Kostenloses eBook

LERNEN

Julia Language

Free unaffiliated eBook created from
Stack Overflow contributors.

#julia-lang

Inhaltsverzeichnis

Über

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.

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

Kapitel 1: Erste Schritte mit Julia Language

Versionen

Ausführung	Veröffentlichungsdatum
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

Hallo Welt!

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

Um Julia auszuführen, besorgen Sie sich zuerst den Interpreter von der Downloadseite der Website. Die aktuelle stabile Version ist Version 0.5.0. Diese Version wird für die meisten Benutzer empfohlen. Bestimmte Paketentwickler oder Power-User verwenden möglicherweise den nächtlichen Build, der weitaus weniger stabil ist.

Wenn Sie den Interpreter haben, schreiben Sie Ihr Programm in eine Datei namens `hello.jl` . Es kann dann von einem Systemendgerät aus wie folgt ausgeführt werden:

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

Julia kann auch interaktiv ausgeführt werden, indem das Programm `julia` wird. Sie sollten einen Header und eine Aufforderung wie folgt sehen:

```
               _
   _       _ _(_)_     |  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>
```

Sie können jeden Julia-Code in dieser REPL ausführen. Versuchen Sie daher Folgendes:

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

In diesem Beispiel wird die Zeichenfolge `"Hello, World!"` Verwendet. und der Funktion `println` - eine von vielen in der Standardbibliothek. Versuchen Sie die folgenden Quellen, um weitere Informationen oder Hilfe zu erhalten:

- Die REPL verfügt über einen integrierten Hilfemodus für den Zugriff auf die Dokumentation.
- Die offizielle Dokumentation ist sehr umfangreich.
- Stack Overflow enthält eine kleine, aber wachsende Sammlung von Beispielen.
- Benutzer von Gitter helfen gerne bei kleinen Fragen.
- Der primäre Online-Diskussionsort für Julia ist das Discourse-Forum unter discourse.julialang.org . Weitere ausführliche Fragen sollten hier veröffentlicht werden.
- Eine Sammlung von Tutorials und Büchern finden Sie hier .

Erste Schritte mit Julia Language online lesen: https://riptutorial.com/de/julia-lang/topic/485/erste-schritte-mit-julia-language

Kapitel 2: @goto und @label

Syntax

- @goto label
- @label label

Bemerkungen

Übermäßiger Gebrauch oder unangemessene Verwendung des erweiterten Kontrollflusses macht den Code schwer lesbar. `@goto` oder seine Entsprechungen in anderen Sprachen führen bei unsachgemäßer Verwendung zu unlesbarem Spaghetti-Code.

Ähnlich wie Sprachen wie C kann man in Julia nicht zwischen Funktionen wechseln. Dies bedeutet auch, dass `@goto` auf der obersten Ebene nicht möglich ist. es funktioniert nur innerhalb einer Funktion. Außerdem kann man nicht von einer inneren Funktion zu ihrer äußeren Funktion oder von einer äußeren Funktion zu einer inneren Funktion springen.

Examples

Eingabeüberprüfung

`@label` Makros `@goto` und `@label` können zwar `@label` nicht als Schleifen betrachtet werden, können jedoch für einen erweiterten Steuerungsfluss verwendet werden. Ein Anwendungsfall liegt vor, wenn der Ausfall eines Teils dazu führen sollte, dass eine gesamte Funktion erneut versucht wird. Dies ist häufig bei der Validierung von Eingaben hilfreich:

```
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

    a, b
end
```

Dieser Anwendungsfall wird jedoch häufig mit Rekursion klarer:

```
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
```

Obwohl beide Beispiele dasselbe tun, ist das zweite verständlicher. Der erste ist jedoch performanter (weil er den rekursiven Aufruf vermeidet). In den meisten Fällen spielen die Kosten des Anrufs keine Rolle. In begrenzten Situationen ist die erste Form jedoch akzeptabel.

Fehler beim Bereinigen

In Sprachen wie C wird die `@goto` Anweisung häufig verwendet, um sicherzustellen, dass eine Funktion die erforderlichen Ressourcen auch im Fehlerfall aufräumt. Dies ist bei Julia weniger wichtig, da stattdessen oft Ausnahmen und `try finally` Blöcke verwendet werden.

Es ist jedoch möglich, dass Julia-Code eine Schnittstelle zu C-Code und C-APIs herstellt, sodass Funktionen manchmal noch wie C-Code geschrieben werden müssen. Das nachfolgende Beispiel ist zwar entwickelt, zeigt aber einen allgemeinen Anwendungsfall. Der Julia-Code ruft `Libc.malloc` auf, um Speicherplatz zuzuordnen (dies simuliert einen C-API-Aufruf). Wenn nicht alle Zuordnungen erfolgreich sind, sollte die Funktion die bisher erhaltenen Ressourcen freigeben. Andernfalls wird der zugewiesene Speicher zurückgegeben.

```
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
```

@goto und @label online lesen: https://riptutorial.com/de/julia-lang/topic/5564/-goto-und--label

Kapitel 3: 2ind

Syntax

- 2ind (dims :: Tuple {Vararg {Integer}}, I :: Integer ...)
- 2ind {T <: Integer} (Dims :: Tuple {Vararg {Integer}}, I :: AbstractArray {T <: Integer, 1} ...)

Parameter

Parameter	Einzelheiten
dims :: Tuple {Vararg {Integer}}	Größe des Arrays
I :: Integer ...	Indizes (Skalar) des Arrays
I :: AbstractArray {T <: Integer, 1} ...	Indexe (Vektor) des Arrays

Bemerkungen

Das zweite Beispiel zeigt, dass das Ergebnis von, `sub2ind` in bestimmten Fällen sehr `sub2ind` sein kann.

Examples

Konvertieren Sie Indizes in lineare Indizes

```
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
```

Pits & Falls

```
# no error, even the subscript is out of range.
julia> sub2ind((3,3), 3, 4)
12
```

Man kann nicht feststellen, ob sich ein Index im Bereich eines Arrays befindet, indem man seinen Index vergleicht:

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

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

2ind online lesen: https://riptutorial.com/de/julia-lang/topic/1914/2ind

Kapitel 4: Arithmetik

Syntax

- + x
- -x
- a + b
- a - b
- a * b
- a / b
- a ^ b
- a% b
- 4a
- Quadrat (a)

Examples

Quadratische Formel

Julia verwendet für grundlegende Rechenoperationen ähnliche binäre Operatoren wie Mathematik oder andere Programmiersprachen. Die meisten Operatoren können in Infix-Notation geschrieben werden (dh zwischen den berechneten Werten platziert werden). Julia hat eine Reihenfolge von Operationen, die der üblichen Konvention in der Mathematik entspricht.

Der folgende Code implementiert beispielsweise die quadratische Formel , die die Operatoren `+` , `-` , `*` und `/` für Addition, Subtraktion, Multiplikation und Division veranschaulicht. Ebenfalls gezeigt ist die *implizite Multiplikation* , bei der eine Zahl direkt vor einem Symbol platziert werden kann, um die Multiplikation zu bedeuten; das heißt, `4a` bedeutet dasselbe wie `4*a` .

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

Verwendungszweck:

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

Sieb von Eratosthenes

Der Restoperator in Julia ist der Operator `%` . Dieser Operator verhält sich ähnlich wie `%` in Sprachen wie C und C ++. `a % b` ist der unterschriebene Rest, der nach der Division von `a` durch `b` übrig bleibt.

Dieser Operator ist sehr nützlich, um bestimmte Algorithmen zu implementieren, beispielsweise die folgende Implementierung des Sieve of Eratosthenes .

```
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
```

Verwendungszweck:

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

Matrix-Arithmetik

Julia verwendet die mathematischen Standardbedeutungen von arithmetischen Operationen, wenn sie auf Matrizen angewendet werden. Manchmal sind elementweise Operationen stattdessen erwünscht. Diese sind mit einem Punkt (`.`) Vor dem Operator markiert, der elementweise erfolgen soll. (Beachten Sie, dass elementweise Operationen oft nicht so effizient wie Schleifen sind.)

Summen

Der Operator `+` auf Matrizen ist eine Matrixsumme. Es ähnelt einer elementweisen Summe, überträgt jedoch keine Form. Das heißt, wenn `A` und `B` die gleiche Form haben, dann ist `A + B` das Gleiche wie `A .+ B` ; Andernfalls ist `A + B` ein Fehler, wohingegen `A .+ B` nicht unbedingt der Fall sein muss.

```
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
```

Ebenso berechnet `-` eine Matrixdifferenz. Sowohl `+` als auch `-` können auch als unäre Operatoren verwendet werden.

Produkte

Der Operator `*` auf Matrizen ist das Matrixprodukt (nicht das Elementweise Produkt). Verwenden Sie für ein elementweises Produkt den Operator. `.*` . Vergleiche (mit den gleichen Matrizen wie oben):

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

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

Befugnisse

Der Operator `^` berechnet die Exponentiation der Matrix . Die Matrix-Exponentiation kann hilfreich sein, um Werte bestimmter Wiederholungen schnell zu berechnen. Zum Beispiel können die

Fibonacci-Zahlen durch den Matrixausdruck generiert werden

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

Wie üblich kann der Operator `.^` Verwendet werden, wenn die Elementweise Potenzierung die gewünschte Operation ist.

Arithmetik online lesen: https://riptutorial.com/de/julia-lang/topic/3848/arithmetik

Kapitel 5: Arrays

Syntax

- [1,2,3]
- [1 2 3]
- [1 2 3; 4 5 6; 7 8 9]
- Array (Typ, Dimensionen ...)
- one (typ, dims ...)
- Nullen (Typ, Dims ...)
- trues (typ, dims ...)
- Falses (Typ, Dims ...)
- schieben! (A, x)
- Pop! (A)
- nicht verschieben! (A, x)
- Schicht! (A)

Parameter

Parameter	Bemerkungen
Zum	`push!(A, x)` , nicht `unshift!(A, x)`
`A`	Das Array, das hinzugefügt werden soll.
`x`	Das Element, das dem Array hinzugefügt werden soll.

Examples

Manueller Aufbau eines einfachen Arrays

Man kann ein Julia-Array von Hand mit der eckigen Klammer-Syntax initialisieren:

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

Die erste Zeile nach dem Befehl zeigt die Größe des von Ihnen erstellten Arrays. Es zeigt auch die Art seiner Elemente und ihre Dimensionalität (in diesem Fall `Int64` bzw. `1`). Für ein zweidimensionales Array können Sie Leerzeichen und Semikolon verwenden:

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

```
 1  2  3
 4  5  6
```

Um ein nicht initialisiertes Array zu erstellen, können Sie die `Array(type, dims...)` -Methode verwenden:

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

Die Funktionen `zeros` , `ones` , `trues` , `falses` weisen Methoden auf, die sich genau gleich verhalten, aber Arrays erzeugen, die mit `0.0` , `1.0` , `True` oder `False` `falses` sind.

Array-Typen

In Julia haben Arrays Typen, die durch zwei Variablen parametrisiert werden: einen Typ `T` und eine Dimensionalität `D` (`Array{T, D}`). Für ein 1-dimensionales Array von Ganzzahlen lautet der Typ:

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

Wenn das Array eine 2-dimensionale Matrix ist, ist `D` gleich 2:

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

Der Elementtyp kann auch abstrakt sein:

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

Hier ist `Any` (ein abstrakter Typ) der Typ des resultierenden Arrays.

Festlegen von Typen beim Erstellen von Arrays

Wenn wir ein Array auf die oben beschriebene Weise erstellen, versucht Julia, den richtigen Typ zu ermitteln, den wir möglicherweise benötigen. In den ersten Beispielen oben haben wir Eingaben eingegeben, die wie Ganzzahlen aussahen, und so war Julia standardmäßig der Standardtyp `Int64` . Manchmal möchten wir jedoch genauer sein. Im folgenden Beispiel geben wir an, dass der Typ statt `Int8` :

```
x1 = Int8[1 2 3; 4 5 6; 7 8 9]
```

```
typeof(x1)  ## Array{Int8,2}
```

Wir können den Typ sogar als etwas wie `Float64` , selbst wenn wir die Eingaben auf eine Weise schreiben, die ansonsten standardmäßig als ganze Zahlen interpretiert werden könnte (z. B. `1` statt `1.0`). z.B

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

Arrays von Arrays - Eigenschaften und Aufbau

In Julia können Sie ein Array haben, das andere Array-Typobjekte enthält. Beachten Sie die folgenden Beispiele für das Initialisieren verschiedener Arten von Arrays:

```
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
```

Betrachten Sie zum Beispiel die Unterschiede zwischen C und D hier:

```
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)` erzeugt ein Objekt vom Typ `Array{Float64,1}` . Da die einzigen Elemente für die Elemente von `C` sind, dass sie Arrays mit Elementen vom Typ Float64 sind, passt dies in die Definition von `C` Für `D` wir jedoch angegeben, dass die Elemente zweidimensionale Arrays sein müssen. Da `rand(3)` kein zweidimensionales Array erzeugt, können wir es daher nicht verwenden, um einem bestimmten Element von `D` einen Wert zuzuweisen

Spezifizieren Sie spezifische Abmessungen von Arrays innerhalb eines Arrays

Obwohl wir angeben können, dass ein Array Elemente vom Typ Array aufnehmen soll, und wir können angeben, dass diese Elemente z. B. zweidimensionale Arrays sein sollen, können wir die Dimensionen dieser Elemente nicht direkt angeben. Wir können beispielsweise nicht direkt angeben, dass ein Array 10 Arrays enthalten soll, von denen jedes 5,5 ist. Wir können dies anhand der Syntax für die `Array()` Funktion sehen, die zum Erstellen eines Arrays verwendet wird:

> **Array {T} (Dims)**

Konstruiert ein nicht initialisiertes dichtes Array mit dem Elementtyp T. dims kann ein Tupel oder eine Reihe von ganzzahligen Argumenten sein. Das Syntax-Array (T, dims) ist ebenfalls verfügbar, aber veraltet.

Der Typ eines Arrays in Julia umfasst die Anzahl der Dimensionen, nicht jedoch die Größe dieser Dimensionen. Daher gibt es in dieser Syntax keinen Platz, um die genauen Abmessungen anzugeben. Ein ähnlicher Effekt könnte jedoch mit einem Array-Verständnis erzielt werden:

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

Hinweis: Diese Dokumentation spiegelt die folgende SO-Antwort wider

Initialisieren Sie ein leeres Array

Wir können das `[]` , um ein leeres Array in Julia zu erstellen. Das einfachste Beispiel wäre:

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

Arrays vom Typ `Any` funktionieren im Allgemeinen nicht so gut wie Arrays mit einem angegebenen Typ. So können wir zum Beispiel verwenden:

```
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}
```

Die Quelle des letzten Beispiels finden Sie unter Ein leeres Array von Tupeln in Julia initialisieren .

Vektoren

Vektoren sind eindimensionale Arrays und unterstützen meistens die gleiche Schnittstelle wie ihre mehrdimensionalen Gegenstücke. Vektoren unterstützen jedoch auch zusätzliche Operationen.

Beachten Sie zunächst, dass `Vector{T}` wobei `T` ein Typ ist, mit `Array{T,1}` identisch ist.

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

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

Man liest `Array{Int64,1}` als "eindimensionales Array von `Int64` ".

Im Gegensatz zu mehrdimensionalen Arrays kann die Größe von Vektoren geändert werden. Elemente können vor oder hinter dem Vektor hinzugefügt oder entfernt werden. Diese Vorgänge sind alle konstant amortisiert .

```
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
```

Wie üblich, jede dieser Funktionen `push!` `pop!` , nicht `unshift!` und `shift!` endet mit einem Ausrufezeichen, um anzuzeigen, dass sie ihre Argumente mutieren. Die Funktionen `push!` und nicht `unshift!` das Array zurückgeben, während `pop!` und `shift!` das entfernte Element zurückgeben

Verkettung

Es ist oft nützlich, Matrizen aus kleineren Matrizen zu erstellen.

Horizontale Verkettung

Matrizen (und Vektoren, die als Spaltenvektoren behandelt werden) können mit der `hcat` Funktion horizontal verkettet werden.

```
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
```

Es ist eine bequeme Syntax verfügbar, die die eckige Klammer und Leerzeichen verwendet:

```
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
```

Diese Notation kann der Notation für Blockmatrizen, die in der linearen Algebra verwendet werden, sehr ähnlich sein:

```
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
```

Beachten Sie, dass Sie eine einzelne Matrix nicht horizontal mit der `[]` -Syntax verketten können, da dies einen Ein-Element-Vektor von Matrizen erzeugen würde:

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

Vertikale Verkettung

Vertikale Verkettung ist wie horizontale Verkettung, jedoch in vertikaler Richtung. Die Funktion für die vertikale Verkettung ist `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
```

Alternativ kann die Notation mit eckigen Klammern auch mit Semikolons verwendet werden ; als Trennzeichen:

```
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
```

Vektoren können auch vertikal verkettet werden. Das Ergebnis ist ein Vektor:

```
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
```

Horizontale und vertikale Verkettung können kombiniert werden:

```
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
```

Arrays online lesen: https://riptutorial.com/de/julia-lang/topic/5437/arrays

Kapitel 6: Aufzählungen

Syntax

- @enum EnumType val = 1 val val
- :Symbol

Bemerkungen

Es ist manchmal nützlich, Aufzählungstypen zu haben, bei denen jede Instanz einen anderen Typ hat (häufig ein unveränderlicher Singleton-Typ). Dies kann für die Typstabilität wichtig sein. Merkmale werden typischerweise mit diesem Paradigma implementiert. Dies führt jedoch zu zusätzlichem Aufwand für die Kompilierung.

Examples

Einen Aufzählungstyp definieren

Ein Aufzählungstyp ist ein Typ , der eine endliche Liste möglicher Werte enthalten kann. In Julia werden Aufzählungstypen normalerweise "Aufzählungstypen" genannt. Zum Beispiel könnte man Aufzählungstypen verwenden, um die sieben Tage der Woche, die zwölf Monate des Jahres, die vier Farben eines Standard-Decks mit 52 Karten oder andere ähnliche Situationen zu beschreiben.

Wir können aufgezählte Typen definieren, um die Anzüge und Ränge eines Standard-Decks mit 52 Karten zu modellieren. Das `@enum` enum-Makro wird zur Definition von Aufzählungstypen verwendet.

```
@enum Suit ♣♦♥♠
@enum Rank ace=1 two three four five six seven eight nine ten jack queen king
```

Dies definiert zwei Typen: `Suit` und `Rank` . Wir können überprüfen, ob die Werte tatsächlich den erwarteten Typen entsprechen:

```
julia> ♦
♦::Suit = 1

julia> six
six::Rank = 6
```

Beachten Sie, dass jeder Farbe und Rang eine Nummer zugeordnet wurde. Standardmäßig beginnt diese Nummer bei Null. Daher wurde der zweiten Farbe, Diamanten, die Nummer 1 zugewiesen. Im Fall von `Rank` kann es sinnvoller sein, die Nummer bei eins zu beginnen. Dies wurde durch Annotation der Definition von `ace` mit einer Annotation `=1` .

Aufzählungsarten verfügen über eine Vielzahl von Funktionen, z. B. Gleichheit (und sogar Identität) und integrierte Vergleiche:

```
julia> seven === seven
true

julia> ten ≠ jack
true

julia> two < three
true
```

Wie Werte eines anderen unveränderlichen Typs können auch Werte von aufgezählten Typen in `Dict` gespeichert und gespeichert werden.

Wir können dieses Beispiel vervollständigen, indem Sie einen `Card` definieren, der ein Feld für `Rank` und ein Feld für `Suit` :

```
immutable Card
    rank::Rank
    suit::Suit
end
```

und somit können wir Karten mit erstellen

```
julia> Card(three, ♣)
Card(three::Rank = 3,♣::Suit = 0)
```

Aufzählungsarten verfügen jedoch auch über eigene `convert` , sodass wir dies in der Tat tun können

```
julia> Card(7, ♠)
Card(seven::Rank = 7,♠::Suit = 3)
```

Da `7` direkt in `Rank` konvertiert werden kann, ist dieser Konstruktor sofort einsatzbereit.

Wir möchten möglicherweise syntaktischen Zucker für die Konstruktion dieser Karten definieren. Die implizite Multiplikation bietet eine bequeme Möglichkeit, dies zu tun. Definieren

```
julia> import Base.*

julia> r::Int * s::Suit = Card(r, s)
* (generic function with 156 methods)
```

und dann

```
julia> 10♣
Card(ten::Rank = 10,♣::Suit = 0)

julia> 5♠
Card(five::Rank = 5,♠::Suit = 3)
```

wieder die eingebauten `convert` .

Verwendung von Symbolen als leichtes Enum

Obwohl das `@enum` Makro für die meisten Anwendungsfälle recht nützlich ist, kann es in einigen Anwendungsfällen übermäßig sein. Nachteile von `@enum` sind:

- Es wird ein neuer Typ erstellt
- Es ist etwas schwieriger zu erweitern
- Sie bietet Funktionen wie Konvertierung, Aufzählung und Vergleich, die in manchen Anwendungen überflüssig sein können

Wenn eine leichtere Alternative gewünscht wird, kann der `Symbol` verwendet werden. Symbole sind interne Zeichenfolgen . Sie stellen Zeichenfolgen dar, ähnlich wie Zeichenfolgen , aber sie sind eindeutig mit Zahlen verknüpft. Diese einzigartige Zuordnung ermöglicht einen schnellen Vergleich der Symbolgleichheit.

Wir können einen `Card` erneut implementieren, diesmal mit `Symbol` :

```
const ranks = Set([:ace, :two, :three, :four, :five, :six, :seven, :eight, :nine,
                   :ten, :jack, :queen, :king])
const suits = Set([:♣, :♦, :♥, :♠])
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
```

Wir implementieren den inneren Konstruktor, um zu überprüfen, ob falsche Werte an den Konstruktor übergeben werden. Anders als in dem Beispiel, in dem `@enum` Typen verwendet werden, können `Symbol` eine beliebige Zeichenfolge enthalten. `@enum` müssen wir darauf achten, welche Arten von `Symbol` wir akzeptieren. Beachten Sie hier die Verwendung der bedingten Kurzschlussoperatoren .

Jetzt können wir `Card` Objekte wie erwartet `Card` :

```
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
```

Ein großer Vorteil von `Symbol` ist die Laufzeiterweiterung. Wenn wir zur Laufzeit (zum Beispiel) `:eleven` als neuen Rang akzeptieren möchten, genügt es, einfach `push!(ranks, :eleven)` auszuführen. Eine solche Laufzeiterweiterung ist bei `@enum` Typen nicht möglich.

Aufzählungen online lesen: https://riptutorial.com/de/julia-lang/topic/7104/aufzahlungen

Kapitel 7: Ausdrücke

Examples

Einführung in Ausdrücke

Ausdrücke sind ein spezieller Objekttyp in Julia. Sie können sich einen Ausdruck als einen Teil des Julia-Codes vorstellen, der noch nicht ausgewertet wurde (dh ausgeführt wurde). Es gibt dann bestimmte Funktionen und Operationen, wie `eval()` die den Ausdruck auswerten.

Zum Beispiel könnten wir ein Skript schreiben oder in den Interpreter folgendes eingeben: julia> 1 + 1 2

Eine Möglichkeit zum Erstellen eines Ausdrucks ist die Syntax `:()` . Zum Beispiel:

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

Wir haben jetzt ein `Expr` Typ `Expr` . Gerade erst gebildet, tut es nichts - es sitzt einfach wie jedes andere Objekt herum, bis es reagiert wird. In diesem Fall können wir diesen Ausdruck mit der Funktion `eval()` *auswerten* :

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

So sehen wir, dass die folgenden zwei gleichwertig sind:

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

Warum sollten wir die viel kompliziertere Syntax in `eval(:(1+1))` durchgehen, wenn wir nur herausfinden wollen, was 1 + 1 gleich ist? Der Hauptgrund ist, dass wir einen Ausdruck an einer Stelle in unserem Code definieren, ihn möglicherweise später ändern und ihn später noch auswerten können. Dies kann dem Julia-Programmierer potentiell neue Möglichkeiten eröffnen. Ausdrücke sind eine Schlüsselkomponente der Metaprogrammierung in Julia.

Ausdrücke erstellen

Es gibt verschiedene Methoden, um denselben Ausdruckstyp zu erstellen. Das Intro für Ausdrücke erwähnte die `:()` Syntax. Der beste Startplatz ist jedoch vielleicht mit Schnüren. Dies hilft, einige der grundlegenden Ähnlichkeiten zwischen Ausdrücken und Strings in Julia aufzuzeigen.

Erstellen Sie einen Ausdruck aus Zeichenfolge

Aus der Julia- Dokumentation :

Jedes Julia-Programm beginnt als Zeichenfolge

Mit anderen Worten, jedes Julia-Skript wird einfach in eine Textdatei geschrieben, die nur aus einer Zeichenfolge besteht. Ebenso ist jeder Julia-Befehl, der in einen Interpreter eingegeben wird, nur eine Zeichenfolge. Die Rolle von Julia oder einer anderen Programmiersprache besteht dann darin, Zeichenfolgen auf logische und vorhersagbare Weise zu interpretieren und zu bewerten, sodass diese Zeichenfolgen verwendet werden können, um zu beschreiben, was der Programmierer vom Computer erreichen möchte.

Daher besteht eine Möglichkeit zum Erstellen eines Ausdrucks darin, die Funktion `parse()` als auf einen String angewendet zu verwenden. Der folgende Ausdruck weist nach der Auswertung dem Symbol `x` den Wert 2 zu.

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

Erstellen Sie einen Ausdruck mit `:()` Syntax

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

Beachten Sie, dass Julia mit dieser Syntax die Namen von Objekten automatisch als Verweis auf Symbole behandelt. Wir können das sehen, wenn wir die `args` des Ausdrucks betrachten. (Weitere Informationen zum Feld `args` in einem Ausdruck finden Sie unter Felder mit Ausdrucksobjekten.)

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

Erstellen Sie einen Ausdruck mit der Funktion `Expr()`

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

Diese Syntax basiert auf der Präfixnotation . Mit anderen Worten ist das erste Argument der für die Funktion `Expr()` Funktion der `head` oder das Präfix. Die restlichen sind die `arguments` des Ausdrucks. Der `head` bestimmt, welche Operationen an den Argumenten ausgeführt werden.

Weitere Informationen hierzu finden Sie unter Felder mit Ausdrucksobjekten

Bei der Verwendung dieser Syntax ist es wichtig, zwischen Objekten und Symbolen für Objekte zu unterscheiden. Im obigen Beispiel weist der Ausdruck beispielsweise dem Symbol den Wert `2` zu `:x` , eine absolut sinnvolle Operation. Wenn wir in einem solchen Ausdruck `x` selbst verwenden,

erhalten wir das unsinnige Ergebnis:

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

Wenn wir die `args` , sehen wir:

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

Daher führt die Funktion `Expr()` nicht die gleiche automatische Umwandlung in Symbole durch wie die Syntax `:()` zum Erstellen von Ausdrücken.

Erstellen Sie mehrzeilige Ausdrücke mit `quote...end`

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

Beachten Sie, dass wir mit `quote...end` Ausdrücke erstellen können, die andere Ausdrücke in ihrem `args` :

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

Weitere `args` zu diesem `args` Sie unter Felder mit Ausdrucksobjekten .

Weitere Informationen zum Erstellen von Ausdrücken

Dieses Beispiel enthält lediglich die Grundlagen zum Erstellen von Ausdrücken. Weitere Informationen zum Erstellen komplexerer und erweiterter Ausdrücke finden Sie auch unter Interpolation und Ausdrücke und Felder von Ausdrucksobjekt.

Felder von Ausdrucksobjekten

Wie in der Einführung zu Ausdrücken erwähnt, sind Ausdrücke ein spezieller Objekttyp in Julia. Als solche haben sie Felder. Die zwei am häufigsten verwendeten Felder eines Ausdrucks sind der `head` und die `args` . Betrachten Sie zum Beispiel den Ausdruck

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

in Erstellen von Ausdrücken behandelt . Wir können den `head` und die `args` wie folgt sehen:

```
julia> MyExpr3.head
```

```
:(=)

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

Ausdrücke basieren auf der Präfixnotation . Der `head` spezifiziert im Allgemeinen die Operation, die an den `args` . Der Kopf muss ein Julia- `Symbol` .

Wenn ein Ausdruck einen Wert zuweisen soll (wenn er ausgewertet wird), verwendet er im Allgemeinen einen Kopf von `:(=)` . Es gibt natürlich offensichtliche Variationen, die verwendet werden können, zB:

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

: Aufruf zum Ausdruck Köpfe

Ein weiterer gemeinsamer `head` für Ausdrücke ist `:call` . Z.B

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

Nach den Konventionen der Präfixnotation werden Operatoren von links nach rechts ausgewertet. Daher bedeutet dieser Ausdruck hier, dass wir die Funktion aufrufen, die im ersten Element von `args` in den nachfolgenden Elementen angegeben ist. In ähnlicher Weise könnten wir

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

Oder andere, möglicherweise interessantere Funktionen, z

```
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
```

Automatische Ermittlung des `head` bei Verwendung von `:()` Ausdruckserstellungsnotation

Beachten Sie, dass `:call` in bestimmten Ausdruckskonstruktionen der `:call` implizit als Kopf verwendet wird, z

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

Mit der Syntax `:()` zum Erstellen von Ausdrücken versucht Julia daher automatisch, den richtigen zu verwendenden Kopf zu ermitteln. Ähnlich:

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

Wenn Sie nicht sicher sind, was der richtige Kopf für einen Ausdruck ist, den Sie zum Beispiel mit `Expr()` formen, kann dies ein hilfreiches Werkzeug sein, um Tipps und Ideen zu erhalten.

Interpolation und Ausdrücke

Das Erstellen von Ausdrücken erwähnt, dass Ausdrücke eng mit Strings zusammenhängen. Daher sind die Interpolationsprinzipien in Strings auch für Ausdrücke relevant. Bei der grundlegenden String-Interpolation können wir beispielsweise Folgendes haben:

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

Wir verwenden das `$` -Zeichen, um den Wert von `n` in die Zeichenfolge einzufügen. Wir können dieselbe Technik mit Ausdrücken verwenden. Z.B

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

Im Gegensatz dazu dies:

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

Im ersten Beispiel setzen wir also vorab den Wert von `a` , der zum Zeitpunkt der Auswertung des Ausdrucks verwendet wird. Mit dem zweiten Beispiel werden jedoch die Julia - Compiler nur schauen `a` , um seinen Wert *zum Zeitpunkt der Auswertung* für unseren Ausdruck zu finden.

Externe Verweise auf Ausdrücke

Es gibt eine Reihe nützlicher Webressourcen, mit denen Sie Ihr Wissen über Ausdrücke in Julia erweitern können. Diese schließen ein:

- Julia Docs - Metaprogrammierung
- Wikibooks - Julia Metaprogrammierung
- Julias Makros, Ausdrücke usw. für und von den Verwirrten von Gray Calhoun
- Monat Julia - Metaprogrammierung, von Andrew Collier
- Symbolische Unterscheidung in Julia von John Myles White

SO Beiträge:

- Was ist ein "Symbol" in Julia? Antwort von Stefan Karpinski
- Warum bringt Julia diesen Ausdruck auf diese komplexe Weise zum Ausdruck?
- Erläuterung des Beispiels für die Julia-Ausdrucksinterpolation

Ausdrücke online lesen: https://riptutorial.com/de/julia-lang/topic/5805/ausdrucke

Kapitel 8: Conditionals

Syntax

- wenn cond; Karosserie; Ende
- wenn cond; Karosserie; sonst; Karosserie; Ende
- wenn cond; Karosserie; elseif cond; Karosserie; sonst; Ende
- wenn cond; Karosserie; elseif cond; Karosserie; Ende
- cond iftrue: iffalse
- cond && iftrue
- cond || iffalse
- ifelse (cond, iftrue, iffalse)

Bemerkungen

Alle bedingten Operatoren und Funktionen erfordern die Verwendung boolescher Bedingungen (`true` oder `false`). In Julia ist der Typ von Booleans `Bool` . Im Gegensatz zu anderen Sprachen können andere Arten von Zahlen (wie `1` oder `0`), Zeichenfolgen, Arrays usw. *nicht* direkt in Bedingungen verwendet werden.

Normalerweise verwendet man entweder Prädikatfunktionen (Funktionen, die ein `Bool`) oder Vergleichsoperatoren im Zustand eines bedingten Operators oder einer Funktion.

Examples

wenn ... sonst Ausdruck

Die häufigste Bedingung in Julia ist der `if` ... `else` Ausdruck. Im Folgenden implementieren wir zum Beispiel den Euklidischen Algorithmus zur Berechnung des größten gemeinsamen Divisors unter Verwendung einer Bedingung, um den Basisfall zu behandeln:

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

Das `if` ... `else` Formular in Julia ist tatsächlich ein Ausdruck und hat einen Wert. Der Wert ist der Ausdruck in Endposition (d. h. der letzte Ausdruck) in der Verzweigung. Betrachten Sie die folgende Beispieleingabe:

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

Hier ist `a 0` und `b` ist `-10` . Die Bedingung `a == 0` ist `true` , also wird der erste Zweig genommen. Der

zurückgegebene Wert ist `abs(b)` , also `10` .

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

Hier ist `a` `2` und `b` `3` . Die Bedingung `a == 0` ist falsch, also wird der zweite Zweig genommen und wir berechnen `mygcd(b % a, a)` , was `mygcd(3 % 2, 2)` . Der Operator `%` gibt den Rest zurück, wenn `3` durch `2` dividiert wird, in diesem Fall `1` . Somit berechnen wir `mygcd(1, 2)` , und diesmal `a` ist `1` , und `b` ist `2` . Wieder ist `a == 0` falsch, also wird der zweite Zweig genommen, und wir berechnen `mygcd(b % a, a)` , was `mygcd(0, 1)` . Diesmal wird endlich `a == 0` und so wird `abs(b)` zurückgegeben, was das Ergebnis `1` ergibt.

wenn ... sonst eine Aussage

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

Jeder Ausdruck, beispielsweise der `if` ... `else` Ausdruck, kann an die Anweisungsposition gesetzt werden. Dies ignoriert seinen Wert, führt jedoch immer noch den Ausdruck für die Nebeneffekte aus.

wenn Aussage

Der Rückgabewert eines `if` ... `else` Ausdrucks kann wie jeder andere Ausdruck ignoriert (und somit verworfen werden). Dies ist im Allgemeinen nur nützlich, wenn der Hauptteil des Ausdrucks Nebenwirkungen hat, z. B. das Schreiben in eine Datei, das Variieren von Variablen oder das Drucken auf dem Bildschirm.

Außerdem ist der `else` Zweig eines `if` ... `else` Ausdrucks optional. Zum Beispiel können wir den folgenden Code nur für die Ausgabe auf dem Bildschirm schreiben, wenn eine bestimmte Bedingung erfüllt ist:

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

Im obigen Beispiel verwenden wir Zeit- und Datumsfunktionen , um die aktuelle Sekunde abzurufen. Wenn es zum Beispiel 10:55:27 ist, wird die `second` Variable `27` halten. Wenn diese Anzahl gerade ist, wird eine Zeile auf den Bildschirm gedruckt. Sonst wird nichts unternommen.

Ternärer bedingter Operator

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

Der ternäre Bedingungsoperator ist ein weniger wortreicher `if` ... `else` Ausdruck.

Die Syntax lautet speziell:

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

In diesem Beispiel fügen wir `x` auf die Sammlung `A` nur dann , wenn `x` nicht bereits in ist `A` . Ansonsten lassen wir `A` einfach unverändert.

Ternärer Betreiber Referenzen:

- Julia-Dokumentation
- Wikibooks

Kurzschlussoperatoren: && und ||

Zum Verzweigen

Die kurzschließenden bedingten Operatoren `&&` und `||` kann als leichter Ersatz für die folgenden Konstrukte verwendet werden:

- `x && y` ist äquivalent zu `x ? y : x`
- `x || y` ist äquivalent zu `x ? x : y`

Eine Anwendung für Kurzschlussoperatoren ist die präzisere Möglichkeit, eine Bedingung zu testen und abhängig von dieser Bedingung eine bestimmte Aktion auszuführen. Im folgenden Code wird beispielsweise mit dem Operator `&&` ein Fehler `&&` , wenn das Argument `x` negativ ist:

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

Die `||` Der Operator kann auch zur Fehlerüberprüfung verwendet werden, außer dass er den Fehler auslöst, es *sei denn,* eine Bedingung gilt, und nicht, *wenn* die Bedingung gilt:

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

Eine weitere nützliche Anwendung ist, einem Objekt einen Standardwert zu übergeben, sofern es nicht zuvor definiert wurde:

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

Hier wird geprüft, ob das Symbol x definiert ist (dh ob dem Objekt `x` ein Wert zugeordnet ist). Wenn ja, passiert nichts. Wenn nicht, wird `x` `NEW_VALUE` zugewiesen. Beachten Sie, dass dieses Beispiel nur im Toplevel-Bereich funktioniert.

Unter Bedingungen

Die Operatoren sind auch nützlich, da mit ihnen zwei Bedingungen getestet werden können, von denen die zweite nur in Abhängigkeit vom Ergebnis der ersten Bedingung ausgewertet wird. Aus der Julia- Dokumentation :

> Im Ausdruck `a && b` wird der Unterausdruck `b` nur ausgewertet, wenn `a` Wert `true` ergibt
>
> In dem Ausdruck `a || b` wird der Unterausdruck `b` nur ausgewertet, wenn `a` zu `false` ausgewertet wird

Während sowohl `a & b` als auch `a && b true` wenn sowohl `a` als auch `b true` , ist ihr Verhalten anders, wenn `a false` ist.

Nehmen wir zum Beispiel an, wir möchten prüfen, ob ein Objekt eine positive Zahl ist, wobei es sich möglicherweise um eine Zahl handelt. Berücksichtigen Sie die Unterschiede zwischen diesen beiden versuchten Implementierungen:

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

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

`CheckPositive1()` gibt einen Fehler aus, wenn ein nicht numerischer Typ als Argument angegeben wird. Dies liegt daran, dass *beide* Ausdrücke unabhängig vom Ergebnis des ersten ausgewertet werden und der zweite Ausdruck einen Fehler ausgibt, wenn versucht wird, ihn für einen nicht numerischen Typ auszuwerten.

`CheckPositive2()` liefert jedoch `false` (anstatt eines Fehlers), wenn ein nicht numerischer Typ angegeben wird, da der zweite Ausdruck nur ausgewertet wird, wenn der erste `true` .

Es können mehrere Kurzschlussoperatoren aneinandergereiht werden. Z.B:

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

if-Anweisung mit mehreren Zweigen

```
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
```

Mit einer `if` Anweisung kann eine beliebige Anzahl von `elseif` Zweigen verwendet werden,

möglicherweise mit oder ohne `elseif` `else` Zweig. Nachfolgende Bedingungen werden nur ausgewertet, wenn alle vorherigen Bedingungen als `false` befunden wurden.

Die ifelse-Funktion

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

Verwendungszweck:

```
julia> shift(10)
9

julia> shift(11)
12

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

Die `ifelse` Funktion wertet beide Zweige aus, auch den nicht ausgewählten. Dies kann nützlich sein, wenn die Zweige Nebenwirkungen haben, die bewertet werden müssen, oder weil es schneller sein kann, wenn beide Zweige selbst billig sind.

Conditionals online lesen: https://riptutorial.com/de/julia-lang/topic/4356/conditionals

Kapitel 9: Eingang

Syntax

- Zeile lesen()
- readlines ()
- Lesefolge (STDIN)
- chomp (str)
- öffnen (f, Datei)
- Jede Zeile (io)
- Lesezeichenfolge (Datei)
- lesen (Datei)
- readcsv (Datei)
- readdlm (Datei)

Parameter

Parameter	Einzelheiten
`chomp(str)`	**Entfernen Sie bis zu eine nachgestellte Zeile aus einer Zeichenfolge.**
`str`	Die Zeichenfolge, aus der eine nachgestellte Zeile entfernt wird. Beachten Sie, dass Zeichenfolgen durch Konvention unveränderlich sind. Diese Funktion gibt eine neue Zeichenfolge zurück.
`open(f, file)`	**Öffnen Sie eine Datei, rufen Sie die Funktion auf und schließen Sie die Datei anschließend.**
`f`	Die Funktion, die für den IO-Stream aufgerufen werden soll, der die Datei öffnet.
`file`	Der Pfad der zu öffnenden Datei.

Examples

Ein String aus der Standardeingabe lesen

Der `STDIN` Stream in Julia bezieht sich auf die Standardeingabe . Dies kann entweder eine Benutzereingabe für interaktive Befehlszeilenprogramme oder eine Eingabe aus einer Datei oder Pipeline darstellen , die in das Programm umgeleitet wurde.

Wenn die `readline` Funktion keine Argumente `STDIN` , liest sie Daten aus `STDIN` bis eine neue Zeile gefunden wird oder der `STDIN` Stream den Dateiendungstatus erreicht. Diese beiden Fälle unterscheiden sich darin, ob das Zeichen `\n` als letztes Zeichen gelesen wurde:

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

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

Für interaktive Programme ist uns der EOF-Status oft nicht wichtig, und wir möchten nur eine Zeichenfolge. Zum Beispiel können wir den Benutzer zur Eingabe auffordern:

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

Dies ist jedoch aufgrund des zusätzlichen Newlines nicht ganz zufriedenstellend:

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

Die `chomp` Funktion ist verfügbar, um bis zu eine nachgestellte Zeile aus einer Zeichenfolge zu entfernen. Zum Beispiel:

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

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

Wir können daher unsere Funktion mit `chomp` so erweitern, dass das Ergebnis wie erwartet ist:

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

was ein wünschenswerteres Ergebnis hat:

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

Manchmal möchten wir möglicherweise so viele Zeilen wie möglich lesen (bis der Eingabestrom in den Dateiendungszustand wechselt). Die Funktion `readlines` bietet diese Funktion.

```
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

Wenn wir die Zeilenumbrüche am Zeilenende, die von `readlines` gelesen werden, nicht `chomp` , können wir sie mit der `chomp` Funktion entfernen. Diesmal senden wir die `chomp` Funktion über das gesamte Array:

```
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"
```

In anderen Fällen interessieren wir uns vielleicht überhaupt nicht für Zeilen und möchten einfach so viel wie möglich als einzelne Zeichenfolge lesen. Die `readstring` Funktion `readstring` dies:

```
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"
```

(Das `# [END OF INPUT]` ist nicht Teil der ursprünglichen Eingabe; es wurde aus Gründen der Übersichtlichkeit hinzugefügt.)

Beachten Sie, dass `readstring` muss übergeben werden `STDIN` Argument.

Zahlen aus der Standardeingabe lesen

Das Lesen von Zahlen aus der Standardeingabe ist eine Kombination aus dem Lesen von Zeichenfolgen und dem Parsen solcher Zeichenfolgen als Zahlen.

Die `parse` - Funktion wird verwendet , um eine Zeichenkette in den gewünschten Anzahl Typen zu analysieren:

```
julia> parse(Int, "17")
17

julia> parse(Float32, "-3e6")
-3.0f6
```

Das von `parse(T, x)` erwartete Format ist ähnlich, aber nicht genau dasselbe, wie das Format, das Julia von Zahlenlitern erwartet:

```
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
```

Durch die Kombination der `parse` und `readline` Funktionen können wir eine einzelne Nummer aus einer Zeile lesen:

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

was wie erwartet funktioniert:

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

Es gelten die üblichen Vorbehalte bezüglich der Fließkomma-Genauigkeit . Beachten Sie, dass `parse` kann mit verwendet werden `BigInt` und `BigFloat` zu entfernen oder Verlust an Präzision zu minimieren.

Manchmal ist es nützlich, mehrere Nummern aus derselben Zeile zu lesen. In der Regel kann die Zeile mit Leerzeichen geteilt werden:

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

```
end
```

was kann wie folgt verwendet werden:

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

Daten aus einer Datei lesen

Strings oder Bytes lesen

Dateien können zum Lesen mit der `open` Funktion `open` werden, die häufig zusammen mit do-Block-Syntax verwendet wird :

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

Angenommen, `myfile` existiert und sein Inhalt ist

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

Dann würde dieser Code das folgende Ergebnis erzeugen:

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

Beachten Sie, dass `eachline` faul iterable über die Zeilen der Datei ist. Es wird bevorzugt, `readlines` aus Leistungsgründen zu `readlines` .

Da `do` Block-Syntax für anonyme Funktionen nur syntaktischer Zucker ist, können wir auch benannte Funktionen an `open` :

```
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
```

Die Funktionen `read` und `readstring` bieten komfortable Methoden, mit denen eine Datei automatisch geöffnet wird:

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

Strukturierte Daten lesen

Nehmen wir an, wir hätten eine CSV-Datei mit folgendem Inhalt in einer Datei namens `file.csv` :

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

Dann können wir die `readcsv` Funktion verwenden, um diese Daten in eine `Matrix` zu lesen:

```
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
```

Wenn die Datei stattdessen durch Tabulatoren in einer Datei namens `file.tsv` , kann stattdessen die Funktion `readdlm` verwendet werden, wobei das Argument `delim` auf `'\t'` . Für fortgeschrittenere Workloads sollte das Paket CSV.jl verwendet werden .

Eingang online lesen: https://riptutorial.com/de/julia-lang/topic/7201/eingang

Kapitel 10: Funktionen

Syntax

- f (n) = ...
- Funktion f (n) ... Ende
- n :: Typ
- x -> ...
- f (n) do ... end

Bemerkungen

Neben generischen Funktionen (die am häufigsten vorkommen) gibt es auch integrierte Funktionen. Solche Funktionen umfassen `is` , `isa` , `typeof` , `throw` und ähnliche Funktionen. Integrierte Funktionen werden normalerweise in C anstelle von Julia implementiert. Sie können daher nicht auf Argumenttypen für den Versand spezialisiert werden.

Examples

Platzieren Sie eine Zahl

Dies ist die einfachste Syntax zum Definieren einer Funktion:

```
square(n) = n * n
```

Um eine Funktion aufzurufen, verwenden Sie runde Klammern (ohne Leerzeichen dazwischen):

```
julia> square(10)
100
```

Funktionen sind Objekte in Julia und wir können sie in der REPL wie alle anderen Objekte zeigen:

```
julia> square
square (generic function with 1 method)
```

Alle Julia-Funktionen sind standardmäßig generisch (auch als polymorph bezeichnet). Unsere `square` funktioniert genauso gut mit Fließkommazahlen:

```
julia> square(2.5)
6.25
```

... oder sogar Matrizen :

```
julia> square([2 4
               2 1])
```

```
2×2 Array{Int64,2}:
 12  12
  6   9
```

Rekursive Funktionen

Einfache Rekursion

Mit Rekursion und dem ternären bedingten Operator können wir eine alternative Implementierung der eingebauten `factorial` Funktion erstellen:

```
myfactorial(n) = n == 0 ? 1 : n * myfactorial(n - 1)
```

Verwendungszweck:

```
julia> myfactorial(10)
3628800
```

Mit Bäumen arbeiten

Rekursive Funktionen sind oft am nützlichsten für Datenstrukturen, insbesondere für Baumdatenstrukturen. Da Ausdrücke in Julia Baumstrukturen sind, kann Rekursion für die Metaprogrammierung sehr nützlich sein. Die folgende Funktion sammelt beispielsweise eine Menge aller Köpfe, die in einem Ausdruck verwendet werden.

```
heads(ex::Expr) = reduce(∪, Set((ex.head,)), (heads(a) for a in ex.args))
heads(::Any) = Set{Symbol}()
```

Wir können überprüfen, ob unsere Funktion wie vorgesehen funktioniert:

```
julia> heads(:(7 + 4x > 1 > A[0]))
Set(Symbol[:comparison,:ref,:call])
```

Diese Funktion ist kompakt und verwendet eine Vielzahl fortgeschrittener Techniken, z. B. die Funktion zum `reduce` höherer Ordnung , den Datentyp `Set` und Generatorausdrücke.

Einführung in den Versand

Wir können die `::` -Syntax verwenden, um den Typ eines Arguments abzusetzen.

```
describe(n::Integer) = "integer $n"
describe(n::AbstractFloat) = "floating point $n"
```

Verwendungszweck:

```
julia> describe(10)
"integer 10"
```

```
julia> describe(1.0)
"floating point 1.0"
```

Im Gegensatz zu vielen Sprachen, die normalerweise entweder statische Mehrfachzustellung oder dynamische Einzelzustellung bereitstellen, verfügt Julia über vollständige dynamische Mehrfachzustellung. Das heißt, Funktionen können auf mehrere Argumente spezialisiert sein. Dies ist praktisch, wenn Sie spezielle Methoden für Operationen mit bestimmten Typen und Fallback-Methoden für andere Typen definieren.

```
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"
```

Verwendungszweck:

```
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"
```

Optionale Argumente

Julia erlaubt Funktionen, optionale Argumente zu übernehmen. Hinter den Kulissen wird dies als ein weiterer Sonderfall des Mehrfachversands implementiert. Lösen wir zum Beispiel das beliebte Fizz Buzz-Problem . Standardmäßig machen wir dies für Zahlen im Bereich `1:10` , aber wenn nötig, erlauben wir einen anderen Wert. Wir werden auch verschiedene Sätze für `Fizz` oder `Buzz` zulassen.

```
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
```

Wenn wir `fizzbuzz` in der REPL untersuchen, heißt es, dass es vier Methoden gibt. Für jede zulässige Kombination von Argumenten wurde eine Methode erstellt.

```
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
```

Wir können überprüfen, ob unsere Standardwerte verwendet werden, wenn keine Parameter angegeben werden:

```
julia> fizzbuzz()
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
```

Die optionalen Parameter werden jedoch akzeptiert und respektiert, wenn wir sie angeben:

```
julia> fizzbuzz(5:8, "fuzz", "bizz")
bizz
fuzz
7
8
```

Parametrischer Versand

Es ist häufig der Fall, dass eine Funktion auf parametrische Typen wie `Vector{T}` oder `Dict{K,V}`, aber die Typparameter sind nicht festgelegt. Dieser Fall kann mit dem parametrischen Versand behandelt werden:

```
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"
```

Man könnte versucht sein, einfach `xs::Vector{Number}` zu schreiben. Dies funktioniert jedoch nur für Objekte, deren Typ explizit `Vector{Number}` :

```
julia> isa(Number[1, 2], Vector{Number})
true

julia> isa(Int[1, 2], Vector{Number})
false
```

Dies liegt an der parametrischen Invarianz : Das Objekt `Int[1, 2]` ist *kein* `Vector{Number}` , da es nur `Int` s enthalten kann, wohingegen ein `Vector{Number}` alle möglichen Zahlen enthalten kann.

Generischen Code schreiben

Dispatch ist eine unglaublich leistungsstarke Funktion, aber häufig ist es besser, generischen Code zu schreiben, der für alle Typen geeignet ist, anstatt den Code für jeden Typ zu spezialisieren. Durch das Schreiben von generischem Code wird die Duplizierung von Code vermieden.

Hier ist zum Beispiel Code, um die Summe der Quadrate eines Vektors von Ganzzahlen zu berechnen:

```
function sumsq(v::Vector{Int})
    s = 0
    for x in v
        s += x ^ 2
    end
    s
end
```

Dieser Code funktioniert jedoch *nur* für einen Vektor von `Int` s. Es funktioniert nicht bei einem `UnitRange` :

```
julia> sumsq(1:10)
ERROR: MethodError: no method matching sumsq(::UnitRange{Int64})
Closest candidates are:
  sumsq(::Array{Int64,1}) at REPL[8]:2
```

Es funktioniert nicht auf einem `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
```

Eine bessere Methode zum Schreiben dieser `sumsq` Funktion sollte sein

```
function sumsq(v::AbstractVector)
    s = zero(eltype(v))
    for x in v
        s += x ^ 2
    end
    s
end
```

Dies funktioniert in den beiden oben aufgeführten Fällen. Aber es gibt einige Sammlungen, in denen wir vielleicht die Quadrate davon zusammenfassen wollen, die in keiner Weise Vektoren sind. Zum Beispiel,

```
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
```

zeigt, dass wir die Quadrate eines faulen Iterablen nicht summieren können.

Eine noch allgemeinere Implementierung ist einfach

```
function sumsq(v)
    s = zero(eltype(v))
    for x in v
        s += x ^ 2
    end
    s
end
```

Was funktioniert in allen Fällen:

```
julia> sumsq(take(countfrom(1), 100))
338350
```

Dies ist der idiomatischste Julia-Code und kann mit allen möglichen Situationen umgehen. In einigen anderen Sprachen kann das Entfernen von Typanmerkungen die Leistung beeinträchtigen. Dies ist jedoch in Julia nicht der Fall. Nur die Typenstabilität ist wichtig für die Leistung.

Imperative Fakultät

Für die Definition von mehrzeiligen Funktionen steht eine Langform-Syntax zur Verfügung. Dies kann nützlich sein, wenn wir imperative Strukturen wie Schleifen verwenden. Der Ausdruck in Endposition wird zurückgegeben. Zum Beispiel verwendet die untenstehende Funktion eine `for`

Schleife , um die Fakultät einiger Integer `n` zu berechnen:

```
function myfactorial(n)
    fact = one(n)
    for m in 1:n
        fact *= m
    end
    fact
end
```

Verwendungszweck:

```
julia> myfactorial(10)
3628800
```

Bei längeren Funktionen wird häufig die verwendete `return` Anweisung angezeigt. Die `return` Anweisung ist in Endposition nicht erforderlich, wird jedoch aus Gründen der Klarheit manchmal verwendet. Eine andere Schreibweise für die obige Funktion wäre beispielsweise

```
function myfactorial(n)
    fact = one(n)
    for m in 1:n
        fact *= m
    end
    return fact
end
```

was im Verhalten mit der Funktion oben identisch ist.

Anonyme Funktionen

Pfeilsyntax

Anonyme Funktionen können mit der Syntax `->` erstellt werden. Dies ist hilfreich, wenn Sie Funktionen an Funktionen höherer Ordnung übergeben , z. B. die `map` Funktion. Die folgende Funktion berechnet das Quadrat jeder Zahl in einem Feld `A`

```
squareall(A) = map(x -> x ^ 2, A)
```

Ein Beispiel für die Verwendung dieser Funktion:

```
julia> squareall(1:10)
10-element Array{Int64,1}:
   1
   4
   9
  16
  25
  36
  49
  64
```

```
  81
 100
```

Mehrzeilige Syntax

Anonyme Funktionen mit mehreren Linien können mithilfe der `function` erstellt werden. Im folgenden Beispiel werden beispielsweise die Fakultäten der ersten `n` Zahlen berechnet, wobei jedoch eine anonyme Funktion anstelle der eingebauten `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
```

Blockieren Sie die Syntax

Da es so üblich ist, eine anonyme Funktion als erstes Argument an eine Funktion zu übergeben, gibt es eine `do` Block-Syntax. Die Syntax

```
map(A) do x
    x ^ 2
end
```

ist äquivalent zu

```
map(x -> x ^ 2, A)
```

Ersteres kann jedoch in vielen Situationen klarer sein, insbesondere wenn in der anonymen Funktion viel berechnet wird. `do` Blocksyntax ist besonders für die Dateieingabe und -ausgabe aus Gründen der Ressourcenverwaltung hilfreich.

Funktionen online lesen: https://riptutorial.com/de/julia-lang/topic/3079/funktionen

Kapitel 11: Funktionen höherer Ordnung

Syntax

- foreach (f, xs)
- Karte (f, xs)
- filter (f, xs)
- reduzieren (f, v0, xs)
- foldl (f, v0, xs)
- foldr (f, v0, xs)

Bemerkungen

Funktionen können als Parameter akzeptiert und auch als Rückgabetypen erzeugt werden. In der Tat können Funktionen innerhalb des Körpers anderer Funktionen erstellt werden. Diese inneren Funktionen werden als Verschlüsse bezeichnet .

Examples

Funktionen als Argumente

Funktionen sind Objekte in Julia. Sie können wie alle anderen Objekte als Argumente an andere Funktionen übergeben werden. Funktionen, die Funktionen akzeptieren, werden als Funktionen höherer Ordnung bezeichnet.

Beispielsweise können wir ein Äquivalent der `foreach` Funktion der Standardbibliothek implementieren, indem wir als ersten Parameter eine Funktion `f` .

```
function myforeach(f, xs)
    for x in xs
        f(x)
    end
end
```

Wir können testen, dass diese Funktion tatsächlich wie erwartet funktioniert:

```
julia> myforeach(println, ["a", "b", "c"])
a
b
c
```

Wenn Sie statt eines späteren Parameters eine Funktion als *ersten* Parameter verwenden, können Sie die do-Block-Syntax von Julia verwenden. Die do-Block-Syntax ist nur eine bequeme Möglichkeit, eine anonyme Funktion als erstes Argument an eine Funktion zu übergeben.

```
julia> myforeach([1, 2, 3]) do x
```

```
            println(x^x)
        end
1
4
27
```

`myforeach` oben beschriebene Implementierung von `myforeach` entspricht in etwa der integrierten `foreach` Funktion. Es gibt auch viele andere integrierte Funktionen höherer Ordnung.

Funktionen höherer Ordnung sind ziemlich mächtig. Beim Arbeiten mit Funktionen höherer Ordnung werden die genauen ausgeführten Operationen manchmal unwichtig, und Programme können sehr abstrakt werden. Kombinatoren sind Beispiele für Systeme mit hoch abstrakten Funktionen höherer Ordnung.

Zuordnen, filtern und reduzieren

Zwei der grundlegendsten Funktionen höherer Ordnung, die in der Standardbibliothek enthalten sind, sind `map` und `filter` . Diese Funktionen sind generisch und können auf jeder Iteration ausgeführt werden . Sie eignen sich insbesondere für Berechnungen an Arrays .

Angenommen, wir haben einen Datensatz von Schulen. Jede Schule unterrichtet ein bestimmtes Fach, hat eine Anzahl von Klassen und eine durchschnittliche Anzahl von Schülern pro Klasse. Wir können eine Schule mit dem folgenden unveränderlichen Typ modellieren:

```
immutable School
    subject::Symbol
    nclasses::Int
    nstudents::Int  # average no. of students per class
end
```

Unser Datensatz von Schulen wird eine `Vector{School}` :

```
dataset = [School(:math, 3, 30), School(:math, 5, 20), School(:science, 10, 5)]
```

Angenommen, wir möchten herausfinden, wie viele Schüler insgesamt in einem Mathematikprogramm eingeschrieben sind. Dazu benötigen wir mehrere Schritte:

- wir müssen den Datensatz auf nur Schulen beschränken, die Mathematik unterrichten (`filter`)
- Wir müssen die Anzahl der Schüler an jeder Schule berechnen (`map`)
- und wir müssen diese Liste der Schülerzahlen auf einen einzigen Wert `reduce` , die Summe (`reduce`)

Eine naive (nicht sehr performante) Lösung wäre einfach, diese drei Funktionen höherer Ordnung direkt zu verwenden.

```
function nmath(data)
    maths = filter(x -> x.subject === :math, data)
    students = map(x -> x.nclasses * x.nstudents, maths)
    reduce(+, 0, students)
```

```
end
```

Wir überprüfen, dass sich in unserem Datensatz 190 Mathematikstudenten befinden:

```
julia> nmath(dataset)
190
```

Es gibt Funktionen, um diese Funktionen zu kombinieren und somit die Leistung zu verbessern. Wir hätten beispielsweise die `mapreduce` Funktion verwenden können, um das Mapping und die Reduktion in einem Schritt durchzuführen, was Zeit und Speicher sparen würde.

Die `reduce` ist nur für assoziative Operationen wie `+` , gelegentlich ist es jedoch hilfreich, eine Reduktion mit einer nicht assoziativen Operation durchzuführen. Die Funktionen `foldl` und `foldr` höherer Ordnung sind vorgesehen, um eine bestimmte Reduktionsordnung zu erzwingen.

Funktionen höherer Ordnung online lesen: https://riptutorial.com/de/julia-lang/topic/6955/funktionen-hoherer-ordnung

Kapitel 12: für Loops

Syntax

- denn ich in iter; ... Ende
- während cond; ... Ende
- brechen
- fortsetzen
- @parallel (op) für i in iter; ... Ende
- @parallel für i in iter; ... Ende
- @goto label
- @label label

Bemerkungen

Immer dann , wenn es Code kürzer und leichter lesen macht, sollten Sie mit Funktionen höherer Ordnung, wie `map` oder `filter` , anstelle von Schleifen.

Examples

Fizz Buzz

Ein häufiger Anwendungsfall für eine `for` Schleife besteht darin, einen vordefinierten Bereich oder eine vordefinierte Sammlung zu durchlaufen und dieselbe Aufgabe für alle Elemente auszuführen. Zum Beispiel kombinieren wir hier eine `for` Schleife mit einer bedingten `if elseif else` Anweisung :

```
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
```

Dies ist die klassische Fizz Buzz- Interviewfrage. Die (abgeschnittene) Ausgabe ist:

```
1
2
Fizz
4
Buzz
Fizz
7
8
```

Finde den kleinsten Primfaktor

In einigen Situationen möchten Sie möglicherweise von einer Funktion zurückkehren, bevor Sie eine gesamte Schleife abschließen. Die `return` Anweisung kann dazu verwendet werden.

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

Verwendungszweck:

```
julia> primefactor(100)
2

julia> primefactor(97)
97
```

Schleifen können auch mit der `break` Anweisung vorzeitig beendet werden, wodurch nur die einschließende Schleife anstelle der gesamten Funktion beendet wird.

Mehrdimensionale Iteration

In Julia kann eine for-Schleife ein Komma (`,`) enthalten, um die Iteration über mehrere Dimensionen anzugeben. Dies funktioniert ähnlich wie das Schachteln einer Schleife in einer anderen, kann jedoch kompakter sein. Mit der Funktion unten werden beispielsweise Elemente des kartesischen Produkts aus zwei iterierbaren Elementen generiert:

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

Verwendungszweck:

```
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)
```

Die Indizierung von Arrays einer beliebigen Dimension sollte jedoch mit `eachindex` , nicht mit einer

mehrdimensionalen Schleife (falls möglich):

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

Reduktion und Parallelschleifen

Julia bietet Makros zur Vereinfachung der Verteilung von Berechnungen auf mehrere Maschinen oder Worker. Im Folgenden wird beispielsweise die Summe einiger Quadrate berechnet, möglicherweise parallel.

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

Verwendungszweck:

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

Weitere `@parallel` zu diesem Thema finden Sie im Beispiel zu `@parallel` im Thema Parallele Verarbeitung.

für Loops online lesen: https://riptutorial.com/de/julia-lang/topic/4355/fur-loops

Kapitel 13: Iterables

Syntax

- start (itr)
- nächstes (itr, s)
- fertig (itr, s)
- nimm (itr, n)
- drop (itr, n)
- Zyklus (itr)
- Basisprodukt (xs, ys)

Parameter

Parameter	Einzelheiten
Zum	**Alle Funktionen**
`itr`	Die iterable zu operieren.
Zum	**`next` und `done`**
`s`	Ein Iteratorstatus, der die aktuelle Position der Iteration beschreibt.
Zum	**`take` und `drop`**
`n`	Die Anzahl der Elemente, die genommen oder fallen sollen.
Zum	**`Base.product`**
`xs`	Das iterable, um erste Elemente von Paaren zu nehmen.
`ys`	Das iterable, um zweite Elemente von Paaren zu nehmen.
...	(Beachten Sie, dass das `product` eine beliebige Anzahl von Argumenten akzeptiert. Wenn mehr als zwei angegeben werden, werden Tupel mit einer Länge von mehr als zwei erstellt.)

Examples

Neuer iterierbarer Typ

In Julia, wenn `I` ein iterierbares Objekt durchläuft, habe `I` die `for` Syntax:

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

```
    # body
end
```

Hinter den Kulissen heißt das:

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

Deshalb, wenn Sie wollen , `I` ein iterable sein, müssen Sie definieren `start` , `next` und `done` Methoden für seine Art. Angenommen, Sie definieren einen Typ `Foo` , der ein Array enthält, als eines der Felder:

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

Wir instantiieren ein `Foo` Objekt, indem wir Folgendes tun:

```
julia> I = Foo([1,2,3])
Foo([1,2,3])

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

Wenn wir durch iterieren wollen `Foo` , wobei jedes Element `bar` von jeder Iteration zurückgegeben werden, definieren wir die Methoden:

```
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
```

Beachten Sie, dass , da diese Funktionen in denen gehören `Base` müssen wir zuerst `import` , um sie vor dem Hinzufügen von neuen Methoden , um ihre Namen.

Nachdem die Methoden definiert sind, ist `Foo` mit der Iteratorschnittstelle kompatibel:

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

```
    end

1
2
3
```

Lazy Iterables kombinieren

Die Standardbibliothek verfügt über eine umfangreiche Sammlung von Lazy-Iterables (und Bibliotheken wie Iterators.jl bieten noch mehr). Lazy iterables können erstellt werden, um in konstanter Zeit leistungsfähigere iterables zu erstellen. Die wichtigsten faulen Iterables sind take and drop , aus denen viele andere Funktionen erstellt werden können.

Lazy Slice eine iterable

Arrays können mit Slice-Notation geschnitten werden. Das Folgende gibt beispielsweise das 10. bis 15. Element eines Arrays inklusive zurück:

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

Die Slice-Notation funktioniert jedoch nicht mit allen Iterables. Zum Beispiel können wir keinen Generatorausdruck schneiden:

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

Das Aufteilen von Zeichenfolgen weist möglicherweise nicht das erwartete Unicode-Verhalten auf:

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

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

Wir können eine Funktion `lazysub(itr, range::UnitRange)` , um diese Art von Slicing auf beliebigen `lazysub(itr, range::UnitRange)` . Dies ist in Bezug auf `take` und `drop` :

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

Die Implementierung funktioniert hier, da für den `UnitRange` Wert `a:b` die folgenden Schritte ausgeführt werden:

- löscht die ersten `a-1` Elemente
- nimmt das `a` te Element, `a+1` -Element usw., bis das `a+(ba)=b` te Element

Insgesamt werden `ba` Elemente genommen. Wir können bestätigen, dass unsere Implementierung jeweils oben korrekt ist:

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

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

Verschieben Sie eine iterable Zirkel

Die `circshift` Operation von Arrays verschiebt das Array wie einen Kreis und zeigt es dann erneut an. Zum Beispiel,

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

Können wir das faul für alle Iterationen machen? Wir können den `cycle` , das `drop` und `take` Durchlaufen von iterablen Funktionen verwenden, um diese Funktionalität zu implementieren.

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

Neben lazy-Typen, die in vielen Situationen leistungsfähiger sind, können wir damit eine `circshift` Funktion für Typen `circshift` , die sie sonst nicht unterstützen würden:

```
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"
```

0,5,0

Multiplikationstabelle erstellen

Lassen Sie uns eine Multiplikationstabelle erstellen, indem Sie faul iterierbare Funktionen verwenden, um eine Matrix zu erstellen.

Die wichtigsten Funktionen, die hier verwendet werden können, sind:

- `Base.product` , das ein kartesisches Produkt berechnet.
- `prod` , der ein reguläres Produkt berechnet (wie bei der Multiplikation)
- `:` , wodurch ein Bereich entsteht
- `map` , eine Funktion höherer Ordnung, die auf jedes Element einer Sammlung eine Funktion anwendet

Die Lösung ist:

```
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
```

Faul bewertete Listen

Es ist möglich, eine einfache, faul bewertete Liste mit veränderlichen Typen und Schließungen zu erstellen . Eine Liste mit langsamer Auswertung ist eine Liste, deren Elemente nicht beim Erstellen, sondern beim Zugriff ausgewertet werden. Die Vorteile von faul bewerteten Listen beinhalten die Möglichkeit, unendlich zu sein.

```
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()
```

Was tatsächlich so funktioniert wie in einer Sprache wie Haskell , in der alle Listen faul bewertet werden:

```
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
```

In der Praxis ist es besser, das Lazy.jl- Paket zu verwenden. Die Implementierung der Lazy-Liste oben beleuchtet jedoch wichtige Details zum Konstruieren des eigenen iterierbaren Typs.

Iterables online lesen: https://riptutorial.com/de/julia-lang/topic/5466/iterables

Kapitel 14: JSON

Syntax

- mit JSON
- JSON.parse (str)
- JSON.json (obj)
- JSON.print (io, obj, einzug)

Bemerkungen

Da weder Julia `Dict` noch JSON-Objekte inhärent geordnet sind, sollten Sie sich nicht auf die Reihenfolge der Schlüssel-Wert-Paare in einem JSON-Objekt verlassen.

Examples

JSON.jl installieren

JSON ist ein verbreitetes Datenaustauschformat. Die beliebteste JSON-Bibliothek für Julia ist JSON.jl. Um dieses Paket zu installieren, verwenden Sie den Paketmanager:

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

Im nächsten Schritt testen Sie, ob das Paket auf Ihrem Computer funktioniert:

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

Wenn alle Tests bestanden sind, ist die Bibliothek einsatzbereit.

JSON analysieren

JSON, das als Zeichenfolge codiert wurde, kann leicht in einen Standard-Julia-Typ analysiert werden:

```
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]
```

Es gibt einige unmittelbare Eigenschaften von JSON.jl:

- JSON-Typen werden in Julia sinnvollen Typen zugeordnet: Objekt wird zu `Dict` , Array wird zu `Vector` , Nummer wird zu `Int64` oder `Float64` , Boolean wird zu `Bool` und Null wird zu `nothing::Void` .
- JSON ist ein nicht typisiertes Containerformat: Die zurückgegebenen Julia-Vektoren sind vom Typ `Vector{Any}` und die zurückgegebenen Wörterbücher vom Typ `Dict{String, Any}` .
- Der JSON-Standard unterscheidet nicht zwischen Ganzzahlen und Dezimalzahlen, JSON.jl jedoch. Eine Zahl ohne Dezimalpunkt oder wissenschaftliche Notation wird in `Int64` analysiert, während eine Zahl mit Dezimalpunkt in `Float64` analysiert `Float64` . Dies stimmt eng mit dem Verhalten von JSON-Parsern in vielen anderen Sprachen überein.

Serialisierung von JSON

Die `JSON.json` Funktion serialisiert ein Julia-Objekt in einen Julia- `String` , der JSON enthält:

```
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}
```

Wenn keine Zeichenfolge gewünscht wird, kann JSON direkt in einen E / A-Stream gedruckt werden:

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

Beachten Sie, dass `STDOUT` die Standardeinstellung ist und bei dem obigen Aufruf weggelassen werden kann.

Ein besserer Druck kann durch Übergeben des optionalen `indent` werden:

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

Es gibt eine normale Standardserialisierung für komplexe Julia-Typen:

```
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
}
```

JSON online lesen: https://riptutorial.com/de/julia-lang/topic/5468/json

Kapitel 15: Kombinatoren

Bemerkungen

Obwohl Kombinatoren nur einen begrenzten praktischen Nutzen haben, sind sie ein nützliches Instrument in der Ausbildung, um zu verstehen, wie die Programmierung grundsätzlich mit der Logik verknüpft ist und wie sehr einfache Bausteine kombiniert werden können, um sehr komplexes Verhalten zu erzeugen. Im Zusammenhang mit Julia wird das Lernen, wie man Kombinatoren erstellt und verwendet, das Verständnis für das Programmieren in einem funktionalen Stil in Julia stärken.

Examples

Der Y- oder Z-Kombinator

Obwohl Julia keine rein funktionale Sprache ist, werden viele Eckpfeiler der funktionalen Programmierung vollständig unterstützt: erstklassige Funktionen , lexikalischer Umfang und Schließungen .

Der Festkomma-Kombinator ist ein Schlüsselkombinator für die Funktionsprogrammierung. Da Julia eine begierige Evaluierungssemantik hat (wie viele funktionale Sprachen, darunter auch Scheme, von dem Julia stark inspiriert ist), wird der ursprüngliche Y-Kombinator von Curry nicht sofort funktionieren:

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

Ein enger Verwandter des Y-Kombinators, der Z-Kombinator, wird jedoch tatsächlich funktionieren:

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

Dieser Kombinator akzeptiert eine Funktion und gibt eine Funktion zurück, die bei Aufruf mit Argument `x` selbst und `x` . Warum sollte es sinnvoll sein, eine Funktion selbst zu übergeben? Dies ermöglicht eine Rekursion, ohne den Namen der Funktion überhaupt zu referenzieren!

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

Daher wird `Z(fact)` eine rekursive Implementierung der Faktorfunktion, obwohl in dieser Funktionsdefinition keine Rekursion sichtbar ist. (Rekursion ist natürlich in der Definition des `Z` Kombinators offensichtlich, aber in einer eifrigen Sprache ist dies unvermeidlich.) Wir können überprüfen, ob unsere Funktion tatsächlich funktioniert:

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

Nicht nur das, aber es ist so schnell, wie wir es von einer rekursiven Implementierung erwarten können. Der LLVM-Code demonstriert, dass das Ergebnis in einen einfachen alten Zweig, Subtrahieren, Aufruf und Multiplizieren kompiliert wird:

```
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"
}
```

Das SKI Combinator System

Das SKI-Kombinatorsystem reicht aus, um alle Begriffe der Lambda-Berechnung darzustellen. (In der Praxis werden Lambda-Abstraktionen natürlich auf exponentielle Größe gebracht, wenn sie in SKI übersetzt werden.) Aufgrund der Einfachheit des Systems ist die Implementierung der Kombinatoren S, K und I außerordentlich einfach:

Eine direkte Übersetzung aus dem Lambda-Kalkül

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

Mit dem Unit-Testing- System können wir bestätigen, dass jeder Kombinator das erwartete Verhalten aufweist.

Der I-Kombinator ist am einfachsten zu überprüfen. Der angegebene Wert sollte unverändert zurückgegeben werden:

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

Der K-Kombinator ist auch recht einfach: er sollte sein zweites Argument verwerfen.

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

Der S-Kombinator ist der komplexeste; Sein Verhalten kann als Anwendung der ersten beiden Argumente auf das dritte Argument zusammengefasst werden, wobei das erste Ergebnis auf das zweite angewendet wird. Wir können den S-Kombinator am einfachsten testen, indem wir einige der aktuellen Formen testen. `S(K)` zum Beispiel sollte einfach das zweite Argument zurückgeben und das erste Argument verwerfen, wie wir sehen:

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

`S(I)(I)` sollte sein Argument auf sich selbst anwenden:

```
@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)))(K)` wendet sein zweites Argument auf das erste an:

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

Die I Kombinator oben beschrieben hat einen Namen in Standard - `Base` - Julia: `identity` . Daher könnten wir die obigen Definitionen mit der folgenden alternativen Definition von `I` umschreiben:

```
const I = identity
```

SKI-Kombinatoren anzeigen

Eine Schwachstelle bei dem oben beschriebenen Ansatz ist, dass unsere Funktionen nicht so gut angezeigt werden, wie wir es wünschen. Könnten wir ersetzen?

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

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

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

mit einigen informativen Anzeigen? Die Antwort ist ja! Lassen Sie uns die REPL neu starten, und definieren Sie diesmal, wie die einzelnen Funktionen angezeigt werden sollen:

```
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
```

Es ist wichtig zu vermeiden, dass etwas angezeigt wird, bis die Definition der Funktionen abgeschlossen ist. Andernfalls besteht die Gefahr, dass der Methodencache ungültig wird, und unsere neuen Methoden scheinen nicht sofort wirksam zu sein. Deshalb haben wir in den obigen Definitionen Semikola eingefügt. Die Semikolons unterdrücken die Ausgabe der REPL.

Dadurch werden die Funktionen gut angezeigt:

```
julia> S
S

julia> K
K

julia> I
I
```

Es gibt jedoch immer noch Probleme, wenn wir versuchen, eine Schließung anzuzeigen:

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

Es wäre schöner, das als `S(K)` anzuzeigen. Um dies zu erreichen, müssen wir ausnutzen, dass die Verschlüsse ihre eigenen individuellen Typen haben. Wir können diese Arten zugreifen und fügen Sie Methoden , um sie durch Reflexion, mit `typeof` und das `primary` Feld des `name` Feld des Typs. Starten Sie die REPL erneut. Wir werden weitere Änderungen vornehmen:

```
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)
```

Und jetzt endlich zeigen sich die Dinge so, wie wir es gerne hätten:

```
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
```

Kombinatoren online lesen: https://riptutorial.com/de/julia-lang/topic/5758/kombinatoren

Kapitel 16: Lesen eines DataFrame aus einer Datei

Examples

Lesen eines Datenrahmens aus durch Trennzeichen getrennten Daten

Sie können einen `DataFrame` aus einer CSV-Datei (durch `DataFrame` getrennte Werte) oder sogar aus einem TSV oder WSV (durch Tabulatoren und Leerzeichen getrennte Dateien) lesen. Wenn Ihre Datei die richtige Erweiterung hat, können Sie die `readtable` Funktion verwenden, um den Datenrahmen einzulesen:

```
readtable("dataset.CSV")
```

Was ist, wenn Ihre Datei nicht die richtige Erweiterung hat? Sie können das von Ihrer Datei verwendete Trennzeichen (Komma, Tabulator, Leerzeichen usw.) als Schlüsselwortargument für die Funktion `readtable` :

```
readtable("dataset.txt", separator=',')
```

Umgang mit verschiedenen Kommentarkennzeichen

Datensätze enthalten häufig Kommentare, die das Datenformat erläutern oder die Lizenz- und Nutzungsbedingungen enthalten. Normalerweise möchten Sie diese Zeilen ignorieren, wenn Sie den `DataFrame` .

Die Funktion `readtable` geht davon aus, dass Kommentarzeilen mit dem Zeichen '#' beginnen. Ihre Datei kann jedoch Kommentarzeichen wie `%` oder `//` . Um sicherzustellen, dass `readtable` diese korrekt verarbeitet, können Sie die Kommentarmarke als Schlüsselwortargument angeben:

```
readtable("dataset.csv", allowcomments=true, commentmark='%')
```

Lesen eines DataFrame aus einer Datei online lesen: https://riptutorial.com/de/julia-lang/topic/7340/lesen-eines-dataframe-aus-einer-datei

Kapitel 17: Metaprogrammierung

Syntax

- Makroname (ex) ... Ende
- Zitat ... Ende
- : (...)
- $ x
- Meta.quot (x)
- QuoteNode (x)
- esc (x)

Bemerkungen

Julias Metaprogrammierungsfunktionen sind stark von denen von Lisp-ähnlichen Sprachen inspiriert und werden denen mit Lisp-Hintergrund bekannt vorkommen. Metaprogrammierung ist sehr mächtig. Bei korrekter Verwendung kann es zu prägnanterem und lesbarerem Code führen.

Das `quote ... end` ist quasiquote Syntax. Anstelle der Ausdrücke, die ausgewertet werden, werden sie einfach analysiert. Der Wert des `quote ... end` ist der resultierende Abstract Syntax Tree (AST).

Die Syntax `:(...)` ähnelt der Syntax `quote ... end` , ist jedoch leichter. Diese Syntax ist knapper als `quote ... end` .

In einer Quasiquote ist der Operator `$` Besonderes und *interpoliert* sein Argument in die AST. Es wird erwartet, dass das Argument ein Ausdruck ist, der direkt mit dem AST verbunden ist.

Die `Meta.quot(x)` -Funktion zitiert ihr Argument. Dies ist häufig in Kombination mit `$` für die Interpolation hilfreich, da Ausdrücke und Symbole buchstäblich mit dem AST verbunden werden können.

Examples

Das @show-Makro erneut implementieren

In Julia ist das `@show` Makro häufig für Debugging-Zwecke hilfreich. Es zeigt sowohl den auszuwertenden Ausdruck als auch das Ergebnis an und gibt schließlich den Wert des Ergebnisses zurück:

```
julia> @show 1 + 1
1 + 1 = 2
2
```

Es ist einfach, eine eigene Version von `@show` zu erstellen:

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

Um die neue Version zu verwenden, verwenden Sie einfach das `@myshow` Makro:

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

julia> x
2
```

Bis zur Schleife

Wir sind alle an die `while` Syntax gewöhnt, die ihren Körper ausführt, während die Bedingung als `true` bewertet wird. Was passiert , wenn wir wollen , eine implementieren , `until` Schleife führt , dass eine Schleife , bis die Bedingung ausgewertet wird `true` ?

In Julia können Sie dies tun, indem Sie ein `@until` Makro `@until` , das seinen Körper stoppt, wenn die Bedingung erfüllt ist:

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

Hier haben wir die Funktionsverkettungssyntax `|>` , die dem Aufruf der `esc` Funktion für den gesamten `quote` . Die `esc` Funktion verhindert, dass Makrohygiene auf den Inhalt des Makros angewendet wird. Andernfalls werden Variablen im Makro umbenannt, um Kollisionen mit externen Variablen zu vermeiden. Weitere Informationen finden Sie in der Julia-Dokumentation zur Makrohygiene .

Sie können mehrere Ausdrücke in dieser Schleife verwenden, indem Sie einfach alles in einen `begin ... end` einfügen:

```
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 und Expr (: Quote)

Es gibt drei Möglichkeiten, etwas mit einer Julia-Funktion zu zitieren:

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

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

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

Was bedeutet "Zitieren" und wofür ist es gut? Durch Zitieren können wir Ausdrücke davor schützen, von Julia als Sonderformen interpretiert zu werden. Ein häufiger Anwendungsfall ist das Generieren von Ausdrücken , die Elemente enthalten sollten, die Symbole auswerten. (Dieses Makro muss beispielsweise einen Ausdruck zurückgeben, der ein Symbol auswertet.) Es funktioniert nicht einfach, um das Symbol zurückzugeben:

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

julia> @mysym
ERROR: UndefVarError: x not defined

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

Was ist denn hier los? `@mysym` erweitert sich zu `:x` , das als Ausdruck als Variable `x` interpretiert wird. `x` noch nichts zugewiesen, so dass wir einen `x not defined` Fehler erhalten.

Um dies zu umgehen, müssen wir das Ergebnis unseres Makros zitieren:

```
julia> macro mysym2(); Meta.quot(:x); end
@mysym2 (macro with 1 method)

julia> @mysym2
:x

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

Hier haben wir die `Meta.quot` Funktion verwendet, um unser Symbol in ein in Anführungszeichen

gesetztes Symbol zu verwandeln. Dies ist das gewünschte Ergebnis.

Was ist der Unterschied zwischen `Meta.quot` und `QuoteNode` und welchen sollte ich verwenden? In fast allen Fällen spielt der Unterschied keine Rolle. Es ist vielleicht etwas sicherer, `QuoteNode` anstelle von `Meta.quot` . Die Untersuchung des Unterschieds ist jedoch aufschlussreich, wie Julia-Ausdrücke und -Makros funktionieren.

Der Unterschied zwischen `Meta.quot` und `QuoteNode` wird erläutert

Hier ist eine Faustregel:

- Wenn Sie die Interpolation benötigen oder unterstützen möchten, verwenden Sie `Meta.quot` .
- Wenn Sie die Interpolation nicht zulassen können oder wollen, verwenden Sie `QuoteNode` .

Kurz gesagt, der Unterschied besteht darin, dass `Meta.quot` die Interpolation innerhalb der zitierten Sache erlaubt, während `QuoteNode` sein Argument vor jeder Interpolation schützt. Um die Interpolation zu verstehen, muss der Ausdruck `$` . Es gibt eine Art Ausdruck in Julia, den Ausdruck `$` . Diese Ausdrücke ermöglichen die Flucht. Betrachten Sie zum Beispiel den folgenden Ausdruck:

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

Bei der Auswertung wertet dieser Ausdruck `1` ordnet es `x` und erstellt dann einen Ausdruck der Form `_ + _` wobei _ durch den Wert von `x` . Daher sollte das Ergebnis der *Ausdruck* `1 + 1` (der noch nicht ausgewertet wurde und sich daher vom *Wert* `2`). In der Tat ist dies der Fall:

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

Nehmen wir an, wir schreiben ein Makro, um diese Art von Ausdrücken zu erstellen. Unser Makro wird ein Argument annehmen, das die `1` im obigen `ex` ersetzen wird. Dieses Argument kann natürlich jeder Ausdruck sein. Hier ist etwas, was wir nicht wollen:

```
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
```

Der zweite Fall ist falsch, weil wir `1 + 1` ausgewertet lassen sollten. Wir beheben das, indem wir das Argument mit `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
```

Die Makrohygiene gilt nicht für den Inhalt eines Angebots. In diesem Fall ist eine Flucht nicht erforderlich (und in der Tat nicht legal).

Wie bereits erwähnt, ermöglicht `Meta.quot` die Interpolation. Also lassen Sie uns das ausprobieren:

```
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
```

Im ersten Beispiel sehen wir, dass die Interpolation es uns ermöglicht, die `sin(1)` zu inline zu setzen, anstatt dass der Ausdruck eine wörtliche `sin(1)` . Das zweite Beispiel zeigt, dass diese Interpolation im Bereich des Makroaufrufs erfolgt, nicht im eigenen Bereich des Makros. Das liegt daran, dass unser Makro keinen Code ausgewertet hat. Alles, was es tut, ist Code zu generieren. Die Auswertung des Codes (der in den Ausdruck gelangt) wird ausgeführt, wenn der vom Makro generierte Ausdruck tatsächlich ausgeführt wird.

Was wäre, wenn wir stattdessen `QuoteNode` verwendet `QuoteNode` ? Wie Sie sich `QuoteNode` können, da `QuoteNode` die Interpolation überhaupt nicht `QuoteNode` , bedeutet dies, dass es nicht funktioniert.

```
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
```

In diesem Beispiel könnten wir zustimmen, dass `Meta.quot` mehr Flexibilität bietet, da es die Interpolation ermöglicht. Warum können wir `QuoteNode` überhaupt in Betracht `QuoteNode` ? In einigen Fällen wünschen wir uns eigentlich keine Interpolation und den buchstäblichen `$` -Ausdruck. Wann wäre das wünschenswert? Lassen Sie sich eine Verallgemeinerung betrachten `@makeex` wo wir zusätzliche Argumente bestimmen , was kommt nach links und rechts neben dem passieren können `+` Zeichen:

```
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)
```

Eine Einschränkung unserer Implementierung von `@makeex4` besteht darin, dass wir Ausdrücke nicht direkt als linke oder rechte Seite des Ausdrucks verwenden können, da sie interpoliert werden. Mit anderen Worten, die Ausdrücke werden möglicherweise für die Interpolation ausgewertet, wir möchten jedoch, dass sie beibehalten werden. (Da es hier viele Ebenen des Zitierens und der Bewertung gibt, lassen Sie uns klarstellen: Unser Makro generiert *Code* , der einen *Ausdruck erstellt* , der bei Auswertung einen anderen *Ausdruck* erzeugt. Puh!)

```
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)
```

Wir sollten dem Benutzer erlauben zu bestimmen, wann Interpolation stattfinden soll und wann nicht. Theoretisch ist das eine einfache Lösung: Wir können einfach eines der `$` -Zeichen in unserer Anwendung entfernen und den Benutzer selbst einbringen. Dies bedeutet, dass wir eine zitierte Version des vom Benutzer eingegebenen Ausdrucks interpolieren (die wir bereits einmal zitiert und interpoliert haben). Dies führt zu dem folgenden Code, der aufgrund der verschachtelten Ebenen des Zitierens und des Nicht-Zitierens zunächst etwas verwirrend sein kann. Versuche zu lesen und zu verstehen, wozu jede Flucht dient.

```
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
```

Die Dinge haben gut angefangen, aber etwas ist schief gelaufen. Der generierte Code des Makros versucht, die Kopie von `y` im Makroaufrufbereich zu interpolieren. Im Makroaufrufbereich befindet sich jedoch *keine* Kopie von `y` . Unser Fehler ist die Interpolation mit den zweiten und dritten Argumenten im Makro. Um diesen Fehler zu beheben, müssen wir `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)
```

Durch die Verwendung von `QuoteNode` haben wir unsere Argumente vor Interpolation geschützt. Da `QuoteNode` nur als zusätzlicher Schutz wirkt, ist die Verwendung von `QuoteNode` niemals schädlich, es sei denn, Sie `QuoteNode` Interpolation. `Meta.quot` Sie den Unterschied verstehen, können Sie jedoch verstehen, wo und warum `Meta.quot` die bessere Wahl sein könnte.

Diese lange Übung ist an einem Beispiel, das offensichtlich zu komplex ist, um in einer vernünftigen Anwendung gezeigt zu werden. Daher haben wir die bereits erwähnte folgende Faustregel gerechtfertigt:

- Wenn Sie die Interpolation benötigen oder unterstützen möchten, verwenden Sie `Meta.quot` .
- Wenn Sie die Interpolation nicht zulassen können oder wollen, verwenden Sie `QuoteNode` .

Was ist mit Expr (: Zitat)?

`Expr(:quote, x)` entspricht `Meta.quot(x)` . Letzteres ist jedoch eher idiomatisch und wird bevorzugt. Für Code, den metaprogramming, eine stark nutzt `using Base.Meta` Linie wird oft verwendet, die ermöglicht `Meta.quot` wird bezeichnet als einfach `quot` .

Führen

π's Metaprogrammier-Bits und -Bobs

Tore:

- Lehren Sie durch minimale gezielte funktionale / nützliche / nicht abstrakte Beispiele (z. B. `@swap` oder `@assert`), die Konzepte in geeigneten Kontexten einführen
- Lassen Sie den Code lieber die Konzepte als Erklärungsabschnitte veranschaulichen / veranschaulichen
- Vermeiden Sie das Verlinken des "erforderlichen Lesens" auf andere Seiten, da dies die Erzählung unterbricht
- Präsentieren Sie die Dinge in einer sinnvollen Reihenfolge, die das Lernen am einfachsten macht

Ressourcen:

julialang.org
wikibook (@Cormullion)
5 Schichten (Leah Hanson)
SO-Doc-Notierung (@TotalVerb)
SO-Doc - Symbole, die keine zulässigen Bezeichner sind (@TotalVerb)
SO: Was ist ein Symbol in Julia (@StefanKarpinski)
Diskursthread (@ pi-) Metaprogrammierung

Das meiste Material stammt aus dem Diskurskanal, das meiste stammt von fcard ... bitte stoße mich, wenn ich Attributionen vergessen hätte.

Symbol

```
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
```

Flaggen an Funktionen übergeben:

```
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
```

Ein Hashschlüssel-Beispiel:

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

(Fortgeschritten) (@fcard) `:foo` aka `:(foo)` ergibt ein Symbol, wenn `foo` ein gültiger Bezeichner ist, andernfalls ein Ausdruck.

```
# 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
...
```

Grundsätzlich können Sie Symbole als leichte Zeichenfolgen behandeln. Dafür gibt es sie nicht, aber Sie können es tun, also warum nicht. Julias Base selbst macht es, `print_with_color(:red, "abc")` druckt ein rotes abc.

Ausdruck (AST)

(Fast) alles in Julia ist ein Ausdruck, dh eine Instanz von `Expr` , die einen AST enthalten wird .

```
# 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
```

Nesting `Expr` s:

```
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))
```

mehrzeilige `Expr` mit `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
```

... also `quote` ist funktionell das gleiche, bietet aber zusätzliche Debug-Informationen.

(*) **TIPP** : Verwenden Sie `let` , um `x` innerhalb des Blocks zu halten

`quote` -ing ein `quote`

`Expr(:quote, x)` werden verwendet, um Anführungszeichen innerhalb von Anführungszeichen darzustellen.

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

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

`QuoteNode(x)` ähnelt `Expr(:quote, x)` , verhindert jedoch Interpolation.

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

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

(Erläutern Sie die verschiedenen Zitiermechanismen in der Julia-Metaprogrammierung

Sind $ und : (...) sich irgendwie gegenseitig umkehren?

`:(foo)` bedeutet "nicht auf den Wert schauen, sondern auf den Ausdruck" `$foo` bedeutet "den Ausdruck in seinen Wert ändern"

`:($(foo)) == foo` . `$(:(foo))` ist ein Fehler. `$(...)` ist keine Operation und macht nichts von sich aus, es ist ein "interpoliere das!" kennzeichnen, dass die Anführungszeichen-Syntax verwendet.

dh es existiert nur innerhalb eines Zitats.

Ist `$ foo` dasselbe wie `eval( foo )` ?

Nein! `$foo` wird gegen die Compile-Time- `eval(foo)` - `eval(foo)` ausgetauscht, um dies zur Laufzeit zu tun

`eval` wird im globalen Gültigkeitsbereich auftreten. Interpolation ist lokal

`eval(:<expr>)` sollte das gleiche wie nur `<expr>` (vorausgesetzt, `<expr>` ist ein gültiger Ausdruck im aktuellen globalen Bereich).

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

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

`macro` S

Bereit? :)

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

Lassen Sie uns unser eigenes `@show` Makro `@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` , um einen `Expr`

5 Schichten (Leah Hanson) <- erklärt, wie Julia Quellcode als Zeichenfolge verwendet, in einen `Expr` Baum (AST) einkennzeichnet, alle Makros (immer noch AST) ausdehnt, ***senkt*** (***verringert*** AST) und konvertiert dann in LLVM (und darüber hinaus - im Moment brauchen wir uns keine Sorgen zu machen, was dahinter liegt!)

F: `code_lowered` wirkt auf Funktionen. Ist es möglich, einen `Expr` zu senken? A: yup!

```
# 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)
```

Wenn Sie nur Makros erweitern möchten, können Sie **`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
```

... das eine nicht abgesenkte AST zurückgibt, aber alle Makros erweitert sind.

`esc()`

`esc(x)` gibt einen Ausdruck zurück, der besagt "Keine Hygiene anwenden", es ist dasselbe wie `Expr(:escape, x)` . Hygiene ist das, was ein Makro in sich hält, und Sie müssen Dinge `esc` wenn Sie möchten, dass sie "auslaufen". z.B

Beispiel: `swap` Makro zur Veranschaulichung von `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
```

`$` können wir dem `quote` entkommen. Warum also nicht einfach `$p` und `$q` ? dh

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

Da dies zunächst nach dem `macro` Gültigkeitsbereich für `p` suchen würde, würde es ein lokales `p` dh den Parameter `p` (ja, wenn Sie anschließend ohne `esc` -ing auf `p` zugreifen, betrachtet das Makro den `p` Parameter als lokale Variable).

`$p = ...` ist also nur eine Zuordnung zum lokalen `p` . Es wirkt sich nicht auf die Variable aus, die im aufrufenden Kontext übergeben wurde.

Ok, wie wäre es mit:

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

So `esc(p)` ist 'undichte' `p` in den anrufenden Kontext. *"Die Sache, die in das Makro übergeben wurde, das **wir als `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
```

Wie Sie sehen, erhält `tmp` die Hygienebehandlung `#10#tmp` , während `x` und `y` dies nicht tun. Julia macht einen eindeutigen Bezeichner für `tmp` , den Sie manuell mit `gensym` tun `gensym` , z. `gensym` .:

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

Aber: **Es gibt ein Problem:**

```
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
```

Eine weitere Sache, die Julias Makrohygiene bewirkt, ist, dass das Makro, wenn es aus einem anderen Modul stammt, alle Variablen (die nicht im zurückgegebenen Ausdruck des Makros, wie in diesem Fall `tmp` zugewiesen wurden) zu globalen Werten des aktuellen Moduls macht, sodass `$p` zu `Swap.$p` , ebenfalls `$q` -> `Swap.$q` .

Wenn Sie eine Variable außerhalb des Gültigkeitsbereichs des Makros benötigen, sollten Sie esc esc machen. `esc(p)` sollten Sie `esc(p)` und `esc(q)` unabhängig davon, ob sie sich auf der LHS oder der RHS eines Ausdrucks befinden oder sogar für sich alleine.

Die Leute haben `gensym` einige Male erwähnt, und schon bald werden Sie von der dunklen Seite des `gensym` verführt, dem ganzen Ausdruck mit ein paar `gensym` s hier und da zu entgehen, aber ... Machen Sie sich mit Hygiene `gensym` , bevor Sie versuchen zu sein schlauer als es! Es ist kein besonders komplexer Algorithmus, also sollte es nicht zu lange dauern, aber übertreiben Sie es nicht! Nutze diese Macht erst, wenn du alle Auswirkungen davon verstanden hast ... (@fcard)

Beispiel: `until` Makro

(@ Ismael-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) `|>` ist jedoch umstritten. Ich bin überrascht, dass sich ein Mob noch nicht zur Diskussion

gestellt hat. (Vielleicht ist jeder einfach müde). Es wird empfohlen, dass die meisten, wenn nicht alle, nur einen Aufruf einer Funktion haben, also:

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

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

... ist eine sicherere Alternative.

@ fcard's einfache Makro-Herausforderung

Aufgabe: Tauschen Sie die Operanden aus, also ergibt `swaps(1/2) 2.00` dh `2/1`

```
macro swaps(e)
    e.args[2:3] = e.args[3:-1:2]
    e
end
@swaps(1/2)
2.00
```

Weitere Makro-Herausforderungen von @fcard hier

Interpolations- und `assert` Makro

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

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

Q: warum der letzte `$` ? A: Es interpoliert, dh zwingt Julia, diese `string(ex) eval` , wenn die Ausführung den Aufruf dieses Makros durchläuft. Wenn Sie diesen Code nur ausführen, wird keine Bewertung erzwungen. Aber in dem Moment, in dem Sie `assert(foo)` Julia dieses Makro ***aufrufen*** und seinen 'AST-Token / Ausdruck' durch das ersetzen, was es zurückgibt, und `$` *wird* in Aktion treten.

Ein lustiger Hack für die Verwendung von {} für Blöcke

(@fcard) Ich glaube nicht, dass es technische Gründe gibt, `{}` die als Blöcke verwendet werden. Tatsächlich kann man sogar die restliche `{}` -Syntax unterdrücken, damit es funktioniert:

```
julia> macro c(block)
```

```
          @assert block.head == :cell1d
          esc(quote
            $(block.args...)
          end)
        end
@c (macro with 1 method)

julia> @c {
          print(1)
          print(2)
          1+2
        }
123
```

* (funktioniert wahrscheinlich nicht, wenn / wenn die {} -Syntax neu verwendet wird)

Also sieht Julia zuerst das Makro-Token, liest / parst Token bis zum passenden `end` und erstellt was? Ein `Expr` mit `.head=:macro` oder etwas? Speichert es `"a+1"` als Zeichenfolge oder zerlegt es in `:+(:a, 1)` ? Wie sehen Sie?

?

(@fcard) In diesem Fall ist a wegen des lexikalischen Gültigkeitsbereichs im `@M` undefiniert, daher wird die globale Variable verwendet. Ich habe tatsächlich vergessen, den flipplin'-Ausdruck in meinem blöden Beispiel zu ignorieren, aber das *"funktioniert nur innerhalb von Dasselbe Modul "* Teil davon gilt noch.

```
julia> module M
       macro m()
         :(a+1)
       end
       end
M

julia> a = 1
1

julia> M.@m
ERROR: UndefVarError: a not defined
```

Der Grund ist, dass, wenn das Makro in einem anderen Modul als dem verwendet wird, in dem es definiert wurde, alle Variablen, die nicht im zu erweiternden Code definiert sind, als Globals des Moduls des Makros behandelt werden.

```
julia> macroexpand(:(M.@m))
:(M.a + 1)
```

ERWEITERT

@ Ismael-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 until i == 5
i = 0
i = 1
i = 2
i = 3
i = 4
```

Scott's Makro:

```
"""
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
```

Junk / unverarbeitet ...

ein Makro anzeigen / ausgeben

(@ pi-) Angenommen, ich mache einfach `macro m(); a+1; end` in einer frischen REPL. Ohne `a` definiertes. Wie kann ich es "sehen"? Gibt es eine Möglichkeit, ein Makro zu "entleeren"? Ohne es tatsächlich auszuführen

(@fcard) Der gesamte Code in Makros wird tatsächlich in Funktionen eingefügt, sodass Sie nur den herabgesetzten oder vom Typ abgeleiteten Code anzeigen können.

```
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!
```

Andere Möglichkeiten, eine Makrofunktion zu erhalten:

```
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)
```

^ `code_typed` kann `code_typed` stattdessen `code_typed` verwenden

Wie kann man `eval(Symbol("@M"))` verstehen?

(@fcard) Derzeit ist jedem Makro eine Funktion zugeordnet. Wenn Sie ein Makro mit dem Namen `M` , heißt die Funktion des Makros `@M` . Im Allgemeinen können Sie einen Funktionswert mit zB `eval(:print)` aber mit einer Makrofunktion müssen Sie `Symbol("@M") :@M` , da aus `:@M` ein `Expr(:macrocall, Symbol("@M"))` und Auswertung, die eine Makro-Erweiterung bewirkt.

Warum zeigt `code_typed` keine `code_typed` an?

(@Pi-)

```
julia> code_typed( x -> x^2 )[1]
LambdaInfo for (::##5#6)(::Any)
:(begin
        return x ^ 2
    end)
```

^ hier sehe ich einen `::Any` Parameter, aber er scheint nicht mit dem Token `x` .

```
 julia> code_typed( print )[1]
LambdaInfo for print(::IO, ::Char)
:(begin
        (Base.write)(io,c)
        return Base.nothing
    end::Void)
```

^ hier ähnlich; Es gibt nichts, was `io` mit dem `::IO` verbinden kann. Dies kann jedoch kein vollständiger Speicherauszug der AST-Darstellung dieser speziellen `print` …?

(@fcard) `print(::IO, ::Char)` sagt Ihnen nur, um welche Methode es sich handelt, sie ist nicht Teil des AST. Es ist nicht einmal mehr in master vorhanden:

```
julia> code_typed(print)[1]
CodeInfo(:(begin
        (Base.write)(io,c)
        return Base.nothing
    end))=>Void
```

(@ pi-) Ich verstehe nicht, was du damit meinst. Es scheint, die AST für den Körper dieser Methode zu entleeren, nein? Ich dachte, `code_typed` gibt den AST für eine Funktion an. Der erste Schritt scheint jedoch zu fehlen, dh das Einrichten von Token für Params.

(@fcard) `code_typed` soll nur die AST des Körpers anzeigen, aber `code_typed` gibt es den vollständigen AST der Methode in Form einer `LambdaInfo` (0.5) oder `CodeInfo` (0.6) an, aber viele Informationen werden weggelassen wenn auf der Replik gedruckt. Sie müssen das `LambdaInfo` Feld nach Feld untersuchen, um alle Details zu erhalten. `dump` wird Ihre Antwort überfluten, so dass Sie versuchen könnten:

```
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])
```

Gibt alle Werte der benannten Felder der AST einer Methode an:

```
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)
```

Sehen Sie das lil ' `def = print(io::IO, c::Char)` ? Da gehts! (auch die `slotnames = [..., :io, :c]` part) Auch ja, der Unterschied in der Ausgabe liegt darin, dass ich die Ergebnisse auf Master `slotnames = [..., :io, :c]` .

???

(@ Ismael-VC) meinst du so? Generischer Versand mit Symbolen

Sie können es so machen:

```
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)
```

Dies ist für den Euler-Algorithmus! Ich frage mich, was @fcard über den generischen Symbolversand denkt! --- ^: Engel:

Modul Gotcha

```
@def m begin
  a+2
end

@m # replaces the macro at compile-time with the expression a+2
```

Genauer, funktioniert nur innerhalb der obersten Ebene des Moduls, in dem das Makro definiert wurde.

```
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` dies geschieht, aber wenn es standardmäßig verwendet wird, widerspricht es dem Sprachdesign. Eine gute Verteidigung dafür ist, dass man keine Namen in Makros verwenden und einführen kann, was es schwierig macht, sie einem menschlichen Leser zu finden.

Python `dict` / JSON-ähnliche Syntax für` Dict`-Literale.

Einführung

Julia verwendet die folgende Syntax für Wörterbücher:

```
Dict({k₁ => v₁, k₂ => v₂, …, k□₋₁ => v□₋₁, k□ => v□)
```

Während Python und JSON so aussehen:

```
{k₁: v₁, k₂: v₂, …, k□₋₁: v□₋₁, k□: v□}
```

Zu **Illustrationszwecken könnten** wir diese Syntax auch in Julia verwenden und neue Semantik hinzufügen (`Dict` Syntax ist die idiomatische Art in Julia, die empfohlen wird).

Lassen Sie uns zunächst sehen , welche *Art* von Ausdruck ist:

```
julia> parse("{1:2 , 3: 4}") |> Meta.show_sexpr
(:cell1d, (:(:), 1, 2), (:(:), 3, 4))
```

Das bedeutet, dass wir `:cell1d` nehmen müssen `:cell1d` Ausdruck und entweder transformieren oder einen neuen Ausdruck zurückgeben, der folgendermaßen aussehen sollte:

```
julia> parse("Dict(1 => 2 , 3 => 4)") |> Meta.show_sexpr
(:call, :Dict, (:(=>), 1, 2), (:(=>), 3, 4))
```

Makrodefinition

Mit dem folgenden Makro können Sie zwar eine solche Codegenerierung und -transformation demonstrieren:

```
macro dict(expr)
    # Check the expression has the correct form:
    if expr.head ≠ :cell1d || any(sub_expr.head ≠ :(:) for sub_expr ∈ expr.args)
        error("syntax: expected `{k₁: v₁, k₂: v₂, …, k□₋₁: v□₋₁, k□: v□}`")
    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, …, k□₋₁ => v□₋₁, k□ => v□))

    # Block is escaped so it can reach variables from it's calling scope:
    return esc(block)
end
```

Schauen wir uns die resultierende Makro-Erweiterung an:

```
julia> :(@dict {"a": :b, 'c': 1, :d: 2.0}) |> macroexpand
:(Dict("a" => :b,'c' => 1,:d => 2.0))
```

Verwendungszweck

```
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
```

Das letzte Beispiel ist genau gleichbedeutend mit:

```
Dict(
    "string" => :b,
    'c'      => 1,
    :symbol  => π,
    Function => print,
    (1:10)   => range(1, 10)
)
```

Missbrauch

```
julia> @dict {"one": 1, "two": 2, "three": 3, "four": 4, "five" => 5}
syntax: expected `{k₁: v₁, k₂: v₂, …, k□₋₁: v□₋₁, k□: v□}`

julia> @dict ["one": 1, "two": 2, "three": 3, "four": 4, "five" => 5]
syntax: expected `{k₁: v₁, k₂: v₂, …, k□₋₁: v□₋₁, k□: v□}`
```

Beachten Sie, dass Julia andere Verwendungsmöglichkeiten für Doppelpunkte hat `:` Als solche müssen Sie Bereichsliteralausdrücke mit Klammern umschließen oder beispielsweise die `range` verwenden.

Metaprogrammierung online lesen: https://riptutorial.com/de/julia-lang/topic/1945/metaprogrammierung

Kapitel 18: Module

Syntax

- Modul Modul; ... Ende
- unter Verwendung des Moduls
- Modul importieren

Examples

Code in ein Modul einschließen

Das `module` kann verwendet werden, um ein Modul zu beginnen, mit dem Code organisiert und Namespaces erstellt werden kann. Module können eine externe Schnittstelle definieren, die typischerweise aus `export` Symbolen besteht. Um diese externe Schnittstelle zu unterstützen, können Module nicht exportierte interne Funktionen und Typen aufweisen, die nicht für die öffentliche Verwendung bestimmt sind.

Einige Module existieren hauptsächlich, um einen Typ und zugehörige Funktionen zu umschließen. Solche Module werden normalerweise durch die Pluralform des Namens des Typs benannt. Wenn wir beispielsweise ein Modul haben, das einen `Building` bereitstellt, können wir ein solches Modul `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
```

Das Modul kann dann mit der `using` Anweisung verwendet werden:

```
julia> using Buildings

julia> Building("Burj Khalifa", 163, 830)
163-story Burj Khalifa with height 830m
```

```
julia> height(ans)
830
```

Verwenden von Modulen zum Verwalten von Paketen

In der Regel bestehen Pakete aus einem oder mehreren Modulen. Wenn Pakete größer werden, kann es sinnvoll sein, das Hauptmodul des Pakets in kleinere Module zu organisieren. Eine gängige Redewendung ist das Definieren dieser Module als Submodule des Hauptmoduls:

```
module RootModule

module SubModule1

...

end

module SubModule2

...

end

end
```

Anfangs haben weder das Wurzelmodul noch die Submodule Zugriff auf die exportierten Symbole der jeweils anderen. Es werden jedoch relative Importe unterstützt, um dieses Problem zu beheben:

```
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
```

In diesem Beispiel beträgt der Wert von `RootModule.z` `30` .

Module online lesen: https://riptutorial.com/de/julia-lang/topic/7368/module

Kapitel 19: Pakete

Syntax

- Pkg.add (Paket)
- Pkg.checkout (package, branch = "master")
- Pkg.clone (URL)
- Pkg.dir (Paket)
- Pkg.pin (Paket, Version)
- Pkg.rm (Paket)

Parameter

Parameter	Einzelheiten
`Pkg.add(` *`package`* `)`	Laden Sie das angegebene registrierte Paket herunter und installieren Sie es.
`Pkg.checkout(` *`package`* `,` *`branch`* `)`	Schauen Sie sich die angegebene Niederlassung für das angegebene registrierte Paket an. *`branch`* ist optional und standardmäßig auf `"master"`.
`Pkg.clone(` *`url`* `)`	Klonen Sie das Git-Repository unter der angegebenen URL als Paket.
`Pkg.dir(` *`package`* `)`	Rufen Sie den Speicherort für das angegebene Paket auf der Festplatte ab.
`Pkg.pin(` *`package`* `,` *`version`* `)`	Erzwinge, dass das Paket bei der angegebenen Version bleibt. *`version`* ist optional und wird standardmäßig auf die aktuelle Version des Pakets gesetzt.
`Pkg.rm(` *`package`* `)`	Entfernen Sie das angegebene Paket aus der Liste der erforderlichen Pakete.

Examples

Installieren, verwenden und entfernen Sie ein registriertes Paket

Nachdem Sie ein offizielles Julia-Paket gefunden haben, können Sie das Paket problemlos herunterladen und installieren. Zunächst wird empfohlen, die lokale Kopie von METADATA zu aktualisieren:

```
julia> Pkg.update()
```

Dadurch wird sichergestellt, dass Sie die neuesten Versionen aller Pakete erhalten.

Nehmen wir an, das zu installierende Paket heißt `Currencies.jl` . Der Befehl zum Installieren dieses Pakets lautet:

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

Dieser Befehl installiert nicht nur das Paket selbst, sondern auch alle Abhängigkeiten.

Wenn die Installation erfolgreich ist, können Sie testen, ob das Paket ordnungsgemäß funktioniert :

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

Verwenden Sie dann, um das Paket zu verwenden

```
julia> using Currencies
```

und fahren Sie wie in der Dokumentation des Pakets beschrieben fort, die normalerweise mit ihrer README.md-Datei verknüpft oder darin enthalten ist.

Verwenden `Pkg.rm` zum Deinstallieren eines Pakets, das nicht mehr benötigt wird, die Funktion `Pkg.rm` :

```
julia> Pkg.rm("Currencies")
```

Beachten Sie, dass das Paketverzeichnis dadurch möglicherweise nicht wirklich entfernt wird. Stattdessen wird das Paket lediglich als nicht mehr benötigt markiert. Dies ist oft vollkommen in Ordnung - es spart Zeit, falls Sie das Paket in Zukunft erneut benötigen. Wenn Sie das Paket jedoch physisch entfernen `Pkg.resolve` , rufen Sie die Funktion `rm` und anschließend `Pkg.resolve` :

```
julia> rm(Pkg.dir("Currencies"); recursive=true)

julia> Pkg.resolve()
```

Überprüfen Sie einen anderen Zweig oder eine andere Version

Manchmal ist die neueste getaggte Version eines Pakets fehlerhaft oder es fehlen einige erforderliche Funktionen. Fortgeschrittene Benutzer können auf die neueste Entwicklung Version eines Pakets aktualisieren mögen (manchmal als „Master“ bezeichnet, benannt nach dem üblichen Namen für einen Entwicklungszweig in Git). Die Vorteile davon sind:

- Entwickler, die zu einem Paket beitragen, sollten zur neuesten Entwicklungsversion beitragen.
- Die neueste Entwicklungsversion kann nützliche Funktionen, Bugfixes oder Leistungsverbesserungen enthalten.
- Benutzer, die einen Fehler melden, möchten möglicherweise prüfen, ob in der neuesten Entwicklungsversion ein Fehler auftritt.

Die Ausführung der neuesten Entwicklungsversion hat jedoch viele Nachteile:

- Die neueste Entwicklungsversion ist möglicherweise schlecht getestet und weist ernsthafte Fehler auf.
- Die neueste Entwicklungsversion kann sich häufig ändern und Ihren Code beschädigen.

`JSON.jl` zum Beispiel den neuesten Entwicklungszweig eines Pakets mit dem Namen `JSON.jl`

```
Pkg.checkout("JSON")
```

Verwenden Sie zum Auschecken eines anderen Zweigs oder Tags (nicht "Master" genannt)

```
Pkg.checkout("JSON", "v0.6.0")
```

Wenn das Tag jedoch eine Version darstellt, ist es normalerweise besser zu verwenden

```
Pkg.pin("JSON", v"0.6.0")
```

Beachten Sie, dass hier ein Versionsliteral verwendet wird und keine einfache Zeichenfolge. Die `Pkg.pin` Version informiert den Paketmanager über die Versionseinschränkung, sodass der Paketmanager Rückmeldungen zu möglichen Problemen geben kann.

Um zur neuesten getaggten Version zurückzukehren,

```
Pkg.free("JSON")
```

Installieren Sie ein nicht registriertes Paket

Einige experimentelle Pakete sind nicht im METADATA-Paket-Repository enthalten. Diese Pakete können durch direktes Klonen ihrer Git-Repositorys installiert werden. Beachten Sie, dass Abhängigkeiten von nicht registrierten Paketen bestehen können, die selbst nicht registriert sind. Diese Abhängigkeiten können nicht vom Paketmanager aufgelöst werden und müssen manuell aufgelöst werden. So installieren Sie beispielsweise das nicht registrierte Paket `OhMyREPL.jl` :

```
Pkg.clone("https://github.com/KristofferC/Tokenize.jl")
Pkg.clone("https://github.com/KristofferC/OhMyREPL.jl")
```

Dann wird , wie üblich, verwenden Sie `using` dem Paket verwenden:

```
using OhMyREPL
```

Pakete online lesen: https://riptutorial.com/de/julia-lang/topic/5815/pakete

Kapitel 20: Parallelverarbeitung

Examples

pmap

`pmap` übernimmt eine Funktion (die Sie angeben) und wendet diese auf alle Elemente in einem Array an. Diese Arbeit wird unter den verfügbaren Arbeitern aufgeteilt. `pmap` dann die Ergebnisse dieser Funktion in ein anderes Array zurück.

```
addprocs(3)
sqrts = pmap(sqrt, 1:10)
```

Wenn Sie mehrere Argumente verwenden, können Sie `pmap` mehrere Vektoren zur Verfügung `pmap`

```
dots = pmap(dot, 1:10, 11:20)
```

Wie bei `@parallel` jedoch, wenn die Funktion gegeben `pmap` (ist also benutzerdefiniert oder in einem Paket definiert ist) ist nicht in der Basis Julia dann müssen Sie sicherstellen, dass Funktion für alle Arbeitnehmer zur Verfügung steht zuerst:

```
@everywhere begin
    function rand_det(n)
        det(rand(n,n))
    end
end

determinants = pmap(rand_det, 1:10)
```

Siehe auch diese SO Q & A.

@parallel

Mit @parallel können Sie eine Schleife parallelisieren, indem Sie die Schritte der Schleife auf verschiedene Worker verteilen. Als sehr einfaches Beispiel:

```
addprocs(3)

a = collect(1:10)

for idx = 1:10
    println(a[idx])
end
```

Für ein etwas komplexeres Beispiel sollten Sie Folgendes berücksichtigen:

```
@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
```

Wir sehen also, wenn wir diese Schleife ohne `@parallel` hätten, hätte es 55 statt 27 Sekunden `@parallel` .

Wir können auch einen Reduktionsoperator für das `@parallel` Makro `@parallel` . Angenommen, wir haben ein Array, wir wollen jede Spalte des Arrays summieren und diese Summen dann mit einander multiplizieren:

```
A = rand(100,100);

@parallel (*) for idx = 1:size(A,1)
    sum(A[:,idx])
end
```

Bei der Verwendung von `@parallel` einige wichtige `@parallel` , um unerwartetes Verhalten zu vermeiden.

Erstens: Wenn Sie Funktionen in Ihren Schleifen verwenden möchten, die sich nicht in der Basis Julia befinden (z. B. eine von Ihnen in Ihrem Skript definierte Funktion oder die Sie aus Paketen importieren), müssen Sie diese Funktionen den Arbeitern zugänglich machen. Folgendes würde beispielsweise *nicht* funktionieren:

```
myprint(x) = println(x)
for idx = 1:10
    myprint(a[idx])
end
```

Stattdessen müssten wir Folgendes verwenden:

```
@everywhere begin
    function myprint(x)
        println(x)
    end
end

@parallel for idx in 1:length(a)
    myprint(a[idx])
end
```

Zweitens Obwohl jeder Arbeiter auf die Objekte im Bereich des Controllers zugreifen kann, kann er sie *nicht* ändern. Somit

```
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
```

Wenn wir jedoch die Schleife ohne @parallel ausgeführt hätten, wäre das Array `a` erfolgreich geändert worden.

Um dem abzuhelfen, können wir stattdessen machen `a` ein `SharedArray` Typ - Objekt , sodass jeder Arbeiter zugreifen können und ändern Sie es:

```
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 und @ Spawnat

Die Makros `@spawn` und `@spawnat` sind zwei der Werkzeuge, die Julia zur Verfügung stellt, um Arbeitern Aufgaben zuzuweisen. Hier ist ein Beispiel:

```
julia> @spawnat 2 println("hello world")
RemoteRef{Channel{Any}}(2,1,3)

julia>  From worker 2:  hello world
```

Beide Makros bewerten einen Ausdruck in einem Arbeitsprozess. Der einzige Unterschied zwischen den beiden besteht darin, dass Sie in `@spawnat` auswählen können, welcher Worker den Ausdruck auswerten soll (im obigen Beispiel wurde Worker 2 angegeben), während mit `@spawn` ein Worker automatisch `@spawn` von der Verfügbarkeit ausgewählt wird.

In dem obigen Beispiel mussten wir einfach Arbeiter 2 die println-Funktion ausführen. Es gab nichts Interessantes, um davon zurückzukommen. Oft wird jedoch der Ausdruck, den wir dem Arbeiter senden, etwas ergeben, das wir abrufen möchten. Beachten Sie im obigen Beispiel, als wir `@spawnat` , bevor wir den Ausdruck von Worker 2 erhielten, wir Folgendes gesehen haben:

```
RemoteRef{Channel{Any}}(2,1,3)
```

Dies zeigt an, dass das `@spawnat` Makro ein `RemoteRef` Objekt `RemoteRef` . Dieses Objekt enthält wiederum die Rückgabewerte aus unserem Ausdruck, die an den Worker gesendet werden. Wenn wir diese Werte abrufen möchten, können wir zuerst das von `RemoteRef` `@spawnat` einem Objekt zuweisen und dann die Funktion `fetch()` verwenden, die ein `RemoteRef` Objekt `RemoteRef` , um die Ergebnisse einer Auswertung `RemoteRef` , für die eine Auswertung durchgeführt wurde ein Arbeiter.

```
julia> result = @spawnat 2 2 + 5
```

```
RemoteRef{Channel{Any}}(2,1,26)

julia> fetch(result)
7
```

Der Schlüssel für die effektive Verwendung von `@spawn` ist das Verständnis der Natur hinter den Ausdrücken , mit denen es arbeitet. Die Verwendung von `@spawn` zum Senden von Befehlen an Worker ist etwas komplizierter als nur das direkte Eingeben, was Sie eingeben würden, wenn Sie einen "Interpreter" auf einem der Worker ausführen oder Code nativ auf ihnen ausführen würden. Nehmen wir zum Beispiel an, wir wollten `@spawnat` , um einer Variablen eines Arbeiters einen Wert zuzuweisen. Wir könnten versuchen:

```
@spawnat 2 a = 5
RemoteRef{Channel{Any}}(2,1,2)
```

Hat es funktioniert? Nun, lass uns sehen, indem Arbeiter 2 versucht, `a` zu drucken.

```
julia> @spawnat 2 println(a)
RemoteRef{Channel{Any}}(2,1,4)

julia>
```

Nichts ist passiert. Warum? Wir können dies mehr mithilfe von `fetch()` wie oben untersuchen. `fetch()` kann sehr praktisch sein, da es nicht nur erfolgreiche Ergebnisse, sondern auch Fehlermeldungen abruft. Ohne sie wissen wir vielleicht gar nicht, dass etwas schief gelaufen ist.

```
julia> result = @spawnat 2 println(a)
RemoteRef{Channel{Any}}(2,1,5)

julia> fetch(result)
ERROR: On worker 2:
UndefVarError: a not defined
```

Die Fehlermeldung besagt, dass `a` nicht für Worker 2 definiert ist. Aber warum ist das so? Der Grund ist, dass wir unsere Zuweisungsoperation in einen Ausdruck `@spawn` , den wir dann `@spawn` , um den Worker mit der Auswertung zu beauftragen. Unten ist ein Beispiel mit der folgenden Erklärung:

```
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
```

Die Syntax `:()` verwendet Julia, um Ausdrücke zu bezeichnen. Wir verwenden dann die Funktion `eval()` in Julia, die einen Ausdruck auswertet, und verwenden das `@spawnat` Makro, um anzuweisen, dass der Ausdruck auf Worker 2 ausgewertet wird.

Wir könnten auch das gleiche Ergebnis erzielen wie:

```
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
```

Dieses Beispiel zeigt zwei zusätzliche Begriffe. Zunächst sehen wir, dass wir auch einen Ausdruck erstellen können, indem Sie die Funktion `parse()` verwenden, die für einen String aufgerufen wird. Zweitens sehen wir, dass wir beim Aufruf von `@spawnat` Klammern verwenden können, wenn `@spawnat` unsere Syntax klarer und handhabbarer wird.

Wann Sie @parallel vs. pmap verwenden sollten

Die Julia- Dokumentation weist darauf hin

> pmap () ist für den Fall konzipiert, in dem jeder Funktionsaufruf viel Arbeit verrichtet. Im Gegensatz dazu kann @parallel for Situationen bewältigen, in denen jede Iteration klein ist und vielleicht nur zwei Zahlen summiert.

Dafür gibt es mehrere Gründe. `pmap` verursacht `pmap` höhere Anlaufkosten für die `pmap` Arbeitsplätzen. Wenn die Jobs sehr klein sind, können diese Anlaufkosten ineffizient werden. Umgekehrt `pmap` jedoch die Aufgabe, Arbeitsplätze unter den Arbeitnehmern zu `pmap` . Insbesondere wird eine Warteschlange mit Aufträgen erstellt und jedes Mal, wenn er verfügbar ist, ein neuer Auftrag an jeden Mitarbeiter gesendet. `@parallel` dazu gibt es alle Arbeiten, die unter den Arbeitern erledigt werden müssen, wenn sie aufgerufen werden. Wenn also einige Arbeiter länger arbeiten als andere, können Sie in einer Situation enden, in der die meisten Ihrer Mitarbeiter beendet sind und im Leerlauf sind, während einige für eine übermäßig lange Zeit aktiv bleiben und ihre Jobs beenden. Es ist jedoch weniger wahrscheinlich, dass eine solche Situation bei sehr kleinen und einfachen Jobs auftritt.

Das Folgende veranschaulicht dies: Angenommen, wir haben zwei Arbeiter, von denen einer langsam und der andere doppelt so schnell ist. Idealerweise möchten wir dem schnellen Arbeiter doppelt so viel Arbeit geben wie dem langsamen Arbeiter. (oder wir könnten schnelle und langsame Jobs haben, aber das Prinzip ist das gleiche). `pmap` dies, aber `@parallel` nicht.

Für jeden Test initialisieren wir Folgendes:

```
addprocs(2)

@everywhere begin
    function parallel_func(idx)
        workernum = myid() - 1
        sleep(workernum)
        println("job $idx")
    end
end
```

Beim `@parallel` Test führen wir nun Folgendes aus:

```
@parallel for idx = 1:12
    parallel_func(idx)
end
```

Und Druckausgabe zurückbekommen:

```
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
```

Es ist fast süß. Die Arbeiter haben die Arbeit gleichmäßig "geteilt". Beachten Sie, dass jeder Arbeiter 6 Jobs ausgeführt hat, obwohl Arbeiter 2 doppelt so schnell wie Arbeiter 3 ist. Er kann sich berühren, ist aber ineffizient.

Für den `pmap` Test `pmap` ich Folgendes aus:

```
pmap(parallel_func, 1:12)
```

und erhalte die Ausgabe:

```
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
```

Nun ist zu beachten, dass Arbeiter 2 acht Jobs und Arbeiter 3 4 ausgeführt hat. Dies ist genau im Verhältnis zu ihrer Geschwindigkeit und dem, was wir für eine optimale Effizienz wünschen. `pmap` ist ein harter Aufgabenmeister - von jedem nach seinen Fähigkeiten.

@async und @sync

Gemäß der Dokumentation unter `?@async` " `@async` einen Ausdruck in einer Task." Dies bedeutet, dass Julia diese Aufgabe für das, was in ihren Geltungsbereich fällt, startet, aber dann mit dem nächsten Schritt im Skript fortfährt, ohne auf den Abschluss der Aufgabe zu warten. So erhalten Sie beispielsweise ohne das Makro:

```
julia> @time sleep(2)
  2.005766 seconds (13 allocations: 624 bytes)
```

Mit dem Makro erhalten Sie jedoch:

```
julia> @time @async sleep(2)
  0.000021 seconds (7 allocations: 657 bytes)
Task (waiting) @0x0000000112a65ba0

julia>
```

Julia erlaubt dem Skript daher, fortzufahren (und das `@time` Makro vollständig auszuführen), ohne darauf zu warten, dass die Aufgabe (in diesem Fall zwei Sekunden im `@time`) abgeschlossen ist.

Im Gegensatz dazu wird das `@sync` Makro "warten, bis alle dynamisch eingeschlossenen Verwendungen von `@async` , `@spawn` , `@spawnat` und `@parallel` sind." (gemäß der Dokumentation unter `?@sync`). So sehen wir:

```
julia> @time @sync @async sleep(2)
  2.002899 seconds (47 allocations: 2.986 KB)
Task (done) @0x0000000112bd2e00
```

In diesem einfachen Beispiel ist es nicht `@async` eine einzelne Instanz von `@async` und `@sync` zusammenzufügen. `@sync` kann jedoch nützlich sein, wenn `@async` auf mehrere Vorgänge angewendet werden soll, die alle gleichzeitig starten sollen, ohne darauf zu warten, bis die einzelnen Operationen abgeschlossen sind.

Angenommen, wir haben mehrere Mitarbeiter, und wir möchten, dass jeder von ihnen gleichzeitig an einer Aufgabe arbeitet und dann die Ergebnisse aus diesen Aufgaben abruft. Ein erster (aber falscher) Versuch könnte sein:

```
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)
```

Das Problem hierbei ist, dass die Schleife auf die `remotecall_fetch()` jeder `remotecall_fetch()` Operation wartet, dh dass jeder Prozess seine Arbeit beendet (in diesem Fall für 2 Sekunden `remotecall_fetch()`), bevor die nächste `remotecall_fetch()` Operation fortgesetzt wird. In einer praktischen Situation haben wir hier nicht die Vorteile der Parallelität, da unsere Prozesse nicht gleichzeitig arbeiten (dh schlafen).

Wir können dies jedoch korrigieren, indem Sie eine Kombination der `@async` und `@sync` Makros verwenden:

```
@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)
```

Wenn wir nun jeden Schritt der Schleife als separate Operation zählen, sehen wir, dass dem `@async` Makro zwei separate Operationen vorangestellt sind. Das Makro ermöglicht das Starten jedes dieser Elemente, und der Code wird fortgesetzt (in diesem Fall bis zum nächsten Schritt der Schleife), bevor er beendet wird. Die Verwendung des `@sync` Makros, dessen Gültigkeitsbereich die gesamte Schleife umfasst, bedeutet jedoch, dass das Skript nicht an dieser Schleife vorbeigehen kann, bis alle Vorgänge, denen `@async` , abgeschlossen sind.

Sie können die Funktionsweise dieser Makros noch genauer verstehen, indem Sie das obige Beispiel weiter anpassen, um zu sehen, wie es sich bei bestimmten Modifikationen ändert. Angenommen, wir haben nur `@async` ohne `@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)
```

Mit dem `@async` Makro können wir in unserer Schleife `remotecall_fetch()` auch bevor die Ausführung von `remotecall_fetch()` . Wir haben jedoch kein `@sync` Makro, um zu verhindern, dass der Code nach dieser Schleife fortgesetzt wird, bis alle `remotecall_fetch()` .

Trotzdem `remotecall_fetch()` jede `remotecall_fetch()` Operation immer noch parallel, auch wenn wir `remotecall_fetch()` . Wir können das sehen, denn wenn wir zwei Sekunden warten, enthält das Array a, das die Ergebnisse enthält, Folgendes:

```
sleep(2)
julia> a
2-element Array{Any,1}:
 nothing
 nothing
```

(Das Element "nothing" ist das Ergebnis eines erfolgreichen Abrufs der Ergebnisse der Sleep-Funktion, die keine Werte zurückgibt.)

Sehen wir auch , dass die beiden `remotecall_fetch()` Operationen im wesentlichen zur gleichen Zeit gestartet werden, da die `print` , die sie auch in schnellen Folge ausführen vorausgehen (Ausgabe von diesen Befehlen hier nicht dargestellt). Man vergleiche dies mit dem nächsten Beispiel , bei dem die `print` in einer 2 Sekunde Verzögerung voneinander auszuführen:

Wenn wir das `@async` Makro in die gesamte Schleife `@async` (anstatt nur den inneren Schritt), wird unser Skript sofort fortgesetzt, ohne zu warten, bis die `remotecall_fetch()` . Jetzt erlauben wir jedoch nur, dass das Skript über die gesamte Schleife hinausgeht. Wir lassen nicht zu, dass jeder

einzelne Schritt der Schleife beginnt, bevor der vorherige abgeschlossen ist. Anders als im obigen Beispiel hat das `results` Array zwei Sekunden, nachdem das Skript nach der Schleife `#undef` , noch ein Element als `#undef` , das `#undef` hinweist, dass die zweite `remotecall_fetch()` noch nicht abgeschlossen ist.

```
@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
```

Und nicht überraschend, wenn wir `@sync` und `@async` direkt nebeneinander stellen, wird `@sync` , dass jeder `remotecall_fetch()` sequentiell (und nicht gleichzeitig) ausgeführt wird. Der Code wird jedoch erst fortgesetzt, wenn er fertig ist. Mit anderen Worten, dies wäre im Wesentlichen das Äquivalent dazu, wenn wir keines der Makros hätten, genauso wie sich `sleep(2)` Wesentlichen identisch zu `@sync @async sleep(2)` verhält.

```
@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
```

Beachten Sie auch, dass im `@async` Makro `@async` Operationen möglich sind. Die Dokumentation enthält ein Beispiel, das eine gesamte Schleife im Rahmen von `@async` .

Es sei daran erinnert, dass die Hilfe für die Synchronisationsmakros besagt, dass "es warten wird, bis alle dynamisch eingeschlossenen Anwendungen von `@async` , `@spawn` , `@spawnat` und `@parallel` vollständig sind." Für das, was als "abgeschlossen" gilt, ist es wichtig, wie Sie die Aufgaben im Rahmen der `@sync` und `@async` Makros definieren. Betrachten Sie das folgende Beispiel, das eine geringfügige Abweichung von einem der oben angegebenen Beispiele darstellt:

```
@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)
```

Die Ausführung des vorherigen Beispiels dauerte ungefähr zwei Sekunden. Dies zeigt an, dass die beiden Tasks parallel ausgeführt wurden und das Skript darauf wartete, dass die Ausführung der Funktionen abgeschlossen war, bevor es fortfuhr. Dieses Beispiel hat jedoch eine wesentlich geringere Zeitauswertung. Der Grund dafür ist, dass die `@sync remotecall()` für `@sync` "beendet" ist, sobald der Arbeiter den Job gesendet hat. (Beachten Sie, dass das resultierende Array, a, hier nur `RemoteRef` Objekttypen enthält, die nur darauf hinweisen, dass bei einem bestimmten Prozess etwas `RemoteRef` , das theoretisch irgendwann in der Zukunft abgerufen werden könnte.) Im Gegensatz dazu ist der `remotecall_fetch()` " `remotecall_fetch()` " nur "beendet", wenn er die Nachricht vom Worker erhält, dass seine Aufgabe abgeschlossen ist.

Wenn Sie also nach Möglichkeiten suchen, sicherzustellen, dass bestimmte Operationen mit Arbeitern abgeschlossen sind, bevor Sie mit Ihrem Skript fortfahren (wie zum Beispiel in diesem Beitrag beschrieben), müssen Sie sorgfältig darüber nachdenken, was als "abgeschlossen" gilt und wie Sie dies tun werden Messen und dann operationalisieren Sie das in Ihrem Skript.

Arbeiter hinzufügen

Wenn Sie Julia zum ersten Mal starten, wird standardmäßig nur ein einziger Prozess ausgeführt, der für die Arbeit verfügbar ist. Sie können dies überprüfen mit:

```
julia> nprocs()
1
```

Um die parallele Verarbeitung zu nutzen, müssen Sie zunächst weitere Worker hinzufügen, die dann für die ihnen zugewiesene Arbeit zur Verfügung stehen. Sie können dies in Ihrem Skript (oder über den Interpreter) mithilfe von: `addprocs(n)` wobei `n` die Anzahl der Prozesse ist, die Sie verwenden möchten.

Alternativ können Sie Prozesse hinzufügen, wenn Sie Julia von der Befehlszeile aus starten:

```
$ julia -p n
```

Dabei ist `n` die Anzahl der *zusätzlichen* Prozesse, die Sie hinzufügen möchten. Also, wenn wir mit Julia anfangen

```
$ julia -p 2
```

Wenn Julia anfängt, bekommen wir:

```
julia> nprocs()
3
```

Parallelverarbeitung online lesen: https://riptutorial.com/de/julia-lang/topic/4542/parallelverarbeitung

Kapitel 21: Regexes

Syntax

- Regex ("[Regex]")
- r "[Regex]"
- Streichholz (Nadel, Heuhaufen)
- Matchall (Nadel, Heuhaufen)
- Eachmatch (Nadel, Heuhaufen)
- Ismatch (Nadel, Heuhaufen)

Parameter

Parameter	Einzelheiten
`needle`	die `Regex` im `haystack` suchen
`haystack`	der Text, in dem nach der `needle`

Examples

Regex-Literale

Julia unterstützt reguläre Ausdrücke [1] . Die PCRE-Bibliothek wird als Regex-Implementierung verwendet. Regexe sind wie eine Minisprache innerhalb einer Sprache. Da die meisten Sprachen und viele Texteditoren Unterstützung für reguläre Ausdrücke bieten, sind Dokumentation und Beispiele für die Verwendung von regulärer Expression im Allgemeinen nicht in diesem Beispiel enthalten.

Es ist möglich, eine `Regex` mithilfe des Konstruktors aus einer Zeichenfolge zu `Regex` :

```
julia> Regex("(cat|dog)s?")
```

Der `@r_str` kann jedoch das `@r_str` String-Makro verwendet werden:

```
julia> r"(cat|dog)s?"
```

[1] : Technisch unterstützt Julia Ausdrücke, die sich von den in der Sprachtheorie genannten regulären Ausdrücken unterscheiden. Häufig wird der Begriff "regulärer Ausdruck" auch für Regex verwendet.

Übereinstimmungen finden

Es gibt vier Hauptfunktionen für reguläre Ausdrücke, die alle Argumente in `needle, haystack`

annehmen. Die Begriffe "Nadel" und "Heuhaufen" stammen aus der englischen Redewendung "Nadel im Heuhaufen finden". Im Zusammenhang mit Regex ist der Regex die Nadel und der Text der Heuhaufen.

Die `match` Funktion kann verwendet werden, um die erste Übereinstimmung in einer Zeichenfolge zu finden:

```
julia> match(r"(cat|dog)s?", "my cats are dogs")
RegexMatch("cats", 1="cat")
```

Mit der `matchall` Funktion können Sie alle Übereinstimmungen eines regulären Ausdrucks in einem String suchen:

```
julia> matchall(r"(cat|dog)s?", "The cat jumped over the dogs.")
2-element Array{SubString{String},1}:
 "cat"
 "dogs"
```

Die `ismatch` Funktion gibt einen Booleschen `ismatch` zurück, der angibt, ob eine Übereinstimmung in der Zeichenfolge gefunden wurde:

```
julia> ismatch(r"(cat|dog)s?", "My pigs")
false

julia> ismatch(r"(cat|dog)s?", "My cats")
true
```

Die `eachmatch` Funktion gibt einen Iterator über `RegexMatch` Objekte zurück, die für `for` Schleifen geeignet sind :

```
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
```

Gruppen erfassen

Auf die von Capture-Gruppen erfassten `RegexMatch` kann von `RegexMatch` Objekten mithilfe der Indexierungsnotation `RegexMatch` werden.

Beispielsweise analysiert der folgende Regex nordamerikanische Telefonnummern, die im `(555)-555-5555` Format geschrieben wurden:

```
julia> phone = r"\((\d{3})\)-(\d{3})-(\d{4})"
```

Angenommen, wir möchten die Telefonnummern aus einem Text extrahieren:

```
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"
```

Mit der `matchall` Funktion können wir ein Array der Teilzeichenfolgen selbst erhalten:

```
julia> matchall(phone, text)
2-element Array{SubString{String},1}:
 "(555)-505-1000"
 "(555)-999-9999"
```

Angenommen, wir möchten auf die Vorwahlnummern (die ersten drei Ziffern, in Klammern eingeschlossen) zugreifen. Dann können wir den `eachmatch` Iterator verwenden:

```
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
```

Beachten Sie hierbei, dass wir `m[1]` da die Vorwahl die erste Erfassungsgruppe in unserem regulären Ausdruck ist. Wir können alle drei Komponenten der Telefonnummer über eine Funktion als Tupel abrufen:

```
julia> splitmatch(m) = m[1], m[2], m[3]
splitmatch (generic function with 1 method)
```

Dann können wir eine solche Funktion auf ein bestimmtes `RegexMatch` :

```
julia> splitmatch(match(phone, text))
("555","505","1000")
```

Oder wir könnten `map` es in jedem Spiel:

```
julia> map(splitmatch, eachmatch(phone, text))
2-element Array{Tuple{SubString{String},SubString{String},SubString{String}},1}:
 ("555","505","1000")
 ("555","999","9999")
```

Regexes online lesen: https://riptutorial.com/de/julia-lang/topic/5890/regexes

Kapitel 22: REPL

Syntax

- Julia>
- Hilfe?>
- Schale>
- \[Latex]

Bemerkungen

Andere Pakete können zusätzlich zu den Standardmodi eigene REPL-Modi definieren. Das `Cxx` Paket definiert beispielsweise den Shell-Modus `cxx>` für eine C ++ - REPL. Diese Modi sind normalerweise mit ihren eigenen Sondertasten erreichbar. Weitere Informationen finden Sie in der Paketdokumentation.

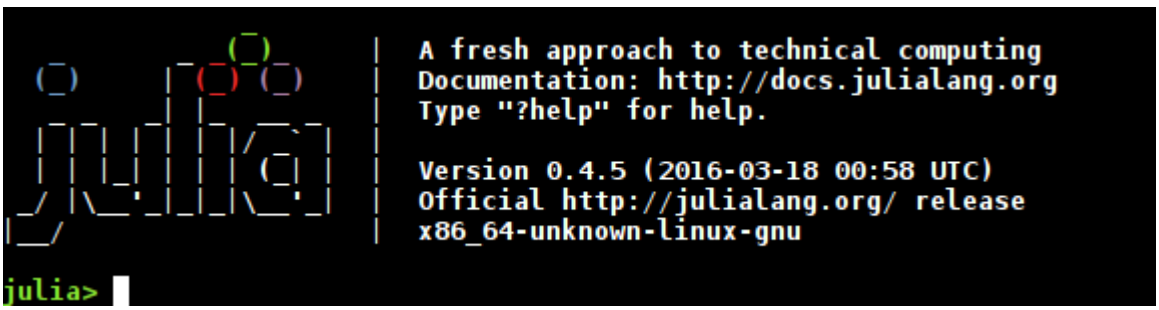
Examples

Starten Sie die REPL

Nach der Installation von Julia starten Sie die Read-Eval-Print-Schleife (REPL):

Auf Unix-Systemen

Öffnen Sie ein Terminal - Fenster, geben Sie dann `julia` an der Eingabeaufforderung, dann drücken Sie die `Eingabetaste`. Sie sollten so etwas sehen:



Unter Windows

Suchen Sie das Julia-Programm in Ihrem Startmenü und klicken Sie darauf. Die REPL sollte gestartet werden.

Verwenden der REPL als Taschenrechner

Der Julia REPL ist ein ausgezeichneter Rechner. Wir können mit einigen einfachen Operationen

beginnen:

```
julia> 1 + 1
2

julia> 8 * 8
64

julia> 9 ^ 2
81
```

Die Variable `ans` enthält das Ergebnis der letzten Berechnung:

```
julia> 4 + 9
13

julia> ans + 9
22
```

Wir können unsere eigenen Variablen mit der Zuordnung definieren `=` Operator:

```
julia> x = 10
10

julia> y = 20
20

julia> x + y
30
```

Julia hat eine implizite Multiplikation für numerische Literale, wodurch einige Berechnungen schneller geschrieben werden können:

```
julia> 10x
100

julia> 2(x + y)
60
```

Wenn wir einen Fehler machen und etwas tun, das nicht zulässig ist, gibt die Julia REPL einen Fehler aus, oft mit einem hilfreichen Tipp zur Behebung des Problems:

```
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
```

Um auf vorherige Befehle zuzugreifen oder sie zu bearbeiten, verwenden Sie die Taste ↑ (Nach

oben), die zum letzten Eintrag im Verlauf wechselt. Das `↓` `wechselt` zum nächsten Element in der Historie. Mit den Tasten `←` und `→` können Sie eine Zeile verschieben und bearbeiten.

Julia hat einige eingebaute mathematische Konstanten, einschließlich `e` und `pi` (oder `π`).

```
julia> e
e = 2.7182818284590...

julia> pi
π = 3.1415926535897...

julia> 3π
9.42477796076938
```

Wir können Zeichen wie `π` schnell `π` , indem Sie ihre LaTeX-Codes verwenden: Drücken Sie `\` , dann `p` und `i` , und drücken Sie die `Tabulatortaste, um das` eingegebene `\pi` durch `π` zu ersetzen. Dies funktioniert für andere griechische Buchstaben und zusätzliche Unicode-Symbole.

Wir können alle integrierten mathematischen Funktionen von Julia verwenden, die von einfach bis ziemlich mächtig reichen:

```
julia> cos(π)
-1.0

julia> besselh(1, 1, 1)
0.44005058574493355 - 0.7812128213002889im
```

Komplexe Zahlen werden mit `im` als imaginäre Einheit unterstützt:

```
julia> abs(3 + 4im)
5.0
```

Einige Funktionen geben kein komplexes Ergebnis zurück, es sei denn, Sie geben eine komplexe Eingabe ab, selbst wenn die Eingabe echt ist:

```
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
```

Genaue Operationen mit rationalen Zahlen sind mit dem `//` rationalen Divisionsoperator möglich:

```
julia> 1//3 + 1//3
2//3
```

Unter Arithmetik finden Sie weitere Informationen darüber, welche arithmetischen Operatoren von Julia unterstützt werden.

Umgang mit Maschinengenauigkeit

Beachten Sie, dass Maschinen-Ganzzahlen in der Größe eingeschränkt sind und überlaufen, wenn das Ergebnis zu groß zum Speichern ist:

```
julia> 2^62
4611686018427387904

julia> 2^63
-9223372036854775808
```

Dies kann durch die Verwendung von Ganzzahlen mit beliebiger Genauigkeit bei der Berechnung verhindert werden:

```
julia> big"2"^62
4611686018427387904

julia> big"2"^63
9223372036854775808
```

Maschinenfließpunkte sind auch in der Genauigkeit begrenzt:

```
julia> 0.1 + 0.2
0.30000000000000004
```

Mehr (aber immer noch begrenzte) Präzision ist durch erneutes Verwenden von `big` :

```
julia> big"0.1" + big"0.2"
3.000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000017e-01
```

Genaue Berechnungen können in einigen Fällen mit `Rational` :

```
julia> 1//10 + 2//10
3//10
```

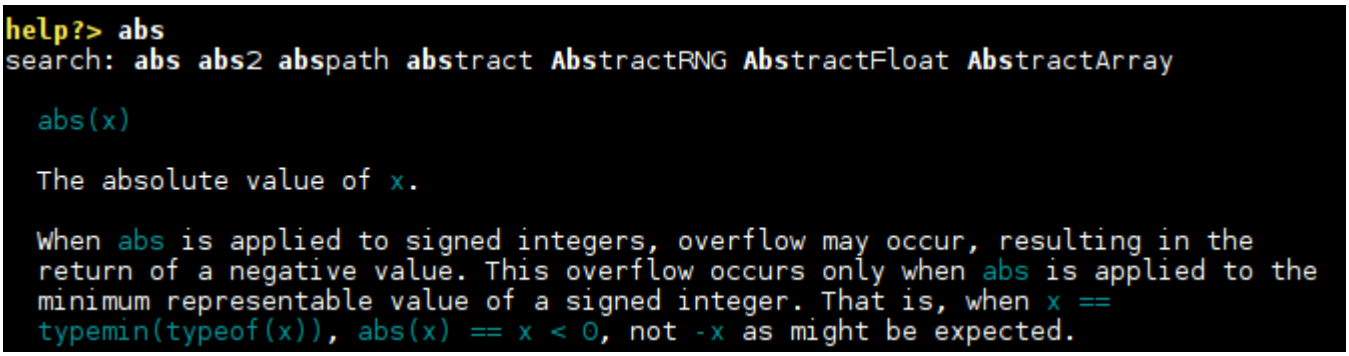
REPL-Modi verwenden

In Julia gibt es drei integrierte REPL-Modi: den Julia-Modus, den Hilfemodus und den Shell-Modus.

Der Hilfemodus

Das Julia REPL verfügt über ein integriertes Hilfesystem. Drücken `?` an der `julia>` Eingabeaufforderung, um auf die `help?>` zuzugreifen.

Geben Sie an der Hilfeaufforderung den Namen einer Funktion oder eines Typs ein, um Hilfe zu erhalten:



Auch wenn Sie die Funktion nicht richtig buchstabieren, kann Julia einige Funktionen vorschlagen, die möglicherweise Ihre Bedeutung hatten:

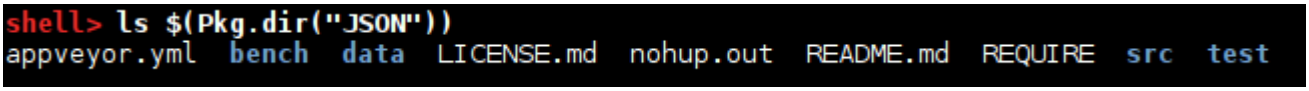
```
help?> printline
search:

Couldn't find printline
Perhaps you meant println, pipeline, @inline or print
  No documentation found.

  Binding printline does not exist.
```

Diese Dokumentation funktioniert auch für andere Module, sofern sie das Julia-Dokumentationssystem verwenden.

```
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)
```

Der Shell-Modus

Weitere Informationen zur Verwendung von Julias Shell-Modus finden Sie unter Verwenden von Shell in REPL . auf die Aufforderung. Dieser Shell-Modus unterstützt die Interpolation von Daten aus der Julia REPL-Sitzung. Dadurch können Sie leicht Julia-Funktionen aufrufen und ihre Ergebnisse in Shell-Befehle umwandeln:



REPL online lesen: https://riptutorial.com/de/julia-lang/topic/5739/repl

Kapitel 23: Shell Scripting und Piping

Syntax

- Shell-Befehl

Examples

Shell aus dem REPL verwenden

Innerhalb der interativen Julia-Shell (auch als REPL bezeichnet) können Sie durch Eingabe auf die Shell des Systems zugreifen `;` gleich nach der Aufforderung:

```
shell>
```

Von hier aus können Sie einen beliebigen Shell-Befehl eingeben, der innerhalb der REPL ausgeführt wird:

```
shell> ls
Desktop     Documents   Pictures   Templates
Downloads   Music       Public     Videos
```

Um diesen Modus zu verlassen, geben Sie `backspace` wenn die Eingabeaufforderung leer ist.

Aus Julia Code herausschälen

Julia-Code kann Befehlsliterale erstellen, bearbeiten und ausführen, die in der Systemumgebung des Betriebssystems ausgeführt werden. Dies ist leistungsfähig, macht Programme jedoch oft weniger portabel.

Ein Befehlsliteral kann mit dem `` ` `` Literal erstellt werden. Informationen können wie bei String-Literalen mit der `$` -Interpolationssyntax interpoliert werden. Julia-Variablen, die durch Befehlsliterale durchlaufen werden, müssen nicht zuerst maskiert werden. Sie werden nicht wirklich an die Shell übergeben, sondern direkt an den Kernel. Julia zeigt diese Objekte jedoch so an, dass sie korrekt maskiert erscheinen.

```
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(+)
```

Shell Scripting und Piping online lesen: https://riptutorial.com/de/julia-lang/topic/5420/shell-scripting-und-piping

Kapitel 24: String-Normalisierung

Syntax

- normalize_string (s :: String, ...)

Parameter

Parameter	Einzelheiten
`casefold=true`	Falten Sie die Zeichenfolge zu einem kanonischen Fall, der auf dem Unicode- Standard basiert.
`stripmark=true`	Entfernen Sie diakritische Zeichen (dh Akzente) von Zeichen in der Eingabezeichenfolge.

Examples

String-Vergleich ohne Berücksichtigung der Groß- und Kleinschreibung

Zeichenfolgen können mit dem Operator `==` in Julia verglichen werden. Dies ist jedoch abhängig von den Unterschieden. Zum Beispiel werden `"Hello"` und `"hello"` als unterschiedliche Zeichenfolgen betrachtet.

```
julia> "Hello" == "Hello"
true

julia> "Hello" == "hello"
false
```

Um Zeichenfolgen unabhängig von der Groß- und Kleinschreibung zu vergleichen, normalisieren Sie die Zeichenfolgen, indem Sie sie zuerst mit der Groß- / Kleinschreibung falten. Zum Beispiel,

```
equals_ignore_case(s, t) =
    normalize_string(s, casefold=true) == normalize_string(t, casefold=true)
```

Dieser Ansatz behandelt auch Nicht-ASCII-Unicode korrekt:

```
julia> equals_ignore_case("Hello", "hello")
true

julia> equals_ignore_case("Weierstraß", "WEIERSTRASS")
true
```

Beachten Sie, dass in Deutsch die Großbuchstaben des ß-Zeichens SS ist.

Vergleich der diakritischen Zeichenfolgen

Manchmal möchte man Zeichenfolgen wie `"resume"` und `"ré sumé "` gleich vergleichen. Das heißt, Grapheme , die eine grundlegende Glyphe verwenden, sich aber möglicherweise aufgrund von Hinzufügungen zu diesen grundlegenden Glyphen unterscheiden. Ein solcher Vergleich kann durch Abziehen diakritischer Markierungen erreicht werden.

```
equals_ignore_mark(s, t) =
    normalize_string(s, stripmark=true) == normalize_string(t, stripmark=true)
```

Dadurch kann das obige Beispiel korrekt funktionieren. Darüber hinaus funktioniert es auch mit Nicht-ASCII-Unicode-Zeichen.

```
julia> equals_ignore_mark("resume", "ré sumé ")
true

julia> equals_ignore_mark("αβγ", "à β ỹ ")
true
```

String-Normalisierung online lesen: https://riptutorial.com/de/julia-lang/topic/7612/string-normalisierung

Kapitel 25: Tuples

Syntax

- ein,
- a, b
- a, b = xs
- ()
- (ein,)
- (a, b)
- (a, b ...)
- Tupel {T, U, V}
- NTuple {N, T}
- Tupel {T, U, Vararg {V}}

Bemerkungen

Tupel haben aus zwei Gründen eine viel bessere Laufzeitperformance als Arrays : Ihre Typen sind präziser und aufgrund ihrer Unveränderlichkeit können sie auf dem Stapel statt auf dem Heap zugewiesen werden. Diese genauere Typisierung bringt jedoch mehr Overhead bei der Kompilierung mit sich und es ist schwieriger, die Typstabilität zu erreichen .

Examples

Einführung in Tuples

`Tuple` sind unveränderliche geordnete Sammlungen von beliebigen Objekten, entweder vom selben Typ oder von verschiedenen Typen . Typischerweise werden Tupel unter Verwendung der `(x, y)` -Syntax konstruiert.

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

Die einzelnen Objekte eines Tupels können mit der Indexierungssyntax abgerufen werden:

```
julia> tup[1]
1

julia> tup[2]
1.0

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

Sie implementieren die iterierbare Schnittstelle und können daher mithilfe von `for` Schleifen iteriert werden :

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

Tupel unterstützen auch eine Reihe generischer Sammlungsfunktionen, z. B. `reverse` oder `length` :

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

julia> length(tup)
3
```

Darüber hinaus unterstützen Tupel eine Reihe von Erfassungsvorgängen höherer Ordnung , darunter `any` , `all` `map` oder `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
```

Das leere Tupel kann mit `()` :

```
julia> ()
()

julia> isempty(ans)
true
```

Um ein Tupel eines Elements zu erstellen, ist jedoch ein nachfolgendes Komma erforderlich. Dies liegt daran, dass die Klammern (`(` und `)`) sonst als Gruppieren von Operationen behandelt würden, anstatt ein Tupel zu erstellen.

```
julia> (1)
1

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

Aus Gründen der Konsistenz ist ein Nachkomma auch für Tupel mit mehr als einem Element zulässig.

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

Tuple-Typen

Der `typeof` eines Tupels ist ein Untertyp von `Tuple` :

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

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

Im Gegensatz zu anderen Datentypen sind `Tuple` Typen kovariant . Andere Datentypen in Julia sind im Allgemeinen unveränderlich. Somit,

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

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

Dies ist der Fall, da überall ein `Tuple{Number, Number}` akzeptiert wird, ebenso ein `Tuple{Int, Int}` , da er auch zwei Elemente hat, die beide Zahlen sind. Dies ist bei einem `Vector{Int}` gegenüber einem `Vector{Number}` nicht der Fall, da eine Funktion, die einen `Vector{Number}` akzeptiert, möglicherweise einen Gleitkommawert (z. B. `1.0`) oder eine komplexe Zahl (z. B. `1+3im`) in einem solchen `1+3im` ein Vektor.

Die Kovarianz von Tupeltypen bedeutet, dass Tuple `Tuple{Number}` (wiederum im Gegensatz zu `Vector{Number}`) tatsächlich ein abstrakter Typ ist:

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

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

Zu den konkreten Subtypen von `Tuple{Number}` gehören `Tuple{Int}` , `Tuple{Float64}` , `Tuple{Rational{BigInt}}` und so weiter.

`Tuple` Typen können einen abschließenden `Vararg` als letzten Parameter enthalten, um eine unbegrenzte Anzahl von Objekten anzuzeigen. `Tuple{Vararg{Int}}` ist beispielsweise der Typ aller Tupel, die eine beliebige Anzahl von `Int` s enthalten, möglicherweise Null:

```
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
```

`Tuple{String, Vararg{Int}}` akzeptiert Tupel, die aus einem String bestehen , gefolgt von einer beliebigen Anzahl (möglicherweise Null) von `Int` s.

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

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

Kombiniert mit `Tuple{Vararg{Any}}` bedeutet dies, dass `Tuple{Vararg{Any}}` jedes Tupel beschreibt. In der Tat ist `Tuple{Vararg{Any}}` nur eine andere Art, `Tuple` sagen:

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

`Vararg` akzeptiert einen zweiten numerischen Typparameter, der angibt, wie oft genau der erste Typparameter vorkommen soll. (Standardmäßig , wenn nicht spezifiziert, wobei diese zweite Typ Parameter ist ein typevar , die jeden Wert annehmen kann, weshalb eine beliebige Anzahl von `Int` s in der akzeptiert `Vararg` oben s.) `Tuple` Arten in einer bestimmten Endung `Vararg` wird automatisch auf die erweitert werden gewünschte Anzahl von Elementen:

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

Notation existiert für homogene Tupel mit einem angegebenen `Vararg` : `NTuple{N, T}` . In dieser Notation bezeichnet `N` die Anzahl der Elemente im Tupel und `T` den akzeptierten Typ. Zum Beispiel,

```
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)
```

Beachten Sie, dass `NTuple` jenseits einer bestimmten Größe anstelle des erweiterten `Tuple` Formulars einfach als `NTuple{N, T}` , sie sind jedoch immer noch derselbe Typ:

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

Versand auf Tupel-Typen

Da Julia-Funktionsparameterlisten selbst Tupel sind, ist das Dispatching auf verschiedene Arten von Tupeln oft einfacher über die Methodenparameter selbst, oft mit freiem Gebrauch für den Operator "Splatting" `...` Betrachten Sie beispielsweise die Implementierung von `reverse` für Tupel von `Base` :

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

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

Die Implementierung Methoden auf Tupeln bewahrt auf diese Weise Stabilität geben , die für die Leistung entscheidend ist. Wir können sehen, dass dieser Ansatz mit dem `@code_warntype` Makro nicht `@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)::Tupl

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

Obwohl es etwas schwer zu lesen ist, erstellt der Code einfach ein neues Tupel mit den Werten 3., 2. bzw. 1. Element des ursprünglichen Tupels. Bei vielen Maschinen wird dies zu äußerst effizientem LLVM-Code, der aus Ladungen und Speichern besteht.

```
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
}
```

Mehrere Rückgabewerte

Tupel werden häufig für mehrere Rückgabewerte verwendet. Ein Großteil der Standardbibliothek, einschließlich zweier Funktionen der iterierbaren Schnittstelle (`next` und `done`), gibt Tupel zurück, die zwei verwandte, aber unterschiedliche Werte enthalten.

Die Klammern um Tupel können in bestimmten Situationen weggelassen werden, so dass mehrere Rückgabewerte einfacher zu implementieren sind. Zum Beispiel können wir eine

Funktion erstellen, um sowohl positive als auch negative Quadratwurzeln einer reellen Zahl zurückzugeben:

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

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

Mit Hilfe der Destrukturierungszuweisung können mehrere Rückgabewerte entpackt werden. Um die Quadratwurzeln in den Variablen `a` und `b` zu speichern, genügt es zu schreiben:

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

julia> a
3.0

julia> b
-3.0
```

Ein anderes Beispiel dafür sind die Funktionen `divrem` und `fldmod` , die gleichzeitig eine Ganzzahl- (`divrem` oder `fldmod`) Division und eine `divrem` `fldmod` :

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

julia> q
3

julia> r
1
```

Tuples online lesen: https://riptutorial.com/de/julia-lang/topic/6675/tuples

Kapitel 26: Typ Stabilität

Einführung

Typinstabilität tritt auf, wenn der Typ einer Variablen zur Laufzeit geändert werden kann und daher nicht zur Kompilierzeit abgeleitet werden kann. Typinstabilität verursacht häufig Leistungsprobleme, daher ist es wichtig, typstabilen Code schreiben und identifizieren zu können.

Examples

Schreiben Sie typstabilen Code

```
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
```

Das Timing der beiden oben genannten Funktionen zeigt wesentliche Unterschiede hinsichtlich der Zeit- und Speicherzuordnung.

```
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)
```

Dies liegt an dem Typ-instabilen Code in `sumofsins1` bei dem der Typ von `r` für jede Iteration überprüft werden muss.

Typ Stabilität online lesen: https://riptutorial.com/de/julia-lang/topic/6084/typ-stabilitat

Kapitel 27: Typen

Syntax

- unveränderlich MyType; Feld; Feld; Ende
- Geben Sie MyType ein. Feld; Feld; Ende

Bemerkungen

Typen sind der Schlüssel zu Julias Leistung. Eine wichtige Idee für die Leistung ist die Typstabilität , die auftritt, wenn der von einer Funktion zurückgegebene Typ nur von den Typen und nicht von den Werten ihrer Argumente abhängt.

Examples

Versand auf Typen

In Julia können Sie für jede Funktion mehrere Methoden definieren. Nehmen wir an, wir definieren drei Methoden derselben Funktion:

```
foo(x) = 1
foo(x::Number) = 2
foo(x::Int) = 3
```

Bei der Entscheidung, welche Methode verwendet werden soll (als " dispatch" bezeichnet), wählt Julia die spezifischere Methode aus, die den Typen der Argumente entspricht:

```
julia> foo('one')
1

julia> foo(1.0)
2

julia> foo(1)
3
```

Dies erleichtert den Polymorphismus . Beispielsweise können Sie leicht eine verknüpfte Liste erstellen, indem Sie zwei unveränderliche Typen definieren, die als `Nil` und `Cons` . Diese Namen werden traditionell verwendet, um eine leere Liste bzw. eine nicht leere Liste zu beschreiben.

```
abstract LinkedList
immutable Nil <: LinkedList end
immutable Cons <: LinkedList
    first
    rest::LinkedList
end
```

Die leere Liste wird von `Nil()` und alle anderen Listen von `Cons(first, rest)` , wobei `first` das erste Element der verknüpften Liste und `rest` die verknüpfte Liste ist, die aus allen übrigen Elementen besteht. Beispielsweise wird die Liste `[1, 2, 3]` als dargestellt

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

Ist die Liste leer?

Angenommen, wir möchten die `isempty` Funktion der Standardbibliothek `isempty` , die für verschiedene Sammlungen verwendet wird:

```
julia> methods(isempty)
# 29 methods for generic function "isempty":
isempty(v::SimpleVector) at essentials.jl:180
isempty(m::Base.MethodList) at reflection.jl:394
...
```

Wir können einfach die Funktionssendungssyntax verwenden und zwei zusätzliche Methoden für `isempty` . Da diese Funktion von dem ist `Base` haben wir es als qualifizieren `Base.isempty` , um es zu erweitern.

```
Base.isempty(::Nil) = true
Base.isempty(::Cons) = false
```

Hier haben wir die Argumentwerte überhaupt nicht benötigt, um festzustellen, ob die Liste leer ist. Nur der Typ allein reicht aus, um diese Informationen zu berechnen. Julia erlaubt uns, die Namen der Argumente wegzulassen und nur die Typanmerkung beizubehalten, wenn wir ihre Werte nicht verwenden müssen.

Wir können testen, ob unsere `isempty` Methoden funktionieren:

```
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())))))
```

und in der Tat hat sich die Anzahl der Methoden für `isempty` um `2` erhöht:

```
julia> methods(isempty)
# 31 methods for generic function "isempty":
isempty(v::SimpleVector) at essentials.jl:180
isempty(m::Base.MethodList) at reflection.jl:394
```

Die Feststellung, ob eine verknüpfte Liste leer ist oder nicht, ist eindeutig ein triviales Beispiel.

Aber es führt zu etwas Interessanterem:

Wie lang ist die Liste?

Die `length` aus der Standardbibliothek gibt die Länge einer Sammlung oder bestimmte iterierbare Werte an . Es gibt viele Möglichkeiten, um `length` für eine verknüpfte Liste zu implementieren. Insbesondere ist die Verwendung einer `while` Schleife in Julia wahrscheinlich am schnellsten und am meisten Speichereffizienz. Eine vorzeitige Optimierung sollte jedoch vermieden werden. Nehmen wir einmal an, dass unsere verknüpfte Liste nicht effizient sein muss. Was ist der einfachste Weg , um einen schreiben `length` Funktion?

```
Base.length(::Nil) = 0
Base.length(xs::Cons) = 1 + length(xs.rest)
```

Die erste Definition ist einfach: Eine leere Liste hat die Länge `0` . Die zweite Definition ist auch leicht zu lesen: Um die Länge einer Liste zu zählen, zählen wir das erste Element und dann die Länge des Restes der Liste. Wir können diese Methode ähnlich testen, wie wir es getestet `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
```

Nächste Schritte

Dieses Spielzeugbeispiel ist ziemlich weit davon entfernt, alle Funktionen zu implementieren, die in einer verknüpften Liste erwünscht wären. Es fehlt beispielsweise die Iterationsschnittstelle. Es zeigt jedoch, wie mit dem Versand kurzer und klarer Code geschrieben werden kann.

Unveränderliche Typen

Der einfachste zusammengesetzte Typ ist ein unveränderlicher Typ. Instanzen unveränderlicher Typen wie Tupel sind Werte. Ihre Felder können nicht geändert werden, nachdem sie erstellt wurden. In vielerlei Hinsicht ähnelt ein unveränderlicher Typ einem `Tuple` mit Namen für den Typ selbst und für jedes Feld.

Singleton-Typen

Zusammengesetzte Typen enthalten per Definition eine Anzahl einfacherer Typen. In Julia kann diese Zahl Null sein. Das heißt, ein unveränderlicher Typ darf *keine* Felder enthalten. Dies ist vergleichbar mit dem leeren Tupel `()` .

Warum kann das nützlich sein? Solche unveränderlichen Typen werden als "Singleton-Typen" bezeichnet, da nur eine Instanz davon existieren könnte. Die Werte solcher Typen werden als "Singleton-Werte" bezeichnet. Die Standardbibliothek `Base` enthält viele solcher Singleton-Typen. Hier ist eine kurze Liste:

- `Void` , die Art von `nothing` . Wir können überprüfen, dass `Void.instance` (die spezielle Syntax zum Abrufen des Singleton-Werts eines Singleton-Typs) tatsächlich `nothing` .
- Jeder Medientyp wie `MIME"text/plain"` ist ein Singleton-Typ mit einer einzigen Instanz, `MIME("text/plain")` .
- `Irrational{:π}` , `Irrational{:e}` , `Irrational{:ψ}` und ähnliche Typen sind Singleton-Typen, und ihre Singleton-Instanzen sind die irrationalen Werte `π = 3.1415926535897...` usw.
- Die Iteratorgrößenmerkmale `Base.HasLength` , `Base.HasShape` , `Base.IsInfinite` und `Base.SizeUnknown` sind alle Singleton-Typen.

0,5,0

- In Version 0.5 und höher ist jede Funktion eine Singleton-Instanz eines Singleton-Typs! Wie jeder andere Einzelwert können wir die Funktion `sin` beispielsweise aus `typeof(sin).instance` .

Da sie nichts enthalten, sind Singleton-Typen unglaublich leicht und können vom Compiler häufig dahin gehend optimiert werden, dass sie keinen Laufzeit-Overhead haben. Daher sind sie perfekt für Merkmale, spezielle Tag-Werte und für Funktionen wie Funktionen, auf die Sie sich spezialisieren möchten.

Um einen Singleton-Typ zu definieren,

```
julia> immutable MySingleton end
```

So definieren Sie einen benutzerdefinierten Druck für den Singleton-Typ

```
julia> Base.show(io::IO, ::MySingleton) = print(io, "sing")
```

Um auf die Singleton-Instanz zuzugreifen,

```
julia> MySingleton.instance
MySingleton()
```

Oft ordnet man dies einer Konstanten zu:

```
julia> const sing = MySingleton.instance
MySingleton()
```

Wrapper-Typen

Wenn unveränderliche Nullfeldtypen interessant und nützlich sind, sind möglicherweise unveränderliche Einfeldtypen möglicherweise noch nützlicher. Solche Typen werden im

Allgemeinen als "Wrapper-Typen" bezeichnet, da sie einige zugrunde liegende Daten umschließen, wodurch eine alternative Schnittstelle zu den Daten bereitgestellt wird. Ein Beispiel für einen Wrapper-Typ in `Base` ist `String` . Wir definieren einen ähnlichen `String` `MyString` . Dieser Typ wird durch einen Vektor (eindimensionales Array) von Bytes (`UInt8`) `UInt8` .

Zuerst die Typdefinition selbst und einige angepasste Anzeigen:

```
immutable MyString <: AbstractString
    data::Vector{UInt8}
end

function Base.show(io::IO, s::MyString)
    print(io, "MyString: ")
    write(io, s.data)
    return
end
```

Jetzt ist unser `MyString` Typ einsatzbereit! Wir können ihm einige unreife UTF-8-Daten zuführen, und es zeigt an, wie es uns gefällt:

```
julia> MyString([0x48,0x65,0x6c,0x6c,0x6f,0x2c,0x20,0x57,0x6f,0x72,0x6c,0x64,0x21])
MyString: Hello, World!
```

Offensichtlich erfordert dieser String-Typ eine Menge Arbeit, bevor er so nutzbar ist wie der `Base.String` Typ.

Echte zusammengesetzte Typen

Am häufigsten enthalten viele unveränderliche Typen mehr als ein Feld. Ein Beispiel ist die Standardbibliothek `Rational{T}` , die zwei Felder enthält: ein `num` Feld für den Zähler und ein `den` Feld für den Nenner. Es ist ziemlich einfach, diesen Typ zu emulieren:

```
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)
```

Wir haben erfolgreich einen Konstruktor implementiert, der unsere rationalen Zahlen vereinfacht:

```
julia> MyRational(10, 6)
MyRational{Int64}(5,3)
```

Typen online lesen: https://riptutorial.com/de/julia-lang/topic/5467/typen

Kapitel 28: Unit Testing

Syntax

- @test [Ausdruck]
- @test_throws [Ausnahme] [Ausdruck]
- @testset "[Name]" begin; [Tests]; Ende
- Pkg.test ([Paket])

Bemerkungen

Die Standard-Bibliotheksdokumentation für `Base.Test` umfasst zusätzliches Material, das über das in diesen Beispielen gezeigte hinausgeht.

Examples

Paket testen

Verwenden Sie zum `Pkg.test` der `Pkg.test` für ein Paket die Funktion `Pkg.test` . Für ein Paket mit dem Namen `MyPackage` der Befehl

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

Eine erwartete Ausgabe wäre ähnlich

```
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
```

Natürlich kann man nicht erwarten, dass es genau mit den oben genannten übereinstimmt, da verschiedene Pakete unterschiedliche Frameworks verwenden.

Dieser Befehl führt die Datei `test/runtests.jl` des Pakets in einer sauberen Umgebung aus.

Mit können Sie alle installierten Pakete gleichzeitig testen

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

Dies dauert jedoch normalerweise sehr lange.

Einen einfachen Test schreiben

Unit-Tests werden in der Datei `test/runtests.jl` in einem Paket deklariert. Normalerweise beginnt diese Datei

```
using MyModule
using Base.Test
```

Die grundlegende `@test` ist das `@test` Makro. Dieses Makro ist eine Art Behauptung. Jeder boolesche Ausdruck kann im `@test` Makro getestet werden:

```
@test 1 + 1 == 2
@test iseven(10)
@test 9 < 10 || 10 < 9
```

Wir können das `@test` Makro in der REPL ausprobieren:

```
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
```

Das Testmakro kann fast überall verwendet werden, beispielsweise in Schleifen oder Funktionen:

```
# 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)
```

Test-Set schreiben

0,5,0

In Version v0.5 sind Testsätze in die Standardbibliothek `Base.Test` , und Sie müssen keine besonderen `using Base.Test` (außer `using Base.Test`), um sie verwenden zu können.

0,4,0

Test - Sets sind nicht Teil der von Julia v0.4 `Base.Test` Bibliothek. Stattdessen müssen Sie `REQUIRE` das `BaseTestNext` Modul, und fügen Sie `using BaseTestNext` zu Ihrer Datei. Um beide Versionen 0.4 und 0.5 zu unterstützen, können Sie verwenden

```
if VERSION ≥ v"0.5.0-dev+7720"
    using Base.Test
else
    using BaseTestNext
    const Test = BaseTestNext
end
```

Es ist hilfreich, verwandte `@test` Tests in einem Test-Set zusammenzufassen. Zusätzlich zu einer übersichtlicheren Testorganisation bieten Testsets eine bessere Ausgabe und mehr Anpassbarkeit.

Um ein `@test` zu definieren, wickeln Sie einfach eine beliebige Anzahl von `@test` mit einem `@testset` Block ein:

```
@testset "+" begin
    @test 1 + 1 == 2
    @test 2 + 2 == 4
end

@testset "*" begin
    @test 1 * 1 == 1
    @test 2 * 2 == 4
end
```

Beim Ausführen dieser Testsätze wird die folgende Ausgabe gedruckt:

```
Test Summary: | Pass  Total
  +           |    2      2

Test Summary: | Pass  Total
  *           |    2      2
```

Selbst wenn ein Testsatz einen fehlerhaften Test enthält, wird der gesamte Testsatz vollständig ausgeführt und die Fehler werden aufgezeichnet und gemeldet:

```
@testset "-" begin
    @test 1 - 1 == 0
    @test 2 - 2 == 1
    @test 3 - () == 3
    @test 4 - 4 == 0
end
```

Das Ausführen dieses Testsatzes führt zu

```
-: 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.
    ...
```

Testsätze können geschachtelt werden, um eine beliebig tiefe Organisation zu ermöglichen

```
@testset "Int" begin
    @testset "+" begin
        @test 1 + 1 == 2
        @test 2 + 2 == 4
    end
    @testset "-" begin
        @test 1 - 1 == 0
    end
end
```

Wenn die Tests erfolgreich sind, werden nur die Ergebnisse für den äußersten Testsatz angezeigt:

```
Test Summary: | Pass  Total
  Int         |    3      3
```

Wenn die Tests fehlschlagen, wird ein Drilldown in den genauen Testsatz und den Test, der den Fehler verursacht, gemeldet.

Das `@testset` Makro kann mit einer `for` Schleife verwendet werden , um viele Testsätze gleichzeitig zu erstellen:

```
@testset for i in 1:5
    @test 2i == i + i
    @test i^2 == i * i
    @test i ÷ i == 1
end
```

welche berichtet

```
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
```

Eine übliche Struktur besteht darin, äußere Testsätze Komponenten oder Typen prüfen zu lassen. Innerhalb dieser äußeren Testsätze wird durch das innere Testverhalten das Testverhalten festgelegt. Angenommen, wir haben einen Typ `UniversalSet` mit einer Einzelinstanz erstellt, die alles enthält. Bevor wir den Typ überhaupt implementieren, können wir testgetriebene Entwicklungsprinzipien anwenden und die Tests implementieren:

```
@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
```

Wir können dann mit der Implementierung unserer Funktionalität beginnen, bis unsere Tests bestanden sind. Der erste Schritt ist die Definition des Typs:

```
immutable UniversalSet <: Base.AbstractSet end
```

Nur zwei unserer Tests bestehen im Moment. Wir können implementieren `in` :

```
immutable UniversalSet <: Base.AbstractSet end
Base.in(x, ::UniversalSet) = true
```

Damit sind auch einige unserer Teilmengenprüfungen erfolgreich. Das `issubset` (`⊆`) -Fallback funktioniert jedoch nicht für `UniversalSet` , da das Fallback versucht, Elemente zu durchlaufen, was wir nicht tun können. Wir können einfach eine Spezialisierung definieren, die `issubset` , dass `issubset` für jede Menge `true` :

```
immutable UniversalSet <: Base.AbstractSet end
Base.in(x, ::UniversalSet) = true
Base.issubset(x::Base.AbstractSet, ::UniversalSet) = true
```

Und jetzt bestehen alle unsere Tests!

Ausnahmen testen

Ausnahmen, die während eines Tests aufgetreten sind, schlagen fehl, und wenn der Test nicht in einem Testsatz enthalten ist, beenden Sie den Testmotor. Dies ist normalerweise eine gute Sache, da in den meisten Fällen Ausnahmen nicht das gewünschte Ergebnis sind. Aber manchmal möchte man speziell testen, dass eine bestimmte Ausnahme ausgelöst wird. Das Makro `@test_throws` erleichtert dies.

```
julia> @test_throws BoundsError [1, 2, 3][4]
Test Passed
  Expression: ([1,2,3])[4]
      Thrown: BoundsError
```

Wenn die falsche Ausnahme ausgelöst wird, `@test_throws` immer noch fehl:

```
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
```

und wenn keine Ausnahme ausgelöst wird, `@test_throws` ebenfalls fehl:

```
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
```

Prüfung des Fließpunkts Ungefährer Gleichwert

Was ist der Deal mit dem Folgenden?

```
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
```

Der Fehler wird durch die Tatsache verursacht, dass keine der Werte `0.1` , `0.2` und `0.3` im Computer als genau die Werte dargestellt werden - `1//10` , `2//10` und `3//10` . Sie werden stattdessen durch sehr nahe Werte angeglichen. Wie aus dem obigen Testfehler hervorgeht, kann

das Ergebnis beim Hinzufügen von zwei Näherungen eine etwas schlechtere Näherung sein, als dies möglich ist. Es gibt noch viel mehr zu diesem Thema , das hier nicht behandelt werden kann.

Aber wir haben kein Glück! Um zu testen, ob die Kombination aus Rundung auf eine Fließkommazahl und Fließkomma-Arithmetik *annähernd* korrekt ist, auch wenn sie nicht genau ist, können wir die Funktion `isapprox` (die dem Operator `≈`) verwenden. So können wir unseren Test als neu schreiben

```
julia> @test 0.1 + 0.2 ≈ 0.3
Test Passed
  Expression: 0.1 + 0.2 ≈ 0.3
   Evaluated: 0.30000000000000004 isapprox 0.3
```

Wenn unser Code völlig falsch war, wird der Test das natürlich trotzdem feststellen:

```
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
```

Die `isapprox` Funktion verwendet Heuristiken basierend auf der Größe der Zahlen und der Genauigkeit des Gleitkommatyps, um die zu tolerierende Fehlermenge zu bestimmen. Es ist nicht für alle Situationen geeignet, funktioniert aber in den meisten `isapprox` und erspart viel Aufwand bei der Implementierung der eigenen Version von `isapprox` .

Unit Testing online lesen: https://riptutorial.com/de/julia-lang/topic/5632/unit-testing

Kapitel 29: Vergleiche

Syntax

- x <y #, wenn `x` streng weniger als `y`
- x> y # wenn `x` streng größer als `y`
- x == y # wenn `x` gleich `y`
- x === y # alternativ `x ≡ y` , wenn `x` für `y` egal ist
- x ≤ y # alternativ `x <= y` , wenn `x` kleiner oder gleich `y`
- x ≥ y # alternativ `x >= y` , wenn `x` größer oder gleich `y`
- x ≠ y # alternativ `x != y` , wenn `x` nicht gleich `y`
- x ≈ y #, wenn `x` ungefähr gleich `y`

Bemerkungen

Achten Sie darauf, die Vergleichszeichen umzudrehen. Julia definiert standardmäßig viele Vergleichsfunktionen, ohne die entsprechende umgedrehte Version zu definieren. Zum Beispiel kann man laufen

```
julia> Set(1:3) ⊆ Set(0:5)
true
```

aber es funktioniert nicht

```
julia> Set(0:5) ⊇ Set(1:3)
ERROR: UndefVarError: ⊇ not defined
```

Examples

Verkettete Vergleiche

Mehrere miteinander kombinierte Vergleichsoperatoren werden wie über den Operator `&&` verbunden miteinander verkettet. Dies kann nützlich sein für lesbare und mathematisch präzise Vergleichsketten, wie z

```
# same as 0 < i && i <= length(A)
isinbounds(A, i)       = 0 < i ≤ length(A)

# same as Set() != x && issubset(x, y)
isnonemptysubset(x, y) = Set() ≠ x ⊆ y
```

Es gibt jedoch einen wichtigen Unterschied zwischen `a > b > c` und `a > b && b > c` ; in letzterer wird der Begriff `b` zweimal bewertet. Dies spielt für einfache alte Symbole keine Rolle, könnte jedoch von Bedeutung sein, wenn die Begriffe selbst Nebenwirkungen haben. Zum Beispiel,

```
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
```

Schauen wir uns die verketteten Vergleiche und ihre Funktionsweise genauer an, indem wir sehen, wie sie analysiert und in Ausdrücke abgesenkt werden. Betrachten wir zunächst den einfachen Vergleich, den wir als einfachen alten Funktionsaufruf sehen:

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

Wenn wir nun den Vergleich verketten, stellen wir fest, dass sich das Parsing geändert hat:

```
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
```

Nach dem Parsen wird der Ausdruck in seine endgültige Form abgesenkt:

```
julia> expand(:(a > b >= c))
:(begin
        unless a > b goto 3
        return b >= c
        3:
        return false
    end)
```

und wir stellen fest, dass dies dasselbe ist wie für `a > b && b >= c` :

```
julia> expand(:(a > b && b >= c))
:(begin
        unless a > b goto 3
        return b >= c
        3:
```

```
        return false
    end)
```

Ordnungszahlen

Wir werden untersuchen, wie Sie benutzerdefinierte Vergleiche implementieren, indem Sie einen benutzerdefinierten Typ Ordnungszahlen implementieren. Um die Implementierung zu vereinfachen, konzentrieren wir uns auf eine kleine Teilmenge dieser Zahlen: alle Ordnungszahlen bis, jedoch nicht einschließlich ε_0. Unsere Implementierung konzentriert sich auf Einfachheit und nicht auf Geschwindigkeit. Die Implementierung ist jedoch auch nicht langsam.

Wir speichern Ordinalzahlen in ihrer normalen Cantor-Form . Da die ordinale Arithmetik nicht kommutativ ist, verwenden wir die übliche Konvention, zuerst die wichtigsten Bezeichnungen zu speichern.

```
immutable OrdinalNumber <: Number
    βs::Vector{OrdinalNumber}
    cs::Vector{Int}
end
```

Da die Cantor-Normalform einzigartig ist, können wir Gleichheit einfach durch rekursive Gleichheit testen:

0,5,0

In Version v0.5 gibt es eine sehr schöne Syntax, um dies kompakt zu machen:

```
import Base: ==
α::OrdinalNumber == β::OrdinalNumber = α.βs == β.βs && α.cs == β.cs
```

0,5,0

Andernfalls definieren Sie die Funktion wie üblich:

```
import Base: ==
==(α::OrdinalNumber, β::OrdinalNumber) = α.βs == β.βs && α.cs == β.cs
```

Um unsere Bestellung `isless` , sollten Sie die `isless` Funktion überladen, da dieser Typ eine Gesamtbestellung hat.

```
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
    return length(α.cs) < length(β.cs)
end
```

Um unsere Reihenfolge zu testen, können wir einige Methoden erstellen, um Ordnungszahlen zu erstellen. Null wird natürlich dadurch erhalten, dass keine Terme in der Cantor-Normalform vorliegen:

```
const ORDINAL_ZERO = OrdinalNumber([], [])
Base.zero(::Type{OrdinalNumber}) = ORDINAL_ZERO
```

Wir können ein `expω` definieren, um `ω^α` zu berechnen, und daraus 1 und ω berechnen:

```
expω(α) = OrdinalNumber([α], [1])
const ORDINAL_ONE = expω(ORDINAL_ZERO)
Base.one(::Type{OrdinalNumber}) = ORDINAL_ONE
const ω = expω(ORDINAL_ONE)
```

Wir haben jetzt eine voll funktionsfähige Ordnungsfunktion für Ordnungszahlen:

```
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[])],[


OrdinalNumber(OrdinalNumber[OrdinalNumber(OrdinalNumber[OrdinalNumber(OrdinalNumber[OrdinalNumber
```

Im letzten Beispiel sehen wir, dass das Drucken von Ordnungszahlen besser sein könnte, aber das Ergebnis ist wie erwartet.

Standardoperatoren

Julia unterstützt sehr viele Vergleichsoperatoren. Diese schließen ein

1. Alle folgenden Unicode-Sequenzen: `> < >= ≥ <= ≤ == === ≡ != ≠ !== ≢ ∈ ∉ ∋ ∌ ⊆ ⊈ ⊂ ⊄ ⊊ ∝ ∊ ∍ ∥ ∦ ∷ ∺ ∻ ∽ ∾ ≁ ≃ ≄ ≅ ≆ ≇ ≈ ≉ ≊ ≋ ≌ ≍ ≎ ≐ ≑ ≒ ≓ ≔ ≕ ≖ ≗ ≘ ≙ ≚ ≛ ≜ ≝ ≞ ≟ ≣ ≦ ≧ ≨ ≩ ≪ ≫ ≬ ≭ ≮ ≯ ≰ ≱ ≲ ≳ ≴ ≵ ≶ ≷ ≸ ≹ ≺ ≻ ≼ ≽ ≾ ≿ ⊀ ⊁ ⊃ ⊅ ⊇ ⊉ ⊋ ⊏ ⊐ ⊑ ⊒ ⊜ ⊩ ⊬ ⊮ ⊰ ⊱ ⊲ ⊳ ⊴ ⊵ ⊶ ⊷ ⋍ ⋐ ⋑ ⋕ ⋖ ⋗ ⋘ ⋙ ⋚ ⋛ ⋜ ⋝ ⋞ ⋟ ⋠ ⋡ ⋢ ⋣ ⋤ ⋥ ⋦ ⋧ ⋨ ⋩ ⋪ ⋫ ⋬ ⋭ ⋲ ⋳ ⋴ ⋵ ⋶ ⋷ ⋸ ⋹ ⋺ ⋻ ⋼ ⋽ ⋾ ⋿ ⟒ ⦷ ⧀ ⧁ ⧡ ⧣ ⧤ ⧥ ⩦ ⩧ ⩪ ⩫ ⩬ ⩭ ⩮ ⩯ ⩰ ⩱ ⩲ ⩳ ⩵ ⩶ ⩷ ⩸ ⩹ ⩺ ⩻ ⩼ ⩽ ⩾ ⩿ ⪀ ⪁ ⪂ ⪃ ⪄ ⪅ ⪆ ⪇ ⪈ ⪉ ⪊ ⪋ ⪌ ⪍ ⪎ ⪏ ⪐ ⪑ ⪒ ⪓ ⪔ ⪕ ⪖ ⪗ ⪘ ⪙ ⪚ ⪛ ⪜ ⪝ ⪞ ⪟ ⪠ ⪡ ⪢ ⪣ ⪤ ⪥ ⪦ ⪧ ⪨ ⪩ ⪪ ⪫ ⪬ ⪭ ⪮ ⪯ ⪰ ⪱ ⪲ ⪳ ⪴ ⪵ ⪶ ⪷ ⪸ ⪹ ⪺ ⪻ ⪼ ⪽ ⪾ ⪿ ⫀ ⫁ ⫂ ⫃ ⫄ ⫅ ⫆ ⫇ ⫈ ⫉ ⫊ ⫋ ⫌ ⫍ ⫎ ⫏ ⫐ ⫑ ⫒ ⫓ ⫔ ⫕ ⫖ ⫗ ⫘ ⫙ ⫷ ⫸ ⫹ ⫺ ⊢ ⊣`
2. Alle Symbole in Punkt 1, vorangestellt von einem Punkt (`.`), Der elementweise erstellt werden soll;
3. Die Operatoren `<:` , `>:` , `.!` und `in` , dem kein Punkt (`.`) vorangestellt werden kann.

Nicht alle haben eine Definition in der Standard- `Base` Bibliothek. Sie sind jedoch für andere Pakete verfügbar, um sie entsprechend zu definieren und zu verwenden.

Im täglichen Gebrauch sind die meisten dieser Vergleichsoperatoren nicht relevant. Die am häufigsten verwendeten sind die mathematischen Standardfunktionen für die Reihenfolge; Eine Liste finden Sie im Abschnitt Syntax.

Wie die meisten anderen Operatoren in Julia sind Vergleichsoperatoren Funktionen und können als Funktionen aufgerufen werden. Zum Beispiel ist `(<)(1, 2)` in der Bedeutung mit `1 < 2` identisch.

Verwenden Sie ==, === und ist gleich

Es gibt drei Gleichheitsoperatoren: `==` , `===` und `isequal` . (Der letzte ist nicht wirklich ein Operator, aber es ist eine Funktion und alle Operatoren sind Funktionen.)

Wann verwenden `==`

`==` ist *Wertgleichheit.* Sie gibt `true` zurück `true` wenn zwei Objekte in ihrem aktuellen Zustand denselben Wert darstellen.

Zum Beispiel ist es offensichtlich, dass

```
julia> 1 == 1
true
```

aber darüber hinaus

```
julia> 1 == 1.0
true

julia> 1 == 1.0 + 0.0im
true

julia> 1 == 1//1
true
```

Die rechte Seite jeder obigen Gleichheit ist von einem anderen Typ , sie repräsentieren jedoch immer noch denselben Wert.

Bei veränderlichen Objekten wie Arrays vergleicht `==` ihren aktuellen Wert.

```
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
```

Meistens ist `==` die richtige Wahl.

Wann verwenden `===`

`===` ist eine weitaus strengere Operation als `==` . Anstelle von Wertgleichheit misst es die Egalität. Zwei Objekte sind egal, wenn sie vom Programm selbst nicht unterschieden werden können. So haben wir

```
julia> 1 === 1
true
```

da es keine Möglichkeit gibt, eine `1` von einer anderen `1` . Aber

```
julia> 1 === 1.0
false
```

Obwohl `1` und `1.0` den gleichen Wert haben, unterscheiden sie sich, und das Programm kann sie unterscheiden.

Außerdem,

```
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
```

was auf den ersten Blick überraschend erscheinen mag! Wie konnte das Programm zwischen den beiden Vektoren `A` und `B` ? Da Vektoren veränderlich sind, können sie `A` modifizieren und verhalten sich dann anders als `B` Unabhängig davon, wie `A` geändert wird, verhält sich `A` immer wie `A` selbst. Also ist `A` egal zu `A` , aber nicht egal zu `B`

Weiter entlang dieser Ader beobachten

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

julia> A === C
true
```

Durch die Zuordnung von `A` bis `C` , sagen wir , dass `C` *aliased* hat `A` . Das heißt, es ist nur ein anderer Name für `A` . Alle Änderungen an `A` werden von `C` ebenfalls beachtet. Daher gibt es keine Möglichkeit, den Unterschied zwischen `A` und `C` erkennen. Sie sind also egal.

Wann verwendet man `isequal`

Der Unterschied zwischen `==` und `isequal` ist sehr subtil. Der größte Unterschied besteht im Umgang mit Gleitkommazahlen:

```
julia> NaN == NaN
false
```

Dies möglicherweise raschendes Ergebnis ist definiert durch den IEEE - Standard für Gleitkomma - Typen (IEEE-754). Dies ist jedoch in einigen Fällen nicht sinnvoll, beispielsweise beim Sortieren. `isequal` wird für diese Fälle bereitgestellt:

```
julia> isequal(NaN, NaN)
true
```

Auf der anderen Seite des Spektrums behandelt `==` IEEE-negative Null und positive Null als denselben Wert (auch wie von IEEE-754 angegeben). Diese Werte haben jedoch unterschiedliche

Repräsentationen im Speicher.

```
julia> 0.0
0.0

julia> -0.0
-0.0

julia> 0.0 == -0.0
true
```

Auch `isequal` unterscheidet sie sich zu Sortierzwecken.

```
julia> isequal(0.0, -0.0)
false
```

Vergleiche online lesen: https://riptutorial.com/de/julia-lang/topic/5563/vergleiche

Kapitel 30: Verschlüsse

Syntax

- x -> [Körper]
- (x, y) -> [Körper]
- (xs ...) -> [Körper]

Bemerkungen

0,4,0

In älteren Versionen von Julia hatten Schließungen und anonyme Funktionen einen Performance-Nachteil für die Laufzeit. Diese Strafe wurde in 0.5 eliminiert.

Examples

Funktionszusammensetzung

Wir können eine Funktion definieren, um die Funktionszusammenstellung mit anonymer Funktionssyntax durchzuführen:

```
f ∘ g = x -> f(g(x))
```

Beachten Sie, dass diese Definition den folgenden Definitionen entspricht:

```
∘(f, g) = x -> f(g(x))
```

oder

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

unter Hinweis darauf, dass in Julia `f ∘ g` nur Syntaxzucker für `∘(f, g)` .

Wir können sehen, dass diese Funktion richtig komponiert:

```
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

In Version v0.5 ist diese Definition sehr performant. Wir können den erzeugten LLVM-Code untersuchen:

```
julia> @code_llvm sextuple(1)

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

Es ist klar, dass die beiden Multiplikationen zu einer einzigen Multiplikation zusammengefasst wurden und dass diese Funktion so effizient wie möglich ist.

Wie funktioniert diese Funktion höherer Ordnung? Es erstellt einen sogenannten Abschluss , der nicht nur aus dem Code besteht, sondern auch bestimmte Variablen aus ihrem Gültigkeitsbereich heraus verfolgt. Alle Funktionen in Julia, die nicht im Top-Level-Bereich erstellt werden, sind Schließungen.

0,5,0

Man kann die verschlossenen Variablen durch die Felder der Schließung überprüfen. Zum Beispiel sehen wir das:

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

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

Currying implementieren

Eine Anwendung von Verschlüssen ist die teilweise Anwendung einer Funktion; Geben Sie jetzt einige Argumente an und erstellen Sie eine Funktion, die die restlichen Argumente übernimmt. Currying ist eine spezifische Form der Teilanwendung.

Beginnen wir mit der einfachen Funktion `curry(f, x)` , die das erste Argument für eine Funktion bereitstellt, und erwartet später weitere Argumente. Die Definition ist ziemlich einfach:

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

Wieder verwenden wir eine anonyme Funktionssyntax , diesmal in Kombination mit einer variadischen Argumentsyntax.

Mit dieser `curry` Funktion können wir einige grundlegende Funktionen implizit (oder punktfrei)

implementieren.

```
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?
```

Funktionen erhalten den erwarteten Generismus aufrecht:

```
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
```

Einführung in die Verschlüsse

Funktionen sind ein wichtiger Bestandteil der Julia-Programmierung. Sie können direkt in Modulen definiert werden. In diesem Fall werden die Funktionen als *Top-Level bezeichnet* . Funktionen können aber auch in anderen Funktionen definiert werden. Solche Funktionen werden " Schließungen " genannt.

Verschlüsse erfassen die Variablen in ihrer äußeren Funktion. Eine Funktion der obersten Ebene kann nur globale Variablen aus ihren Modul-, Funktionsparametern oder lokalen Variablen verwenden:

```
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
```

Ein Abschluss dagegen kann alle zusätzlich zu den Variablen der äußeren Funktionen verwenden, die er erfasst:

```
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
```

Wenn wir `c = toplevel(10)` , sehen wir das Ergebnis

```
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)
```

Beachten Sie, dass der Endausdruck dieser Funktion eine Funktion an sich ist. das ist eine Schließung. Wir können die Schließung `c` wie jede andere Funktion nennen:

```
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)
```

Beachten Sie, dass `c` immer noch Zugriff auf die Variablen `y` und `z` aus dem Aufruf der `toplevel` `toplevel` hat - obwohl `toplevel` bereits zurückgekehrt ist! Jeder Abschluss, auch der von derselben Funktion zurückgegebene, schließt verschiedene Variablen. Wir können wieder `toplevel` anrufen

```
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)
```

Beachten Sie, dass, obwohl `d` und `c` denselben Code haben und dieselben Argumente übergeben werden, deren Ausgabe unterschiedlich ist. Sie sind unterschiedliche Verschlüsse.

Kapitel 31: Versionsübergreifende Kompatibilität

Syntax

- mit Compat
- Compat.String
- Compat.UTF8String
- @compat f. (x, y)

Bemerkungen

Es ist manchmal sehr schwierig, eine neue Syntax für mehrere Versionen zu erhalten. Da sich Julia noch in der aktiven Entwicklung befindet, ist es oft nützlich, die Unterstützung für ältere Versionen einfach einzustellen und stattdessen nur die neueren zu verwenden.

Examples

Versionsnummern

Julia verfügt über eine integrierte Implementierung der semantischen Versionierung, die durch den `VersionNumber` Typ `VersionNumber` .

Um eine `VersionNumber` als Literal zu `@v_str` , kann das `@v_str` String-Makro verwendet werden:

```
julia> vers = v"1.2.0"
v"1.2.0"
```

Alternativ kann man den `VersionNumber` Konstruktor aufrufen. Beachten Sie, dass der Konstruktor bis zu fünf Argumente akzeptiert, alle außer dem ersten sind jedoch optional.

```
julia> vers2 = VersionNumber(1, 1)
v"1.1.0"
```

Versionsnummern können mit Vergleichsoperatoren verglichen und somit sortiert werden:

```
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"
```

Versionsnummern werden an verschiedenen Stellen in Julia verwendet. Beispielsweise ist die `VERSION` Konstante eine `VersionNumber` :

```
julia> VERSION
v"0.5.0"
```

Dies wird normalerweise für die bedingte Codeauswertung verwendet, abhängig von der Julia-Version. Um zum Beispiel anderen Code auf v0.4 und v0.5 auszuführen, kann man dies tun

```
if VERSION < v"0.5"
    println("v0.5 prerelease, v0.4 or older")
else
    println("v0.5 or newer")
end
```

Jedes installierte Paket ist auch mit einer aktuellen Versionsnummer verknüpft:

```
julia> Pkg.installed("StatsBase")
v"0.9.0"
```

Compat.jl verwenden

Das Compat.jl-Paket ermöglicht die Verwendung einiger neuer Julia-Funktionen und -Syntax mit älteren Versionen von Julia. Seine Funktionen sind in der README-Dokumentation dokumentiert. Nachfolgend finden Sie eine Zusammenfassung nützlicher Anwendungen.

0,5,0

Einheitlicher String-Typ

In Julia v0.4 gab es viele verschiedene Arten von Saiten . Dieses System wurde als zu komplex und verwirrend angesehen, sodass in Julia v0.5 nur noch der `String` Typ erhalten bleibt. `Compat` erlaubt die Verwendung des `String` Typs und `Compat.String` in Version 0.4 unter dem Namen `Compat.String` . Zum Beispiel dieser Code der Version 0.5

```
buf = IOBuffer()
println(buf, "Hello World!")
String(buf)  # "Hello World!\n"
```

kann direkt in diesen Code übersetzt werden, der für v0.5 und v0.4 funktioniert:

```
using Compat
buf = IOBuffer()
println(buf, "Hello World!")
Compat.String(buf)  # "Hello World!\n"
```

Beachten Sie, dass es einige Einschränkungen gibt.

- Auf v0.4, `Compat.String` ist typealiased `ByteString` , was `Union{ASCIIString, UTF8String}` . Daher sind Typen mit `String` Feldern nicht typstabil. In diesen Situationen wird `Compat.UTF8String` empfohlen, da dies `String` in Version 0.5 und `UTF8String` in Version `UTF8String` bedeutet. Beide Typen sind konkrete Typen.
- Man muss vorsichtig sein, `Compat.String` zu verwenden oder `import Compat: String` , da `String` selbst in Version 0.4 eine Bedeutung hat: Es ist ein veralteter Alias für `AbstractString` . Ein Zeichen, dass `String` versehentlich anstelle von `Compat.String` ist, wenn zu irgendeinem Zeitpunkt die folgenden Warnungen `Compat.String` :

```
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
```

Kompakte Broadcasting-Syntax

Julia v0.5 führt syntaktischen Zucker für die `broadcast` . Die Syntax

```
f.(x, y)
```

wird auf `broadcast(f, x, y)` abgesenkt. Beispiele für die Verwendung dieser Syntax sind `sin.([1, 2, 3])` , um den Sinus mehrerer Zahlen gleichzeitig zu erfassen.

In Version 0.5 kann die Syntax direkt verwendet werden:

```
julia> sin.([1.0, 2.0, 3.0])
3-element Array{Float64,1}:
 0.841471
 0.909297
 0.14112
```

Wenn wir dasselbe auf Version 0.4 versuchen, wird jedoch ein Fehler angezeigt:

```
julia> sin.([1.0, 2.0, 3.0])
ERROR: TypeError: getfield: expected Symbol, got Array{Float64,1}
```

Glücklicherweise macht `Compat` diese neue Syntax ab Version 0.4 auch nutzbar. Wieder fügen wir `using Compat` . Diesmal umgeben wir den Ausdruck mit dem `@compat` Makro:

```
julia> using Compat

julia> @compat sin.([1.0, 2.0, 3.0])
3-element Array{Float64,1}:
 0.841471
 0.909297
 0.14112
```

Versionsübergreifende Kompatibilität online lesen: https://riptutorial.com/de/julia-lang/topic/5832/versionsubergreifende-kompatibilitat

Kapitel 32: Verständnis

Examples

Array-Verständnis

Grundlegende Syntax

Julias Array-Verständnis verwendet die folgende Syntax:

```
[expression for element = iterable]
```

Beachten Sie, dass wie bei `for` Schleifen alle `=` , `in` und `∈` für das Verständnis akzeptiert werden.

Dies entspricht in etwa der Erstellung eines leeren Arrays und der Verwendung einer `for` Schleife zum `push!` Artikel dazu.

```
result = []
for element in iterable
    push!(result, expression)
end
```

Die Art eines Array-Verständnisses ist jedoch so eng wie möglich, was für die Leistung besser ist.

Um beispielsweise ein Feld der Quadrate der Ganzzahlen von `1` bis `10` , kann der folgende Code verwendet werden.

```
squares = [x^2 for x=1:10]
```

Dies ist ein sauberer, prägnanter Ersatz für die längere Version `for` -loop.

```
squares = []
for x in 1:10
    push!(squares, x^2)
end
```

Bedingtes Array-Verständnis

Vor dem Julia 0.5 gibt es keine Möglichkeit, Bedingungen innerhalb der Array-Verhältnisse zu verwenden. Aber es stimmt nicht mehr. In Julia 0.5 können wir die Bedingungen innerhalb der folgenden Bedingungen verwenden:

```
julia> [x^2 for x in 0:9 if x > 5]
4-element Array{Int64,1}:
 36
 49
 64
```

```
 81
```

Quelle des obigen Beispiels finden Sie hier .

Wenn wir ein verschachteltes Listenverständnis verwenden möchten:

```
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)
```

Mehrdimensionale Arrayverstehen

Verschachtelte `for` Schleifen können verwendet werden, um mehrere eindeutige iterierbare Elemente zu durchlaufen.

```
result = []
for a = iterable_a
    for b = iterable_b
        push!(result, expression)
    end
end
```

In ähnlicher Weise können mehrere Iterationsspezifikationen zu einem Arrayverständnis geliefert werden.

```
[expression for a = iterable_a, b = iterable_b]
```

Beispielsweise kann das Folgende verwendet werden, um das kartesische Produkt von `1:3` und `1:2` zu erzeugen.

```
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)
```

Abgeflachte mehrdimensionale Array-Verhältnisse sind ähnlich, außer dass sie die Form verlieren. Zum Beispiel,

```
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)
```

ist eine abgeflachte Variante. Der syntaktische Unterschied besteht darin, dass anstelle eines Kommas ein zusätzliches `for` verwendet wird.

Generatorverständnisse

Generatorenverstehen haben ein ähnliches Format wie Arrayverstehen, verwenden jedoch Klammern `()` anstelle von eckigen Klammern `[]` .

```
(expression for element = iterable)
```

Ein solcher Ausdruck gibt ein `Generator` Objekt zurück.

```
julia> (x^2 for x = 1:5)
Base.Generator{UnitRange{Int64},##1#2}(#1,1:5)
```

Funktionsargumente

Generatorenverständnisse können als einziges Argument für eine Funktion bereitgestellt werden, ohne dass zusätzliche Klammern erforderlich sind.

```
julia> join(x^2 for x = 1:5)
"1491625"
```

Wenn jedoch mehr als ein Argument angegeben wird, benötigt das Generatorverständnis seinen eigenen Satz von Klammern.

```
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"
```

Verständnis online lesen: https://riptutorial.com/de/julia-lang/topic/5477/verstandnis

Kapitel 33: während Loops

Syntax

- während cond; Karosserie; Ende
- brechen
- fortsetzen

Bemerkungen

Die `while` Schleife hat keinen Wert. Obwohl es in Ausdrucksposition verwendet werden kann, ist der Typ `Void` und der erhaltene Wert ist `nothing` .

Examples

Collatz-Sequenz

Die `while` Schleife läuft so lange, bis die Bedingung erfüllt ist. Mit dem folgenden Code wird beispielsweise die Collatz-Sequenz aus einer bestimmten Anzahl berechnet und gedruckt:

```
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
```

Verwendungszweck:

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

Es ist möglich, jede Schleife rekursiv zu schreiben, und für komplexe `while` Schleifen ist manchmal die rekursive Variante klarer. In Julia haben Schleifen jedoch einige Vorteile gegenüber der Rekursion:

- Julia garantiert nicht die Beseitigung von Rückrufen, daher verwendet Rekursion zusätzlichen Speicher und kann zu Überlauffehlern führen.
- Aus demselben Grund kann eine Schleife den Overhead verringert haben und schneller laufen.

Einmal ausführen, bevor die Bedingung getestet wird

Manchmal möchte man einmal Initialisierungscode ausführen, bevor eine Bedingung getestet wird. In bestimmten anderen Sprachen hat diese Art von Schleife eine spezielle `do while` Syntax. Allerdings kann diese Syntax mit einer regelmäßigen ersetzt werden , `while` Schleife und `break` Aussage, so Julia hat keine spezielle `do` - `while` Syntax. Stattdessen schreibt man:

```
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
```

Beachten Sie, dass in manchen Situationen solche Schleifen bei Rekursion klarer sein könnten:

```
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
```

Breitensuche

0,5,0

(Obwohl dieses Beispiel mit der in Version v0.5 eingeführten Syntax geschrieben wurde, kann es auch mit einigen Änderungen an älteren Versionen funktionieren.)

Diese Implementierung der Breitensuche (BFS) in einem mit Nachbarschaftslisten dargestellten Graphen verwendet `while` Schleifen und die `return` Anweisung. Die Aufgabe, die wir lösen werden, lautet wie folgt: Wir haben eine Abfolge von Menschen und eine Abfolge von Freundschaften (Freundschaften sind gegenseitig). Wir möchten den Grad der Verbindung zwischen zwei Personen bestimmen. Das heißt, wenn zwei Leute Freunde sind, werden wir `1` ; Wenn einer von beiden befreundet ist, werden wir `2` zurückkehren und so weiter.

Nehmen wir zunächst an, wir haben bereits eine Adjazenzliste: ein `Dict T` auf `Array{T, 1}` `Dict` , wobei die Schlüssel Personen sind und die Werte alle Freunde dieser Person sind. Hier können wir Menschen mit dem von uns gewählten Typ `T` ; In diesem Beispiel verwenden wir `Symbol` . Im BFS-Algorithmus führen wir eine Reihe von Personen, die "Besuch" machen sollen, und markieren deren Entfernung vom Ursprungsknoten.

```
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
```

Nun schreiben wir eine Funktion, um eine Adjazenzliste mit einer Abfolge von Personen und einer Abfolge von `(person, person)` Tupeln zu erstellen:

```
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
```

Wir können jetzt die ursprüngliche Funktion definieren:

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

Lassen Sie uns nun unsere Funktion an einigen Daten testen.

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

```
    (:jean, :marius),
    (:cosette, :éponine),
    (:cosette, :marius),
    (:gavroche, :éponine)
]
```

Jean ist in `0` Schritten mit sich selbst verbunden:

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

Jean und Cosette sind Freunde und haben Grad `1` :

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

Jean und Gavroche sind indirekt durch Cosette und dann über Marius miteinander verbunden, so dass ihr Grad `3` beträgt:

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

Javert und Marius sind über keine Kette miteinander verbunden, daher wird ein Fehler ausgegeben:

```
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
```

während Loops online lesen: https://riptutorial.com/de/julia-lang/topic/5565/wahrend-loops

Kapitel 34: Wörterbücher

Examples

Wörterbücher verwenden

Wörterbücher können erstellt werden, indem beliebig viele Paare übergeben werden.

```
julia> Dict("A"=>1, "B"=>2)
Dict{String,Int64} with 2 entries:
  "B" => 2
  "A" => 1
```

Sie können Einträge in einem Wörterbuch erhalten, indem Sie den Schlüssel in eckige Klammern setzen.

```
julia> dict = Dict("A"=>1, "B"=>2)
Dict{String,Int64} with 2 entries:
  "B" => 2
  "A" => 1

julia> dict["A"]
1
```

Wörterbücher online lesen: https://riptutorial.com/de/julia-lang/topic/9028/worterbucher

Kapitel 35: Zeichenfolge-Makros

Syntax

- Makro "Zeichenfolge" # kurz, Zeichenfolge Makroform
- @macro_str "string" # Lange, reguläre Makroform
- Makro`Befehl`

Bemerkungen

String-Makros sind nicht ganz so leistungsfähig wie einfache alte Strings. Da Interpolation in der Logik des Makros implementiert werden muss, können String-Makros keine String-Literale desselben Trennzeichens für die Interpolation enthalten.

Zum Beispiel obwohl

```
julia> "$("x")"
"x"
```

funktioniert, das String-Makro-Textformular

```
julia> doc"$("x")"
ERROR: KeyError: key :x not found
```

wird falsch analysiert. Dies kann durch die Verwendung von Anführungszeichen als Begrenzungszeichen für die äußeren Zeichenketten etwas gemildert werden.

```
julia> doc"""$("x")"""
"x"
```

funktioniert ja richtig.

Examples

String-Makros verwenden

String-Makros sind syntaktischer Zucker für bestimmte Makroaufrufe. Der Parser erweitert die Syntax wie

```
mymacro"my string"
```

in

```
@mymacro_str "my string"
```

Dieser wird dann wie jeder andere Makroaufruf durch einen beliebigen Ausdruck ersetzt, den das `@mymacro_str` Makro zurückgibt. Base Julia verfügt über mehrere String-Makros wie:

`@b_str`

Dieses Zeichenfolgenmakro erstellt Byte- Arrays anstelle von Zeichenfolgen . Der Inhalt des Strings, der als UTF-8 codiert ist, wird als Byte-Array verwendet. Dies kann nützlich sein, um Schnittstellen mit APIs auf niedriger Ebene herzustellen, von denen viele mit Byte-Arrays anstelle von Zeichenfolgen arbeiten.

```
julia> b"Hello World!"
12-element Array{UInt8,1}:
 0x48
 0x65
 0x6c
 0x6c
 0x6f
 0x20
 0x57
 0x6f
 0x72
 0x6c
 0x64
 0x21
```

`@big_str`

Dieses Makro gibt ein `BigInt` oder ein `BigFloat` aus der angegebenen Zeichenfolge analysiert wird.

```
julia> big"1"
1

julia> big"1.0"
1.000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

Dieses Makro ist vorhanden, weil sich `big(0.1)` nicht so verhält, wie man es anfangs erwarten könnte: Die `0.1` ist eine `Float64` Approximation von True `0.1` (`1//10`). `BigFloat` das für `BigFloat` wird der Approximationsfehler von `Float64` . Bei Verwendung des Makros wird `0.1` direkt in ein `BigFloat` , wodurch der Näherungsfehler reduziert wird.

```
julia> big(0.1)
1.000000000000000005551115123125782702118158340454101562500000000000000000000000000e-01

julia> big"0.1"
1.000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000002e-01
```

`@doc_str`

Dieses Zeichenfolgenmakro erstellt `Base.Markdown.MD` Objekte, die im internen Dokumentationssystem verwendet werden, um eine Rich-Text-Dokumentation für jede Umgebung bereitzustellen. Diese MD-Objekte werden in einem Terminal gut dargestellt:

```
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.
```

und auch in einem Browser:

```
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

Dieses Zeichenfolgenmakro erstellt HTML-Zeichenfolgenliterale, die in einem Browser gut dargestellt werden:

```
In [1]: html"""
        <p><abbr title="Hypertext Markup Language">HTML</abbr> text.</p>
        """
```

Out[1]: HTML text.

@ip_str

Dieses Zeichenfolgenmakro erstellt IP-Adressliterale. Es funktioniert sowohl mit IPv4 als auch mit IPv6:

```
julia> ip"127.0.0.1"
ip"127.0.0.1"

julia> ip"::"
ip"::"
```

@r_str

Dieses String-Makro erstellt `Regex` Literale .

@s_str

Dieses Zeichenfolgenmakro erstellt `SubstitutionString` , die mit `Regex` Literalen zusammenarbeiten, um eine erweiterte `Regex` zu ermöglichen.

@text_str

Dieses Zeichenfolgenmakro ähnelt im Sinne von `@doc_str` und `@html_str` , hat jedoch keine ausgefallenen Formatierungsfunktionen:

```
In [3]: text"""
        This is some plain text.
        """
Out[3]: This is some plain text.
```

@v_str

Dieses Zeichenfolgenmakro erstellt `VersionNumber` Literale. Siehe Versionsnummern für eine Beschreibung ihrer Verwendung und ihrer Verwendung.

@MIME_str

Dieses Zeichenfolgenmakro erstellt die Singleton-Typen von MIME-Typen. Beispielsweise ist `MIME"text/plain"` der Typ von `MIME("text/plain")` .

Symbole, die keine legalen Bezeichnungen sind

Julia-Symbol-Literale müssen gültige Bezeichner sein. Das funktioniert:

```
julia> :cat
:cat
```

Dies gilt jedoch nicht:

```
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,I
::T<:Union{Int128,Int16,Int32,Int64,Int8,UInt128,UInt16,UInt32,UInt64,UInt8}) at int.jl:33
  *(::Real, ::Complex{Bool}) at complex.jl:180
  ...
```

Was hier wie ein Symbolliteral aussieht, wird tatsächlich als implizite Multiplikation von `:2` (was nur `2`) und der Funktion `cat` analysiert, die offensichtlich nicht funktioniert.

Wir können benutzen

```
julia> Symbol("2cat")
Symbol("2cat")
```

um das Problem zu umgehen.

Ein String-Makro könnte dazu beitragen, dies knapper zu machen. Wenn wir das `@sym_str` Makro definieren:

```
macro sym_str(str)
    Meta.quot(Symbol(str))
end
```

dann können wir es einfach tun

```
julia> sym"2cat"
Symbol("2cat")
```

um Symbole zu erstellen, die keine gültigen Julia-Bezeichner sind.

Natürlich können diese Techniken auch Symbole erstellen, *die* gültige Julia-Bezeichner sind. Zum Beispiel,

```
julia> sym"test"
:test
```

Interpolation in einem String-Makro implementieren

String-Makros verfügen nicht über integrierte Interpolationsfunktionen . Es ist jedoch möglich, diese Funktionalität manuell zu implementieren. Beachten Sie, dass das Einbetten nicht möglich ist, ohne Stringliterale zu umgehen, die denselben Begrenzer wie das umgebende Stringmakro haben. Das heißt, obwohl `""" $("x") """` möglich ist, `" $("x") "` nicht. Stattdessen muss dies als `" $(\"x\") "` . Weitere Informationen zu dieser Einschränkung finden Sie im Abschnitt " Anmerkungen" .

Es gibt zwei Ansätze zum manuellen Implementieren der Interpolation: Implementieren Sie das Parsing manuell oder lassen Sie Julia das Parsing durchführen. Der erste Ansatz ist flexibler, der zweite Ansatz ist jedoch einfacher.

Manuelle Analyse

```
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
    quote
        string($(map(esc, components)...))
    end
end
```

Julia beim Parsing

```
macro e_str(s)
    esc(parse("\"$(escape_string(s))\""))
end
```

Diese Methode entgeht der Zeichenfolge (beachte jedoch, dass `escape_string` die `$` -Zeichen *nicht* entgeht) und gibt sie an Julias Parser zurück, um sie zu analysieren. Es ist notwendig, den String zu umgehen, um sicherzustellen, dass `"` und `\` die Analyse des Strings nicht beeinflussen. Der resultierende Ausdruck ist ein `:string` Ausdruck, der für Makrozwecke untersucht und zerlegt werden kann.

Befehlsmakros

0,6,0-dev

In Julia v0.6 und höher werden zusätzlich zu regulären Zeichenfolgenmakros Befehlsmakros unterstützt. Ein Befehlsmakroaufruf wie

```
mymacro`xyz`
```

wird als Makroaufruf analysiert

```
@mymacro_cmd "xyz"
```

Beachten Sie, dass dies den Zeichenfolgenmakros ähnelt, mit Ausnahme von `_cmd` anstelle von `_str` .

Wir verwenden in der Regel Befehlsmakros für Code, der in vielen Sprachen häufig enthält `"` , aber selten enthält `` ` `` So ist es recht einfach , eine einfache Version neu zu implementieren. Quasiquoting mit Befehlsmakros:

```
macro julia_cmd(s)
    esc(Meta.quot(parse(s)))
end
```

Wir können dieses Makro entweder inline verwenden:

```
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)
```

oder mehrzeilig:

```
julia> julia```
       function hello()
           println("Hello, World!")
       end
       ```
:(function hello() # none, line 2:
        println("Hello, World!")
    end)
```

Interpolation mit `$` wird unterstützt:

```
julia> x = 2
2

julia> julia`1 + $x`
:(1 + 2)
```

Die hier angegebene Version erlaubt jedoch nur einen Ausdruck:

```
julia> julia```
       x = 2
       y = 3
       ```
ERROR: ParseError("extra token after end of expression")
```

Die Erweiterung auf mehrere Ausdrücke ist jedoch nicht schwierig.

Zeichenfolge-Makros online lesen: https://riptutorial.com/de/julia-lang/topic/5817/zeichenfolge-makros

Kapitel 36: Zeichenketten

Syntax

- "[string]"
- "[Unicode-Skalarwert]"
- Graphemes ([Zeichenfolge])

Parameter

Parameter	Einzelheiten
Zum	`sprint(f, xs...)`
`f`	Eine Funktion, die ein `IO` Objekt als erstes Argument verwendet.
`xs`	Null oder mehr verbleibende Argumente, die an `f` .

Examples

Hallo Welt!

Zeichenfolgen in Julia werden mit dem Symbol `"` :

```
julia> mystring = "Hello, World!"
"Hello, World!"
```

Beachten Sie, dass im Gegensatz zu anderen Sprachen das `'` Symbol *nicht* verwendet werden kann. `'` definiert ein *Zeichenliteral* ; Dies ist ein `Char` Datentyp und speichert nur einen einzelnen Unicode-Skalarwert :

```
julia> 'c'
'c'

julia> 'character'
ERROR: syntax: invalid character literal
```

Man kann die Unicode-Skalarwerte aus einem String extrahieren, indem man ihn mit einer `for` Schleife durchläuft :

```
julia> for c in "Hello, World!"
           println(c)
       end
H
e
l
```

```
l
o
,

W
o
r
l
d
!
```

Graphemes

Julias `Char` Typ steht für einen Unicode-Skalarwert , der nur in einigen Fällen dem entspricht, was Menschen als "Zeichen" wahrnehmen. Zum Beispiel ist eine Darstellung des Zeichens é wie in einem Lebenslauf tatsächlich eine Kombination aus zwei Unicode-Skalarwerten:

```
julia> collect("é ")
2-element Array{Char,1}:
 'e'
 ' '
```

Die Unicode-Beschreibungen für diese Codepunkte lauten "LATIN SMALL LETTER E" und "COMBINING ACUTE ACCENT". Zusammen definieren sie ein einzelnes "menschliches" Zeichen, bei dem Unicode-Begriffe als Graphem bezeichnet werden . Genauer gesagt, motiviert Unicode-Anhang Nr. 29 die Definition eines Graphem-Clusters aus folgenden Gründen:

> Es ist wichtig zu wissen, dass das, was der Benutzer als "Zeichen" betrachtet - eine Grundeinheit eines Schriftsystems für eine Sprache - möglicherweise nicht nur ein einzelner Unicode-Codepunkt ist. Stattdessen kann diese Basiseinheit aus mehreren Unicode-Codepunkten bestehen. Um Mehrdeutigkeiten bei der Verwendung des Begriffs "Zeichen" durch den Computer zu vermeiden, wird dies als vom Benutzer wahrgenommenes Zeichen bezeichnet. Zum Beispiel ist „G“ + Akzentakzent ein vom Benutzer wahrgenommenes Zeichen: Benutzer halten es für ein einzelnes Zeichen, werden jedoch tatsächlich durch zwei Unicode-Codepunkte dargestellt. Diese vom Benutzer wahrgenommenen Zeichen werden durch einen sogenannten Graphem-Cluster approximiert, der programmgesteuert bestimmt werden kann.

Julia bietet die `graphemes` Funktion zum Durchlaufen der Graphem-Cluster in einer Zeichenfolge:

```
julia> for c in graphemes("ré sumé ")
           println(c)
       end
r
é
s
u
m
é
```

Beachten Sie, dass das Ergebnis, wenn Sie jedes Zeichen in einer eigenen Zeile drucken, besser

ist, als wenn wir die Unicode-Skalarwerte durchlaufen hätten:

```
julia> for c in "ré sumé "
           println(c)
       end
r
e

s
u
m
e
```

Normalerweise ist es bei der Arbeit mit Zeichen im Sinne des Benutzers sinnvoller, mit Graphem-Clustern zu arbeiten als mit Unicode-Skalarwerten. Angenommen, wir möchten eine Funktion schreiben, um die Länge eines einzelnen Wortes zu berechnen. Eine naive Lösung wäre zu verwenden

```
julia> wordlength(word) = length(word)
wordlength (generic function with 1 method)
```

Das Ergebnis ist nicht intuitiv, wenn das Wort Graphem-Cluster enthält, die aus mehr als einem Codepunkt bestehen:

```
julia> wordlength("ré sumé ")
8
```

Wenn wir die korrektere Definition mit der Funktion `graphemes` verwenden, erhalten wir das erwartete Ergebnis:

```
julia> wordlength(word) = length(graphemes(word))
wordlength (generic function with 1 method)

julia> wordlength("ré sumé ")
6
```

Konvertieren Sie numerische Typen in Strings

Es gibt zahlreiche Möglichkeiten, numerische Typen in Strings in Julia zu konvertieren:

```
julia> a = 123
123

julia> string(a)
"123"

julia> println(a)
123
```

Die Funktion `string()` kann auch mehr Argumente annehmen:

```
julia> string(a, "b")
"123b"
```

Sie können Integer (auch als Interpolation bezeichnet) (und bestimmte andere Typen) mit `$` in Strings einfügen:

```
julia> MyString = "my integer is $a"
"my integer is 123"
```

Leistungstipp: Die oben genannten Methoden können zuweilen recht bequem sein. Wenn Sie jedoch viele, viele derartige Vorgänge ausführen und sich Gedanken über die Ausführungsgeschwindigkeit Ihres Codes machen, empfiehlt der Julia- Leistungsführer dagegen und stattdessen die folgenden Methoden:

Sie können für `print()` und `println()` mehrere Argumente `println()` die genau so funktionieren, wie `string()` für mehrere Argumente:

```
julia> println(a, "b")
123b
```

Oder wenn Sie in eine Datei schreiben, können Sie auf ähnliche Weise verwenden, z

```
open("/path/to/MyFile.txt", "w") do file
    println(file, a, "b", 13)
end
```

oder

```
file = open("/path/to/MyFile.txt", "a")
println(file, a, "b", 13)
close(file)
```

Diese sind schneller, da sie es vermeiden, zuerst eine Zeichenfolge aus bestimmten Teilen zu formen und dann auszugeben (entweder an die Konsolenanzeige oder in eine Datei) und stattdessen einfach die verschiedenen Teile nacheinander ausgeben.

Credits: Antwort basierend auf SO Frage Wie kann ich ein Int in einen String in Julia am besten konvertieren? mit Antwort von Michael Ohlrogge und Input von Fengyang Wang

String-Interpolation (Einfügung eines Wertes durch eine Variable in einen String)

In Julia ist es wie in vielen anderen Sprachen möglich, durch Einfügen von durch Variablen definierten Werten in Strings zu interpolieren. Für ein einfaches Beispiel:

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

Wir können andere Typen als numerische verwenden, z

```
Result = false
julia> println("test results is $Result")
test results is false
```

Sie können mehrere Interpolationen innerhalb einer gegebenen Zeichenfolge haben:

```
MySubStr = "a32"
MyNum = 123.31
println("$MySubStr  ,   $MyNum")
```

Leistungstipp Interpolation ist ziemlich bequem. Aber wenn Sie es oft sehr schnell machen, ist es nicht das effizienteste. Siehe unter Konvertieren numerischer Typen in Zeichenfolgen für Vorschläge, wenn Leistung ein Problem ist.

Verwenden von sprint zum Erstellen von Zeichenfolgen mit E / A-Funktionen

Zeichenfolgen können aus Funktionen erstellt werden, die mit `IO` Objekten arbeiten, indem Sie die `sprint` Funktion verwenden. Zum Beispiel kann die `code_llvm` nimmt Funktion ein `IO` - Objekt als erstes Argument. Normalerweise wird es gerne verwendet

```
julia> code_llvm(STDOUT, *, (Int, Int))

define i64 @"jlsys_*_46115"(i64, i64) #0 {
top:
  %2 = mul i64 %1, %0
  ret i64 %2
}
```

Angenommen, wir möchten diese Ausgabe stattdessen als Zeichenfolge. Dann können wir es einfach tun

```
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
}
```

Umwandeln der Ergebnisse der „interaktiven“ Funktionen wie `code_llvm` in Strings können für die automatisierte Analyse, wie nützlich sein Testen , ob generierte Code haben regressiert kann.

Die `sprint` Funktion ist eine Funktion höherer Ordnung, die als erstes Argument die Funktion der `IO` Objekte verwendet. Hinter den Kulissen erstellt es einen `IOBuffer` im RAM, ruft die angegebene Funktion auf und übernimmt die Daten aus dem Puffer in ein `String` Objekt.

Kapitel 37: Zeit

Syntax

- jetzt()
- Dates.today ()
- Termine.Jahr (t)
- Termine.Monat (t)
- Termine.Tag (t)
- Termine.Stunde (t)
- Termine.Minute (T)
- Termine.sekunde (t)
- Termine.Millisekunde (t)
- Datumsformat (t, s)

Examples

Aktuelle Uhrzeit

Um das aktuelle Datum und die aktuelle Uhrzeit abzurufen, verwenden Sie die `now` Funktion:

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

Dies ist die Ortszeit, die die konfigurierte Zeitzone des Computers enthält. Verwenden Sie `now(Dates.UTC)` um die Uhrzeit in der UTC- Zeitzone (UTC) zu `now(Dates.UTC)` :

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

Um das aktuelle Datum ohne Uhrzeit abzurufen, verwenden Sie `today()` :

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

Der Rückgabewert von `now` ist ein `DateTime` Objekt. Es gibt Funktionen, um die einzelnen Komponenten einer `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
```

Es ist möglich, `DateTime` mit einer speziell formatierten `DateTime` formatieren:

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

Da viele der `Dates` Funktionen aus dem exportiert werden `Base.Dates` Modul , kann es einige Tipparbeit ersparen zu schreiben

```
using Base.Dates
```

Dies ermöglicht dann den Zugriff auf die oben genannten qualifizierten Funktionen ohne die `Dates.` Qualifikation.

Zeit online lesen: https://riptutorial.com/de/julia-lang/topic/5812/zeit

Credits

S. No	Kapitel	Contributors
1	Erste Schritte mit Julia Language	Andrew Piliser, becko, Community, Dawny33, Fengyang Wang , Kevin Montrose, prcastro
2	@goto und @label	Fengyang Wang
3	2ind	Fengyang Wang, Gnimuc
4	Arithmetik	Fengyang Wang
5	Arrays	Fengyang Wang, Michael Ohlrogge, prcastro
6	Aufzählungen	Fengyang Wang
7	Ausdrücke	Michael Ohlrogge
8	Conditionals	Fengyang Wang, Michael Ohlrogge, prcastro
9	Eingang	Fengyang Wang
10	Funktionen	Fengyang Wang, Harrison Grodin, Michael Ohlrogge, Sebastialonso
11	Funktionen höherer Ordnung	Fengyang Wang, mnoronha
12	für Loops	Fengyang Wang, Michael Ohlrogge
13	Iterables	Fengyang Wang, prcastro
14	JSON	4444, Fengyang Wang
15	Kombinatoren	Fengyang Wang
16	Lesen eines DataFrame aus einer Datei	Pranav Bhat
17	Metaprogrammierung	Fengyang Wang, Ismael Venegas Castelló, P i, prcastro
18	Module	Fengyang Wang
19	Pakete	Fengyang Wang
20	Parallelverarbeitung	Fengyang Wang, Harrison Grodin, Michael Ohlrogge, prcastro

21	Regexes	Fengyang Wang
22	REPL	Fengyang Wang
23	Shell Scripting und Piping	2Cubed, Fengyang Wang, mnoronha, prcastro
24	String-Normalisierung	Fengyang Wang
25	Tuples	Fengyang Wang
26	Typ Stabilität	Abhijith, Fengyang Wang
27	Typen	Fengyang Wang, prcastro
28	Unit Testing	Fengyang Wang
29	Vergleiche	Fengyang Wang
30	Verschlüsse	Fengyang Wang
31	Versionsübergreifende Kompatibilität	Fengyang Wang
32	Verständnis	2Cubed, Fengyang Wang, zwlayer
33	während Loops	Fengyang Wang
34	Wörterbücher	B Roy Dawson
35	Zeichenfolge-Makros	Fengyang Wang
36	Zeichenketten	Fengyang Wang, Michael Ohlrogge
37	Zeit	Fengyang Wang