
    end
end
## 0.001429 seconds (27 allocations: 2.234 KB)
```

Здесь макрос `@async` позволяет нам продолжить работу в нашем цикле еще до `remotecall_fetch()` каждой `remotecall_fetch()`. Но, к лучшему или худшему, у нас нет макроса `@sync`

чтобы предотвратить продолжение кода после этого цикла до тех пор, пока все операции `remotecall_fetch()` закончатся.

Тем не менее, каждая `remotecall_fetch()` все еще работает параллельно, даже если мы продолжим. Мы можем это видеть, потому что если мы подождем две секунды, то массив `a`, содержащий результаты, будет содержать:

```
sleep(2)
julia> a
2-element Array{Any,1}:
 nothing
 nothing
```

(Элемент «ничего» является результатом успешной выборки результатов функции сна, которая не возвращает никаких значений)

Мы также видим, что две `remotecall_fetch()` начинаются практически в одно и то же время, потому что команды `print` которые предшествуют им, также выполняются с быстрой последовательностью (вывод этих команд не показан здесь). Сравните это со следующим примером, когда команды `print` выполняются с интервалом в 2 секунды друг от друга:

Если мы `@async` макрос `@async` во весь цикл (а не только на его внутренний шаг), то снова наш скрипт будет продолжен немедленно, не дожидаясь завершения `remotecall_fetch()`. Однако теперь мы разрешаем сценарию продолжать цикл в целом. Мы не разрешаем каждому отдельному шагу цикла запускать предыдущий. Таким образом, в отличие от приведенного выше примера, через две секунды после того, как сценарий продолжается после цикла, массив `results` все еще имеет один элемент в качестве `#undef` указывающий, что вторая `remotecall_fetch()` все еще не завершена.

```
@time begin
    a = cell(nworkers())
    @async for (idx, pid) in enumerate(workers())
        println("sending work to $pid")
        a[idx] = remotecall_fetch(pid, sleep, 2)
    end
end
# 0.001279 seconds (328 allocations: 21.354 KB)
# Task (waiting) @0x0000000115ec9120
## This also allows us to continue to

sleep(2)

a
2-element Array{Any,1}:
 nothing
 #undef
```

И неудивительно, что если мы поместим `@sync` и `@async` рядом друг с другом, мы получим, что каждый `remotecall_fetch()` работает последовательно (а не одновременно), но мы не продолжаем в коде до тех пор, пока все не закончится. Другими словами, это было бы

фактически эквивалентно тому, если бы у нас не было макроса на месте, так же как

`sleep(2)` ведет себя по существу идентично `@sync @async sleep(2)`

```
@time begin
    a = cell(nworkers())
    @sync @async for (idx, pid) in enumerate(workers())
        a[idx] = remotecall_fetch(pid, sleep, 2)
    end
end
# 4.019500 seconds (4.20 k allocations: 216.964 KB)
# Task (done) @0x0000000115e52a10
```

Также обратите внимание, что в `@async` макроса `@async` можно выполнять более сложные операции. В [документации](#) приведен пример, содержащий полный цикл в области `@async`.

Напомним, что справка для макросов синхронизации утверждает, что она будет «Подождать, пока все динамически закрытые приложения `@async`, `@spawn`, `@spawnat` и `@parallel` будут завершены». Для целей, которые считаются «полными», важно, как вы определяете задачи в рамках макросов `@sync` и `@async`. Рассмотрим приведенный ниже пример, который является небольшим изменением на одном из приведенных выше примеров:

```
@time begin
    a = cell(nworkers())
    @sync for (idx, pid) in enumerate(workers())
        @async a[idx] = remotecall(pid, sleep, 2)
    end
end
## 0.172479 seconds (93.42 k allocations: 3.900 MB)

julia> a
2-element Array{Any,1}:
 RemoteRef{Channel{Any}}{2,1,3}
 RemoteRef{Channel{Any}}{3,1,4}
```

Более ранний пример занял примерно 2 секунды для выполнения, что указывает на то, что две задачи выполнялись параллельно и что сценарий ожидает завершения каждого выполнения своих функций перед продолжением. Однако этот пример имеет гораздо более низкую оценку времени. Причина в том, что для целей `@sync remotecall()` была «завершена», как только она отправила работнику работу. (Обратите внимание, что результирующий массив, здесь, просто содержит `RemoteRef` объектов `RemoteRef`, которые просто указывают на то, что происходит что-то происходящее с конкретным процессом, который теоретически может быть получен в какой-то момент в будущем). Напротив, `remotecall_fetch()` только «закончена», когда получает сообщение от работника о завершении своей задачи.

Таким образом, если вы ищете способы обеспечения того, чтобы определенные операции с работниками были завершены до перехода в ваш сценарий (как, например, обсуждается в [этом сообщении](#)), необходимо тщательно подумать о том, что считается «полным» и как вы

будете измерьте, а затем примените это в своем сценарии.

Добавление рабочих

Когда вы впервые запускаете Julia, по умолчанию будет работать только один процесс и доступен для работы. Вы можете проверить это, используя:

```
julia> nprocs()  
1
```

Чтобы воспользоваться параллельной обработкой, вы должны сначала добавить дополнительных работников, которые затем будут доступны для выполнения работы, которую вы им назначаете. Вы можете сделать это в своем скрипте (или из интерпретатора), используя: `addprocs(n)` где `n` - количество процессов, которые вы хотите использовать.

Кроме того, вы можете добавлять процессы, когда вы запускаете Julia из командной строки, используя:

```
$ julia -p n
```

где `n` - количество *дополнительных* процессов, которые вы хотите добавить. Таким образом, если мы начнем Джулию с

```
$ julia -p 2
```

Когда Джулия начнет, мы получим:

```
julia> nprocs()  
3
```

Прочитайте Параллельная обработка онлайн: <https://riptutorial.com/ru/julia-lang/topic/4542/параллельная-обработка>

глава 22: Перечисления

Синтаксис

- `@enum EnumType val = 1 val val`
- `:условное обозначение`

замечания

Иногда бывает полезно перечислять типы, в которых каждый экземпляр имеет другой тип (часто один [неизменный тип](#)); это может быть важно для стабильности типа. Черты обычно реализуются с помощью этой парадигмы. Однако это приводит к дополнительным издержкам времени компиляции.

Examples

Определение перечислимого типа

[Перечислимый тип](#) - это [тип](#), который может содержать один из конечных списков возможных значений. В Julia перечисляемые типы обычно называются «перечисляемыми типами». Например, можно использовать типы перечислений для описания семи дней недели, двенадцати месяцев года, четырех костюмов [стандартной колоды с 52 карточками](#) или других подобных ситуаций.

Мы можем определить перечисленные типы для моделирования костюмов и рангов стандартной колоды с 52 карточками. Макрос `@enum` используется для определения типов перечислений.

```
@enum Suit ♣ ♦ ♥ ♠  
@enum Rank ace=1 two three four five six seven eight nine ten jack queen king
```

Это определяет два типа: `Suit` и `Rank`. Мы можем проверить, действительно ли значения ожидаемых типов:

```
julia> ♦  
♦::Suit = 1  
  
julia> six  
six::Rank = 6
```

Обратите внимание, что каждый костюм и ранг связаны с числом. По умолчанию это число начинается с нуля. Таким образом, второй костюм, бриллианты, был присвоен номер 1. В случае `Rank` может возникнуть больше смысла начинать число в одном. Это было

достигнуто путем аннотации определения `ace` с аннотации `a = 1` .

Перечисленные типы имеют много функциональных возможностей, таких как равенство (и действительно идентичность) и встроенные сравнения:

```
julia> seven === seven
true

julia> ten ≠ jack
true

julia> two < three
true
```

Подобно значениям любого другого [неизменяемого типа](#) , значения перечисленных типов также могут быть хэшированы и сохранены в `Dict{S}` .

Мы можем завершить этот пример, указав тип `Card` который имеет поле `Rank` и `Suit` :

```
immutable Card
    rank::Rank
    suit::Suit
end
```

и, следовательно, мы можем создавать карточки с

```
julia> Card(three, ♣)
Card(three::Rank = 3, ♣::Suit = 0)
```

Но перечисляемые типы также имеют свои собственные методы `convert` , поэтому мы действительно можем просто сделать

```
julia> Card(7, ♠)
Card(seven::Rank = 7, ♠::Suit = 3)
```

и поскольку `7` может быть напрямую преобразован в `Rank` , этот конструктор работает из коробки.

Мы могли бы определить синтаксический сахар для построения этих карт; неявное умножение обеспечивает удобный способ сделать это. определять

```
julia> import Base.*

julia> r::Int * s::Suit = Card(r, s)
* (generic function with 156 methods)
```

а потом

```
julia> 10♣
Card(ten::Rank = 10, ♣::Suit = 0)
```

```
julia> 5♠  
Card(five::Rank = 5,♠::Suit = 3)
```

снова воспользовавшись встроенными функциями `convert` .

Использование символов в виде легких перечислений

Хотя макрос `@enum` весьма полезен для большинства случаев использования, он может быть чрезмерным в некоторых случаях использования. Недостатки `@enum` включают:

- Он создает новый тип
- Немного сложнее расширить
- Он включает такие функции, как преобразование, перечисление и сравнение, которые могут быть излишними в некоторых приложениях

В случаях, когда требуется альтернатива с более легким весом, можно использовать тип `Symbol` . Символы - [интернированные строки](#) ; они представляют последовательности символов, как и [строки](#) , но они однозначно связаны с числами. Эта уникальная ассоциация обеспечивает быстрое сравнение равенства символов.

Мы можем снова реализовать тип `Card` , на этот раз используя поля `Symbol` :

```
const ranks = Set{Symbol}([:ace, :two, :three, :four, :five, :six, :seven, :eight, :nine,  
                           :ten, :jack, :queen, :king])  
const suits = Set{Symbol}([:♣, :♦, :♥, :♠])  
immutable Card  
    rank::Symbol  
    suit::Symbol  
    function Card(r::Symbol, s::Symbol)  
        r in ranks || throw(ArgumentError("invalid rank: $r"))  
        s in suits || throw(ArgumentError("invalid suit: $s"))  
        new(r, s)  
    end  
end
```

Мы реализуем внутренний конструктор, чтобы проверить любые неверные значения, переданные конструктору. В отличие от примера, использующего типы `@enum` , `Symbol` `s` может содержать любую строку, поэтому мы должны быть осторожны с тем, какие символы `Symbol` мы принимаем. Обратите внимание на использование условных операторов [короткого замыкания](#) .

Теперь мы можем создавать объекты `Card` как мы ожидаем:

```
julia> Card(:ace, :♦)  
Card(:ace,:♦)  
  
julia> Card(:nine, :♠)  
Card(:nine,:♠)
```

```
julia> Card(:eleven, :♠)
ERROR: ArgumentError: invalid rank: eleven
in Card(::Symbol, ::Symbol) at ./REPL[17]:5

julia> Card(:king, :X)
ERROR: ArgumentError: invalid suit: X
in Card(::Symbol, ::Symbol) at ./REPL[17]:6
```

Основным преимуществом `Symbol` является его расширяемость. Если во время выполнения мы хотим принять (например) `:eleven` в качестве нового ранга, достаточно просто запустить `push!(ranks, :eleven)`. Такая расширяемость невозможна при `@enum` типов `@enum`.

Прочитайте Перечисления онлайн: <https://riptutorial.com/ru/julia-lang/topic/7104/перечисления>

глава 23: постижения

Examples

Массивное понимание

Основной синтаксис

В подходах массива Julia используется следующий синтаксис:

```
[expression for element = iterable]
```

Обратите внимание, что, как и `for` ЦИКЛОВ, все значения `=`, `in` и `∈` принимаются для понимания.

Это примерно эквивалентно созданию пустого массива и использованию цикла `for` для `push!` предметов к нему.

```
result = []
for element in iterable
    push!(result, expression)
end
```

однако тип понимания массива настолько узкий, насколько это возможно, что лучше для производительности.

Например, чтобы получить массив квадратов целых чисел от 1 до 10, может использоваться следующий код.

```
squares = [x^2 for x=1:10]
```

Это чистая, лаконичная замена дольше `for` -loop версии.

```
squares = []
for x in 1:10
    push!(squares, x^2)
end
```

Условное определение массива

Перед Julia 0.5 невозможно использовать условия внутри массива. Но это уже не так. В Julia 0.5 мы можем использовать условия в следующих условиях:

```
julia> [x^2 for x in 0:9 if x > 5]
```

```
4-element Array{Int64,1}:
 36
 49
 64
 81
```

Источник приведенного выше примера можно найти [здесь](#) .

Если мы хотим использовать понимание вложенного списка:

```
julia> [(x,y) for x=1:5 , y=3:6 if y>4 && x>3 ]
4-element Array{Tuple{Int64,Int64},1}:
 (4,5)
 (5,5)
 (4,6)
 (5,6)
```

Многомерное понимание массива

Вложенные `for` циклов могут использоваться для повторения нескольких уникальных итераций.

```
result = []
for a = iterable_a
    for b = iterable_b
        push!(result, expression)
    end
end
```

Аналогичным образом, множественные спецификации итераций могут быть предоставлены для понимания массива.

```
[expression for a = iterable_a, b = iterable_b]
```

Например, для генерации декартова произведения `1:3` и `1:2` может быть использовано следующее.

```
julia> [(x, y) for x = 1:3, y = 1:2]
3×2 Array{Tuple{Int64,Int64},2}:
 (1,1) (1,2)
 (2,1) (2,2)
 (3,1) (3,2)
```

Сглаженные многомерные измерения массива аналогичны, за исключением того, что они теряют форму. Например,

```
julia> [(x, y) for x = 1:3 for y = 1:2]
6-element Array{Tuple{Int64,Int64},1}:
 (1, 1)
 (1, 2)
 (2, 1)
```

```
(2, 2)
(3, 1)
(3, 2)
```

является сплюснутым вариантом вышеприведенного. Синтаксическая разница заключается в том, что вместо запятой используется дополнительная `for`.

Понимание генератора

Генератор постижения следует формату `(expression for element = iterable)`, аналогичные постижения массива, но использовать круглые скобки `()` вместо того, чтобы квадратные скобки `[]`.

```
(expression for element = iterable)
```

Такое выражение возвращает объект `Generator`.

```
julia> (x^2 for x = 1:5)
Base.Generator{UnitRange{Int64},##1#2} (#1,1:5)
```

Аргументы функции

Понимание генератора может быть предоставлено как единственный аргумент функции, без необходимости в дополнительном наборе круглых скобок.

```
julia> join(x^2 for x = 1:5)
"1491625"
```

Однако, если предоставляется более одного аргумента, понимание генератора требует собственный набор круглых скобок.

```
julia> join(x^2 for x = 1:5, ", ")
ERROR: syntax: invalid iteration specification

julia> join((x^2 for x = 1:5), ", ")
"1, 4, 9, 16, 25"
```

Прочитайте постижения онлайн: <https://riptutorial.com/ru/julia-lang/topic/5477/постижения>

глава 24: Регулярные выражения

Синтаксис

- `Regex ("[регулярное выражение]")`
- `г «[регулярное выражение]»`
- `матч (игла, стог сена)`
- `matchall (игла, стога сена)`
- `eachmatch (игла, стог сена)`
- `ismatch (игла, стог сена)`

параметры

параметр	подробности
<code>needle</code>	<code>Regex</code> искать в <code>haystack</code>
<code>haystack</code>	текст, в котором нужно искать <code>needle</code>

Examples

Литералы регулярных выражений

Джулия поддерживает регулярные выражения ¹. Библиотека PCRE используется как реализация регулярного выражения. Регулярные выражения похожи на мини-язык внутри языка. Поскольку большинство языков и многие текстовые редакторы предоставляют некоторую поддержку регулярного выражения, документация и примеры того, как использовать [регулярное выражение](#) вообще, выходят за рамки этого примера.

Можно построить `Regex` из строки с помощью конструктора:

```
julia> Regex("(cat|dog)s?")
```

Но для удобства и более простого экранирования вместо этого можно использовать [строковый макрос](#) `@r_str`:

```
julia> r"(cat|dog)s?"
```

¹ : Технически, Julia поддерживает регулярные выражения, которые отличаются от и более мощными, чем те, которые называются [регулярными выражениями](#) в теории языка. Часто термин «регулярное выражение» также используется для обозначения регулярных

выражений.

Поиск матчей

Существует четыре основных полезных функции для регулярных выражений, каждая из которых принимает аргументы в порядке `needle`, `haystack`. Терминология «игла» и «стога сена» исходят из английской идиомы «найти иглу в стоге сена». В контексте регулярных выражений регулярное выражение представляет собой иглу, а текст - стог сена.

Функция `match` может использоваться для поиска первого совпадения в строке:

```
julia> match(r"(cat|dog)s?", "my cats are dogs")
RegexMatch("cats", 1="cat")
```

Функция `matchall` может использоваться для поиска всех совпадений регулярного выражения в строке:

```
julia> matchall(r"(cat|dog)s?", "The cat jumped over the dogs.")
2-element Array{SubString{String},1}:
 "cat"
 "dogs"
```

Функция `ismatch` возвращает логическое значение, указывающее, было ли совпадение найдено внутри строки:

```
julia> ismatch(r"(cat|dog)s?", "My pigs")
false

julia> ismatch(r"(cat|dog)s?", "My cats")
true
```

`eachmatch` функция возвращает итератор над `RegexMatch` объектами, пригодный для использования с `for` **петель** :

```
julia> for m in eachmatch(r"(cat|dog)s?", "My cats and my dog")
    println("Matched $(m.match) at index $(m.offset)")
end
Matched cats at index 4
Matched dog at index 16
```

Группы захвата

Подстроки, захваченные **группами захвата**, доступны из объектов `RegexMatch` с использованием нотации индексации.

Например, следующее регулярное выражение анализирует североамериканские телефонные номера, написанные в формате `(555)-555-5555` :

```
julia> phone = r"((\d{3})\)-(\d{3})-(\d{4})"
```

и предположим, что мы хотим извлечь телефонные номера из текста:

```
julia> text = """
    My phone number is (555)-505-1000.
    Her phone number is (555)-999-9999.
    """
"My phone number is (555)-505-1000.\nHer phone number is (555)-999-9999.\n"
```

Используя функцию `matchall`, мы можем получить массив подстрок, соответствующих самим себе:

```
julia> matchall(phone, text)
2-element Array{SubString{String},1}:
 "(555)-505-1000"
 "(555)-999-9999"
```

Но предположим, что мы хотим получить доступ к кодам областей (первые три цифры, заключенные в скобки). Затем мы можем использовать итератор `eachmatch`:

```
julia> for m in eachmatch(phone, text)
    println("Matched $(m.match) with area code $(m[1])")
end
Matched (555)-505-1000 with area code 555
Matched (555)-999-9999 with area code 555
```

Обратите внимание, что мы используем `m[1]` потому что код области является первой группой захвата в нашем регулярном выражении. Мы можем получить все три компонента телефонного номера в виде кортежа, используя функцию:

```
julia> splitmatch(m) = m[1], m[2], m[3]
splitmatch (generic function with 1 method)
```

Затем мы можем применить такую функцию к определенному `RegexMatch`:

```
julia> splitmatch(match(phone, text))
("555", "505", "1000")
```

Или мы могли бы `map` его в каждом матче:

```
julia> map(splitmatch, eachmatch(phone, text))
2-element Array{Tuple{SubString{String},SubString{String},SubString{String}},1}:
 ("555", "505", "1000")
 ("555", "999", "9999")
```

Прочитайте Регулярные выражения онлайн: <https://riptutorial.com/ru/julia-lang/topic/5890/регулярные-выражения>

глава 25: РЕПЛ

Синтаксис

- джулия>
- помочь?>
- оболочка>
- \[латекс]

замечания

Другие пакеты могут определять свои собственные режимы REPL в дополнение к режимам по умолчанию. Например, пакет `сxx` определяет режим оболочки `сxx>` для C ++ REPL. Эти режимы обычно доступны с помощью собственных специальных клавиш; см. документацию по упаковке для более подробной информации.

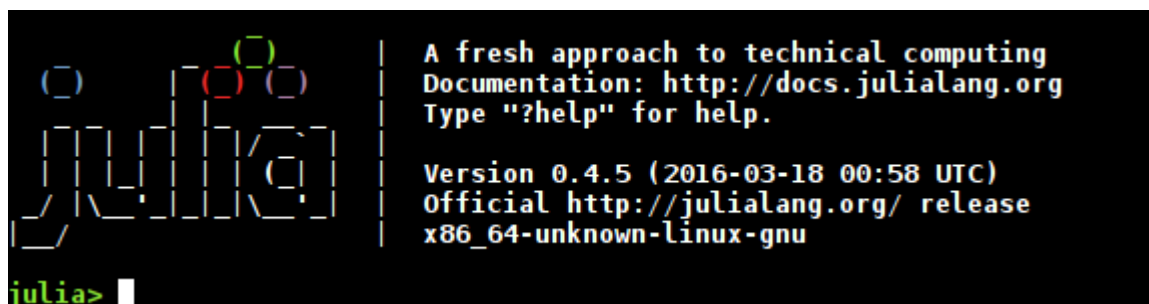
Examples

Запустить REPL

После [установки Julia](#) для запуска цикла read-eval-print-loop (REPL):

В Unix-системах

Откройте окно терминала, затем введите `julia` в приглашении, а затем нажмите `Return` . Вы должны увидеть что-то вроде этого:



В Windows

Найдите программу Julia в меню «Пуск» и щелкните по ней. REPL должен быть запущен.

Использование REPL в качестве калькулятора

Julia REPL - отличный калькулятор. Мы можем начать с простых операций:

```
julia> 1 + 1
2

julia> 8 * 8
64

julia> 9 ^ 2
81
```

Переменная `ans` содержит результат последнего вычисления:

```
julia> 4 + 9
13

julia> ans + 9
22
```

Мы можем определить собственные переменные, используя оператор присваивания `=`:

```
julia> x = 10
10

julia> y = 20
20

julia> x + y
30
```

У Джулии есть неявное умножение для числовых литералов, что позволяет некоторым вычислениям быстрее писать:

```
julia> 10x
100

julia> 2(x + y)
60
```

Если мы допустим ошибку и сделаем то, что не разрешено, Julia REPL выдает ошибку, часто с полезным советом о том, как исправить проблему:

```
julia> 1 ^ -1
ERROR: DomainError:
Cannot raise an integer x to a negative power -n.
Make x a float by adding a zero decimal (e.g. 2.0^-n instead of 2^-n), or write
1/x^n, float(x)^-n, or (x//1)^-n.
in power_by_squaring at ./intfuncs.jl:82
in ^ at ./intfuncs.jl:106

julia> 1.0 ^ -1
1.0
```

Для доступа или редактирования предыдущих команд используйте клавишу ↑ (Вверх), которая перемещается к последнему элементу в истории. ↓ переходит к следующему элементу в истории. Клавиши ← и → можно использовать для перемещения и редактирования в строке.

У Джулии есть некоторые встроенные математические константы, включая e и π (или π).

```
julia> e
e = 2.7182818284590...

julia> pi
π = 3.1415926535897...

julia> 3π
9.42477796076938
```

Мы можем набирать символы типа π быстро, используя их коды LaTeX: нажмите \ , затем p и i , затем нажмите клавишу Tab, чтобы подставить \pi просто набранный π . Это работает для других греческих букв и дополнительных символов Юникода.

Мы можем использовать любые встроенные математические функции Julia, которые варьируются от простых до довольно мощных:

```
julia> cos(π)
-1.0

julia> besselh(1, 1, 1)
0.44005058574493355 - 0.7812128213002889im
```

Сложные числа поддерживаются с помощью im как мнимой единицы:

```
julia> abs(3 + 4im)
5.0
```

Некоторые функции не возвратят сложный результат, если вы не дадите ему сложный ввод, даже если вход является реальным:

```
julia> sqrt(-1)
ERROR: DomainError:
sqrt will only return a complex result if called with a complex argument. Try
sqrt(complex(x)).
in sqrt at math.jl:146

julia> sqrt(-1+0im)
0.0 + 1.0im

julia> sqrt(complex(-1))
0.0 + 1.0im
```

Точные операции над рациональными числами возможны с использованием оператора // рационального деления:

```
julia> 1//3 + 1//3
2//3
```

См. Раздел « [Арифметика](#)» для получения дополнительной информации о том, какие типы арифметических операторов поддерживаются Джулией.

Работа с точностью машины

Обратите внимание, что целые числа машины ограничены по размеру и будут **переполняться**, если результат слишком велик для хранения:

```
julia> 2^62
4611686018427387904

julia> 2^63
-9223372036854775808
```

Этого можно предотвратить с помощью целых чисел произвольной точности в вычислении:

```
julia> big"2"^62
4611686018427387904

julia> big"2"^63
9223372036854775808
```

Точки с плавающей запятой также ограничены точностью:

```
julia> 0.1 + 0.2
0.30000000000000004
```

Больше (но все же ограничено) точность возможна за счет использования `big` :

[illegible]

Точная арифметика может быть выполнена в некоторых случаях с использованием Rational S:

```
julia> 1//10 + 2//10
3//10
```

Использование режимов REPL

В Julia есть три встроенных режима REPL: режим Julia, режим помощи и режим оболочки.

Режим справки

Julia REPL поставляется со встроенной системой помощи. Нажмите `?` в приглашении `julia>` для доступа к `help?>`.

В командной строке введите имя какой-либо функции или типа, чтобы получить справку для:

```
help?> abs
search: abs abs2 abspath abstract AbstractRNG AbstractFloat AbstractArray

abs(x)

The absolute value of x.

When abs is applied to signed integers, overflow may occur, resulting in the
return of a negative value. This overflow occurs only when abs is applied to the
minimum representable value of a signed integer. That is, when x ==
typemin(typeof(x)), abs(x) == x < 0, not -x as might be expected.
```

Даже если вы не правильно произнесете функцию, Джулия может предложить некоторые функции, которые, возможно, вы имели в виду:

```
help?> printline
search:

Couldn't find printline
Perhaps you meant println, pipeline, @inline or print
No documentation found.

Binding printline does not exist.
```

Эта документация работает и для других модулей, если они используют систему документации Julia.

```
julia> using Currencies

help?> @usingcurrencies
Export each given currency symbol into the current namespace. The individual unit
exported will be a full unit of the currency specified, not the smallest possible
unit. For instance, @usingcurrencies EUR will export EUR, a currency unit worth
1€, not a currency unit worth 0.01€.

@usingcurrencies EUR, GBP, AUD
7AUD # 7.00 AUD

There is no sane unit for certain currencies like XAU or XAG, so this macro does
not work for those. Instead, define them manually:

const XAU = Monetary(:XAU; precision=4)
```

Режим оболочки

См [Использование Shell внутри РЕПЛ](#) для получения более подробной информации о том, как использовать режим оболочки Джулии, которая доступна, [нажав](#); в подсказке. Этот

режим оболочки поддерживает интерполяцию данных из сеанса Julia REPL, что позволяет легко вызвать функции Julia и сделать их результаты в командах оболочки:

```
shell> ls $(Pkg.dir("JSON"))  
appveyor.yml bench data LICENSE.md nohup.out README.md REQUIRE src test
```

Прочитайте РЕПЛ онлайн: <https://riptutorial.com/ru/julia-lang/topic/5739/репл>

глава 26: Скрипты и трубопроводы оболочки

Синтаксис

- команда оболочки

Examples

Использование оболочки из REPL

Изнутри оболочки Interactive (также известной как REPL) вы можете получить доступ к оболочке системы, набрав ; сразу после подсказки:

```
shell>
```

С этого момента вы можете ввести любой командой оболочки, и они будут запущены из REPL:

```
shell> ls
Desktop      Documents  Pictures   Templates
Downloads    Music      Public     Videos
```

Чтобы выйти из этого режима, введите `backspace` когда приглашение пуст.

Исключение из кода Юлии

Код Julia может создавать, манипулировать и выполнять командные литералы, которые выполняются в системной среде ОС. Это мощно, но часто делает программы менее переносимыми.

Литерал команды может быть создан с использованием ``` literal`. Информацию можно интерполировать с использованием синтаксиса `$` интерполяции, как со строковыми литералами. Переменные Julia, прошедшие через командные литералы, не обязательно должны быть экранированы первыми; они фактически не передаются в оболочку, а скорее непосредственно в ядро. Однако Джулия отображает эти объекты так, чтобы они выглядели правильно экранированными.

```
julia> msg = "a commit message"
"a commit message"

julia> command = `git commit -am $msg`
`git commit -am 'a commit message'`
```

```
julia> cd("/directory/where/there/are/unstaged/changes")

julia> run(command)
[master (root-commit) 0945387] add a
 4 files changed, 1 insertion(+)
```

Прочитайте Скрипты и трубопроводы оболочки онлайн: <https://riptutorial.com/ru/julia-lang/topic/5420/скрипты-и-трубопроводы-оболочки>

глава 27: Словари

Examples

Использование словарей

Словари можно построить, передав ему любое количество пар.

```
julia> Dict{"A"=>1, "B"=>2}
Dict{String,Int64} with 2 entries:
  "B" => 2
  "A" => 1
```

Вы можете получить записи в словаре, помещая ключ в квадратные скобки.

```
julia> dict = Dict{"A"=>1, "B"=>2}
Dict{String,Int64} with 2 entries:
  "B" => 2
  "A" => 1

julia> dict["A"]
1
```

Прочитайте Словари онлайн: <https://riptutorial.com/ru/julia-lang/topic/9028/словари>

глава 28: Совместимость с несколькими версиями

Синтаксис

- использование сокетов
- `Compat.String`
- `Compat.UTF8String`
- `@compat f. (x, y)`

замечания

Иногда очень сложно получить новый синтаксис, чтобы хорошо играть с несколькими версиями. Поскольку Julia все еще активно развивается, часто полезно просто отказаться от поддержки старых версий и вместо этого нацелить только более новые.

Examples

Номера версий

У Джулии есть встроенная реализация [семантического управления версиями](#), открытая через тип `VersionNumber`.

Чтобы построить `VersionNumber` как литерал, можно использовать [строковый макрос](#) `@v_str`:

```
julia> vers = v"1.2.0"  
v"1.2.0"
```

В качестве альтернативы можно вызвать конструктор `VersionNumber`; обратите внимание, что конструктор принимает до пяти аргументов, но все, кроме первого, являются необязательными.

```
julia> vers2 = VersionNumber(1, 1)  
v"1.1.0"
```

Номера версий можно сравнивать с помощью [операторов сравнения](#) и, следовательно, их можно сортировать:

```
julia> vers2 < vers  
true  
  
julia> v"1" < v"0"  
false
```

```
julia> sort([v"1.0.0", v"1.0.0-dev.100", v"1.0.1"])
3-element Array{VersionNumber,1}:
 v"1.0.0-dev.100"
 v"1.0.0"
 v"1.0.1"
```

Номера версий используются в нескольких местах по всей Юлии. Например, константа `VERSION` является `VersionNumber` :

```
julia> VERSION
v"0.5.0"
```

Это обычно используется для оценки условного кода, в зависимости от версии Julia. Например, для запуска другого кода на v0.4 и v0.5 можно сделать

```
if VERSION < v"0.5"
    println("v0.5 prerelease, v0.4 or older")
else
    println("v0.5 or newer")
end
```

Каждый установленный **пакет** также связан с текущим номером версии:

```
julia> Pkg.installed("StatsBase")
v"0.9.0"
```

Использование `Compat.jl`

Пакет [Compat.jl](#) позволяет использовать некоторые новые функции и синтаксис Julia со старыми версиями Julia. Его функции документированы на его README, но ниже приводится описание полезных приложений.

0.5.0

Унифицированный тип строки

В Julia v0.4 было много разных типов **строк** . Эта система считалась слишком сложной и запутанной, поэтому в Julia v0.5 остается только тип `String` . `Compat` позволяет использовать тип и конструктор `String` версии 0.4 под именем `Compat.String` . Например, этот код v0.5

```
buf = IOBuffer()
println(buf, "Hello World!")
String(buf) # "Hello World!\n"
```

могут быть непосредственно переведены на этот код, который работает как с v0.5, так и с v0.4:

```
using Compat
buf = IOBuffer()
println(buf, "Hello World!")
Compat.String(buf) # "Hello World!\n"
```

Обратите внимание, что есть некоторые оговорки.

- На v0.4, `Compat.String` typealiased в `ByteString`, который является `Union{ASCIIString, UTF8String}`. Таким образом, типы с `String` полями не будут стабильными по типу. В этих ситуациях рекомендуется `Compat.UTF8String`, так как это будет означать `String` на v0.5 и `UTF8String` на v0.4, оба из которых являются конкретными типами.
- Нужно быть осторожным, чтобы использовать `Compat.String` или `import Compat: String`, потому что сама `String` имеет значение в v0.4: это устаревший псевдоним для `AbstractString`. Знак того, что `String` был случайно использован вместо `Compat.String`, если в любой момент появляются следующие предупреждения:

```
WARNING: Base.String is deprecated, use AbstractString instead.
likely near no file:0
WARNING: Base.String is deprecated, use AbstractString instead.
likely near no file:0
```

Компактный синтаксис вещания

Julia v0.5 вводит синтаксический сахар для `broadcast`. Синтаксис

```
f.(x, y)
```

для `broadcast(f, x, y)`. Примеры использования этого синтаксиса включают `sin.([1, 2, 3])` чтобы взять синус из нескольких чисел одновременно.

На v0.5 синтаксис можно использовать напрямую:

```
julia> sin.([1.0, 2.0, 3.0])
3-element Array{Float64,1}:
 0.841471
 0.909297
 0.14112
```

Однако, если мы попробуем то же самое на v0.4, мы получим ошибку:

```
julia> sin.([1.0, 2.0, 3.0])
ERROR: TypeError: getfield: expected Symbol, got Array{Float64,1}
```

К счастью, `Compat` делает этот новый синтаксис применимым и от v0.4. Еще раз добавим `using Compat`. На этот раз мы окружаем выражение `@compat`:

```
julia> using Compat
```

```
julia> @compat sin.([1.0, 2.0, 3.0])  
3-element Array{Float64,1}:  
 0.841471  
 0.909297  
 0.14112
```

Прочитайте Совместимость с несколькими версиями онлайн: <https://riptutorial.com/ru/julia-lang/topic/5832/совместимость-с-несколькими-версиями>

глава 29: Сравнения

Синтаксис

- $x < y$ #, если x строго меньше y
- $x > y$ #, если x строго больше y
- $x == y$ #, если x равно y
- $x === y$ # альтернативно $x \equiv y$, если x равно y
- $x \leq y$ # альтернативно $x \leq y$, если x меньше или равно y
- $x \geq y$ # альтернативно $x \geq y$, если x больше или равно y
- $x \neq y$ # альтернативно $x \neq y$, если x не равно y
- $x \approx y$ #, если x приблизительно равно y

замечания

Будьте осторожны с переворачиванием сравнительных знаков. Julia определяет множество функций сравнения по умолчанию без определения соответствующей перевернутой версии. Например, можно запустить

```
julia> Set{Int}(1:3) ⊆ Set{Int}(0:5)
true
```

но это не работает

```
julia> Set{Int}(0:5) ⊇ Set{Int}(1:3)
ERROR: UndefVarError: ⊇ not defined
```

Examples

Связанные сравнения

Операторы множественного сравнения, используемые вместе, связаны цепями, как если бы они были связаны с помощью [оператора &&](#). Это может быть полезно для читаемых и математически сжатых цепочек сравнения, таких как

```
# same as 0 < i && i <= length(A)
isinbounds(A, i) = 0 < i <= length(A)

# same as Set{Int}() != x && issubset(x, y)
isnonemptysubset(x, y) = Set{Int}() != x && x ⊆ y
```

Однако существует важное различие между $a > b > c$ и $a > b \ \&\& \ b > c$; в последнем, термин b оценивается дважды. Это не имеет большого значения для простых старых символов, но

может иметь значение, если сами термины имеют побочные эффекты. Например,

```
julia> f(x) = (println(x); 2)
f (generic function with 1 method)

julia> 3 > f("test") > 1
test
true

julia> 3 > f("test") && f("test") > 1
test
test
true
```

Давайте более подробно рассмотрим скоординированные сравнения и как они работают, видя, как они анализируются и опускаются в [выражения](#) . Во-первых, рассмотрим простое сравнение, которое мы видим только как простой старый вызов функции:

```
julia> dump(: (a > b))
Expr
  head: Symbol call
  args: Array{Any}((3,))
    1: Symbol >
    2: Symbol a
    3: Symbol b
  typ: Any
```

Теперь, если мы свяжем сравнение, мы замечаем, что синтаксический анализ изменился:

```
julia> dump(: (a > b >= c))
Expr
  head: Symbol comparison
  args: Array{Any}((5,))
    1: Symbol a
    2: Symbol >
    3: Symbol b
    4: Symbol >=
    5: Symbol c
  typ: Any
```

После разбора выражение затем опускается до его окончательной формы:

```
julia> expand(: (a > b >= c))
:(begin
    unless a > b goto 3
    return b >= c
  3:
    return false
end)
```

и мы действительно отмечаем, что это то же самое, что для `a > b && b >= c` :

```
julia> expand(: (a > b && b >= c))
:(begin
```

```

        unless a > b goto 3
        return b >= c
    3:
        return false
end)

```

Порядковые номера

Мы рассмотрим, как реализовать пользовательские сравнения путем реализации пользовательского типа, [порядковых номеров](#). Чтобы упростить реализацию, мы сосредоточимся на небольшом подмножестве этих чисел: все порядковые числа до, но не включая ϵ_0 . Наша реализация ориентирована на простоту, а не на скорость; однако реализация также не замедляется.

Мы храним порядковые числа по их [обычной форме Кантора](#). Поскольку порядковая арифметика не является коммутативной, мы сначала возьмем общее соглашение о сохранении наиболее значимых терминов.

```

immutable OrdinalNumber <: Number
  βs::Vector{OrdinalNumber}
  cs::Vector{Int}
end

```

Так как нормальная форма Кантора единственна, мы можем проверить равенство просто через рекурсивное равенство:

0.5.0

В версии v0.5 есть очень хороший синтаксис для этого компактно:

```

import Base: ==
α::OrdinalNumber == β::OrdinalNumber = α.βs == β.βs && α.cs == β.cs

```

0.5.0

В противном случае определите функцию как более типичную:

```

import Base: ==
==(α::OrdinalNumber, β::OrdinalNumber) = α.βs == β.βs && α.cs == β.cs

```

Чтобы закончить наш заказ, потому что этот тип имеет полный порядок, мы должны перегрузить функцию `isless`:

```

import Base: isless
function isless(α::OrdinalNumber, β::OrdinalNumber)
    for i in 1:min(length(α.cs), length(β.cs))
        if α.βs[i] < β.βs[i]
            return true
        elseif α.βs[i] == β.βs[i] && α.cs[i] < β.cs[i]
            return true
        end
    end
end

```

```

    end
  end
  return length(a.cs) < length( $\beta$ .cs)
end

```

Чтобы проверить наш заказ, мы можем создать некоторые методы для создания порядковых номеров. Нуль, конечно, получается, если в нормальной кантонской форме нет терминов:

```
const ORDINAL_ZERO = OrdinalNumber([], [])
Base.zero(::Type{OrdinalNumber}) = ORDINAL_ZERO
```

Мы можем определить $\exp \omega$ для вычисления ω^α и использовать его для вычисления 1 и ω :

```
expw(α) = OrdinalNumber([α], [1])
const ORDINAL_ONE = expw(ORDINAL_ZERO)
Base.one(::Type{OrdinalNumber}) = ORDINAL_ONE
const ω = expw(ORDINAL_ONE)
```

Теперь у нас есть полностью функциональная функция упорядочения по порядковым номерам:

```
julia> ORDINAL_ZERO < ORDINAL_ONE < ω < expω(ω)
true

julia> ORDINAL_ONE > ORDINAL_ZERO
true

julia> sort([ORDINAL_ONE, ω, expω(ω), ORDINAL_ZERO])

4-element Array{OrdinalNumber,1}:

OrdinalNumber{OrdinalNumber[],Int64[]})

OrdinalNumber{OrdinalNumber[OrdinalNumber{OrdinalNumber[],Int64[]}], [1]}

OrdinalNumber{OrdinalNumber[OrdinalNumber{OrdinalNumber[OrdinalNumber{OrdinalNumber[],Int64[]}], [1]}], [1]}

OrdinalNumber{OrdinalNumber[OrdinalNumber{OrdinalNumber[OrdinalNumber{OrdinalNumber[OrdinalNumbe
```

В последнем примере мы видим, что печать порядковых номеров может быть лучше, но результат такой, как ожидалось.

Стандартные операторы

Джулия поддерживает очень большой набор операторов сравнения. Они включают

[illegible]

□ □ □ □ □ □ □ ⊥ ⊥ ;

3. Операторы $<: , >: , .! ,$ и in , которому не может предшествовать точка $(.)$.

доступны для других пакетов для определения и использования по мере необходимости.

функции для упорядочения; см. раздел «Синтаксис» для списка.

могут быть вызваны как функции. Например, $(<)(1, 2)$ идентично по значению $1 < 2$.

Использование ==, === и isequal

является оператором, но это функция, и все операторы являются функциями.)

Когда использовать ==

текущем состоянии одно и то же значение.

Например, очевидно, что

```
julia> 1 == 1
true
```

НО, КРОМЕ ТОГО,

```
julia> 1 == 1.0
true

julia> 1 == 1.0 + 0.0im
true

julia> 1 == 1//1
true
```

представляют одно и то же значение.

Для изменяемых объектов, например **массивов**, == сравнивает их текущее значение.

```

julia> A = [1, 2, 3]
3-element Array{Int64,1}:
 1
 2
 3

julia> B = [1, 2, 3]
3-element Array{Int64,1}:
 1
 2
 3

julia> C = [1, 3, 2]
3-element Array{Int64,1}:
 1
 3
 2

julia> A == B
true

julia> A == C
false

julia> A[2], A[3] = A[3], A[2] # swap 2nd and 3rd elements of A
(3,2)

julia> A
3-element Array{Int64,1}:
 1
 3
 2

julia> A == B
false

julia> A == C
true

```

В большинстве случаев `==` правильный выбор.

Когда использовать `===`

`===` намного более строгая операция, чем `==`. Вместо равенства ценности он измеряет эгоизм. Два объекта являются эгальными, если они не могут отличаться друг от друга самой программой. Таким образом, мы имеем

```

julia> 1 === 1
true

```

так как нет возможности рассказать `1` отдельно от другого `1`. Но

```

julia> 1 === 1.0
false

```

потому что хотя 1 и 1.0 имеют одинаковое значение, они имеют разные типы, поэтому программа может отличать их друг от друга.

Более того,

```
julia> A = [1, 2, 3]
3-element Array{Int64,1}:
 1
 2
 3

julia> B = [1, 2, 3]
3-element Array{Int64,1}:
 1
 2
 3

julia> A === B
false

julia> A === A
true
```

что может показаться на первый взгляд удивительным! Как программа могла различать два вектора `A` и `B`? Поскольку векторы являются изменяемыми, он может изменять `A`, а затем он будет вести себя иначе, чем `B`. Но независимо от того, как он изменяет `A`, `A` всегда будет вести себя так же, как сам `A`. Таким образом, `A` является эгалным для `A`, но не имеет значения для `B`.

Продолжая в этом ключе, наблюдайте

```
julia> C = A
3-element Array{Int64,1}:
 1
 2
 3

julia> A === C
true
```

Назначая `A` на `C`, мы говорим, что `C` имеет *псевдоним* `A`. То есть, это стало просто другим именем для `A`. Любые изменения, сделанные для `A` будут также наблюдаться с помощью `C`. Поэтому нет никакого способа рассказать разницу между `A` и `C`, поэтому они являются эгалными.

Когда использовать `isequal`

Разница между `==` и `isequal` очень тонкая. Самая большая разница в том, как обрабатываются числа с плавающей запятой:

```
julia> NaN == NaN
```

```
false
```

Этот неожиданный результат **определяется** стандартом IEEE для типов с плавающей точкой (IEEE-754). Но это не полезно в некоторых случаях, например, для сортировки. `isequal` предоставляется для таких случаев:

```
julia> isequal(NaN, NaN)
true
```

На оборотной стороне спектра `==` обрабатывает отрицательный ноль IEEE и положительный ноль как одно и то же значение (также как указано в IEEE-754). Однако эти значения имеют различные представления в памяти.

```
julia> 0.0
0.0

julia> -0.0
-0.0

julia> 0.0 == -0.0
true
```

Опять же, для сортировки, между ними `isequal` различие.

```
julia> isequal(0.0, -0.0)
false
```

Прочитайте Сравнения онлайн: <https://riptutorial.com/ru/julia-lang/topic/5563/сравнения>

глава 30: Стабильность типа

Вступление

Нестабильность типа возникает, когда **тип** переменной может меняться во время выполнения и, следовательно, не может быть выведен во время компиляции. Типичная нестабильность часто вызывает проблемы с производительностью, поэтому важно писать и идентифицировать стабилизирующий код.

Examples

Создать стабильный код типа

```
function sumofsins1(n::Integer)
    r = 0
    for i in 1:n
        r += sin(3.4)
    end
    return r
end

function sumofsins2(n::Integer)
    r = 0.0
    for i in 1:n
        r += sin(3.4)
    end
    return r
end
```

Сроки выполнения вышеуказанных двух функций показывают значительные различия в распределении времени и памяти.

```
julia> @time [sumofsins1(100_000) for i in 1:100];
0.638923 seconds (30.12 M allocations: 463.094 MB, 10.22% gc time)

julia> @time [sumofsins2(100_000) for i in 1:100];
0.163931 seconds (13.60 k allocations: 611.350 KB)
```

Это происходит из-за нестабильного типа кода в `sumofsins1` где тип `r` должен быть проверен для каждой итерации.

Прочитайте **Стабильность типа онлайн**: <https://riptutorial.com/ru/julia-lang/topic/6084/стабильность-типа>

глава 31: Строковые макросы

Синтаксис

- макрос «строка» # короткая, строковая макроформа
- `@macro_str "string"` # long, регулярная макроформа
- `macro`command``

замечания

Макросы String не так сильны, как простые старые строки - поскольку интерполяция должна быть реализована в логике макроса, макросы строк не могут содержать строковые литералы одного и того же разделителя для интерполяции.

Например, хотя

```
julia> "$("x") "  
"x"
```

работает, текстовая форма строки

```
julia> doc"$("x") "  
ERROR: KeyError: key :x not found
```

неправильно анализируется. Это можно несколько смягчить, используя тройные кавычки в качестве внешнего ограничителя строк;

```
julia> doc"""$("x") """  
"x"
```

действительно работает правильно.

Examples

Использование строковых макросов

Макросы строк - это синтаксический сахар для определенных макросов. Синтаксический анализатор расширяет синтаксис как

```
mymacro"my string"
```

В

```
@mymacro_str "my string"
```

который затем, как и любой другой макрокоманд, заменяется любым выражением, возвращаемым макросом `@mymacro_str`. База Julia поставляется с несколькими строковыми макросами, такими как:

@b_str

Этот строковый макрос строит байтовые **массивы** вместо **строк** . Содержимое строки, закодированной как UTF-8, будет использоваться как массив байтов. Это может быть полезно для взаимодействия с низкоуровневыми API-интерфейсами, многие из которых работают с байтовыми массивами вместо строк.

```
julia> b"Hello World!"
12-element Array{UInt8,1}:
 0x48
 0x65
 0x6c
 0x6c
 0x6f
 0x20
 0x57
 0x6f
 0x72
 0x6c
 0x64
 0x21
```

```
@big_str
```

Этот макрос вернет `BigInt` или `BigFloat` проанализированный из строки, которую он дал.

[illegible]

Этот макрос существует, потому что `big(0.1)` не ведет себя так, как изначально ожидалось: `0.1` - это аппроксимация `Float64` истины `0.1 (1/10)`, и продвижение этого `BigFloat` в `BigFloat` будет поддерживать ошибку аппроксимации `Float64`. Использование макроса будет анализировать `0.1` непосредственно на `BigFloat`, уменьшая погрешность аппроксимации.

[illegible]

```
@doc_str
```

Этот строковый макрос `Base.Markdown.MD` объекты `Base.Markdown.MD` , которые используются во внутренней документации для предоставления полнотекстовой документации для любой среды. Эти объекты MD хорошо отображаются в терминале:

```
julia> doc"""
  This is a markdown documentation string.

  ## Heading

  Math ``1 + 2`` and `code` are supported.
  """
This is a markdown documentation string.

  Heading
  =====

  Math 1 + 2 and code are supported.
```

а также в браузере:

```
In [2]: doc"""
  This is a markdown documentation string.

  ## Heading

  Math ``1 + 2`` and `code` are supported.
  """
```

Out[2]: This is a markdown documentation string.

Heading

Math 1 + 2 and code are supported.

@html_str

Этот строковый макрос создает строковые литералы HTML, которые хорошо отображаются в браузере:

```
In [1]: html"""
  <p><abbr title="Hypertext Markup Language">HTML</abbr> text.</p>
  """
```

Out[1]: HTML text.

@ip_str

Этот строковый макрос создает литералы IP-адреса. Он работает как с IPv4, так и с IPv6:

```
julia> ip"127.0.0.1"
ip"127.0.0.1"
```

```
julia> ip::"  
ip::"
```

@r_str

Этот строковый макрос [Regex](#) [литералы](#) [Regex](#) .

@s_str

Этот строковый макрос [SubstitutionString](#) [литералы](#) [SubstitutionString](#) , которые работают вместе с [литералами](#) [Regex](#) чтобы обеспечить более продвинутое текстовое замещение.

@text_str

Этот строковый макрос похож по духу на @doc_str и @html_str , но не имеет каких-либо причудливых особенностей форматирования:

```
In [3]: text"""  
        This is some plain text.  
        """>  
Out[3]: This is some plain text.
```

@v_str

Этот макрос строка [VersionNumber](#) [литералы](#) [VersionNumber](#) . См. [Номера версий](#) для описания того, что они есть и как их использовать.

@MIME_str

Этот строковый макрос создает односторонние типы MIME-типов. Например, `MIME"text/plain"` - это тип `MIME("text/plain")` .

Символы, которые не являются юридическими идентификаторами

Литералы Джулии Символы должны быть юридическими идентификаторами. Это работает:

```
julia> :cat  
:cat
```

Но это не так:

```
julia> :2cat  
ERROR: MethodError: no method matching *(::Int64, ::Base.#cat)  
Closest candidates are:  
  *(::Any, ::Any, ::Any, ::Any...) at operators.jl:288
```

```
*{T<:Union{Int128,Int16,Int32,Int64,Int8,UInt128,UInt16,UInt32,UInt64,UInt8}}(::T<:Union{Int128,Int16,Int32,Int64,Int8,UInt128,UInt16,UInt32,UInt64,UInt8}) at int.jl:33
*{::Real, ::Complex{Bool}} at complex.jl:180
...
```

То, что похоже на буквенный символ здесь, фактически анализируется как неявное умножение `:2` (это всего лишь `2`) и функция `cat`, которая, очевидно, не работает.

Мы можем использовать

```
julia> Symbol("2cat")
Symbol("2cat")
```

для решения этой проблемы.

Строковый макрос может помочь сделать это более кратким. Если мы определяем макрос `@sym_str`:

```
macro sym_str(str)
    Meta.quot(Symbol(str))
end
```

то мы можем просто сделать

```
julia> sym"2cat"
Symbol("2cat")
```

для создания символов, которые не являются действительными идентификаторами Юлии.

Конечно, эти методы также могут создавать символы, которые *являются* действительными идентификаторами Юлии. Например,

```
julia> sym"test"
:test
```

Реализация интерполяции в строковом макросе

Макросы `String` не оснащены встроенными средствами [интерполяции](#). Тем не менее, это можно реализовать вручную. Обратите внимание, что невозможно вставлять без экранирования строковых литералов, которые имеют тот же разделитель, что и окружающий строковый макрос; то есть, хотя возможно `"" $("x") ""`, `" $("x") "` нет. Вместо этого это должно быть экранировано как `" $("\x") "`. Дополнительную информацию об этом ограничении см. В разделе [примечаний](#).

Существует два подхода к внедрению интерполяции вручную: реализовать парсинг вручную или заставить Джулию выполнять синтаксический анализ. Первый подход более гибкий, но второй подход проще.

Ручной анализ

```
macro interp_str(s)
  components = []
  buf = IOBuffer(s)
  while !eof(buf)
    push!(components, rstrip(readuntil(buf, '$'), '$'))
    if !eof(buf)
      push!(components, parse(buf; greedy=false))
    end
  end
  end
  quote
    string($(map(esc, components)...))
  end
end
```

Анализ Юлии

```
macro e_str(s)
  esc(parse("\"$(escape_string(s))\""))
end
```

Этот метод экранирует строку (но учтите, что `escape_string` *не* избежать \$ знаков) и передает его обратно в синтаксический анализатор Джулии разобрать. Экранирование строки необходимо для обеспечения того, чтобы " и \ не влияли на синтаксический анализ строки. Полученное выражение представляет собой `:string` выражение, которое можно анализировать и разлагать для целей макроса.

Командные макросы

0.6.0-DEV

В Julia v0.6 и более поздних версиях макросы команд поддерживаются в дополнение к регулярным строковым макросам. Вызов макроса команды, например

```
mymacro`xyz`
```

анализируется как вызов макроса

```
@mymacro_cmd "xyz"
```

Обратите внимание, что это похоже на строковые макросы, за исключением `_cmd` вместо `_str`.

Обычно мы используем команду макросы для кода, который во многих языках часто содержит " , но редко содержит ` . Например, это довольно просто переопределить простой вариант. [Quasiquoting](#) с помощью команды макросов:

```
macro julia_cmd(s)
    esc(Meta.quot(parse(s)))
end
```

Мы можем использовать этот макрос либо inline:

```
julia> julia`1+1`
:(1 + 1)

julia> julia`hypot2(x,y)=x^2+y^2`
:(hypot2(x,y) = begin # none, line 1:
    x ^ 2 + y ^ 2
end)
```

или многострочный:

```
julia> julia```
function hello()
    println("Hello, World!")
end
```
:(function hello() # none, line 2:
 println("Hello, World!")
end)
```

Поддерживается интерполяция с использованием \$ :

```
julia> x = 2
2

julia> julia`1 + $x`
:(1 + 2)
```

но приведенная здесь версия допускает только одно выражение:

```
julia> julia```
x = 2
y = 3
```
ERROR: ParseError("extra token after end of expression")
```

Однако расширить его для обработки нескольких выражений не сложно.

Прочитайте **Строковые макросы** онлайн: <https://riptutorial.com/ru/julia-lang/topic/5817/строковые-макросы>

глава 32: Струны

Синтаксис

- "[Строка]"
- '[Сканирующее значение Unicode]'
- графемы ([строка])

параметры

параметр	подробности
За	<code>sprint(f, xs...)</code>
f	Функция, которая принимает объект <code>IO</code> качестве первого аргумента.
xs	Нулевой или более оставшиеся аргументы переходят к <code>f</code> .

Examples

Привет, мир!

Строки в Джулии разделяются с помощью " символа:

```
julia> mystring = "Hello, World!"  
"Hello, World!"
```

Обратите внимание , что в отличие от некоторых других языков, ' символ *не может* быть использован вместо. ' определяет *буквенный символ* ; это тип данных `Char` и будет хранить только одно [сканирующее значение Unicode](#) :

```
julia> 'c'  
'c'  
  
julia> 'character'  
ERROR: syntax: invalid character literal
```

Можно извлекать скалярные значения юникода из строки путем итерации по ней с помощью [цикла for](#) :

```
julia> for c in "Hello, World!"  
    println(c)  
end  
H
```

```
e  
l  
l  
o  
,  
  
w  
o  
r  
l  
d  
!
```

графем

Тип Julia's `Char` представляет собой [скалярное значение Unicode](#), которое только в некоторых случаях соответствует тому, что люди воспринимают как «характер». Например, одно представление символа `é`, как и в предыдущем, на самом деле представляет собой комбинацию двух сканирующих значений Unicode:

```
julia> collect("é ")  
2-element Array{Char,1}:  
 'e'  
 ' '
```

Описание Unicode для этих кодовых точек - «LATIN SMALL LETTER E» и «КОМБИНИРОВАНИЕ ОСТРАЯ АКЦЕНТ». Вместе они определяют один «человеческий» характер, который является символом Юникода, называется [графемой](#). Более конкретно, приложение № Юникод № 29 мотивирует определение [кластера графем](#), поскольку:

Важно понимать, что то, что пользователь считает «символом», основным элементом системы написания языка, может быть не только одной кодовой точкой Юникода. Вместо этого эта базовая единица может состоять из нескольких кодовых точек Unicode. Чтобы избежать двусмысленности использования компьютером символа термина, это называется воспринимаемым пользователем символом. Например, «G» + острый акцент воспринимается пользователем: пользователи считают его единственным символом, но на самом деле представлены двумя кодами Unicode. Эти воспринимаемые пользователем символы аппроксимируются так называемым кластером графем, который можно определить программным путем.

Юлия предоставляет `graphemes` функционировать перебрать графемы кластеров в строке:

```
julia> for c in graphemes("résumé ")  
    println(c)  
end  
  
r  
é  
s  
u
```

```
m
é
```

Обратите внимание, как результат, печатающий каждый символ в отдельной строке, лучше, чем если бы мы повторяли скалярные значения Unicode:

```
julia> for c in "ré sumé "
           println(c)
       end
r
e

s
u
m
e
```

Как правило, при работе с символами в воспринимаемом пользователем смысле более полезно иметь дело с кластерами графем, чем с помощью сканирующих значений Unicode. Например, предположим, что мы хотим написать функцию для вычисления длины одного слова. Наивное решение было бы использовать

```
julia> wordlength(word) = length(word)
wordlength (generic function with 1 method)
```

Заметим, что результат противоречит интуиции, когда слово включает кластеры grapheme, которые состоят из нескольких кодовых точек:

```
julia> wordlength("ré sumé ")
8
```

Когда мы используем более правильное определение, используя функцию `graphemes`, получаем ожидаемый результат:

```
julia> wordlength(word) = length(graphemes(word))
wordlength (generic function with 1 method)

julia> wordlength("ré sumé ")
6
```

Преобразование числовых типов в строки

Существует множество способов преобразования числовых типов в строки в Julia:

```
julia> a = 123
123

julia> string(a)
"123"
```

```
julia> println(a)
123
```

Функция `string()` также может принимать больше аргументов:

```
julia> string(a, "b")
"123b"
```

Вы также можете вставить (ака интерполировать) целые числа (и некоторые другие типы) в строки, используя `$` :

```
julia> MyString = "my integer is $a"
"my integer is 123"
```

Совет по производительности . Вышеупомянутые методы могут быть довольно удобными в разы. Но если вы будете выполнять много и много таких операций, и вас беспокоит скорость выполнения вашего кода, руководство по [эффективности](#) Julia рекомендует против этого и вместо этого использует следующие методы:

Вы можете предоставить несколько аргументов `print()` и `println()` которые будут работать с ними точно так же, как `string()` работает с несколькими аргументами:

```
julia> println(a, "b")
123b
```

Или, когда вы пишете файл, вы можете аналогичным образом использовать, например

```
open("/path/to/MyFile.txt", "w") do file
    println(file, a, "b", 13)
end
```

или же

```
file = open("/path/to/MyFile.txt", "a")
println(file, a, "b", 13)
close(file)
```

Это быстрее, потому что они избегают необходимости сначала формировать строку из заданных частей, а затем выводить ее (либо на консольный дисплей, либо на файл), а вместо этого просто выводить различные фрагменты.

Кредиты: Ответ основан на SO Вопрос [Каков наилучший способ преобразования Int в строку в Julia?](#) с Answer by Michael Ohlrogge и вход от Fengyang Wang

Строковая интерполяция (значение вставки, заданное переменной в строку)

В Julia, как и во многих других языках, можно интерполировать, вставив значения, определенные переменными в строки. Для простого примера:

```
n = 2
julia> MyString = "there are $n ducks"
"there are 2 ducks"
```

Мы можем использовать другие типы, кроме числовых, например

```
Result = false
julia> println("test results is $Result")
test results is false
```

Вы можете иметь несколько интерполяций в заданной строке:

```
MySubStr = "a32"
MyNum = 123.31
println("$MySubStr , $MyNum")
```

Интерполяция **производительности подсказка** довольно удобна. Но, если вы собираетесь делать это много раз очень быстро, это не самый эффективный. Вместо этого см. [Преобразование числовых типов в строки](#) для предложений, когда производительность является проблемой.

Использование `sprint` для создания строк с функциями ввода-вывода

Строки могут быть выполнены из функций, которые работают с объектами `IO`, используя функцию `sprint`. Например, функция `code_llvm` принимает объект `IO` в качестве первого аргумента. Как правило, он используется как

```
julia> code_llvm(STDOUT, *, (Int, Int))

define i64 @"jlsys_*_46115"(i64, i64) #0 {
top:
    %2 = mul i64 %1, %0
    ret i64 %2
}
```

Предположим, мы хотим, чтобы этот вывод был как строка. Тогда мы можем просто сделать

```
julia> sprint(code_llvm, *, (Int, Int))
"\ndefine i64 @"jlsys_*_46115\"(i64, i64) #0 {\ntop:\n    %2 = mul i64 %1, %0\n    ret i64 %2\n}\n"

julia> println(ans)

define i64 @"jlsys_*_46115"(i64, i64) #0 {
top:
    %2 = mul i64 %1, %0
```

```
ret i64 %2  
}
```

Преобразование результатов «интерактивных» функций, таких как `code_llvm` в строки, может быть полезно для автоматического анализа, например, [проверка](#) того, может ли сгенерированный код регрессировать.

Функция `sprint` - это функция более [высокого порядка](#), которая в качестве первого аргумента возвращает функцию, действующую на объекты `IO`. За кулисами он создает `IOBuffer` в ОЗУ, вызывает данную функцию и берет данные из буфера в объект `String`.

Прочитайте [Струны онлайн](#): <https://riptutorial.com/ru/julia-lang/topic/5562/струны>

глава 33: Тестирование устройства

Синтаксис

- `@test [expr]`
- `@test_throws [Исключение] [expr]`
- `@testset "[name]" begin; [Тесты]; конец`
- `Pkg.test ([пакет])`

замечания

Стандартная библиотечная документация для `Base.Test` охватывает дополнительные материалы, кроме тех, что показаны в этих примерах.

Examples

Тестирование пакета

Для запуска модульных тестов для пакета используйте функцию `Pkg.test`. Для пакета с именем `MyPackage` команда будет

```
julia> Pkg.test("MyPackage")
```

Ожидаемый результат будет аналогичен

```
INFO: Computing test dependencies for MyPackage...
INFO: Installing BaseTestNext v0.2.2
INFO: Testing MyPackage
Test Summary: | Pass Total
Data          | 66    66
Test Summary: | Pass Total
Monetary      | 107   107
Test Summary: | Pass Total
Basket        | 47    47
Test Summary: | Pass Total
Mixed         | 13    13
Test Summary: | Pass Total
Data Access   | 35    35
INFO: MyPackage tests passed
INFO: Removing BaseTestNext v0.2.2
```

хотя, очевидно, нельзя ожидать, что он будет точно соответствовать указанному выше, поскольку разные пакеты используют разные фреймворки.

Эта команда запускает файл `test/runtests.jl` в чистой среде.

Можно проверить все установленные пакеты одновременно с помощью

```
julia> Pkg.test()
```

но это обычно занимает очень много времени.

Написание простого теста

`test/runtests.jl` тесты объявляются в файле `test/runtests.jl` в пакете. Как правило, этот файл начинается

```
using MyModule
using Base.Test
```

Базовой единицей тестирования является макрос `@test`. Этот макрос подобен утверждению. Любое булевское выражение может быть проверено в макросе `@test`:

```
@test 1 + 1 == 2
@test iseven(10)
@test 9 < 10 || 10 < 9
```

Мы можем попробовать макрос `@test` в REPL:

```
julia> using Base.Test

julia> @test 1 + 1 == 2
Test Passed
  Expression: 1 + 1 == 2
  Evaluated: 2 == 2

julia> @test 1 + 1 == 3
Test Failed
  Expression: 1 + 1 == 3
  Evaluated: 2 == 3
ERROR: There was an error during testing
in record(::Base.Test.FallbackTestSet, ::Base.Test.Fail) at ./test.jl:397
in do_test(::Base.Test.Returned, ::Expr) at ./test.jl:281
```

Тест-макрос можно использовать практически в любом месте, например, в циклах или функциях:

```
# For positive integers, a number's square is at least as large as the number
for i in 1:10
    @test i^2 ≥ i
end

# Test that no two of a, b, or c share a prime factor
function check_pairwise_coprime(a, b, c)
    @test gcd(a, b) == 1
    @test gcd(a, c) == 1
    @test gcd(b, c) == 1
end
```



```
check_pairwise_coprime(10, 23, 119)
```

Написание тестового набора

0.5.0

В версии v0.5 тестовые наборы встроены в стандартный модуль `Base.Test` библиотеки, и вам не нужно ничего делать (кроме `using Base.Test`), чтобы использовать их.

0.4.0

Тестовые наборы не являются частью библиотеки `Base.Test` Julia `Base.Test`. Вместо этого, вы должны `REQUIRE` в `BaseTestNext` модуль и добавить `using BaseTestNext` в файл. Чтобы поддерживать как версии 0.4, так и 0.5, вы можете использовать

```
if VERSION ≥ v"0.5.0-dev+7720"
    using Base.Test
else
    using BaseTestNext
    const Test = BaseTestNext
end
```

Полезно группировать связанные `@test` s вместе в тестовом наборе. В дополнение к более четкой организации тестирования тестовые комплекты обеспечивают лучшую производительность и большую настраиваемость.

Чтобы определить тестовый набор, просто оберните любое количество `@test` s блоком `@testset`:

```
@testset "+" begin
    @test 1 + 1 == 2
    @test 2 + 2 == 4
end

@testset "*" begin
    @test 1 * 1 == 1
    @test 2 * 2 == 4
end
```

Запуск этих тестовых наборов выводит следующий результат:

```
Test Summary: | Pass  Total
+             |    2     2

Test Summary: | Pass  Total
*             |    2     2
```

Даже если тестовый набор содержит тест с ошибкой, весь тестовый набор будет запущен до завершения, и сбои будут записываться и сообщаться:

```
@testset "-" begin
    @test 1 - 1 == 0
    @test 2 - 2 == 1
    @test 3 - () == 3
    @test 4 - 4 == 0
end
```

Выполнение этого набора тестов приводит к

```
 -: Test Failed
   Expression: 2 - 2 == 1
   Evaluated: 0 == 1
   in record(::Base.Test.DefaultTestSet, ::Base.Test.Fail) at ./test.jl:428
   ...
 -: Error During Test
   Test threw an exception of type MethodError
   Expression: 3 - () == 3
   MethodError: no method matching -(::Int64, ::Tuple{})
   ...
Test Summary: | Pass  Fail  Error  Total
-             |    2    1      1      4
ERROR: Some tests did not pass: 2 passed, 1 failed, 1 errored, 0 broken.
...
```

Наборы тестов могут быть вложенными, позволяя произвольно глубокую организацию

```
@testset "Int" begin
    @testset "+" begin
        @test 1 + 1 == 2
        @test 2 + 2 == 4
    end
    @testset "-" begin
        @test 1 - 1 == 0
    end
end
```

Если тесты пройдут, то это покажет только результаты для самого внешнего тестового набора:

```
Test Summary: | Pass  Total
Int           |    3      3
```

Но если тесты терпят неудачу, то сообщается о детализации тестового набора и теста, вызывающего отказ.

Макрос `@testset` можно использовать с [ЦИКЛОМ for](#) для создания множества тестовых наборов одновременно:

```
@testset for i in 1:5
    @test 2i == i + i
    @test i^2 == i * i
    @test i ÷ i == 1
end
```

который сообщает

```
Test Summary: | Pass  Total
i = 1         |    3    3
Test Summary: | Pass  Total
i = 2         |    3    3
Test Summary: | Pass  Total
i = 3         |    3    3
Test Summary: | Pass  Total
i = 4         |    3    3
Test Summary: | Pass  Total
i = 5         |    3    3
```

Общей структурой является наличие внешних тестовых наборов для тестирования компонентов или типов. Внутри этих внешних наборов тестов внутренний тест устанавливает поведение теста. Например, предположим, что мы создали тип `UniversalSet` с экземпляром `singleton`, который содержит все. Прежде чем мы даже реализуем этот тип, мы можем использовать **основанные** на тестах принципы **разработки** и выполнить тесты:

```
@testset "UniversalSet" begin
  U = UniversalSet.instance
  @testset "egal/equal" begin
    @test U === U
    @test U == U
  end

  @testset "in" begin
    @test 1 in U
    @test "Hello World" in U
    @test Int in U
    @test U in U
  end

  @testset "subset" begin
    @test Set() ⊆ U
    @test Set(["Hello World"]) ⊆ U
    @test Set(1:10) ⊆ U
    @test Set([:a, 2.0, "w", Set()]) ⊆ U
    @test U ⊆ U
  end
end
```

Затем мы можем начать реализацию нашей функциональности до тех пор, пока она не пройдет наши тесты. Первый шаг - определить тип:

```
immutable UniversalSet <: Base.AbstractSet end
```

Только два наших теста проходят прямо сейчас. Мы можем реализовать `in` :

```
immutable UniversalSet <: Base.AbstractSet end
Base.in(x, ::UniversalSet) = true
```

Это также делает некоторые из наших тестов подмножества. Однако `issubset (⊆)` не

работает для `UniversalSet`, потому что резервное копирование пытается перебрать элементы, чего мы не сможем сделать. Мы можем просто определить специализацию, которая делает `issubset return true` для любого набора:

```
immutable UniversalSet <: Base.AbstractSet end
Base.in(x, ::UniversalSet) = true
Base.issubset(x::Base.AbstractSet, ::UniversalSet) = true
```

И теперь все наши тесты проходят!

Исключения для тестирования

Исключения, возникающие при запуске теста, не пройдут проверку, и если тест не находится в тестовом наборе, завершите работу механизма тестирования. Обычно это хорошо, потому что в большинстве ситуаций исключения не являются желаемым результатом. Но иногда хочется конкретно проверить, что возникает определенное исключение. Макрос `@test_throws` облегчает это.

```
julia> @test_throws BoundsError [1, 2, 3][4]
Test Passed
  Expression: ([1,2,3])[4]
    Thrown: BoundsError
```

Если `@test_throws` неправильное исключение, `@test_throws` все равно не будет выполнено:

```
julia> @test_throws TypeError [1, 2, 3][4]
Test Failed
  Expression: ([1,2,3])[4]
    Expected: TypeError
    Thrown: BoundsError
ERROR: There was an error during testing
in record(::Base.Test.FallbackTestSet, ::Base.Test.Fail) at ./test.jl:397
in do_test_throws(::Base.Test.Threw, ::Expr, ::Type{T}) at ./test.jl:329
```

и если исключение не будет `@test_throws`, `@test_throws` также не будет выполнено:

```
julia> @test_throws BoundsError [1, 2, 3, 4][4]
Test Failed
  Expression: ([1,2,3,4])[4]
    Expected: BoundsError
    No exception thrown
ERROR: There was an error during testing
in record(::Base.Test.FallbackTestSet, ::Base.Test.Fail) at ./test.jl:397
in do_test_throws(::Base.Test.Returned, ::Expr, ::Type{T}) at ./test.jl:329
```

Приближенное равенство

Какова сделка со следующим?

```
julia> @test 0.1 + 0.2 == 0.3
```

```
Test Failed
 Expression: 0.1 + 0.2 == 0.3
  Evaluated: 0.30000000000000004 == 0.3
ERROR: There was an error during testing
 in record(::Base.Test.FallbackTestSet, ::Base.Test.Fail) at ./test.jl:397
 in do_test(::Base.Test.Returned, ::Expr) at ./test.jl:281
```

Ошибка вызвана тем фактом, что ни один из 0.1 , 0.2 и 0.3 не представлен на компьютере как именно эти значения - $1/10$, $2/10$ и $3/10$. Вместо этого они аппроксимируются значениями, которые очень близки. Но, как видно из вышеприведенного теста, при добавлении двух приближений результат может быть немного хуже приближения, чем это возможно. [Этой теме гораздо больше](#), чем здесь.

Но нам не повезло! Чтобы проверить, что комбинация округления с числом с плавающей запятой и арифметикой с плавающей точкой является *приблизительно* правильной, даже если она не является точной, мы можем использовать функцию `isapprox` (что соответствует оператору \approx). Поэтому мы можем переписать наш тест как

```
julia> @test 0.1 + 0.2 ≈ 0.3
Test Passed
 Expression: 0.1 + 0.2 ≈ 0.3
  Evaluated: 0.30000000000000004 isapprox 0.3
```

Конечно, если наш код был полностью неправильным, тест все равно поймет, что:

```
julia> @test 0.1 + 0.2 ≈ 0.4
Test Failed
 Expression: 0.1 + 0.2 ≈ 0.4
  Evaluated: 0.30000000000000004 isapprox 0.4
ERROR: There was an error during testing
 in record(::Base.Test.FallbackTestSet, ::Base.Test.Fail) at ./test.jl:397
 in do_test(::Base.Test.Returned, ::Expr) at ./test.jl:281
```

Функция `isapprox` использует эвристику, основанную на размере чисел и точности типа с плавающей запятой, чтобы определить допустимую погрешность. Он не подходит для всех ситуаций, но он работает в большинстве случаев и экономит много усилий, `isapprox` собственную версию `isapprox`.

Прочитайте Тестирование устройства онлайн: <https://riptutorial.com/ru/julia-lang/topic/5632/тестирование-устройства>

глава 34: Типы

Синтаксис

- неизменный `MyType`; поле; поле; конец
- тип `MyType`; поле; поле; конец

замечания

Типы являются ключевыми для работы Юлии. Важной идеей для производительности является [стабильность типа](#), которая возникает, когда возвращаемый тип функции зависит только от типов, а не от значений его аргументов.

Examples

Отправка по типам

В Julia вы можете определить несколько методов для каждой функции. Предположим, что мы определяем три метода одной и той же функции:

```
foo(x) = 1
foo(x::Number) = 2
foo(x::Int) = 3
```

Когда вы решаете, какой метод использовать (называется [диспетчером](#)), Джулия выбирает более конкретный метод, который соответствует типам аргументов:

```
julia> foo('one')
1

julia> foo(1.0)
2

julia> foo(1)
3
```

Это облегчает [полиморфизм](#). Например, мы можем легко создать [связанный список](#), указав два неизменных типа с именами `Nil` и `Cons`. Эти имена традиционно используются для описания пустого списка и непустого списка, соответственно.

```
abstract LinkedList
immutable Nil <: LinkedList end
immutable Cons <: LinkedList
    first
    rest::LinkedList
end
```

Мы представляем пустой список `Nil()` и любые другие списки с помощью `Cons(first, rest)`, где `first` - это первый элемент связанного списка, а `rest` - связанный список, состоящий из всех остальных элементов. Например, список `[1, 2, 3]` будет представлен как

```
julia> Cons(1, Cons(2, Cons(3, Nil())))  
Cons{1, Cons{2, Cons{3, Nil{}}}}
```

Список пуст?

Предположим, мы хотим расширить функцию стандартного библиотечного `isempty`, которая работает с множеством различных коллекций:

```
julia> methods(isempty)  
# 29 methods for generic function "isempty":  
isempty(v::SimpleVector) at essentials.jl:180  
isempty(m::Base.MethodList) at reflection.jl:394  
...
```

Мы можем просто использовать синтаксис функции отправки и определить две дополнительные методы `isempty`. Так как эта функция из `Base` модуля, мы должны квалифицировать как `Base.isempty` для того, чтобы продлить его.

```
Base.isempty(::Nil) = true  
Base.isempty(::Cons) = false
```

Здесь нам вообще не нужны значения аргументов, чтобы определить, пуст ли список. Просто одного типа достаточно, чтобы вычислить эту информацию. Джулия позволяет нам опускать имена аргументов, сохраняя только аннотацию типа, если нам не нужны их значения.

Мы можем [проверить](#), `isempty` наши `isempty` методы:

```
julia> using Base.Test  
  
julia> @test isempty(Nil())  
Test Passed  
Expression: isempty(Nil())  
  
julia> @test !isempty(Cons(1, Cons(2, Cons(3, Nil()))))  
Test Passed  
Expression: !(isempty(Cons(1, Cons(2, Cons(3, Nil())))))
```

и действительно, количество методов для `isempty` увеличилось на 2:

```
julia> methods(isempty)  
# 31 methods for generic function "isempty":  
isempty(v::SimpleVector) at essentials.jl:180  
isempty(m::Base.MethodList) at reflection.jl:394
```

Ясно, что определение того, является ли связанный список пустым или нет, является тривиальным примером. Но это приводит к чему-то более интересному:

Как долго этот список?

Функция `length` из стандартной библиотеки дает нам длину коллекции или определенные [итерации](#). Существует множество способов реализации `length` для связанного списка. В частности, используя `while` цикл, вероятно, самый быстрый и наиболее эффективно использует память в Джулию. Но [преждевременной оптимизации](#) следует избегать, так что давайте предположим, что наш связанный список не должен быть эффективным. Каков самый простой способ написать функцию `length`?

```
Base.length(::Nil) = 0
Base.length(xs::Cons) = 1 + length(xs.rest)
```

Первое определение прост: пустой список имеет длину 0. Второе определение также легко читается: чтобы подсчитать длину списка, мы подсчитываем первый элемент, а затем подсчитываем длину остальной части списка. Мы можем проверить этот метод так же, как мы тестировали `isempty`:

```
julia> @test length(Nil()) == 0
Test Passed
Expression: length(Nil()) == 0
Evaluated: 0 == 0

julia> @test length(Cons(1, Cons(2, Cons(3, Nil())))) == 3
Test Passed
Expression: length(Cons(1, Cons(2, Cons(3, Nil())))) == 3
Evaluated: 3 == 3
```

Следующие шаги

Этот пример игрушек довольно далек от реализации всех функций, которые желательны в связанном списке. Его отсутствует, например, итерационный интерфейс. Тем не менее, это иллюстрирует, как отправка может быть использована для написания короткого и четкого кода.

Неизменяемые типы

Самый простой составной тип - неизменный тип. Экземпляры неизменяемых типов, например [кортежи](#), являются значениями. Их поля не могут быть изменены после их создания. Во многих отношениях неизменяемый тип подобен `Tuple` с именами для самого типа и для каждого поля.

Типы Singleton

Композитные типы, по определению, содержат ряд более простых типов. В Julia это число может быть нулем; то есть непреложный типа допускается не содержать *никаких* полей. Это сопоставимо с пустым кортежем `()`.

Почему это может быть полезно? Такие неизменные типы известны как «одноэлементные типы», поскольку только один из них может когда-либо существовать. Значения таких типов известны как «одиночные значения». Стандартная библиотека `Base` содержит много таких одноэлементных типов. Вот краткий список:

- `Void`, тип `nothing`. Мы можем проверить, что `Void.instance` (который является специальным синтаксисом для получения одноэлементного значения одноэлементного типа) действительно `nothing`.
- Любой тип мультимедиа, такой как `MIME"text/plain"`, представляет собой одноэлементный тип с одним экземпляром `MIME("text/plain")`.
- `Irrational{π}`, `Irrational{e}`, `Irrational{φ}` и подобные типы являются одноточечными типами, а их одноэлементные экземпляры - иррациональные значения $\pi = 3.1415926535897\dots$ и т. д.
- Характеристики размера итератора `Base.HasLength`, `Base.HasShape`, `Base.IsInfinite` и `Base.SizeUnknown` - это все одноэлементные типы.

0.5.0

- В версии 0.5 и более поздней, каждая **функция** представляет собой одноэлементный экземпляр одноэлементного типа! Как и любое другое значение singleton, мы можем восстановить функцию `sin`, например, из `typeof(sin).instance`.

Поскольку они не содержат ничего, одноэлементные типы невероятно легкие, и их часто можно оптимизировать компилятором, чтобы не иметь накладных расходов во время работы. Таким образом, они идеально подходят для черт, специальных значений тегов и для таких функций, как функции, на которые стоит специализироваться.

Чтобы определить одноэлементный тип,

```
julia> immutable MySingleton end
```

Чтобы определить пользовательскую печать для одноэлементного типа,

```
julia> Base.show(io::IO, ::MySingleton) = print(io, "sing")
```

Чтобы получить доступ к экземпляру singleton,

```
julia> MySingleton.instance
MySingleton()
```

Часто это присваивается константе:

```
julia> const sing = MySingleton.instance
MySingleton()
```

Типы оберток

Если неизменяемые типы нулевого поля интересны и полезны, то, возможно, еще более полезны однотипные неизменные типы. Такие типы обычно называют «типами обертки», потому что они оборачивают некоторые базовые данные, обеспечивая альтернативный интерфейс к указанным данным. Примером типа оболочки в `Base` является `String`. Мы будем определять аналогичный тип для `String` именем `MyString`. Этот тип будет поддерживаться векторным (одномерный массив) байтов (`UInt8`).

Во-первых, само определение типа и некоторые настройки показывают:

```
immutable MyString <: AbstractString
    data::Vector{UInt8}
end

function Base.show(io::IO, s::MyString)
    print(io, "MyString: ")
    write(io, s.data)
    return
end
```

Теперь наш тип `MyString` готов к использованию! Мы можем прокормить его некоторыми сырыми данными UTF-8, и он отображает, как нам нравится:

```
julia> MyString([0x48, 0x65, 0x6c, 0x6c, 0x6f, 0x2c, 0x20, 0x57, 0x6f, 0x72, 0x6c, 0x64, 0x21])
MyString: Hello, World!
```

Очевидно, что этот тип строки требует большой работы, прежде чем он станет таким же удобным, как и тип `Base.String`.

Истинные составные типы

Возможно, чаще всего многие неизменяемые типы содержат более одного поля. Примером может служить стандартная библиотека `Rational{T}` type, которая содержит два fields: поле `num` для числителя и поле `den` для знаменателя. Эту модель такого типа довольно просто:

```
immutable MyRational{T}
    num::T
    den::T
    MyRational(n, d) = (g = gcd(n, d); new(n÷g, d÷g))
end
MyRational{T}(n::T, d::T) = MyRational{T}(n, d)
```

Мы успешно внедрили конструктор, который упрощает наши рациональные числа:

```
julia> MyRational(10, 6)  
MyRational{Int64}(5, 3)
```

Прочитайте Типы онлайн: <https://riptutorial.com/ru/julia-lang/topic/5467/типы>

глава 35: функции

Синтаксис

- $f(n) = \dots$
- функция $f(n) \dots \text{end}$
- $n :: \text{Тип}$
- $x \rightarrow \dots$
- $f(n) \text{ do } \dots \text{end}$

замечания

Помимо общих функций (которые являются наиболее распространенными), есть также встроенные функции. Такие функции включают в себя `is`, `isa`, `typeof`, `throw`, и аналогичные функции. Встроенные функции обычно реализуются в C вместо Julia, поэтому они не могут быть специализированы по типам аргументов для отправки.

Examples

Квадратное число

Это самый простой синтаксис для определения функции:

```
square(n) = n * n
```

Чтобы вызвать функцию, используйте круглые скобки (без пробелов между ними):

```
julia> square(10)
100
```

Функции - это объекты в Julia, и мы можем показать их в [REPL](#) как с любыми другими объектами:

```
julia> square
square (generic function with 1 method)
```

По умолчанию все функции Julia являются родовыми (иначе известными как [полиморфные](#)). Наша `square` функция работает так же хорошо, как и значения с плавающей запятой:

```
julia> square(2.5)
6.25
```

... или даже [матрицы](#) :

```
julia> square([2 4
               2 1])
2×2 Array{Int64,2}:
 12  12
  6   9
```

Рекурсивные функции

Простая рекурсия

Используя рекурсию и [тернарный условный оператор](#), мы можем создать альтернативную реализацию встроенной `factorial` функции:

```
myfactorial(n) = n == 0 ? 1 : n * myfactorial(n - 1)
```

Использование:

```
julia> myfactorial(10)
3628800
```

Работа с деревьями

Рекурсивные функции часто наиболее полезны для структур данных, особенно для древовидных структур данных. Поскольку [выражения](#) в Julia являются древовидными структурами, рекурсия может быть весьма полезна для [метапрограммирования](#). Например, нижняя функция собирает набор всех головок, используемых в выражении.

```
heads(ex::Expr) = reduce(U, Set{(ex.head,)}, (heads(a) for a in ex.args))
heads(::Any) = Set{Symbol}()
```

Мы можем проверить, что наша функция работает по назначению:

```
julia> heads(:(7 + 4x > 1 > A[0]))
Set{Symbol{=:comparison,:ref,:call}}
```

Эта функция является компактной и использует множество более совершенных методов, таких как `reduce` [функции более высокого порядка](#), тип `Set` данных и выражения генератора.

Введение в диспетчеризацию

Мы можем использовать синтаксис `::` для отправки по [типу](#) аргумента.

```
describe(n::Integer) = "integer $n"
describe(n::AbstractFloat) = "floating point $n"
```

Использование:

```
julia> describe(10)
"integer 10"

julia> describe(1.0)
"floating point 1.0"
```

В отличие от многих языков, которые обычно предоставляют либо статичную множественную отправку, либо динамическую разовую отправку, Julia имеет полную динамическую множественную отправку. То есть функции могут быть специализированы для более чем одного аргумента. Это полезно при определении специализированных методов для операций над определенными типами и методов резервного копирования для других типов.

```
describe(n::Integer, m::Integer) = "integers n=$n and m=$m"
describe(n, m::Integer) = "only m=$m is an integer"
describe(n::Integer, m) = "only n=$n is an integer"
```

Использование:

```
julia> describe(10, 'x')
"only n=10 is an integer"

julia> describe('x', 10)
"only m=10 is an integer"

julia> describe(10, 10)
"integers n=10 and m=10"
```

Необязательные аргументы

Julia позволяет выполнять необязательные аргументы. За кулисами это реализуется как еще один частный случай множественной отправки. Например, давайте решим популярную [проблему Fizz Buzz](#) . По умолчанию мы будем делать это для чисел в диапазоне 1:10 , но при необходимости мы допустим другое значение. Мы также разрешим использовать разные фразы для Fizz или Buzz .

```
function fizzbuzz(xs=1:10, fizz="Fizz", buzz="Buzz")
    for i in xs
        if i % 15 == 0
            println(fizz, buzz)
        elseif i % 3 == 0
            println(fizz)
        elseif i % 5 == 0
            println(buzz)
        else
            println(i)
        end
    end
end
```

Если мы `fizzbuzz` в REPL, в нем сказано, что существует четыре метода. Для каждой комбинации аргументов был создан один метод.

```
julia> fizzbuzz
fizzbuzz (generic function with 4 methods)

julia> methods(fizzbuzz)
# 4 methods for generic function "fizzbuzz":
fizzbuzz() at REPL[96]:2
fizzbuzz(xs) at REPL[96]:2
fizzbuzz(xs, fizz) at REPL[96]:2
fizzbuzz(xs, fizz, buzz) at REPL[96]:2
```

Мы можем проверить, что наши значения по умолчанию используются, когда параметры не предоставляются:

```
julia> fizzbuzz()
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
```

но что дополнительные параметры принимаются и соблюдаются, если мы их предоставим:

```
julia> fizzbuzz(5:8, "fuzz", "bizz")
bizz
fuzz
7
8
```

Параметрическая отправка

Часто бывает, что функция должна отправлять параметрические типы, такие как `Vector{T}` или `Dict{K,V}`, но параметры типа не фиксированы. Этот случай может быть рассмотрен с помощью параметрической отправки:

```
julia> foo{T<:Number}(xs::Vector{T}) = @show xs .+ 1
foo (generic function with 1 method)

julia> foo(xs::Vector) = @show xs
foo (generic function with 2 methods)

julia> foo([1, 2, 3])
xs .+ 1 = [2,3,4]
3-element Array{Int64,1}:
 2
 3
```

4

```
julia> foo([1.0, 2.0, 3.0])
xs .+ 1 = [2.0,3.0,4.0]
3-element Array{Float64,1}:
 2.0
 3.0
 4.0

julia> foo(["x", "y", "z"])
xs = String["x","y","z"]
3-element Array{String,1}:
"x"
"y"
"z"
```

Может возникнуть соблазн просто написать `xs::Vector{Number}` . Но это работает только для объектов, тип которых явно `Vector{Number}` :

```
julia> isa(Number[1, 2], Vector{Number})
true

julia> isa(Int[1, 2], Vector{Number})
false
```

Это связано с [параметрической инвариантностью](#) : объект `Int[1, 2]` *не* является `Vector{Number}` , поскольку он может содержать только `Int` s, тогда как ожидается, что `Vector{Number}` сможет содержать любые виды чисел.

Создание общего кода

Диспетчер - невероятно мощная функция, но часто лучше писать общий код, который работает для всех типов, вместо специализированного кода для каждого типа. Написание общего кода позволяет избежать дублирования кода.

Например, вот код для вычисления суммы квадратов вектора целых чисел:

```
function sumsq(v::Vector{Int})
    s = 0
    for x in v
        s += x ^ 2
    end
    s
end
```

Но этот код работает *только* для вектора `Int` s. Он не будет работать на `UnitRange` :

```
julia> sumsq(1:10)
ERROR: MethodError: no method matching sumsq(::UnitRange{Int64})
Closest candidates are:
  sumsq(::Array{Int64,1}) at REPL[8]:2
```


Он не будет работать с `Vector{Float64}` :

```
julia> sumsq([1.0, 2.0])
ERROR: MethodError: no method matching sumsq(::Array{Float64,1})
Closest candidates are:
  sumsq(::Array{Int64,1}) at REPL[8]:2
```

Лучшим способом написать эту функцию `sumsq` должно быть

```
function sumsq(v::AbstractVector)
    s = zero(eltype(v))
    for x in v
        s += x ^ 2
    end
    s
end
```

Это будет работать в двух случаях, перечисленных выше. Но есть некоторые коллекции, которые мы могли бы захотеть суммировать квадраты, которые вообще не являются векторами, в любом смысле. Например,

```
julia> sumsq(take(countfrom(1), 100))
ERROR: MethodError: no method matching sumsq(::Base.Take{Base.Count{Int64}})
Closest candidates are:
  sumsq(::Array{Int64,1}) at REPL[8]:2
  sumsq(::AbstractArray{T,1}) at REPL[11]:2
```

показывает, что мы не можем суммировать квадраты [ленивого итерации](#) .

Еще более общая реализация - это просто

```
function sumsq(v)
    s = zero(eltype(v))
    for x in v
        s += x ^ 2
    end
    s
end
```

Что работает во всех случаях:

```
julia> sumsq(take(countfrom(1), 100))
338350
```

Это самый идиоматический код Юлии и может обрабатывать всевозможные ситуации. В некоторых других языках удаление аннотаций типа может повлиять на производительность, но это не относится к Julia; для производительности важна [стабильность](#) только [типа](#) .

Императивный факторный

Синтаксис длинной формы доступен для определения многострочных функций. Это может быть полезно, когда мы используем императивные структуры, такие как циклы.

Возвращается выражение в хвостовом положении. Например, функция ниже использует **цикл** `for` для вычисления **факториала** некоторого целого `n` :

```
function myfactorial(n)
    fact = one(n)
    for m in 1:n
        fact *= m
    end
    fact
end
```

Использование:

```
julia> myfactorial(10)
3628800
```

В более длинных функциях обычно используется оператор `return` . Оператор `return` не нужен в положении хвоста, но он по-прежнему используется для ясности. Например, другим способом написания вышеуказанной функции было бы

```
function myfactorial(n)
    fact = one(n)
    for m in 1:n
        fact *= m
    end
    return fact
end
```

который идентичен поведению функции выше.

Анонимные функции

Синтаксис стрелок

Анонимные функции могут быть созданы с помощью синтаксиса `->` . Это полезно для передачи функций **более высоким функциям** , таким как функция `map` . Функция ниже вычисляет квадрат каждого числа в **массиве** `A`

```
squareall(A) = map(x -> x ^ 2, A)
```

Пример использования этой функции:

```
julia> squareall(1:10)
10-element Array{Int64,1}:
 1
 4
 9
```

```
16
25
36
49
64
81
100
```

Многострочный синтаксис

Многолинейные анонимные функции могут быть созданы с использованием синтаксиса `function`. Например, следующий пример вычисляет [факториалы](#) первых `n` чисел, но использует анонимную функцию вместо встроенного `factorial`.

```
julia> map(function (n)
            product = one(n)
            for i in 1:n
                product *= i
            end
            product
        end, 1:10)
10-element Array{Int64,1}:
 1
 2
 6
24
120
720
5040
40320
362880
3628800
```

Блочный синтаксис

Поскольку так часто передается анонимная функция в качестве первого аргумента функции, существует синтаксис `do block`. Синтаксис

```
map(A) do x
    x ^ 2
end
```

ЭКВИВАЛЕНТНО

```
map(x -> x ^ 2, A)
```

но первое может быть более понятным во многих ситуациях, особенно если в анонимной функции выполняется много вычислений. `do` блок синтаксиса особенно полезно для [ввода](#) имени [файла и вывода](#) по причинам управления ресурсами.

Прочитайте функции онлайн: <https://riptutorial.com/ru/julia-lang/topic/3079/функции>

глава 36: Функции более высокого порядка

Синтаксис

- `foreach (f, xs)`
- отображение `(f, xs)`
- фильтр `(f, xs)`
- уменьшить `(f, v0, xs)`
- `foldl (f, v0, xs)`
- `foldr (f, v0, xs)`

замечания

Функции могут приниматься в качестве параметров и также могут быть получены в качестве типов возврата. Действительно, функции могут быть созданы внутри тела других функций. Эти внутренние функции известны как [замыкания](#) .

Examples

Функции как аргументы

[Функции](#) - объекты в Джулии. Как и любые другие объекты, они могут передаваться как аргументы другим функциям. Функции, которые принимают функции, известны как функции [более высокого порядка](#) .

Например, мы можем реализовать эквивалент стандартной функции `foreach` стандартной библиотеки, используя функцию `f` в качестве первого параметра.

```
function myforeach(f, xs)
    for x in xs
        f(x)
    end
end
```

Мы можем проверить, что эта функция действительно работает так, как мы ожидаем:

```
julia> myforeach(println, ["a", "b", "c"])
a
b
c
```

Принимая функцию в качестве *первого* параметра, вместо более позднего параметра мы

можем использовать синтаксис блока `do` Julia. Синтаксис `do block` - это просто удобный способ передать [анонимную функцию](#) в качестве первого аргумента функции.

```
julia> myforeach([1, 2, 3]) do x
    println(x^x)
end
1
4
27
```

Наша реализация `myforeach` выше выше примерно эквивалентна встроенной функции `foreach`. Также существуют многие другие встроенные функции более высокого порядка.

Функции более высокого порядка достаточно сильны. Иногда, работая с функциями более высокого порядка, точные выполняемые операции становятся несущественными, и программы могут стать довольно абстрактными. [Комбинаторы](#) являются примерами систем высоко абстрактных функций более высокого порядка.

Карта, фильтр и сокращение

Две из наиболее фундаментальных функций более высокого порядка, включенные в стандартную библиотеку, - это `map` и `filter`. Эти функции являются универсальными и могут работать с любым [итерабельным](#). В частности, они хорошо подходят для вычислений на [массивах](#).

Предположим, у нас есть набор данных о школах. Каждая школа учит конкретному предмету, имеет несколько классов и среднее число учащихся в классе. Мы можем моделировать школу со следующим [неизменным типом](#):

```
immutable School
    subject::Symbol
    nclasses::Int
    nstudents::Int # average no. of students per class
end
```

Наш набор учебных пособий будет `Vector{School}`:

```
dataset = [School(:math, 3, 30), School(:math, 5, 20), School(:science, 10, 5)]
```

Предположим, мы хотим найти количество студентов, общее число которых записано в математической программе. Для этого нам требуется несколько шагов:

- мы должны сузить набор данных только до школ, которые учат математике (`filter`)
- мы должны вычислить количество учащихся в каждой школе (`map`)
- и мы должны сократить этот список чисел учащихся до одного значения, сумму (`reduce`)

Наивное (не наиболее эффективное) решение просто заключалось бы в том, чтобы

напрямую использовать эти три функции более высокого порядка.

```
function nmath(data)
    maths = filter(x -> x.subject === :math, data)
    students = map(x -> x.nclasses * x.nstudents, maths)
    reduce(+, 0, students)
end
```

и мы проверяем, что в нашем наборе данных есть 190 математиков:

```
julia> nmath(dataset)
190
```

Существуют функции для объединения этих функций и, таким образом, повышения производительности. Например, мы могли бы использовать функцию `mapreduce` для выполнения отображения и сокращения за один шаг, что позволило бы сэкономить время и память.

`reduce` имеет смысл только для **ассоциативных операций**, таких как `+`, но иногда полезно выполнить сокращение с помощью неассоциативной операции. Функции высокого порядка `foldl` и `foldr` предоставляются для принудительного выполнения определенного порядка восстановления.

Прочитайте Функции более высокого порядка онлайн: <https://riptutorial.com/ru/julia-lang/topic/6955/функции-более-высокого-порядка>

глава 37: Чтение DataFrame из файла

Examples

Чтение данных из разделенных разделителями данных

Возможно, вы захотите прочитать DataFrame из DataFrame CSV (значения, разделенные запятой), а может быть, даже из TSV или WSV (вкладки и разделенные пробелы файлы). Если ваш файл имеет правильное расширение, вы можете использовать функцию `readtable` для чтения в фрейме данных:

```
readtable("dataset.CSV")
```

Но что, если ваш файл не имеет правильного расширения? Вы можете указать разделитель, который использует ваш файл (запятая, табуляция, пробел и т. Д.) В качестве аргумента ключевого слова для функции `readtable` :

```
readtable("dataset.txt", separator=',')
```

Обработка комментариев комментария комментария комментария

Наборы данных часто содержат комментарии, которые объясняют формат данных или содержат условия лицензии и использования. Обычно вы игнорируете эти строки, когда читаете в DataFrame .

Функция `readtable` предполагает, что строки комментариев начинаются с символа «#». Однако ваш файл может использовать метки комментариев, например, % или // . Чтобы убедиться, что `readtable` обрабатывает их правильно, вы можете указать метку комментария как аргумент ключевого слова:

```
readtable("dataset.csv", allowcomments=true, commentmark='%')
```

Прочитайте Чтение DataFrame из файла онлайн: <https://riptutorial.com/ru/julia-lang/topic/7340/чтение-dataframe-из-файла>

кредиты

S. No	Главы	Contributors
1	Начало работы с Julia Language	Andrew Piliser , becko , Community , Dawny33 , Fengyang Wang , Kevin Montrose , prcastro
2	@goto и @label	Fengyang Wang
3	Conditionals	Fengyang Wang , Michael Ohlrogge , prcastro
4	JSON	4444 , Fengyang Wang
5	sub2ind	Fengyang Wang , Gnimuc
6	арифметика	Fengyang Wang
7	в то время как циклы	Fengyang Wang
8	Время	Fengyang Wang
9	вход	Fengyang Wang
10	Выражения	Michael Ohlrogge
11	для циклов	Fengyang Wang , Michael Ohlrogge
12	Затворы	Fengyang Wang
13	итерируемые	Fengyang Wang , prcastro
14	Комбинаторы	Fengyang Wang
15	Кортеж	Fengyang Wang
16	Массивы	Fengyang Wang , Michael Ohlrogge , prcastro
17	Метапрограммирование	Fengyang Wang , Ismael Venegas Castelló , P i , prcastro
18	Модули	Fengyang Wang
19	Нормализация строки	Fengyang Wang
20	пакеты	Fengyang Wang
21	Параллельная	Fengyang Wang , Harrison Grodin , Michael Ohlrogge , prcastro

	обработка	
22	Перечисления	Fengyang Wang
23	постижения	2Cubed , Fengyang Wang , zwlayer
24	Регулярные выражения	Fengyang Wang
25	РЕПЛ	Fengyang Wang
26	Скрипты и трубопроводы оболочки	2Cubed , Fengyang Wang , mnoronha , prcastro
27	Словари	B Roy Dawson
28	Совместимость с несколькими версиями	Fengyang Wang
29	Сравнения	Fengyang Wang
30	Стабильность типа	Abhijith , Fengyang Wang
31	Строковые макросы	Fengyang Wang
32	Струны	Fengyang Wang , Michael Ohlrogge
33	Тестирование устройства	Fengyang Wang
34	Типы	Fengyang Wang , prcastro
35	функции	Fengyang Wang , Harrison Grodin , Michael Ohlrogge , Sebastialonso
36	Функции более высокого порядка	Fengyang Wang , mnoronha
37	Чтение DataFrame из файла	Pranav Bhat