eBook Gratuit

APPRENEZ Julia Language

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#julia-lang

Table des matières

À propos

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

Chapitre 1: Démarrer avec Julia Language

Versions

Version	Date de sortie
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

Bonjour le monde!

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

Pour lancer Julia, commencez par récupérer l'interprète sur la page de téléchargement du site Web. La version stable actuelle est la v0.5.0, et cette version est recommandée pour la plupart des utilisateurs. Certains développeurs de paquets ou utilisateurs expérimentés peuvent choisir d'utiliser la construction nocturne, ce qui est beaucoup moins stable.

Lorsque vous avez l'interpréteur, écrivez votre programme dans un fichier nommé `hello.jl` . Il peut alors être exécuté depuis un terminal système en tant que:

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

Julia peut également être exécutée de manière interactive en exécutant le programme `julia` . Vous devriez voir un en-tête et une invite, comme suit:

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

Vous pouvez exécuter n'importe quel code Julia dans cette REPL , alors essayez:

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

Cet exemple utilise une chaîne `"Hello, World!"` , et de la fonction `println` - l'un des nombreux dans la bibliothèque standard. Pour plus d'informations ou d'aide, essayez les sources suivantes:

- Le REPL dispose d'un mode d'aide intégré pour accéder à la documentation.
- La documentation officielle est assez complète.
- Stack Overflow a une collection petite mais croissante d'exemples.
- Les utilisateurs de Gitter sont heureux de vous aider avec de petites questions.
- Le forum de discussion en ligne principal pour Julia est le forum de discussion sur discourse.julialang.org . Les questions plus complexes doivent être affichées ici.
- Une collection de tutoriels et de livres peut être trouvée ici .

Lire Démarrer avec Julia Language en ligne: https://riptutorial.com/fr/julia-lang/topic/485/demarrer-avec-julia-language

Chapitre 2: @goto et @label

Syntaxe

- @goto label
- @label label

Remarques

L'utilisation excessive ou inappropriée d'un flux de contrôle avancé rend le code difficile à lire. `@goto` ou ses équivalents dans d'autres langues, lorsqu'il est utilisé de manière incorrecte, conduit à un code spaghetti illisible.

Semblable à des langages comme C, on ne peut pas sauter entre les fonctions de Julia. Cela signifie également que `@goto` n'est pas possible au niveau supérieur; cela ne fonctionnera que dans une fonction. De plus, on ne peut pas sauter d'une fonction interne à sa fonction externe ou d'une fonction externe à une fonction interne.

Examples

Validation des entrées

Bien `@label` ne soient généralement pas considérées comme des boucles, les macros `@goto` et `@label` peuvent être utilisées pour un flux de contrôle plus avancé. Un cas d'utilisation est le cas où la défaillance d'une partie devrait conduire à la tentative d'une fonction entière, souvent utile pour la validation des entrées:

```
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
```

Cependant, ce cas d'utilisation est souvent plus clair en utilisant la récursion:

```
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
```

Bien que les deux exemples fassent la même chose, le second est plus facile à comprendre. Cependant, le premier est plus performant (car il évite l'appel récursif). Dans la plupart des cas, le coût de l'appel n'a pas d'importance; mais dans des situations limitées, le premier formulaire est acceptable.

Nettoyage d'erreur

Dans des langages tels que C, l'instruction `@goto` est souvent utilisée pour garantir qu'une fonction nettoie les ressources nécessaires, même en cas d'erreur. Ceci est moins important chez Julia, car les exceptions et les blocs `try` - `finally` sont souvent utilisés à la place.

Cependant, il est possible que le code Julia s'interface avec le code C et les API C, et il faut donc parfois écrire des fonctions comme du code C. L'exemple ci-dessous est conçu, mais illustre un cas d'utilisation courant. Le code Julia appellera `Libc.malloc` pour allouer de la mémoire (ceci simule un appel de l'API C). Si toutes les attributions n'aboutissent pas, alors la fonction devrait libérer les ressources obtenues jusqu'à présent; sinon, la mémoire allouée est renvoyée.

```
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
```

Lire @goto et @label en ligne: https://riptutorial.com/fr/julia-lang/topic/5564/-goto-et--label

Chapitre 3: Arithmétique

Syntaxe

- + x
- -X
- a + b
- un B
- un B
- un B
- a ^ b
- un B
- 4a
- sqrt (a)

Examples

Formule quadratique

Julia utilise des opérateurs binaires similaires pour les opérations arithmétiques de base, comme le font les mathématiques ou d'autres langages de programmation. La plupart des opérateurs peuvent être écrits en notation infixe (c'est-à-dire placés entre les valeurs en cours de calcul). Julia a un ordre des opérations qui correspond à la convention commune en mathématiques.

Par exemple, le code ci-dessous implémente la formule quadratique , qui montre les opérateurs `+` , `-` , `*` et `/` pour l'addition, la soustraction, la multiplication et la division, respectivement. La *multiplication implicite* est également indiquée, dans laquelle un nombre peut être placé directement avant un symbole pour signifier la multiplication; c'est-à-dire que `4a` signifie la même chose que `4*a` .

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

Usage:

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

Tamis d'Ératosthène

L'opérateur restant dans Julia est l'opérateur `%` . Cet opérateur se comporte comme le `%` dans les langages tels que C et C ++. `a % b` est le reste signé après avoir divisé `a` par `b` .

Cet opérateur est très utile pour implémenter certains algorithmes, tels que l'implémentation suivante du 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
```

Usage:

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

Arithmétique Matricielle

Julia utilise la signification mathématique standard des opérations arithmétiques lorsqu'elle est appliquée à des matrices. Parfois, des opérations élémentaires sont souhaitées à la place. Celles-ci sont marquées d'un arrêt complet (`.`) Précédant l'opérateur à effectuer élément par élément. (Notez que les opérations élémentaires ne sont souvent pas aussi efficaces que les boucles.)

Les sommes

L'opérateur `+` sur les matrices est une somme matricielle. Il est similaire à une somme élémentaire, mais ne diffuse pas de forme. C'est-à-dire que si `A` et `B` ont la même forme, alors `A + B` est identique à `A .+ B` ; sinon, `A + B` est une erreur, alors que `A .+ B` peut ne pas l'être nécessairement.

```
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
```

De même, `-` calcule une différence de matrice. Les deux `+` et `-` peuvent également être utilisés comme opérateurs unaires.

Des produits

L'opérateur `*` sur les matrices est le produit de la matrice (pas le produit élément par élément). Pour un produit élément par élément, utilisez l'opérateur. `.*` . Comparez (en utilisant les mêmes matrices que ci-dessus):

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

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

Pouvoirs

L'opérateur `^` calcule l' exponentiation de la matrice . Une exponentiation matricielle peut être utile pour calculer rapidement les valeurs de certaines récurrences. Par exemple, les nombres de Fibonacci peuvent être générés par l' expression de la matrice

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

Comme d'habitude, l'opérateur `.^` Peut être utilisé lorsque l'exponentiation élément par élément est l'opération souhaitée.

Lire Arithmétique en ligne: https://riptutorial.com/fr/julia-lang/topic/3848/arithmetique

Chapitre 4: Combinateurs

Remarques

Bien que les combinateurs aient une utilisation pratique limitée, ils constituent un outil utile pour comprendre comment la programmation est fondamentalement liée à la logique et comment des blocs de construction très simples peuvent se combiner pour créer un comportement très complexe. Dans le contexte de Julia, apprendre à créer et utiliser des combinateurs renforcera la compréhension de la programmation dans un style fonctionnel chez Julia.

Examples

Le combinateur Y ou Z

Bien que Julia n'est pas un langage purement fonctionnel, il a un support complet pour la plupart des pierres angulaires de la programmation fonctionnelle: première classe fonctions , portée lexicale et fermetures .

Le combinateur en virgule fixe est un combinateur de clés en programmation fonctionnelle. Étant donné que Julia a une sémantique d' évaluation (comme le font de nombreux langages fonctionnels, y compris Scheme, dont Julia est fortement inspirée), le combinateur Y original de Curry ne fonctionnera pas immédiatement:

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

Cependant, un proche parent du combinateur Y, le Z-combinator, fonctionnera bien:

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

Ce combinateur prend une fonction et retourne une fonction qui, lorsqu'elle est appelée avec l'argument `x` , se fait passer et `x` . Pourquoi serait-il utile qu'une fonction soit transmise elle-même? Cela permet la récursivité sans faire référence au nom de la fonction du tout!

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

Par conséquent, `Z(fact)` devient une implémentation récursive de la fonction factorielle, bien qu'aucune récursivité ne soit visible dans cette définition de fonction. (La récursivité est évidente dans la définition du combinateur `Z` , bien sûr, mais cela est inévitable dans un langage avide.) Nous pouvons vérifier que notre fonction fonctionne bien:

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

Non seulement cela, mais il est aussi rapide que nous pouvons attendre d'une implémentation récursive. Le code LLVM montre que le résultat est compilé dans une ancienne branche, soustrait,

appelle et multiplie:

```
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"
}
```

Le système de combinaison SKI

Le système de combinaison SKI est suffisant pour représenter tous les termes de calcul lambda. (En pratique, bien sûr, les abstractions lambda atteignent une taille exponentielle lorsqu'elles sont traduites en SKI.) En raison de la simplicité du système, l'implémentation des combinateurs S, K et I est extrêmement simple:

Une traduction directe du calcul Lambda

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

Nous pouvons confirmer, à l'aide du système de test unitaire , que chaque combinateur a le comportement attendu.

Le combinateur I est le plus facile à vérifier; il devrait retourner la valeur donnée inchangée:

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

Le combinateur K est également assez simple: il doit ignorer son deuxième argument.

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

Le combinateur S est le plus complexe; son comportement peut être résumé en appliquant les deux premiers arguments au troisième argument, en appliquant le premier résultat au second. Nous pouvons tester le plus facilement le combinateur S en testant certaines de ses formes curry.

`S(K)` , par exemple, devrait simplement renvoyer son deuxième argument et rejeter son premier, comme on le voit:

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

`S(I)(I)` devrait appliquer son argument à lui-même:

```
@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)` applique son second argument à son premier:

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

Le combinateur décrit ci-dessus porte un nom dans la `Base` standard Julia: `identity` . Ainsi, nous aurions pu réécrire les définitions ci-dessus avec la définition alternative suivante de `I` :

```
const I = identity
```

Affichage des combinateurs SKI

Une des faiblesses de l'approche ci-dessus est que nos fonctions ne s'affichent pas aussi bien que nous le souhaiterions. Pouvons-nous remplacer

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

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

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

avec quelques affichages plus informatifs? La réponse est oui! Relançons le REPL, et définissons cette fois comment chaque fonction doit être affichée:

```
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
```

Il est important d'éviter de montrer quoi que ce soit avant d'avoir fini de définir les fonctions. Sinon, nous risquons d'invalider le cache de la méthode et nos nouvelles méthodes ne semblent pas prendre effet immédiatement. C'est pourquoi nous avons mis des points-virgules dans les

définitions ci-dessus. Les points-virgules suppriment la sortie de la REPL.

Cela rend les fonctions bien affichées:

```
julia> S
S

julia> K
K

julia> I
I
```

Cependant, nous rencontrons toujours des problèmes lorsque nous essayons d'afficher une fermeture:

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

Il serait plus intéressant de l'afficher comme `S(K)` . Pour ce faire, nous devons exploiter le fait que les fermetures ont leur propre type. Nous pouvons accéder à ces types et leur ajouter des méthodes par réflexion, en utilisant `typeof` et le champ `primary` champ de `name` du type. Redémarrez le REPL à nouveau; nous allons faire d'autres changements:

```
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)
```

Et maintenant, enfin, les choses s'affichent comme nous aimerions qu'elles:

```
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
```

Lire Combinateurs en ligne: https://riptutorial.com/fr/julia-lang/topic/5758/combinateurs

Chapitre 5: Comparaisons

Syntaxe

- x <y # si `x` est strictement inférieur à `y`
- x> y # si `x` est strictement supérieur à `y`
- x == y # si `x` est égal à `y`
- x === y # alternativement `x ≡ y` , si `x` est égal à `y`
- x ≤ y # alternativement `x <= y` , si `x` est inférieur ou égal à `y`
- x ≥ y # alternativement `x >= y` , si `x` est supérieur ou égal à `y`
- x ≠ y # alternativement `x != y` , si `x` n'est pas égal à `y`
- x ≈ y # si `x` est approximativement égal à `y`

Remarques

Faites attention à ne pas renverser les signes de comparaison. Julia définit de nombreuses fonctions de comparaison par défaut sans définir la version retournée correspondante. Par exemple, on peut courir

```
julia> Set(1:3) ⊆ Set(0:5)
true
```

mais ça ne marche pas

```
julia> Set(0:5) ⊇ Set(1:3)
ERROR: UndefVarError: ⊇ not defined
```

Examples

Comparaisons en chaîne

Plusieurs opérateurs de comparaison utilisés ensemble sont chaînés, comme s'ils étaient connectés via l' opérateur `&&` . Cela peut être utile pour des chaînes de comparaison lisibles et mathématiquement concises, telles que

```
# 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
```

Cependant, il existe une différence importante entre `a > b > c` et `a > b && b > c` ; dans ce dernier cas, le terme `b` est évalué deux fois. Cela n'a pas beaucoup d'importance pour les vieux symboles simples, mais pourrait avoir de l'importance si les termes eux-mêmes ont des effets secondaires. Par exemple,

```
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
```

Examinons de plus près les comparaisons chaînées et leur fonctionnement en observant comment elles sont analysées et abaissées en expressions . Tout d'abord, considérons la simple comparaison, que nous pouvons voir est juste un appel de fonction simple:

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

Maintenant, si nous enchaînons la comparaison, nous remarquons que l'analyse a changé:

```
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
```

Après l'analyse, l'expression est ensuite abaissée à sa forme finale:

```
julia> expand(:(a > b >= c))
:(begin
        unless a > b goto 3
        return b >= c
        3:
        return false
    end)
```

et on remarque en effet que c'est la même chose que pour `a > b && b >= c` :

```
julia> expand(:(a > b && b >= c))
:(begin
        unless a > b goto 3
        return b >= c
        3:
```

```
        return false
    end)
```

Nombres ordinaux

Nous examinerons comment implémenter des comparaisons personnalisées en implémentant un type personnalisé, des nombres ordinaux . Pour simplifier l'implémentation, nous allons nous concentrer sur un petit sous-ensemble de ces nombres: tous les nombres ordinaux jusqu'à mais n'incluant pas ε_0. Notre implémentation est axée sur la simplicité et non sur la rapidité. Cependant, l'implémentation n'est pas lente non plus.

Nous stockons les nombres ordinaux par leur forme normale de Cantor . Parce que l'arithmétique ordinale n'est pas commutative, nous prendrons la convention commune de stocker d'abord les termes les plus significatifs.

```
immutable OrdinalNumber <: Number
    βs::Vector{OrdinalNumber}
    cs::Vector{Int}
end
```

Comme la forme normale de Cantor est unique, nous pouvons tester l'égalité simplement par l'égalité récursive:

0.5.0

Dans la version v0.5, il existe une très bonne syntaxe pour le faire de manière compacte:

```
import Base: ==
α::OrdinalNumber == β::OrdinalNumber = α.βs == β.βs && α.cs == β.cs
```

0.5.0

Sinon, définissez la fonction comme il est plus typique:

```
import Base: ==
==(α::OrdinalNumber, β::OrdinalNumber) = α.βs == β.βs && α.cs == β.cs
```

Pour finir notre commande, car ce type a une commande totale, nous devrions surcharger la fonction `isless` :

```
import Base: isless
function isless(α::OrdinalNumber, β::OrdinalNumber)
    for i in 1:min(length(α.cs), length(β.cs))
        if α.βs[i] < β.βs[i]
            return true
        elseif α.βs[i] == β.βs[i] && α.cs[i] < β.cs[i]
            return true
        end
    end
    return length(α.cs) < length(β.cs)
end
```

Pour tester notre ordre, nous pouvons créer des méthodes pour faire des nombres ordinaux. Zéro, bien sûr, est obtenu en n'ayant pas de termes dans la forme normale de Cantor:

```
const ORDINAL_ZERO = OrdinalNumber([], [])
Base.zero(::Type{OrdinalNumber}) = ORDINAL_ZERO
```

On peut définir une `expω` pour calculer `ω^α` , et l'utiliser pour calculer 1 et ω:

```
expω(α) = OrdinalNumber([α], [1])
const ORDINAL_ONE = expω(ORDINAL_ZERO)
Base.one(::Type{OrdinalNumber}) = ORDINAL_ONE
const ω = expω(ORDINAL_ONE)
```

Nous avons maintenant une fonction de commande entièrement fonctionnelle sur les nombres ordinaux:

```
julia> ORDINAL_ZERO < ORDINAL_ONE < ω < expω(ω)
true

julia> ORDINAL_ONE > ORDINAL_ZERO
true

julia> sort([ORDINAL_ONE, ω, expω(ω), ORDINAL_ZERO])

4-element Array{OrdinalNumber,1}:

OrdinalNumber(OrdinalNumber[],Int64[])

OrdinalNumber(OrdinalNumber[OrdinalNumber(OrdinalNumber[],Int64[])],[1])

OrdinalNumber(OrdinalNumber[OrdinalNumber(OrdinalNumber[OrdinalNumber(OrdinalNumber[],Int64[])],[1])],


OrdinalNumber(OrdinalNumber[OrdinalNumber(OrdinalNumber[OrdinalNumber(OrdinalNumber[OrdinalNumber(Ordin
```

Dans le dernier exemple, nous voyons que l'impression des nombres ordinaux pourrait être meilleure, mais le résultat est comme prévu.

Opérateurs standard

Julia prend en charge un très grand nombre d'opérateurs de comparaison. Ceux-ci inclus

1. Toutes les séquences Unicode suivantes: `> < >= ≥ <= ≤ == === ≡ != ≠ !== ≢ ∈ ∉ ∋ ∌ ⊆ ⊈ ⊂ ⊄ ⊊ ∝ ∊ ∍ ∥ ∦ ∷ ∺ ∻ ∽ ∾ ≁ ≃ ≄ ≅ ≆ ≇ ≈ ≉ ≊ ≋ ≌ ≍ ≎ ≐ ≑ ≒ ≓ ≔ ≕ ≖ ≗ ≘ ≙ ≚ ≛ ≜ ≝ ≞ ≟ ≣ ≦ ≧ ≨ ≩ ≪ ≫ ≬ ≭ ≮ ≯ ≰ ≱ ≲ ≳ ≴ ≵ ≶ ≷ ≸ ≹ ≺ ≻ ≼ ≽ ≾ ≿ ⊀ ⊁ ⊃ ⊅ ⊇ ⊉ ⊋ ⊏ ⊐ ⊑ ⊒ ⊜ ⊩ ⊬ ⊮ ⊰ ⊱ ⊲ ⊳ ⊴ ⊵ ⊶ ⊷ ⋍ ⋐ ⋑ ⋕ ⋖ ⋗ ⋘ ⋙ ⋚ ⋛ ⋜ ⋝ ⋞ ⋟ ⋠ ⋡ ⋢ ⋣ ⋤ ⋥ ⋦ ⋧ ⋨ ⋩ ⋪ ⋫ ⋬ ⋭ ⋲ ⋳ ⋴ ⋵ ⋶ ⋷ ⋹ ⋺ ⋻ ⋼ ⋽ ⋾ ⟂ ⟒ ⦷ ⧀ ⧁ ⧡ ⧣ ⧤ ⧥ ⩦ ⩧ ⩪ ⩫ ⩬ ⩭ ⩮ ⩯ ⩰ ⩱ ⩲ ⩳ ⩴ ⩵ ⩶ ⩷ ⩸ ⩹ ⩺ ⩻ ⩼ ⩽ ⩾ ⩿ ⪀ ⪁ ⪂ ⪃ ⪄ ⪅ ⪆ ⪇ ⪈ ⪉ ⪊ ⪋ ⪌ ⪍ ⪎ ⪏ ⪐ ⪑ ⪒ ⪓ ⪔ ⪕ ⪖ ⪗ ⪘ ⪙ ⪚ ⪛ ⪜ ⪝ ⪞ ⪟ ⪠ ⪡ ⪢ ⪣ ⪤ ⪥ ⪦ ⪧ ⪨ ⪩ ⪪ ⪫ ⪬ ⪭ ⪮ ⪯ ⪰ ⪱ ⪲ ⪳ ⪴ ⪵ ⪶ ⪷ ⪸ ⪹ ⪺ ⪻ ⪼ ⪽ ⪾ ⪿ ⫀ ⫁ ⫂ ⫃ ⫄ ⫅ ⫆ ⫇ ⫈ ⫉ ⫊ ⫋ ⫌ ⫍ ⫎ ⫏ ⫐ ⫑ ⫒ ⫓ ⫔ ⫕ ⫖ ⫗ ⫘ ⫙ ⫷ ⫸ ⫹ ⫺ ⊢ ⊣ ;`
2. Tous les symboles du point 1, précédés d'un point (.), Doivent être faits élément par élément;
3. Les opérateurs `<:` , `>:` , `.!` , et `in` , qui ne peut pas être précédé d'un point (.).

Tous n'ont pas de définition dans la bibliothèque de `Base` standard. Cependant, ils sont disponibles pour d'autres packages à définir et à utiliser selon les besoins.

Dans un usage quotidien, la plupart de ces opérateurs de comparaison ne sont pas pertinents. Les plus courantes sont les fonctions mathématiques standard pour la commande; voir la section Syntaxe pour une liste.

Comme la plupart des opérateurs de Julia, les opérateurs de comparaison sont des fonctions et peuvent être appelés comme fonctions. Par exemple, `(<)(1, 2)` est identique à `1 < 2` .

Utiliser ==, === et isequal

Il y a trois opérateurs d'égalité: `==` , `===` et `isequal` . (Le dernier n'est pas vraiment un opérateur, mais c'est une fonction et tous les opérateurs sont des fonctions.)

Quand utiliser `==`

`==` est une *valeur d'* égalité. Il renvoie `true` lorsque deux objets représentent, dans leur état actuel, la même valeur.

Par exemple, il est évident que

```
julia> 1 == 1
true
```

mais de plus

```
julia> 1 == 1.0
true

julia> 1 == 1.0 + 0.0im
true

julia> 1 == 1//1
true
```

Les côtés droits de chaque égalité ci-dessus sont d'un type différent, mais ils représentent toujours la même valeur.

Pour les objets mutables, comme les tableaux , `==` compare leur valeur actuelle.

```
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
```

La plupart du temps, `==` est le bon choix.

Quand utiliser `===`

`===` est une opération beaucoup plus stricte que `==` . Au lieu d'égalité de valeur, elle mesure l'égalité. Deux objets sont égaux s'ils ne peuvent pas être distingués les uns des autres par le programme lui-même. Ainsi nous avons

```
julia> 1 === 1
true
```

comme il n'y a aucun moyen de distinguer un `1` d'un autre `1` . Mais

```
julia> 1 === 1.0
false
```

car bien que `1` et `1.0` aient la même valeur, ils sont de types différents et le programme peut donc les distinguer.

En outre,

```
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
```

ce qui peut paraître surprenant au premier abord! Comment le programme pourrait-il distinguer les deux vecteurs `A` et `B` ? Comme les vecteurs sont mutables, ils pourraient modifier `A` , et se comporter différemment de `B` Mais peu importe comment cela modifie `A` , `A` se comportera toujours de la même manière que `A` lui-même. Donc, `A` est égal à `A` , mais pas égal à `B`

Continuant dans cette voie, observez

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

julia> A === C
true
```

En assignant `A` à `C` , on dit que `C` a un *alias* `A` C'est-à-dire que c'est devenu juste un autre nom pour `A` Toute modification apportée à `A` sera observée par `C` également. Par conséquent, il n'y a aucun moyen de faire la différence entre `A` et `C` , donc ils sont égaux.

Quand faut-il utiliser `isequal`

La différence entre `==` et `isequal` est très subtile. La plus grande différence réside dans la gestion des nombres à virgule flottante:

```
julia> NaN == NaN
false
```

Ce résultat, probablement surprenant, est défini par la norme IEEE pour les types à virgule flottante (IEEE-754). Mais cela n'est pas utile dans certains cas, comme le tri. `isequal` est fourni pour ces cas:

```
julia> isequal(NaN, NaN)
true
```

D'un autre côté du spectre, `==` traite le zéro négatif de l'IEEE et le zéro positif comme la même valeur (également spécifiée par IEEE-754). Ces valeurs ont cependant des représentations

distinctes en mémoire.

```
julia> 0.0
0.0

julia> -0.0
-0.0

julia> 0.0 == -0.0
true
```

Toujours à des fins de tri, `isequal` distingue.

```
julia> isequal(0.0, -0.0)
false
```

Lire Comparaisons en ligne: https://riptutorial.com/fr/julia-lang/topic/5563/comparaisons

Chapitre 6: Compatibilité des versions croisées

Syntaxe

- en utilisant Compat
- Compat.String
- Compat.UTF8String
- @compat f. (x, y)

Remarques

Il est parfois très difficile d'obtenir une nouvelle syntaxe pour bien jouer avec plusieurs versions. Comme Julia est toujours en développement actif, il est souvent utile de supprimer le support des anciennes versions et de ne cibler que les plus récentes.

Examples

Numéros de version

Julia a une implémentation `VersionNumber` du `VersionNumber` version sémantique à travers le type `VersionNumber` .

Pour construire un `VersionNumber` de `VersionNumber` tant que littéral, `@v_str` pouvez utiliser la macro de chaîne `@v_str` :

```
julia> vers = v"1.2.0"
v"1.2.0"
```

Alternativement, on peut appeler le constructeur `VersionNumber` ; Notez que le constructeur accepte jusqu'à cinq arguments, mais tous sauf le premier sont facultatifs.

```
julia> vers2 = VersionNumber(1, 1)
v"1.1.0"
```

Les numéros de version peuvent être comparés à l'aide d' opérateurs de comparaison et peuvent donc être triés:

```
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"
```

Les numéros de version sont utilisés à plusieurs endroits sur Julia. Par exemple, la constante `VERSION` est un `VersionNumber` :

```
julia> VERSION
v"0.5.0"
```

Ceci est couramment utilisé pour l'évaluation de code conditionnel, en fonction de la version de Julia. Par exemple, pour exécuter un code différent sur v0.4 et v0.5, on peut faire

```
if VERSION < v"0.5"
    println("v0.5 prerelease, v0.4 or older")
else
    println("v0.5 or newer")
end
```

Chaque package installé est également associé à un numéro de version actuel:

```
julia> Pkg.installed("StatsBase")
v"0.9.0"
```

Utiliser Compat.jl

Le package Compat.jl active l'utilisation de nouvelles fonctionnalités et syntaxes Julia avec les anciennes versions de Julia. Ses fonctionnalités sont documentées dans son README, mais un résumé des applications utiles est donné ci-dessous.

0.5.0

Type de chaîne unifiée

Dans Julia v0.4, il y avait beaucoup de types de chaînes différents . Ce système était considéré comme trop complexe et déroutant, donc dans Julia v0.5, il ne reste que le type `String` . `Compat` permet d'utiliser le type `String` et le constructeur sur la version 0.4, sous le nom `Compat.String` . Par exemple, ce code v0.5

```
buf = IOBuffer()
println(buf, "Hello World!")
String(buf)  # "Hello World!\n"
```

peut être directement traduit dans ce code, qui fonctionne à la fois sur v0.5 et v0.4:

```
using Compat
buf = IOBuffer()
println(buf, "Hello World!")
```

```
Compat.String(buf)  # "Hello World!\n"
```

Notez qu'il y a des mises en garde.

- Sur la v0.4, `Compat.String` est typé avec `ByteString` , qui est `Union{ASCIIString, UTF8String}` . Ainsi, les types avec des champs `String` ne seront pas stables. Dans ces situations, `Compat.UTF8String` est conseillé car il signifiera `String` sur v0.5 et `UTF8String` sur v0.4, les deux étant des types concrets.
- Il faut faire attention à utiliser `Compat.String` ou `import Compat: String` , car `String` lui-même a un sens sur v0.4: c'est un alias obsolète pour `AbstractString` . Un signe indiquant que `String` été utilisé accidentellement au lieu de `Compat.String` est que, à tout moment, les avertissements suivants apparaissent:

```
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
```

Syntaxe de diffusion compacte

Julia v0.5 introduit le sucre syntaxique pour la `broadcast` . La syntaxe

```
f.(x, y)
```

est abaissé pour `broadcast(f, x, y)` . Les exemples d'utilisation de cette syntaxe incluent `sin.([1, 2, 3])` pour prendre le sinus de plusieurs nombres à la fois.

Sur la v0.5, la syntaxe peut être utilisée directement:

```
julia> sin.([1.0, 2.0, 3.0])
3-element Array{Float64,1}:
 0.841471
 0.909297
 0.14112
```

Cependant, si nous essayons la même chose avec la version 0.4, nous obtenons une erreur:

```
julia> sin.([1.0, 2.0, 3.0])
ERROR: TypeError: getfield: expected Symbol, got Array{Float64,1}
```

Heureusement, `Compat` rend également cette nouvelle syntaxe utilisable depuis la version 0.4. Encore une fois, nous ajoutons à l' `using Compat` . Cette fois, nous `@compat` l'expression avec la macro `@compat` :

```
julia> using Compat

julia> @compat sin.([1.0, 2.0, 3.0])
3-element Array{Float64,1}:
 0.841471
```

```
0.909297
0.14112
```

Lire Compatibilité des versions croisées en ligne: https://riptutorial.com/fr/julia-lang/topic/5832/compatibilite-des-versions-croisees

Chapitre 7: Compréhensions

Examples

Compréhension de tableau

Syntaxe de base

Les tableaux de Julia utilisent la syntaxe suivante:

```
[expression for element = iterable]
```

Notez que, comme `for` boucles , tous les éléments `=` , `in` et `∈` sont acceptés pour la compréhension.

Cela équivaut à peu près à créer un tableau vide et à utiliser une boucle `for` pour `push!` articles à elle.

```
result = []
for element in iterable
    push!(result, expression)
end
```

Cependant, le type de compréhension du tableau est aussi étroit que possible, ce qui est préférable pour les performances.

Par exemple, pour obtenir un tableau des carrés des entiers compris entre `1` et `10` , le code suivant peut être utilisé.

```
squares = [x^2 for x=1:10]
```

Ceci est un remplacement clair et concis pour la version longue `for` -loop.

```
squares = []
for x in 1:10
    push!(squares, x^2)
end
```

Compréhension des tableaux conditionnels

Avant le Julia 0.5, il n'y a aucun moyen d'utiliser les conditions dans les compréhensions du tableau. Mais ce n'est plus vrai. Dans Julia 0.5, nous pouvons utiliser les conditions dans des conditions telles que les suivantes:

```
julia> [x^2 for x in 0:9 if x > 5]
4-element Array{Int64,1}:
```

```
 36
 49
 64
 81
```

La source de l'exemple ci-dessus peut être trouvée ici .

Si nous aimerions utiliser la compréhension des listes imbriquées:

```
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)
```

Compréhensions multidimensionnelles

Emboîtés `for` les boucles peuvent être utilisées pour itérer sur un certain nombre de iterables uniques.

```
result = []
for a = iterable_a
    for b = iterable_b
        push!(result, expression)
    end
end
```

De même, plusieurs spécifications d'itération peuvent être fournies à une compréhension de tableau.

```
[expression for a = iterable_a, b = iterable_b]
```

Par exemple, les éléments suivants peuvent être utilisés pour générer le produit cartésien de `1:3` et `1:2` .

```
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)
```

La compréhension des tableaux multidimensionnels aplatis est similaire, sauf qu'ils perdent leur forme. Par exemple,

```
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)
```

est une variante aplatie de ce qui précède. La différence syntaxique est qu'un supplément `for` est utilisé à la place d'une virgule.

Compréhension de générateur

Les compréhensions des générateurs suivent un format similaire à celui des compréhensions de tableaux, mais utilisent des parenthèses `()` au lieu de crochets `[]` .

```
(expression for element = iterable)
```

Une telle expression renvoie un objet `Generator` .

```
julia> (x^2 for x = 1:5)
Base.Generator{UnitRange{Int64},##1#2}(#1,1:5)
```

Arguments de fonction

Les compréhensions génératrices peuvent être fournies comme le seul argument d'une fonction, sans avoir besoin d'un ensemble supplémentaire de parenthèses.

```
julia> join(x^2 for x = 1:5)
"1491625"
```

Cependant, si plusieurs arguments sont fournis, la compréhension du générateur nécessite son propre ensemble de parenthèses.

```
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"
```

Lire Compréhensions en ligne: https://riptutorial.com/fr/julia-lang/topic/5477/comprehensions

Chapitre 8: Conditionnels

Syntaxe

- si cond; corps; fin
- si cond; corps; autre; corps; fin
- si cond; corps; sinon, cond. corps; autre; fin
- si cond; corps; sinon, cond. corps; fin
- cond? Iftrue: iffalse
- cond && iftrue
- cond || iffalse
- ifelse (cond, iftrue, iffalse)

Remarques

Tous les opérateurs et fonctions conditionnels impliquent l'utilisation de conditions booléennes (`true` ou `false`). En Julia, le type de booléen est `Bool` . Contrairement à d'autres langages, d'autres types de nombres (comme `1` ou `0`), des chaînes de caractères, des tableaux, etc., *ne peuvent pas* être utilisés directement dans les conditions.

En règle générale, on utilise soit des fonctions de prédicat (fonctions `Bool` un `Bool`), soit des opérateurs de comparaison dans la condition d'un opérateur ou d'une fonction conditionnelle.

Examples

if ... else expression

Le conditionnel le plus courant dans Julia est l'expression `if` ... `else` . Par exemple, nous implémentons ci-dessous l' algorithme euclidien pour calculer le plus grand commun diviseur , en utilisant un conditionnel pour gérer le cas de base:

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

La forme `if` ... `else` de Julia est en réalité une expression et a une valeur; la valeur est l'expression en position de queue (c'est-à-dire la dernière expression) sur la branche qui est prise. Considérez l'exemple d'entrée suivant:

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

Ici, `a` est `0` et `b` est `-10` . La condition `a == 0` est `true` , donc la première branche est prise. La valeur

renvoyée est `abs(b)` qui est `10` .

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

Ici, `a` est `2` et `b` est `3` . La condition `a == 0` est fausse, donc la deuxième branche est prise, et nous calculons `mygcd(b % a, a)` , qui est `mygcd(3 % 2, 2)` . L'opérateur `%` renvoie le reste lorsque `3` est divisé par `2` , dans ce cas `1` . On calcule donc `mygcd(1, 2)` , et cette fois `a` est `1` et `b` est `2` . Encore une fois, `a == 0` est faux, donc la deuxième branche est prise et nous calculons `mygcd(b % a, a)` , qui est `mygcd(0, 1)` . Cette fois, `a == 0` enfin et ainsi `abs(b)` est retourné, ce qui donne le résultat `1` .

if ... statement autre

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

Toute expression, telle que l'expression `if` ... `else` , peut être placée dans la position de l'instruction. Cela ignore sa valeur mais exécute toujours l'expression pour ses effets secondaires.

si déclaration

Comme toute autre expression, la valeur de retour d'une expression `if` ... `else` peut être ignorée (et donc supprimée). Cela n'est généralement utile que lorsque le corps de l'expression a des effets secondaires, tels que l'écriture dans un fichier, la modification de variables ou l'impression sur l'écran.

De plus, la branche `else` d'une expression `if` ... `else` est facultative. Par exemple, nous pouvons écrire le code suivant en sortie à l'écran uniquement si une condition particulière est remplie:

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

Dans l'exemple ci-dessus, nous utilisons les fonctions de date et heure pour obtenir la seconde courante; Par exemple, si elle est actuellement à 10:55:27, la `second` variable contiendra `27` . Si ce nombre est pair, une ligne sera imprimée à l'écran. Sinon, rien ne sera fait.

Opérateur conditionnel ternaire

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

L'opérateur conditionnel ternaire est une expression moins verbeuse `if` ... `else` .

La syntaxe est spécifiquement:

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

Dans cet exemple, nous ajoutons `x` à la collection `A` uniquement si `x` n'est pas déjà dans `A` Sinon, nous ne faisons que laisser `A` inchangé.

Opérateur ternaire Références:

- Julia Documentation
- Wikibooks

Les opérateurs de court-circuit: && et ||

Pour branchement

Les opérateurs conditionnels en court-circuit `&&` et `||` peut être utilisé comme remplacement léger pour les constructions suivantes:

- `x && y` est équivalent à `x ? y : x`
- `x || y` est équivalent à `x ? x : y`

Une des utilisations des opérateurs de court-circuit consiste à tester de manière plus concise une condition et à effectuer une action spécifique en fonction de cette condition. Par exemple, le code suivant utilise l'opérateur `&&` pour générer une erreur si l'argument `x` est négatif:

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

Le `||` L'opérateur peut également être utilisé pour vérifier les erreurs, sauf qu'il déclenche l'erreur à *moins* qu'une condition *ne soit* remplie, au lieu de *si* la condition est remplie:

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

Une autre application utile consiste à fournir une valeur par défaut à un objet, uniquement s'il n'est pas défini précédemment:

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

Ici, cela vérifie si le symbole x est défini (c'est-à-dire s'il y a une valeur affectée à l'objet `x`). Si oui, alors rien ne se passe. Mais sinon, `x` sera assigné à `NEW_VALUE` . Notez que cet exemple ne fonctionnera que sur la portée de haut niveau.

Dans des conditions

Les opérateurs sont également utiles car ils peuvent être utilisés pour tester deux conditions dont la seconde n'est évaluée qu'en fonction du résultat de la première condition. De la documentation Julia:

> Dans l'expression `a && b` , la sous-expression `b` n'est évaluée que si `a` évaluée à `true`
>
> Dans l'expression `a || b` , la sous-expression `b` n'est évaluée que si `a` évalué comme `false`

Ainsi, alors que `a & b` et `a && b` donneront la valeur `true` si `a` et `b` sont tous deux `true` , leur comportement si `a` est `false` est différent.

Par exemple, supposons que nous souhaitons vérifier si un objet est un nombre positif, où il est possible que ce ne soit même pas un nombre. Considérons les différences entre ces deux implémentations tentées:

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

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

`CheckPositive1()` produira une erreur si un type non numérique lui est fourni en tant qu'argument. En effet, il évalue les *deux* expressions, quel que soit le résultat du premier, et la seconde expression génère une erreur lorsque l'on essaie de l'évaluer pour un type non numérique.

`CheckPositive2()` , cependant, donnera un résultat `false` (plutôt qu'une erreur) si un type non numérique lui est fourni, car la seconde expression n'est évaluée que si la première est `true` .

Plus d'un opérateur de court-circuit peut être branché ensemble. Par exemple:

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

si déclaration avec plusieurs branches

```
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
```

Un nombre quelconque de `elseif` branches peut être utilisé avec une `if` déclaration, peut - être avec ou sans une finale d' `else` branche. Les conditions ultérieures ne seront évaluées que si toutes les conditions antérieures se sont révélées `false` .

La fonction ifelse

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

Usage:

```
julia> shift(10)
9

julia> shift(11)
12

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

La fonction `ifelse` évalue les deux branches, même celle qui n'est pas sélectionnée. Cela peut être utile lorsque les branches ont des effets secondaires à évaluer ou parce que les deux branches elles-mêmes sont moins chères.

Lire Conditionnels en ligne: https://riptutorial.com/fr/julia-lang/topic/4356/conditionnels

Chapitre 9: Contribution

Syntaxe

- readline ()
- readlines ()
- readstring (STDIN)
- chomp (str)
- ouvrir (f, fichier)
- chaque ligne (io)
- readstring (fichier)
- lire (fichier)
- readcsv (fichier)
- readdlm (fichier)

Paramètres

Paramètre	Détails
`chomp(str)`	**Supprimez jusqu'à une nouvelle ligne de fin d'une chaîne.**
`str`	La chaîne à partir de laquelle une nouvelle ligne doit être supprimée. Notez que les chaînes sont immuables par convention. Cette fonction renvoie une nouvelle chaîne.
`open(f, file)`	**Ouvrez un fichier, appelez la fonction et fermez le fichier par la suite.**
`f`	La fonction à appeler sur le flux IO qui ouvre le fichier est générée.
`file`	Le chemin du fichier à ouvrir.

Examples

Lecture d'une chaîne à partir d'une entrée standard

Le flux `STDIN` dans Julia fait référence à une entrée standard . Cela peut représenter une entrée utilisateur, des programmes de ligne de commande interactifs ou une entrée d'un fichier ou d'un pipeline qui a été redirigé dans le programme.

La fonction `readline` , lorsqu'elle ne `STDIN` aucun argument, lira les données de `STDIN` jusqu'à ce qu'une nouvelle ligne soit rencontrée ou que le flux `STDIN` passe à l'état de fin de fichier. Ces deux cas peuvent être distingués selon que le caractère `\n` a été lu comme caractère final:

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

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

Souvent, pour les programmes interactifs, nous ne nous soucions pas de l'état EOF et voulons simplement une chaîne. Par exemple, nous pouvons demander à l'utilisateur de saisir des données:

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

Ce n'est pas tout à fait satisfaisant, cependant, en raison de la nouvelle ligne supplémentaire:

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

La fonction `chomp` est disponible pour supprimer jusqu'à une nouvelle ligne de fin d'une chaîne. Par exemple:

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

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

Nous pouvons donc augmenter notre fonction avec `chomp` afin que le résultat soit comme prévu:

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

qui a un résultat plus souhaitable:

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

Parfois, nous pouvons souhaiter lire autant de lignes que possible (jusqu'à ce que le flux d'entrée passe à l'état de fin de fichier). La fonction `readlines` fournit cette fonctionnalité.

```
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

Encore une fois, si nous détestons les nouvelles lignes à la fin des lignes lues par `readlines` , nous pouvons utiliser la fonction `chomp` pour les supprimer. Cette fois -ci , nous avons diffusé la `chomp` fonction à travers l'ensemble du réseau:

```
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"
```

D'autres fois, nous ne nous soucions pas du tout des lignes et voulons simplement lire autant que possible en une seule chaîne. La fonction `readstring` accomplit ceci:

```
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"
```

(le `# [END OF INPUT]` ne fait pas partie de l'entrée d'origine, il a été ajouté pour plus de clarté.)

Notez que l' `readstring` doit être transmis à `STDIN` .

Lecture de nombres à partir d'une entrée standard

La lecture des nombres à partir d'une entrée standard est une combinaison de lecture de chaînes et d'analyse de chaînes telles que des nombres.

La fonction d' `parse` est utilisée pour analyser une chaîne dans le type de numéro souhaité:

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

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

Le format attendu par l' `parse(T, x)` est similaire, mais pas exactement, au format que Julia attend des littéraux numériques :

```
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
```

En combinant les `parse` et `readline` fonctions nous permet de lire un seul numéro d'une ligne:

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

qui fonctionne comme prévu:

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

Les mises en garde habituelles concernant la précision en virgule flottante s'appliquent. Notez que l' `parse` peut être utilisée avec `BigInt` et `BigFloat` pour supprimer ou minimiser la perte de précision.

Parfois, il est utile de lire plusieurs numéros de la même ligne. En règle générale, la ligne peut être fractionnée avec des espaces:

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

qui peut être utilisé comme suit:

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

Lecture de données d'un fichier

Lecture de chaînes ou d'octets

Les fichiers peuvent être ouverts pour la lecture en utilisant la fonction `open` , qui est souvent utilisée avec la syntaxe de bloc :

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

Supposons que `myfile` existe et que son contenu soit

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

Ensuite, ce code produirait le résultat suivant:

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

Notez que `eachline` est un paresseux itérables sur les lignes du fichier. Il est préférable de `readlines` pour des raisons de performance.

Parce `do` syntaxe de bloc est simplement du sucre syntaxique pour les fonctions anonymes, nous pouvons également passer des fonctions nommées à l' `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
```

Les fonctions `read` et `readstring` fournissent des méthodes pratiques qui ouvriront automatiquement un fichier:

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

Lecture de données structurées

Supposons que nous ayons un fichier CSV avec le contenu suivant, dans un fichier nommé `file.csv` :

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

Ensuite, nous pouvons utiliser la fonction `readcsv` pour lire ces données dans une `Matrix` :

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

Si le fichier était plutôt délimité par des tabulations, dans un fichier nommé `file.tsv` , la fonction `readdlm` peut être utilisée à la place, l'argument `delim` défini sur `'\t'` . Les charges de travail plus avancées doivent utiliser le package CSV.jl.

Lire Contribution en ligne: https://riptutorial.com/fr/julia-lang/topic/7201/contribution

Chapitre 10: Cordes

Syntaxe

- "[chaîne]"
- '[Valeur scalaire Unicode]'
- graphemes ([string])

Paramètres

Paramètre	Détails
Pour	`sprint(f, xs...)`
`f`	Une fonction qui prend un objet `IO` comme premier argument.
`xs`	Zéro ou plusieurs arguments restants à transmettre à `f` .

Examples

Bonjour le monde!

Les chaînes de caractères de Julia sont délimitées à l'aide du `"` symbole:

```
julia> mystring = "Hello, World!"
"Hello, World!"
```

Notez que contrairement à d'autres langues, le symbole `'` *ne peut pas* être utilisé à la place. `'` définit un *littéral de caractère* ; c'est un type de données `Char` et ne stockera qu'une seule valeur scalaire Unicode :

```
julia> 'c'
'c'

julia> 'character'
ERROR: syntax: invalid character literal
```

On peut extraire les valeurs scalaires unicode d'une chaîne en l'itérant avec une boucle `for` :

```
julia> for c in "Hello, World!"
           println(c)
       end
H
e
l
l
```

```
o
,

W
o
r
l
d
!
```

Graphemes

Le type `Char` de Julia représente une valeur scalaire Unicode , qui correspond dans certains cas uniquement à ce que les humains perçoivent comme un "caractère". Par exemple, une représentation du caractère é, comme dans résumé, est en fait une combinaison de deux valeurs scalaires Unicode:

```
julia> collect("é ")
2-element Array{Char,1}:
 'e'
 ' '
```

Les descriptions Unicode de ces points de code sont "LETTRE MINUSCULE LATINE E" et "ACCENT COMBINANT AIGU". Ensemble, ils définissent un seul caractère "humain", ce qui signifie que les termes Unicode sont appelés graphèmes . Plus précisément, l'Annexe Unicode n ° 29 motive la définition d'un cluster grapheme car:

> Il est important de reconnaître que ce que l'utilisateur considère comme un «caractère» - une unité de base d'un système d'écriture pour une langue - peut ne pas être un simple point de code Unicode. Au lieu de cela, cette unité de base peut être composée de plusieurs points de code Unicode. Pour éviter toute ambiguïté avec l'utilisation informatique du terme caractère, cela s'appelle un caractère perçu par l'utilisateur. Par exemple, «G» + accent aigu est un caractère perçu par l'utilisateur: les utilisateurs le considèrent comme un seul caractère, mais sont en réalité représentés par deux points de code Unicode. Ces caractères perçus par l'utilisateur sont approximés par ce qu'on appelle un cluster de graphèmes, qui peut être déterminé par programmation.

Julia fournit la fonction `graphemes` pour parcourir les grappes de graphèmes dans une chaîne:

```
julia> for c in graphemes("ré sumé ")
           println(c)
       end
r
é
s
u
m
é
```

Notez que le résultat, en imprimant chaque caractère sur sa propre ligne, est meilleur que si nous

avions itéré sur les valeurs scalaires Unicode:

```
julia> for c in "ré sumé "
           println(c)
       end
r
e

s
u
m
e
```

Généralement, lorsque vous travaillez avec des caractères dans un sens perçu par l'utilisateur, il est plus utile de traiter des grappes de graphèmes qu'avec des valeurs scalaires Unicode. Par exemple, supposons que nous voulions écrire une fonction pour calculer la longueur d'un seul mot. Une solution naïve serait d'utiliser

```
julia> wordlength(word) = length(word)
wordlength (generic function with 1 method)
```

Nous notons que le résultat est contre-intuitif lorsque le mot inclut des grappes de graphèmes composées de plusieurs points de code:

```
julia> wordlength("ré sumé ")
8
```

Lorsque nous utilisons la définition la plus correcte, en utilisant la fonction `graphemes` , nous obtenons le résultat attendu:

```
julia> wordlength(word) = length(graphemes(word))
wordlength (generic function with 1 method)

julia> wordlength("ré sumé ")
6
```

Convertir les types numériques en chaînes

Il existe de nombreuses façons de convertir des types numériques en chaînes dans Julia:

```
julia> a = 123
123

julia> string(a)
"123"

julia> println(a)
123
```

La fonction `string()` peut également prendre plus d'arguments:

```
julia> string(a, "b")
"123b"
```

Vous pouvez également insérer (aka interpoler) des entiers (et certains autres types) dans des chaînes en utilisant `$` :

```
julia> MyString = "my integer is $a"
"my integer is 123"
```

Astuce de performance: Les méthodes ci-dessus peuvent être très pratiques par moments. Mais, si vous exécutez beaucoup d'opérations de ce type et que vous êtes préoccupé par la vitesse d'exécution de votre code, le guide de performance de Julia vous le déconseille, mais plutôt les méthodes ci-dessous:

Vous pouvez fournir plusieurs arguments à `print()` et `println()` qui fonctionneront exactement comme `string()` opère sur plusieurs arguments:

```
julia> println(a, "b")
123b
```

Ou, lorsque vous écrivez dans un fichier, vous pouvez également utiliser, par exemple

```
open("/path/to/MyFile.txt", "w") do file
    println(file, a, "b", 13)
end
```

ou

```
file = open("/path/to/MyFile.txt", "a")
println(file, a, "b", 13)
close(file)
```

Celles-ci sont plus rapides car elles évitent de devoir d'abord former une chaîne à partir de morceaux donnés, puis de la sortir (soit sur l'écran de la console, soit sur un fichier) et de ne sortir que séquentiellement les différentes pièces.

Crédits: Réponse basée sur SO Question Quelle est la meilleure façon de convertir un Int en String dans Julia? avec la réponse de Michael Ohlrogge et la contribution de Fengyang Wang

Interpolation de chaîne (valeur d'insertion définie par la variable dans la chaîne)

Dans Julia, comme dans de nombreux autres langages, il est possible d'interpoler en insérant des valeurs définies par des variables dans des chaînes. Pour un exemple simple:

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

Nous pouvons utiliser d'autres types que numériques, par exemple

```
Result = false
julia> println("test results is $Result")
test results is false
```

Vous pouvez avoir plusieurs interpolations dans une chaîne donnée:

```
MySubStr = "a32"
MyNum = 123.31
println("$MySubStr  ,   $MyNum")
```

Astuce de performance L' interpolation est très pratique. Mais si vous le faites très rapidement, ce n'est pas le plus efficace. Au lieu de cela, voir Convertir les types numériques en chaînes pour obtenir des suggestions lorsque les performances posent problème.

Utilisation du sprint pour créer des chaînes avec des fonctions IO

Les chaînes peuvent être créées à partir de fonctions qui fonctionnent avec des objets `IO` en utilisant la fonction `sprint` . Par exemple, la fonction `code_llvm` accepte un objet `IO` comme premier argument. Typiquement, il est utilisé comme

```
julia> code_llvm(STDOUT, *, (Int, Int))

define i64 @"jlsys_*_46115"(i64, i64) #0 {
top:
  %2 = mul i64 %1, %0
  ret i64 %2
}
```

Supposons que nous voulons cette sortie sous forme de chaîne à la place. Alors on peut simplement faire

```
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
}
```

La conversion des résultats de fonctions "interactives" comme `code_llvm` en chaînes peut être utile pour une analyse automatisée, telle que le test de la régression du code généré.

La fonction `sprint` est une fonction d'ordre supérieur qui prend comme premier argument la fonction opérant sur les objets `IO` . Dans les coulisses, il crée un `IOBuffer` dans la RAM, appelle la fonction donnée et prend les données du tampon dans un objet `String` .

Lire Cordes en ligne: https://riptutorial.com/fr/julia-lang/topic/5562/cordes

Chapitre 11: Dictionnaires

Examples

Utiliser des dictionnaires

Les dictionnaires peuvent être construits en lui passant un nombre quelconque de paires.

```
julia> Dict("A"=>1, "B"=>2)
Dict{String,Int64} with 2 entries:
  "B" => 2
  "A" => 1
```

Vous pouvez obtenir des entrées dans un dictionnaire en mettant la clé entre crochets.

```
julia> dict = Dict("A"=>1, "B"=>2)
Dict{String,Int64} with 2 entries:
  "B" => 2
  "A" => 1

julia> dict["A"]
1
```

Lire Dictionnaires en ligne: https://riptutorial.com/fr/julia-lang/topic/9028/dictionnaires

Chapitre 12: Enums

Syntaxe

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

Remarques

Il est parfois utile d'avoir des types énumérés où chaque instance est d'un type différent (souvent un type singleton immuable); Cela peut être important pour la stabilité du type. Les traits sont généralement implémentés avec ce paradigme. Cependant, cela se traduit par une surcharge supplémentaire au moment de la compilation.

Examples

Définir un type énuméré

Un type énuméré est un type pouvant contenir une liste de valeurs possibles. Dans Julia, les types énumérés sont généralement appelés "types enum". Par exemple, on pourrait utiliser les types enum pour décrire les sept jours de la semaine, les douze mois de l'année, les quatre combinaisons d'un jeu de 52 cartes standard ou d'autres situations similaires.

Nous pouvons définir des types énumérés pour modéliser les combinaisons et les rangs d'un jeu de 52 cartes standard. La macro `@enum` est utilisée pour définir les types enum.

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

Ceci définit deux types: `Suit` et `Rank` . On peut vérifier que les valeurs sont bien des types attendus:

```
julia> ♦
♦::Suit = 1

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

Notez que chaque combinaison et rang a été associé à un numéro. Par défaut, ce nombre commence à zéro. Donc, le deuxième costume, les diamants, a été attribué le numéro 1. Dans le cas de `Rank` , il peut être plus logique de commencer le nombre à un. Cela a été réalisé en annotant la définition de l' `ace` avec une annotation `=1` .

Les types énumérés comportent de nombreuses fonctionnalités, telles que l'égalité (et même l'identité) et les comparaisons intégrées:

```
julia> seven === seven
true

julia> ten ≠ jack
true

julia> two < three
true
```

Comme les valeurs de tout autre type immuable , les valeurs des types énumérés peuvent également être hachées et stockées dans `Dict` s.

Nous pouvons compléter cet exemple en définissant un type de `Card` avec un champ `Rank` et un `Suit` :

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

et donc nous pouvons créer des cartes avec

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

Mais les types énumérés viennent également avec leurs propres méthodes de `convert` , donc nous pouvons en effet simplement faire

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

et puisque `7` peut être directement converti en `Rank` , ce constructeur fonctionne à la perfection.

Nous pourrions souhaiter définir le sucre syntaxique pour construire ces cartes; la multiplication implicite fournit un moyen pratique de le faire. Définir

```
julia> import Base.*

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

et alors

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

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

en profitant une fois de plus des fonctions de `convert` intégrées.

Utiliser des symboles comme énumérations légères

Bien que la macro `@enum` soit très utile dans la plupart des cas d'utilisation, elle peut être excessive dans certains cas d'utilisation. Les inconvénients de `@enum` incluent:

- Il crée un nouveau type
- C'est un peu plus difficile à étendre
- Il est livré avec des fonctionnalités telles que la conversion, l'énumération et la comparaison, qui peuvent être superflues dans certaines applications.

Dans les cas où une alternative plus légère est souhaitée, le type de `Symbol` peut être utilisé. Les symboles sont des chaînes internes ; elles représentent des séquences de caractères, un peu comme les chaînes , mais elles sont associées uniquement aux nombres. Cette association unique permet une comparaison rapide des égalités de symbole.

Nous pouvons encore implémenter un type de `Card` , cette fois en utilisant les champs `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
```

Nous implémentons le constructeur interne pour vérifier les valeurs incorrectes transmises au constructeur. Contrairement à l'exemple utilisant les types `@enum` , `Symbol` s peut contenir n'importe quelle chaîne et nous devons donc faire attention aux types de `Symbol` nous acceptons. Notez ici l'utilisation des opérateurs conditionnels de court-circuit .

Maintenant, nous pouvons construire des objets `Card` comme prévu:

```
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
```

Le principal avantage de `Symbol` est leur extensibilité d'exécution. Si à l'exécution, nous souhaitons accepter (par exemple) `:eleven` comme nouveau grade, il suffit simplement de lancer `push!(ranks,`

`:eleven)` . Une telle extensibilité d'exécution n'est pas possible avec les types `@enum` .

Lire Enums en ligne: https://riptutorial.com/fr/julia-lang/topic/7104/enums

Chapitre 13: Expressions

Examples

Introduction aux expressions

Les expressions sont un type d'objet spécifique dans Julia. Vous pouvez penser à une expression représentant un morceau de code Julia qui n'a pas encore été évalué (c'est-à-dire exécuté). Il y a ensuite des fonctions et des opérations spécifiques, comme `eval()` qui évaluera l'expression.

Par exemple, nous pourrions écrire un script ou entrer dans l'interpréteur les éléments suivants: julia> 1 + 1 2

Une façon de créer une expression consiste à utiliser la syntaxe `:()` . Par exemple:

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

Nous avons maintenant un objet de type `Expr` . Ayant juste été formé, il ne fait rien - il se contente de rester comme tout autre objet jusqu'à ce qu'il soit agi. Dans ce cas, nous pouvons *évaluer* cette expression en utilisant la fonction `eval()` :

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

Ainsi, nous voyons que les deux suivants sont équivalents:

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

Pourquoi voudrions-nous passer par la syntaxe beaucoup plus compliquée dans `eval(:(1+1))` si nous voulons juste trouver ce que 1 + 1 est égal? La raison de base est que nous pouvons définir une expression à un point de notre code, la modifier éventuellement ultérieurement, puis l'évaluer ultérieurement. Cela peut potentiellement ouvrir de nouvelles fonctionnalités puissantes au programmeur Julia. Les expressions sont un élément clé de la métaprogrammation chez Julia.

Créer des expressions

Plusieurs méthodes différentes peuvent être utilisées pour créer le même type d'expression. L' introduction d'expressions a mentionné la syntaxe `:()` . Peut-être que le meilleur endroit pour commencer, cependant, est avec les cordes. Cela aide à révéler certaines des similarités fondamentales entre les expressions et les chaînes de caractères de Julia.

Créer une expression à partir d'une chaîne

De la documentation Julia:

> Chaque programme Julia commence sa vie comme une ficelle

En d'autres termes, tout script Julia est simplement écrit dans un fichier texte, qui n'est rien d'autre qu'une chaîne de caractères. De même, toute commande Julia entrée dans un interpréteur n'est qu'une chaîne de caractères. Le rôle de Julia ou de tout autre langage de programmation est alors d'interpréter et d'évaluer les chaînes de caractères de manière logique et prévisible, de sorte que ces chaînes de caractères puissent être utilisées pour décrire ce que le programmeur souhaite que l'ordinateur accomplisse.

Ainsi, une façon de créer une expression consiste à utiliser la fonction `parse()` appliquée à une chaîne. L'expression suivante, une fois évaluée, affectera la valeur 2 au symbole `x` .

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

Créer une expression à l'aide de `:()` Syntaxe

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

Notez qu'avec cette syntaxe, Julia traitera automatiquement les noms d'objets en référence à des symboles. Nous pouvons voir cela si nous regardons les `args` de l'expression. (Voir Champs des objets d'expression pour plus de détails sur le champ `args` dans une expression.)

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

Créer une expression à l'aide de la fonction `Expr()`

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

Cette syntaxe est basée sur la notation de préfixe . En d'autres termes, le premier argument de la fonction spécifiée pour la fonction `Expr()` est la `head` ou le préfixe. Les autres sont les `arguments` de l'expression. La `head` détermine quelles opérations seront effectuées sur les arguments.

Pour plus de détails à ce sujet, voir Champs des objets d'expression

Lorsque vous utilisez cette syntaxe, il est important de faire la distinction entre l'utilisation d'objets et de symboles pour les objets. Par exemple, dans l'exemple ci-dessus, l'expression attribue la

valeur `2` au symbole `:x` , une opération parfaitement sensible. Si nous utilisions `x` lui-même dans une expression comme celle-là, nous obtiendrions un résultat absurde:

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

De même, si nous examinons les `args` nous voyons:

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

Ainsi, la fonction `Expr()` n'effectue pas la même transformation automatique en symboles que la syntaxe `:()` pour créer des expressions.

Créer des expressions multilignes en utilisant les `quote...end`

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

Notez qu'avec `quote...end` on peut créer des expressions contenant d'autres expressions dans leur champ `args` :

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

Voir Champs des objets d'expression pour plus d'informations sur ce champ `args` .

Plus sur la création d'expressions

Cet exemple donne simplement les bases pour créer des expressions. Voir aussi, par exemple, Interpolation et Expressions et champs des objets d'expression pour plus d'informations sur la création d'expressions plus complexes et avancées.

Champs d'objets d'expression

Comme mentionné dans l' introduction aux expressions, les expressions sont un type d'objet spécifique dans Julia. En tant que tels, ils ont des champs. Les deux champs les plus utilisés d'une expression sont sa `head` et ses `args` . Par exemple, considérons l'expression

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

discuté dans Création d'expressions . Nous pouvons voir la `head` et les `args` comme suit:

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

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

Les expressions sont basées sur la notation par préfixe . En tant que tel, la `head` spécifie généralement l'opération à effectuer sur les `args` . La tête doit être du type Julia `Symbol` .

Lorsqu'une expression doit affecter une valeur (lorsqu'elle est évaluée), elle utilisera généralement une tête de `:(=)` . Il y a bien sûr des variations évidentes à cela que l'on peut employer, par exemple:

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

: appel à la tête d'expression

Une autre commune `head` des expressions est `:call` . Par exemple

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

Suivant les conventions de notation des préfixes, les opérateurs sont évalués de gauche à droite. Ainsi, cette expression signifie ici que nous appellerons la fonction spécifiée sur le premier élément des `args` sur les éléments suivants. Nous pourrions pareillement avoir:

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

Ou d'autres fonctions potentiellement plus intéressantes, par exemple

```
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
```

Détermination automatique de la `head` lors de l'utilisation de `:()` notation de création d'expression

Notez que `:call` est implicitement utilisé comme tête dans certaines constructions d'expressions, par exemple

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

Ainsi, avec la syntaxe `:()` pour créer des expressions, Julia cherchera à déterminer

automatiquement la tête correcte à utiliser. De même:

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

En fait, si vous n'êtes pas certain de ce que la bonne tête à utiliser pour une expression que vous créez en utilisant, par exemple, `Expr()` cela peut être un outil utile pour obtenir des conseils et des idées sur ce qu'il faut utiliser.

Interpolation et Expressions

La création d'expressions mentionne que les expressions sont étroitement liées aux chaînes. En tant que tels, les principes d'interpolation au sein des chaînes sont également pertinents pour les expressions. Par exemple, en interpolation de base des chaînes, on peut avoir quelque chose comme:

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

Nous utilisons le signe `$` pour insérer la valeur de `n` dans la chaîne. Nous pouvons utiliser la même technique avec des expressions. Par exemple

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

Contraste ceci ceci:

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

Ainsi, avec le premier exemple, nous définissons à l'avance la valeur de `a` qui sera utilisée au moment de l'évaluation de l'expression. Avec le second exemple, cependant, le compilateur Julia ne cherchera à `a` pour trouver sa valeur *au moment de* l' *évaluation* de notre expression.

Références externes sur les expressions

Il existe un certain nombre de ressources Web utiles qui peuvent vous aider à mieux connaître les expressions dans Julia. Ceux-ci inclus:

- Julia Docs - Métaprogrammation
- Wikibooks - Métaprogrammation Julia
- Les macros de Julia, les expressions, etc. pour et par les confus, par Gray Calhoun

- Mois de Julia - Métaprogrammation, par Andrew Collier
- Différenciation symbolique en Julia, par John Myles White

SO Messages:

- Qu'est-ce qu'un "symbole" dans Julia? Répondre par Stefan Karpinski
- Pourquoi julia exprime-t-elle cette expression de manière complexe?
- Explication de l'exemple d'interpolation de l'expression Julia

Lire Expressions en ligne: https://riptutorial.com/fr/julia-lang/topic/5805/expressions

Chapitre 14: Fermetures

Syntaxe

- x -> [body]
- (x, y) -> [body]
- (xs ...) -> [body]

Remarques

0.4.0

Dans les anciennes versions de Julia, les fermetures et les fonctions anonymes entraînaient une pénalité d'exécution. Cette pénalité a été éliminée en 0.5.

Examples

Composition de la fonction

Nous pouvons définir une fonction pour effectuer une composition de fonctions en utilisant une syntaxe de fonction anonyme :

```
f ∘ g = x -> f(g(x))
```

Notez que cette définition est équivalente à chacune des définitions suivantes:

```
∘(f, g) = x -> f(g(x))
```

ou

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

rappelant que dans Julia, `f ∘ g` est juste du sucre syntaxique pour `∘(f, g)` .

Nous pouvons voir que cette fonction compose correctement:

```
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

Dans la version v0.5, cette définition est très performante. Nous pouvons examiner le code LLVM généré:

```
julia> @code_llvm sextuple(1)

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

Il est clair que les deux multiplications ont été pliées en une seule multiplication et que cette fonction est aussi efficace que possible.

Comment fonctionne cette fonction d'ordre supérieur? Il crée une soi-disant fermeture , qui consiste non seulement en son code, mais permet également de suivre certaines variables de son étendue. Toutes les fonctions de Julia qui ne sont pas créées au niveau supérieur sont des fermetures.

0.5.0

On peut inspecter les variables fermées dans les champs de la fermeture. Par exemple, nous voyons que:

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

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

Mise en œuvre du curry

Une des applications des fermetures consiste à appliquer partiellement une fonction; c'est-à-dire fournir des arguments maintenant et créer une fonction qui prend les arguments restants. Le curry est une forme spécifique d'application partielle.

Commençons par la fonction simple `curry(f, x)` qui fournira le premier argument à une fonction et attendra des arguments supplémentaires plus tard. La définition est assez simple:

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

Encore une fois, nous utilisons une syntaxe de fonction anonyme , cette fois en combinaison avec la syntaxe des arguments variadiques.

Nous pouvons implémenter certaines fonctions de base dans un style tacite (ou sans point) en

utilisant cette fonction `curry` .

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

julia> double(10)
20

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

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

Les fonctions maintiennent le générisme attendu:

```
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
```

Introduction aux fermetures

Les fonctions sont une partie importante de la programmation de Julia. Ils peuvent être définis directement dans les modules, auquel cas les fonctions sont appelées *niveau supérieur* . Mais les fonctions peuvent également être définies dans d'autres fonctions. Ces fonctions sont appelées " fermetures ".

Les fermetures capturent les variables dans leur fonction externe. Une fonction de niveau supérieur ne peut utiliser que les variables globales de leur module, paramètres de fonction ou variables locales:

```
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
```

Une fermeture, par contre, peut utiliser tous ceux en plus des variables des fonctions externes qu'elle capture:

```
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
```

Si on lance `c = toplevel(10)` , on voit que le résultat est

```
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)
```

Notez que l'expression en queue de cette fonction est une fonction en soi; c'est une fermeture. On peut appeler la fermeture `c` comme s'il s'agissait d'une autre fonction:

```
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)
```

Notez que `c` toujours accès aux variables `y` et `z` de l'appel `toplevel` - même si `toplevel` a déjà été renvoyé! Chaque fermeture, même celles renvoyées par la même fonction, se ferme sur différentes variables. Nous pouvons appeler à nouveau le `toplevel`

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

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

julia> c(22)
v = 22 is a parameter
w = 3 is a local variable
x = 0 is a global variable
y = 10 is a closed variable (a parameter of the outer function)
z = 2 is a closed variable (a local of the outer function)
```

Notez que, bien que `d` et `c` aient le même code, et étant passés les mêmes arguments, leur sortie est différente. Ce sont des fermetures distinctes.

Lire Fermetures en ligne: https://riptutorial.com/fr/julia-lang/topic/5724/fermetures

Chapitre 15: Fonctions d'ordre supérieur

Syntaxe

- foreach (f, xs)
- carte (f, xs)
- filtre (f, xs)
- réduire (f, v0, xs)
- foldl (f, v0, xs)
- foldr (f, v0, xs)

Remarques

Les fonctions peuvent être acceptées en tant que paramètres et peuvent également être produites en tant que types de retour. En effet, des fonctions peuvent être créées à l'intérieur du corps d'autres fonctions. Ces fonctions internes sont appelées fermetures .

Examples

Fonctionne comme des arguments

Les fonctions sont des objets dans Julia. Comme tous les autres objets, ils peuvent être transmis comme arguments à d'autres fonctions. Les fonctions qui acceptent des fonctions sont appelées fonctions d' ordre supérieur.

Par exemple, nous pouvons implémenter un équivalent de la fonction `foreach` de la bibliothèque standard en prenant une fonction `f` comme premier paramètre.

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

Nous pouvons tester que cette fonction fonctionne effectivement comme prévu:

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

En prenant une fonction comme *premier* paramètre, au lieu d'un paramètre ultérieur, nous pouvons utiliser la syntaxe de bloc de commande de Julia. La syntaxe do block est simplement un moyen pratique de passer une fonction anonyme en tant que premier argument à une fonction.

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

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

Notre implémentation de `myforeach` ci-dessus est à peu près équivalente à la fonction intégrée `foreach` . De nombreuses autres fonctions intégrées d'ordre supérieur existent également.

Les fonctions d'ordre supérieur sont très puissantes. Parfois, lorsque vous travaillez avec des fonctions de niveau supérieur, les opérations exactes sont sans importance et les programmes peuvent devenir très abstraits. Les combinateurs sont des exemples de systèmes de fonctions hautement abstraites d'ordre supérieur.

Mapper, filtrer et réduire

Deux des fonctions les plus fondamentales de la bibliothèque standard sont `map` et `filter` . Ces fonctions sont génériques et peuvent fonctionner sur n'importe quel itérable . En particulier, ils conviennent parfaitement aux calculs sur les tableaux .

Supposons que nous ayons un ensemble de données sur les écoles. Chaque école enseigne une matière particulière, a un nombre de classes et un nombre moyen d'élèves par classe. Nous pouvons modéliser une école avec le type immuable suivant:

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

Notre ensemble de données sur les écoles sera un `Vector{School}` :

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

Supposons que nous souhaitons trouver le nombre total d'élèves inscrits dans un programme de mathématiques. Pour ce faire, nous avons besoin de plusieurs étapes:

- nous devons restreindre le jeu de données aux seules écoles qui enseignent les mathématiques (`filter`)
- nous devons calculer le nombre d'étudiants à chaque école (`map`)
- et nous devons réduire cette liste de nombres d'étudiants à une seule valeur, la somme (`reduce`)

Une solution naïve (pas la plus performante) consisterait simplement à utiliser directement ces trois fonctions supérieures.

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

```
end
```

et nous vérifions qu'il y a 190 étudiants en mathématiques dans notre ensemble de données:

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

Des fonctions existent pour combiner ces fonctions et améliorer ainsi les performances. Par exemple, nous aurions pu utiliser la fonction `mapreduce` pour effectuer le mappage et la réduction en une seule étape, ce qui permettrait d'économiser du temps et de la mémoire.

La `reduce` n'a de sens que pour les opérations associatives comme `+` , mais il est parfois utile d'effectuer une réduction avec une opération non associative. Les fonctions d'ordre supérieur `foldl` et `foldr` sont fournies pour forcer un ordre de réduction particulier.

Lire Fonctions d'ordre supérieur en ligne: https://riptutorial.com/fr/julia-lang/topic/6955/fonctions-d-ordre-superieur

Chapitre 16: Iterables

Syntaxe

- commencer (itr)
- suivant (itr, s)
- fait (itr, s)
- prendre (itr, n)
- drop (itr, n)
- cycle (itr)
- Produit de base (xs, ys)

Paramètres

Paramètre	Détails
Pour	**Toutes les fonctions**
`itr`	L'itérable à utiliser
Pour	**`next` et `done`**
`s`	Un état itérateur décrivant la position actuelle de l'itération.
Pour	**`take` et `drop`**
`n`	Le nombre d'éléments à prendre ou à laisser.
Pour	**`Base.product`**
`xs`	L'itérable de prendre les premiers éléments de paires à partir de.
`ys`	L'itérable de prendre des seconds éléments de paires à partir de.
...	(Notez que le `product` accepte un nombre quelconque d'arguments; si plus de deux arguments sont fournis, il construira des tuples de longueur supérieure à deux.)

Examples

Nouveau type itérable

Julia, lors de la boucle à travers un objet itérable `I` est fait avec la `for` la syntaxe:

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

```
    # body
end
```

Dans les coulisses, ceci est traduit par:

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

Par conséquent, si vous voulez que `I` sois itérable, vous devez définir les méthodes `start` , `next` et `done` pour son type. Supposons que vous définissiez un type `Foo` contenant un tableau comme l'un des champs:

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

Nous instancions un objet `Foo` en faisant:

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

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

Si nous voulons effectuer une itération dans `Foo` , avec chaque `bar` éléments renvoyée par chaque itération, nous définissons les méthodes:

```
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
```

Notez que puisque ces fonctions appartiennent au module `Base` , nous devons d'abord `import` leurs noms avant de leur ajouter de nouvelles méthodes.

Une fois les méthodes définies, `Foo` est compatible avec l’interface de l’itérateur:

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

```
       end

1
2
3
```

Combinaison d'Itables Paresseux

La bibliothèque standard est fournie avec une riche collection d'itérables paresseux (et les bibliothèques telles que Iterators.jl en offrent encore plus). Les itérables paresseux peuvent être composés pour créer des itérables plus puissants en temps constant. Les itérables paresseux les plus importants sont Take et Drop , à partir desquels de nombreuses autres fonctions peuvent être créées.

Assez tranché un iterable

Les tableaux peuvent être découpés avec la notation par tranche. Par exemple, les éléments suivants renvoient les éléments 10 à 15 d'un tableau, y compris:

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

Cependant, la notation slice ne fonctionne pas avec toutes les itérables. Par exemple, nous ne pouvons pas découper une expression de générateur:

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

Les chaînes de découpage peuvent ne pas avoir le comportement Unicode attendu:

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

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

Nous pouvons définir une fonction `lazysub(itr, range::UnitRange)` pour effectuer ce type de découpage sur des itérables arbitraires. Ceci est défini en termes de `take` et de `drop` :

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

L'implémentation fonctionne ici car pour la valeur `UnitRange a:b` , les étapes suivantes sont effectuées:

- laisse tomber les premiers éléments a `a-1`
- prend `a` élément, `a+1` élément et ainsi de suite jusqu'à ce que l'élément `a+(ba)=b`

Au total, `ba` éléments sont pris. Nous pouvons confirmer que notre implémentation est correcte

dans chaque cas ci-dessus:

```
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
```

Déplacer paresseusement une circulaire itérative

La `circshift` opération sur les tableaux se déplacera le tableau comme si elle était un cercle, relinéariser il. Par exemple,

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

Pouvons-nous faire cela paresseusement pour tous les itérables? Nous pouvons utiliser le `cycle` , `drop` et `take` itérables pour implémenter cette fonctionnalité.

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

Avec les types paresseux étant plus performants dans de nombreuses situations, cela nous permet de faire des fonctionnalités `circshift` sur des types qui ne le seraient pas autrement:

```
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

Faire une table de multiplication

Faisons une table de multiplication en utilisant des fonctions itérables paresseuses pour créer une matrice.

Les fonctions clés à utiliser ici sont:

- `Base.product` , qui calcule un produit cartésien .
- `prod` , qui calcule un produit régulier (comme dans la multiplication)
- `:` , qui crée une gamme
- `map` , qui est une fonction d'ordre supérieur appliquant une fonction à chaque élément d'une collection

La solution est la suivante:

```
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
```

Listes évaluées par la bouche

Il est possible de créer une liste simple, évaluée paresseusement, en utilisant des types et des fermetures mutables. Une liste évaluée paresseuse est une liste dont les éléments ne sont pas évalués lors de sa construction, mais plutôt lorsqu'on y accède. Les avantages des listes paresseuses évaluées incluent la possibilité d'être infinie.

```
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()
```

Ce qui fonctionne effectivement comme le ferait un langage comme Haskell , où toutes les listes sont évaluées paresseusement:

```
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
```

En pratique, il est préférable d'utiliser le package Lazy.jl. Cependant, l'implémentation de la liste paresseuse ci-dessus met en lumière des détails importants sur la façon de construire son propre type itérable.

Lire Iterables en ligne: https://riptutorial.com/fr/julia-lang/topic/5466/iterables

Chapitre 17: JSON

Syntaxe

- en utilisant JSON
- JSON.parse (str)
- JSON.json (obj)
- JSON.print (io, obj, indent)

Remarques

Puisque ni les objets Julia `Dict` ni les objets JSON ne sont classés de manière inhérente, il est préférable de ne pas utiliser l'ordre des paires clé-valeur dans un objet JSON.

Examples

Installer JSON.jl

JSON est un format d'échange de données populaire. La bibliothèque JSON la plus populaire pour Julia est JSON.jl. Pour installer ce paquet, utilisez le gestionnaire de paquets:

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

L'étape suivante consiste à tester si le package fonctionne sur votre machine:

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

Si tous les tests réussissent, la bibliothèque est prête à être utilisée.

Analyse JSON

JSON qui a été codé en tant que chaîne peut facilement être analysé en un type Julia standard:

```
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]
```

Il y a quelques propriétés immédiates de JSON.jl de note:

- Les types JSON correspondent à des types sensibles dans Julia: L'objet devient `Dict` , le tableau devient `Vector` , le nombre devient `Int64` ou `Float64` , le booléen devient `Bool` et null devient `nothing::Void` .
- JSON est un format de conteneur non typé: ainsi, les vecteurs Julia renvoyés sont de type `Vector{Any}` , et les dictionnaires retournés sont de type `Dict{String, Any}` .
- La norme JSON ne fait pas la distinction entre les nombres entiers et décimaux, mais JSON.jl le fait. Un nombre sans point décimal ou notation scientifique est analysé dans `Int64` , tandis qu'un nombre avec un point décimal est analysé dans `Float64` . Cela correspond étroitement au comportement des analyseurs JSON dans de nombreuses autres langues.

Sérialisation de JSON

La fonction `JSON.json` sérialise un objet Julia en une `String` Julia contenant JSON:

```
julia> using JSON

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

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

Si une chaîne n'est pas souhaitée, JSON peut être imprimé directement dans un flux IO:

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

Notez que `STDOUT` est la valeur par défaut et peut être omis dans l'appel ci-dessus.

Une impression plus jolie peut être obtenue en passant le paramètre d' `indent` facultatif:

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

Il existe une sérialisation sérieuse par défaut pour les types complexes de Julia:

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

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

Lire JSON en ligne: https://riptutorial.com/fr/julia-lang/topic/5468/json

Chapitre 18: Les fonctions

Syntaxe

- f (n) = ...
- fonction f (n) ... fin
- n :: Type
- x -> ...
- f (n) do ... fin

Remarques

Outre les fonctions génériques (les plus courantes), il existe également des fonctions intégrées. Ces fonctions comprennent `is` , `isa` , `typeof` , `throw` , et des fonctions similaires. Les fonctions intégrées sont généralement implémentées dans C au lieu de Julia, elles ne peuvent donc pas être spécialisées sur les types d'argument pour dispatch.

Examples

Carré un numéro

C'est la syntaxe la plus simple pour définir une fonction:

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

Pour appeler une fonction, utilisez des parenthèses (sans espaces entre):

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

Les fonctions sont des objets dans Julia, et on peut les afficher dans la REPL comme avec tout autre objet:

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

Toutes les fonctions de Julia sont génériques (autrement dit polymorphes) par défaut. Notre fonction `square` fonctionne aussi bien avec des valeurs à virgule flottante:

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

... ou même des matrices :

```
julia> square([2 4
               2 1])
2×2 Array{Int64,2}:
 12  12
  6   9
```

Fonctions récursives

Récursion simple

En utilisant la récursivité et l' opérateur conditionnel ternaire , nous pouvons créer une implémentation alternative de la fonction `factorial` intégrée:

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

Usage:

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

Travailler avec des arbres

Les fonctions récursives sont souvent les plus utiles sur les structures de données, en particulier les structures de données arborescentes. Comme les expressions dans Julia sont des structures arborescentes, la récursivité peut être très utile pour la métaprogrammation . Par exemple, la fonction ci-dessous regroupe un ensemble de toutes les têtes utilisées dans une expression.

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

Nous pouvons vérifier que notre fonction fonctionne comme prévu:

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

Cette fonction est compacte et utilise une variété de techniques plus avancées, telles que la fonction de `reduce` l'ordre supérieur , le type de données `Set` et les expressions du générateur.

Introduction à la répartition

Nous pouvons utiliser la syntaxe `::` pour envoyer le type d'un argument.

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

Usage:

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

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

Contrairement à de nombreux langages, qui fournissent généralement soit une distribution multiple statique, soit une distribution unique dynamique, Julia dispose d'une répartition multiple dynamique. C'est-à-dire que les fonctions peuvent être spécialisées pour plusieurs arguments. Cela s'avère pratique lorsque vous définissez des méthodes spécialisées pour des opérations sur certains types et des méthodes de secours pour d'autres types.

```
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"
```

Usage:

```
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"
```

Arguments optionnels

Julia permet aux fonctions de prendre des arguments optionnels. Dans les coulisses, ceci est mis en œuvre comme un autre cas particulier d'expédition multiple. Par exemple, résolvons le problème populaire de Fizz Buzz . Par défaut, nous le ferons pour les nombres dans la gamme `1:10` , mais nous autoriserons une valeur différente si nécessaire. Nous autoriserons également différentes phrases à utiliser pour `Fizz` ou `Buzz` .

```
function fizzbuzz(xs=1:10, fizz="Fizz", buzz="Buzz")
    for i in xs
        if i % 15 == 0
            println(fizz, buzz)
        elseif i % 3 == 0
            println(fizz)
        elseif i % 5 == 0
            println(buzz)
        else
            println(i)
        end
    end
end
```

Si nous inspectons `fizzbuzz` dans le REPL, il y a quatre méthodes. Une méthode a été créée pour chaque combinaison d'arguments autorisée.

```
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
```

Nous pouvons vérifier que nos valeurs par défaut sont utilisées quand aucun paramètre n'est fourni:

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

mais que les paramètres optionnels sont acceptés et respectés si nous les fournissons:

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

Envoi paramétrique

Il est fréquent qu'une fonction soit envoyée sur des types paramétriques, tels que `Vector{T}` ou `Dict{K,V}` , mais les paramètres de type ne sont pas fixes. Ce cas peut être traité en utilisant la répartition paramétrique:

```
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"
```

On peut être tenté d'écrire simplement `xs::Vector{Number}` . Mais cela ne fonctionne que pour les objets dont le type est explicitement `Vector{Number}` :

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

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

Cela est dû à l' invariance paramétrique : l'objet `Int[1, 2]` n'est *pas* un `Vector{Number}` , car il ne peut contenir que des `Int` , alors qu'un `Vector{Number}` devrait pouvoir contenir n'importe quel type de nombres.

Rédaction de code générique

Dispatch est une fonctionnalité incroyablement puissante, mais il est souvent préférable d'écrire du code générique qui fonctionne pour tous les types, au lieu de spécialiser le code pour chaque type. L'écriture de code générique évite la duplication de code.

Par exemple, voici le code pour calculer la somme des carrés d'un vecteur d'entiers:

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

Mais ce code *ne* fonctionne que pour un vecteur de `Int` s. Cela ne fonctionnera pas sur un `UnitRange` :

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

Cela ne fonctionnera pas sur un `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
```

Une meilleure façon d'écrire cette fonction `sumsq` devrait être

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

Cela fonctionnera sur les deux cas énumérés ci-dessus. Mais il y a des collections que nous voudrions peut-être additionner à des carrés qui ne sont pas du tout des vecteurs. Par exemple,

```
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
```

montre que nous ne pouvons pas additionner les carrés d'une itération paresseuse .

Une implémentation encore plus générique est simplement

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

Ce qui fonctionne dans tous les cas:

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

C'est le code de Julia le plus idiomatique, capable de gérer toutes sortes de situations. Dans certaines autres langues, la suppression des annotations de type peut affecter les performances, mais ce n'est pas le cas dans Julia; seule la stabilité de type est importante pour la performance.

Factorielle impérative

Une syntaxe longue est disponible pour définir des fonctions multilignes. Cela peut être utile lorsque nous utilisons des structures impératives telles que des boucles. L'expression en queue est renvoyée. Par exemple, la fonction ci-dessous utilise une boucle `for` pour calculer la factorielle d'un entier `n` :

```
function myfactorial(n)
```

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

Usage:

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

Dans les fonctions plus longues, il est courant de voir la déclaration de `return` utilisée. L'instruction de `return` n'est pas nécessaire en position de queue, mais elle est parfois utilisée pour plus de clarté. Par exemple, une autre manière d'écrire la fonction ci-dessus serait

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

qui a un comportement identique à la fonction ci-dessus.

Fonctions anonymes

Syntaxe de flèche

Les fonctions anonymes peuvent être créées en utilisant la syntaxe `->` . Ceci est utile pour transmettre des fonctions à des fonctions de niveau supérieur , telles que la fonction de `map` . La fonction ci-dessous calcule le carré de chaque nombre dans un tableau `A`

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

Un exemple d'utilisation de cette fonction:

```
julia> squareall(1:10)
10-element Array{Int64,1}:
   1
   4
   9
  16
  25
  36
  49
  64
  81
 100
```

Syntaxe multiligne

Des fonctions anonymes multilignes peuvent être créées à l'aide de la syntaxe de `function` . Par exemple, l'exemple suivant calcule les factorielles des `n` premiers nombres, mais en utilisant une fonction anonyme au lieu de la `factorial` intégrée.

```
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
```

Faire une syntaxe de bloc

Comme il est si courant de passer une fonction anonyme comme premier argument à une fonction, il existe une syntaxe de bloc `do` . La syntaxe

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

est équivalent à

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

mais le premier peut être plus clair dans de nombreuses situations, surtout si beaucoup de calculs sont effectués dans la fonction anonyme. `do` syntaxe de bloc est particulièrement utile pour les entrées et sorties de fichiers pour des raisons de gestion des ressources.

Lire Les fonctions en ligne: https://riptutorial.com/fr/julia-lang/topic/3079/les-fonctions

Chapitre 19: Les types

Syntaxe

- MyType immuable champ; champ; fin
- tapez MyType; champ; champ; fin

Remarques

Les types sont la clé de la performance de Julia. Une idée importante pour la performance est la stabilité des types , qui se produit lorsque le type renvoyé par une fonction dépend uniquement des types, et non des valeurs, de ses arguments.

Examples

Envoi sur types

Sur Julia, vous pouvez définir plusieurs méthodes pour chaque fonction. Supposons que nous définissions trois méthodes de la même fonction:

```
foo(x) = 1
foo(x::Number) = 2
foo(x::Int) = 3
```

Lors du choix de la méthode à utiliser (appelée dispatch), Julia choisit la méthode plus spécifique correspondant aux types d'arguments:

```
julia> foo('one')
1

julia> foo(1.0)
2

julia> foo(1)
3
```

Cela facilite le polymorphisme . Par exemple, nous pouvons facilement créer une liste chaînée en définissant deux types immuables, nommés `Nil` et `Cons` . Ces noms sont traditionnellement utilisés pour décrire une liste vide et une liste non vide, respectivement.

```
abstract LinkedList
immutable Nil <: LinkedList end
immutable Cons <: LinkedList
    first
    rest::LinkedList
end
```

Nous représenterons la liste vide par `Nil()` et par toute autre liste par `Cons(first, rest)` , où `first` est le premier élément de la liste liée et le `rest` est la liste chaînée composée de tous les éléments restants. Par exemple, la liste `[1, 2, 3]` sera représentée comme

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

La liste est-elle vide?

Supposons que nous voulions étendre la fonction `isempty` la bibliothèque standard, qui fonctionne sur différentes collections:

```
julia> methods(isempty)
# 29 methods for generic function "isempty":
isempty(v::SimpleVector) at essentials.jl:180
isempty(m::Base.MethodList) at reflection.jl:394
...
```

Nous pouvons simplement utiliser la syntaxe de répartition de la fonction et définir deux méthodes supplémentaires d' `isempty` . Comme cette fonction provient du module `Base` , nous devons la qualifier de `Base.isempty` afin de l'étendre.

```
Base.isempty(::Nil) = true
Base.isempty(::Cons) = false
```

Ici, nous n'avons pas du tout besoin des valeurs d'argument pour déterminer si la liste est vide. Le seul type suffit pour calculer cette information. Julia nous permet d'omettre les noms des arguments, en gardant uniquement leur annotation de type, si nous n'avons pas besoin d'utiliser leurs valeurs.

Nous pouvons tester que nos méthodes `isempty` fonctionnent:

```
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())))))
```

et en effet le nombre de méthodes pour `isempty` a augmenté de `2` :

```
julia> methods(isempty)
# 31 methods for generic function "isempty":
isempty(v::SimpleVector) at essentials.jl:180
isempty(m::Base.MethodList) at reflection.jl:394
```

De toute évidence, déterminer si une liste chaînée est vide ou non est un exemple trivial. Mais

cela mène à quelque chose de plus intéressant:

Combien de temps dure la liste?

La fonction `length` de la bibliothèque standard nous donne la longueur d'une collection ou de certaines itérations . Il existe plusieurs façons de mettre en œuvre une `length` pour une liste chaînée. En particulier, en utilisant un `while` boucle est probablement le plus rapide et le plus efficace mémoire de Julia. Mais l' optimisation prématurée doit être évitée, alors supposons que notre liste chaînée ne soit pas efficace. Quelle est la manière la plus simple d'écrire une fonction de `length` ?

```
Base.length(::Nil) = 0
Base.length(xs::Cons) = 1 + length(xs.rest)
```

La première définition est simple: une liste vide a une longueur de `0` . La deuxième définition est également facile à lire: pour compter la longueur d'une liste, nous comptons le premier élément, puis comptons la longueur du reste de la liste. Nous pouvons tester cette méthode de la même manière que nous avons testé `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
```

Prochaines étapes

Cet exemple de jouet est loin d'implémenter toutes les fonctionnalités souhaitées dans une liste chaînée. Il manque, par exemple, l'interface d'itération. Cependant, il illustre comment la répartition peut être utilisée pour écrire un code court et clair.

Types immuables

Le type composite le plus simple est un type immuable. Les instances de types immuables, comme les tuples , sont des valeurs. Leurs champs ne peuvent pas être modifiés après leur création. À bien des égards, un type immuable est comme un `Tuple` avec des noms pour le type lui-même et pour chaque champ.

Types singleton

Les types composites, par définition, contiennent un certain nombre de types plus simples. En Julia, ce nombre peut être zéro; c'est-à-dire qu'un type immuable est autorisé à *ne* contenir *aucun* champ. Ceci est comparable au tuple vide `()` .

Pourquoi cela pourrait-il être utile? Ces types immuables sont connus sous le nom de "types singleton", car une seule d'entre eux pourrait exister. Les valeurs de ces types sont appelées "valeurs singleton". La bibliothèque standard `Base` contient de nombreux types singleton. Voici une brève liste:

- `Void` , le type de `nothing` . Nous pouvons vérifier que `Void.instance` (qui est une syntaxe spéciale pour récupérer la valeur singleton d'un type singleton) n'est en effet `nothing` .
- Tout type de média, tel que `MIME"text/plain"` , est un type singleton avec une seule instance, `MIME("text/plain")` .
- Les types `Irrational{:π}` , `Irrational{:e}` , `Irrational{:ψ}` et similaires sont des types singleton, et leurs instances singleton sont les valeurs irrationnelles `π = 3.1415926535897...` , etc.
- Les traits de taille de l'itérateur `Base.HasLength` , `Base.HasShape` , `Base.IsInfinite` et `Base.SizeUnknown` sont tous des types singleton.

0.5.0

- Dans la version 0.5 et les versions ultérieures, chaque fonction est une instance singleton d'un type singleton! Comme toute autre valeur de singleton, nous pouvons récupérer la fonction `sin` , par exemple, de `typeof(sin).instance` .

Comme ils ne contiennent rien, les types singleton sont incroyablement légers et ils peuvent souvent être optimisés par le compilateur pour ne pas avoir de surcharge d'exécution. Ainsi, ils sont parfaits pour les traits, les valeurs d'étiquettes spéciales et pour des choses comme les fonctions que l'on aimerait se spécialiser.

Pour définir un type de singleton,

```
julia> immutable MySingleton end
```

Pour définir une impression personnalisée pour le type singleton,

```
julia> Base.show(io::IO, ::MySingleton) = print(io, "sing")
```

Pour accéder à l'instance singleton,

```
julia> MySingleton.instance
MySingleton()
```

Souvent, on assigne ceci à une constante:

```
julia> const sing = MySingleton.instance
MySingleton()
```

Types d'emballage

Si les types immuables du champ zéro sont intéressants et utiles, alors les types immuables à un

champ sont peut-être encore plus utiles. De tels types sont communément appelés "types de wrappers" car ils encapsulent certaines données sous-jacentes, fournissant une interface alternative auxdites données. Un exemple de type de wrapper dans `Base` est `String` . Nous allons définir un type similaire à `String` , nommé `MyString` . Ce type sera soutenu par un vecteur (tableau à une dimension) d'octets (`UInt8`).

Tout d'abord, la définition de type elle-même et certains éléments personnalisés:

```
immutable MyString <: AbstractString
    data::Vector{UInt8}
end

function Base.show(io::IO, s::MyString)
    print(io, "MyString: ")
    write(io, s.data)
    return
end
```

Maintenant, notre type `MyString` est prêt à être utilisé! Nous pouvons lui fournir des données UTF-8 brutes, et cela s'affiche comme nous l'aimons pour:

```
julia> MyString([0x48,0x65,0x6c,0x6c,0x6f,0x2c,0x20,0x57,0x6f,0x72,0x6c,0x64,0x21])
MyString: Hello, World!
```

Evidemment, ce type de chaîne nécessite beaucoup de travail avant de devenir aussi utilisable que le type `Base.String` .

Véritables types composites

Le plus souvent, de nombreux types immuables contiennent plus d'un champ. Un exemple est le type de bibliothèque standard `Rational{T}` , qui contient deux champs: un champ `num` pour le numérateur et un champ `den` pour le dénominateur. Il est assez simple d'émuler ce type de conception:

```
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)
```

Nous avons implémenté avec succès un constructeur qui simplifie nos nombres rationnels:

```
julia> MyRational(10, 6)
MyRational{Int64}(5,3)
```

Lire Les types en ligne: https://riptutorial.com/fr/julia-lang/topic/5467/les-types

Chapitre 20: Lire un DataFrame à partir d'un fichier

Examples

Lecture d'un dataframe à partir de données séparées par un délimiteur

Vous souhaiterez peut-être lire un `DataFrame` partir d'un fichier CSV (valeurs séparées par des virgules) ou peut-être même d'un fichier TSV ou WSV (fichiers séparés par des tabulations et des espaces). Si votre fichier a la bonne extension, vous pouvez utiliser la fonction `readtable` pour lire dans le dataframe:

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

Mais que faire si votre fichier n'a pas la bonne extension? Vous pouvez spécifier le délimiteur utilisé par votre fichier (virgule, tabulation, espace, etc.) comme argument de mot-clé de la fonction `readtable` :

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

Gestion des différents commentaires de commentaire

Les ensembles de données contiennent souvent des commentaires expliquant le format des données ou contenant les termes de la licence et de l'utilisation. Vous souhaitez généralement ignorer ces lignes lorsque vous lisez dans le `DataFrame` .

La fonction `readtable` suppose que les lignes de commentaires commencent par le caractère '#'. Cependant, votre fichier peut utiliser des marques de commentaires comme `%` ou `//` . Pour vous assurer que `readtable` gère correctement, vous pouvez spécifier la marque de commentaire comme argument de mot clé:

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

Lire Lire un DataFrame à partir d'un fichier en ligne: https://riptutorial.com/fr/julia-lang/topic/7340/lire-un-dataframe-a-partir-d-un-fichier

Chapitre 21: Macros de chaîne

Syntaxe

- macro "string" # short, chaîne de forme de macro
- @macro_str "string" # forme de macro longue et régulière
- macro`command`

Remarques

Les macros de chaînes de caractères ne sont pas aussi puissantes que les anciennes chaînes de caractères ordinaires - car l'interpolation doit être implémentée dans la logique de la macro, les macros de chaînes ne peuvent pas contenir de littéraux de chaîne du même délimiteur pour l'interpolation.

Par exemple, bien que

```
julia> "$("x")"
"x"
```

fonctionne, la forme de texte de macro de chaîne

```
julia> doc"$("x")"
ERROR: KeyError: key :x not found
```

est analysé de manière incorrecte. Cela peut être quelque peu atténué en utilisant des guillemets comme séparateur de chaîne externe;

```
julia> doc"""$("x")"""
"x"
```

fonctionne effectivement correctement.

Examples

Utiliser des macros de chaîne

Les macros de chaîne sont du sucre syntaxique pour certaines invocations de macros. L'analyseur développe la syntaxe comme

```
mymacro"my string"
```

dans

```
@mymacro_str "my string"
```

qui, comme tout autre appel de macro, est remplacé par l'expression que `@mymacro_str` macro `@mymacro_str` . La base Julia est fournie avec plusieurs macros de chaîne, telles que:

@b_str

Cette macro de chaîne construit des tableaux d' octets au lieu de chaînes . Le contenu de la chaîne, codé en UTF-8, sera utilisé comme tableau d'octets. Cela peut être utile pour l'interfaçage avec des API de bas niveau, dont beaucoup fonctionnent avec des tableaux d'octets au lieu de chaînes.

```
julia> b"Hello World!"
12-element Array{UInt8,1}:
 0x48
 0x65
 0x6c
 0x6c
 0x6f
 0x20
 0x57
 0x6f
 0x72
 0x6c
 0x64
 0x21
```

@big_str

Cette macro renvoie un `BigInt` ou un `BigFloat` analysé à partir de la chaîne fournie.

```
julia> big"1"
1

julia> big"1.0"
1.000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

Cette macro existe car `big(0.1)` ne se comporte pas comme on pouvait s'y attendre: le `0.1` est une approximation `Float64` de true `0.1` (`1//10`), et la promotion de cette `BigFloat` à `BigFloat` conservera l'erreur d'approximation de `Float64` . Utiliser la macro analysera `0.1` directement dans un `BigFloat` , réduisant ainsi l'erreur d'approximation.

```
julia> big(0.1)
1.000000000000000055511151231257827021181583404541015625000000000000000000000000e-01

julia> big"0.1"
1.000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000002e-01
```

@doc_str

Cette macro de chaîne `Base.Markdown.MD` objets `Base.Markdown.MD` , qui sont utilisés dans le système de documentation interne pour fournir une documentation en texte riche pour tout environnement. Ces objets MD se rendent bien dans un terminal:

```
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.
```

et aussi dans un navigateur:

```
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

Cette macro de chaîne construit des littéraux de chaîne HTML, qui s'affichent bien dans un navigateur:

```
In [1]: html"""
        <p><abbr title="Hypertext Markup Language">HTML</abbr> text.</p>
        """
```

Out[1]: HTML text.

@ip_str

Cette macro de chaîne construit des littéraux d'adresse IP. Il fonctionne avec IPv4 et IPv6:

```
julia> ip"127.0.0.1"
ip"127.0.0.1"

julia> ip"::"
ip"::"
```

@r_str

Cette macro de chaîne construit des littéraux `Regex` .

`@s_str`

Cette macro de chaîne construit des littéraux `SubstitutionString` , qui fonctionnent avec les littéraux `Regex` pour permettre une substitution textuelle plus avancée.

`@text_str`

Cette macro de chaîne est similaire dans l'esprit à `@doc_str` et `@html_str` , mais ne possède aucune fonctionnalité de formatage sophistiquée:

```
In [3]: text"""
        This is some plain text.
        """
Out[3]: This is some plain text.
```

`@v_str`

Cette macro de chaîne construit des littéraux `VersionNumber` . Voir les numéros de version pour une description de ce qu'ils sont et comment les utiliser.

`@MIME_str`

Cette macro de chaîne construit les types singleton des types MIME. Par exemple, `MIME"text/plain"` est le type de `MIME("text/plain")` .

Symboles qui ne sont pas des identifiants légaux

Les symboles littéraux de Julia doivent être des identificateurs légaux. Cela marche:

```
julia> :cat
:cat
```

Mais cela ne fait pas:

```
julia> :2cat
ERROR: MethodError: no method matching *(::Int64, ::Base.#cat)
Closest candidates are:
  *(::Any, ::Any, ::Any, ::Any...) at operators.jl:288

*{T<:Union{Int128,Int16,Int32,Int64,Int8,UInt128,UInt16,UInt32,UInt64,UInt8}}(::T<:Union{Int128,Int16,I
::T<:Union{Int128,Int16,Int32,Int64,Int8,UInt128,UInt16,UInt32,UInt64,UInt8}) at int.jl:33
  *(::Real, ::Complex{Bool}) at complex.jl:180
  ...
```

Ce qui ressemble à un littéral de symbole ici est en fait analysé comme une multiplication implicite de `:2` (qui est juste `2`) et la fonction `cat` , qui ne fonctionne évidemment pas.

On peut utiliser

```
julia> Symbol("2cat")
Symbol("2cat")
```

pour contourner le problème.

Une macro de chaîne pourrait aider à rendre ceci plus concis. Si nous définissons la macro `@sym_str` :

```
macro sym_str(str)
    Meta.quot(Symbol(str))
end
```

alors on peut simplement faire

```
julia> sym"2cat"
Symbol("2cat")
```

pour créer des symboles qui ne sont pas des identifiants Julia valides.

Bien entendu, ces techniques peuvent également créer des symboles qui *sont des* identifiants Julia valides. Par exemple,

```
julia> sym"test"
:test
```

Implémentation de l'interpolation dans une macro de chaîne

Les macros de chaîne ne sont pas fournies avec des fonctions d' interpolation intégrées. Cependant, il est possible d'implémenter manuellement cette fonctionnalité. Notez qu'il n'est pas possible d'incorporer sans échapper les littéraux de chaîne qui ont le même délimiteur que la macro de chaîne environnante; c'est-à-dire que `""" $("x") """` est possible, `" $("x") "` n'est pas. Au lieu de cela, cela doit être échappé comme `" $(\"x\") "` . Voir la section des remarques pour plus de détails sur cette limitation.

Il existe deux approches pour implémenter l'interpolation manuellement: implémenter l'analyse syntaxique manuellement ou demander à Julia d'effectuer l'analyse. La première approche est plus flexible, mais la seconde approche est plus facile.

Analyse manuelle

```
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 analyse

```
macro e_str(s)
    esc(parse("\"$(escape_string(s))\""))
end
```

Cette méthode échappe à la chaîne (mais notez que la `escape_string` n'échappe *pas* aux signes `$`) et la renvoie à l'analyseur syntaxique de Julia pour l'analyse. Echapper à la chaîne est nécessaire pour s'assurer que `"` et `\` n'affecte pas l'analyse de la chaîne. L'expression résultante est une expression `:string` , qui peut être examinée et décomposée à des fins macro.

Macros de commande

0.6.0-dev

Dans Julia v0.6 et versions ultérieures, les macros de commandes sont prises en charge en plus des macros de chaîne standard. Une invocation de macro de commande comme

```
mymacro`xyz`
```

est analysé comme l'appel de macro

```
@mymacro_cmd "xyz"
```

Notez que cela est similaire aux macros de chaînes, sauf avec `_cmd` au lieu de `_str` .

Nous utilisons généralement des macros de commande pour le code, ce qui dans de nombreuses langues contient souvent `"` mais contient rarement `` ` `` , par exemple, il est assez facile de ré - écrire une version simple. Quasiquoting en utilisant des macros de commande:

```
macro julia_cmd(s)
    esc(Meta.quot(parse(s)))
end
```

Nous pouvons utiliser cette macro soit en ligne:

```
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)
```

ou multiligne:

```
julia> julia```
       function hello()
           println("Hello, World!")
       end
       ```
:(function hello() # none, line 2:
        println("Hello, World!")
    end)
```

L'interpolation à l'aide de `$` est prise en charge:

```
julia> x = 2
2

julia> julia`1 + $x`
:(1 + 2)
```

mais la version donnée ici ne permet qu'une seule expression:

```
julia> julia```
       x = 2
       y = 3
       ```
ERROR: ParseError("extra token after end of expression")
```

Cependant, l'étendre pour gérer plusieurs expressions n'est pas difficile.

Lire Macros de chaîne en ligne: https://riptutorial.com/fr/julia-lang/topic/5817/macros-de-chaine

Chapitre 22: Métaprogrammation

Syntaxe

- nom de la macro (ex) ... fin
- citation ... fin
- : (...)
- $ x
- Meta.quot (x)
- QuoteNode (x)
- esc (x)

Remarques

Les fonctionnalités de métaprogrammation de Julia sont fortement inspirées de celles des langages de type Lisp et sembleront familières à ceux qui ont un fond Lisp. La métaprogrammation est très puissante. Utilisé correctement, il peut conduire à un code plus concis et lisible.

La `quote ... end` est la syntaxe quasiquote. Au lieu des expressions en cours d'évaluation, elles sont simplement analysées. La valeur de l'expression de `quote ... end` est l'arbre de syntaxe abstraite résultant (AST).

La syntaxe `:(...)` est similaire à la syntaxe `quote ... end` , mais elle est plus légère. Cette syntaxe est plus concise que la `quote ... end` .

À l'intérieur d'une quasiquote, l'opérateur `$` est spécial et *interpole* son argument dans l'AST. L'argument devrait être une expression qui est directement raccordée à l'AST.

La fonction `Meta.quot(x)` cite son argument. Ceci est souvent utile en combinaison avec l'utilisation de `$` pour l'interpolation, car cela permet d'épisser littéralement les expressions et les symboles dans l'AST.

Examples

Réimplémenter la macro @show

Dans Julia, la macro `@show` est souvent utile à des fins de débogage. Il affiche à la fois l'expression à évaluer et son résultat, renvoyant finalement la valeur du résultat:

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

Il est simple de créer notre propre version de `@show` :

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

Pour utiliser la nouvelle version, utilisez simplement la macro `@myshow` :

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

julia> x
2
```

Jusqu'à la boucle

Nous sommes tous habitués à la syntaxe `while` , qui exécute son corps alors que la condition est évaluée à `true` . Que faire si nous voulons mettre en œuvre un `until` la boucle, qui exécute une boucle jusqu'à ce que la condition est évaluée à `true` ?

En Julia, nous pouvons le faire en créant une macro `@until` , qui cesse d'exécuter son corps lorsque la condition est remplie:

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

Nous avons utilisé ici la syntaxe de chaînage des fonctions `|>` , ce qui équivaut à appeler la fonction `esc` sur tout le bloc de `quote` . La fonction `esc` empêche l'application d'une macro hygiène au contenu de la macro. sans elle, les variables définies dans la macro seront renommées pour éviter les collisions avec des variables externes. Voir la documentation Julia sur la macro hygiène pour plus de détails.

Vous pouvez utiliser plus d'une expression dans cette boucle, en mettant simplement tout dans un bloc de `begin ... end`

```
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 et Expr (: quote)

Il y a trois façons de citer quelque chose en utilisant une fonction Julia:

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

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

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

Que signifie "citer", et à quoi cela sert-il? La citation nous permet de protéger les expressions contre l'interprétation de formes spéciales par Julia. Un cas d'utilisation courant est lorsque nous générons des expressions qui doivent contenir des éléments qui évaluent les symboles. (Par exemple, cette macro doit renvoyer une expression qui évalue un symbole.) Cela ne fonctionne pas simplement pour renvoyer le symbole:

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

julia> @mysym
ERROR: UndefVarError: x not defined

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

Que se passe t-il ici? `@mysym` développe en `:x` , qui en tant qu'expression devient la variable `x` . Mais rien n'a encore été attribué à `x` , nous obtenons donc une erreur `x not defined` .

Pour contourner ce problème, nous devons citer le résultat de notre macro:

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

julia> @mysym2
:x

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

Ici, nous avons utilisé la fonction `Meta.quot` pour transformer notre symbole en un symbole entre

guillemets, ce qui est le résultat souhaité.

Quelle est la différence entre `Meta.quot` et `QuoteNode` , et que dois-je utiliser? Dans presque tous les cas, la différence importe peu. Il est peut-être un peu plus sûr parfois d'utiliser `QuoteNode` au lieu de `Meta.quot` . Explorer la différence est instructif sur le fonctionnement des expressions et des macros Julia.

La différence entre `Meta.quot` et `QuoteNode` , expliquée

Voici une règle générale:

- Si vous avez besoin ou souhaitez prendre en charge l'interpolation, utilisez `Meta.quot` ;
- Si vous ne pouvez pas ou ne voulez pas autoriser l'interpolation, utilisez `QuoteNode` .

En bref, la différence est que `Meta.quot` permet l'interpolation dans la chose citée, tandis que `QuoteNode` protège son argument de toute interpolation. Pour comprendre l'interpolation, il est important de mentionner l'expression `$` . Il y a une sorte d'expression dans Julia appelée `$` expression. Ces expressions permettent de s'échapper. Par exemple, considérons l'expression suivante:

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

Lorsqu'elle est évaluée, cette expression évalue `1` et l'assigne à `x` , puis construit une expression de la forme `_ + _` où le _ sera remplacé par la valeur de `x` . Ainsi, le résultat devrait être l'*expression* `1 + 1` (qui n'est pas encore évaluée, et donc distincte de la *valeur* `2`). En effet, c'est le cas:

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

Disons maintenant que nous écrivons une macro pour construire ces types d'expressions. Notre macro prendra un argument qui remplacera le `1` dans l' `ex` ci-dessus. Cet argument peut être n'importe quelle expression, bien sûr. Voici quelque chose qui n'est pas tout à fait ce que nous voulons:

```
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
```

Le deuxième cas est incorrect, car nous devrions garder `1 + 1` évalué. Nous corrigeons cela en citant l'argument avec `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
```

La macro hygiène ne s'applique pas au contenu d'un devis, donc s'échapper n'est pas nécessaire dans ce cas (et en fait pas légal) dans ce cas.

Comme mentionné précédemment, `Meta.quot` permet l'interpolation. Alors essayons ça:

```
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
```

À partir du premier exemple, nous voyons que l'interpolation nous permet d'inclure le `sin(1)` , au lieu d'avoir l'expression comme un `sin(1)` littéral `sin(1)` . Le second exemple montre que cette interpolation est effectuée dans la portée de l'invocation de la macro, et non dans la portée de la macro. C'est parce que notre macro n'a pas réellement évalué de code; tout ce qu'il fait génère du code. L'évaluation du code (qui fait son chemin dans l'expression) est effectuée lorsque l'expression générée par la macro est réellement exécutée.

Et si on avait utilisé `QuoteNode` place? Comme vous pouvez le deviner, puisque `QuoteNode` empêche l'interpolation, cela ne fonctionnera pas.

```
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
```

Dans cet exemple, nous pourrions convenir que `Meta.quot` offre une plus grande flexibilité, car il permet une interpolation. Alors, pourquoi pourrions-nous jamais envisager d'utiliser `QuoteNode` ? Dans certains cas, nous ne souhaitons peut-être pas réellement une interpolation et souhaitons en fait l'expression littérale `$` . Quand cela serait-il souhaitable? Considérons une généralisation de `@makeex` où nous pouvons passer des arguments supplémentaires déterminant ce qui se passe à gauche et à droite du signe `+` :

```
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)
```

Une limitation de notre implémentation de `@makeex4` est que nous ne pouvons pas utiliser directement les expressions comme côtés droit et gauche de l'expression, car elles sont interpolées. En d'autres termes, les expressions peuvent être évaluées pour l'interpolation, mais nous pourrions vouloir les conserver. (Comme il y a plusieurs niveaux de cotation et d'évaluation ici, clarifions: notre macro génère du *code* qui construit une *expression* qui, lorsqu'elle est évaluée, produit une autre *expression* . Ouf!)

```
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)
```

Nous devrions permettre à l'utilisateur de spécifier quand l'interpolation doit avoir lieu, et quand elle ne devrait pas. Théoriquement, c'est une solution facile: nous pouvons simplement supprimer l'un des signes `$` dans notre application et laisser l'utilisateur contribuer. Cela signifie que nous interpolons une version citée de l'expression saisie par l'utilisateur (que nous avons déjà citée et interpolée une fois). Cela conduit au code suivant, qui peut être un peu déroutant au début, en raison des multiples niveaux imbriqués de cotation et de non-classement. Essayez de lire et de comprendre à quoi sert chaque évasion.

```
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
```

Les choses ont bien commencé, mais quelque chose a mal tourné. Le code généré par la macro tente d'interpoler la copie de `y` dans la portée de l'invocation de la macro. mais il n'y a *pas de* copie de `y` dans la portée de l'invocation de macro. Notre erreur permet l'interpolation avec les deuxième et troisième arguments de la macro. Pour corriger cette erreur, nous devons utiliser `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)
```

En utilisant `QuoteNode` , nous avons protégé nos arguments de l'interpolation. Comme `QuoteNode` n'a pour effet que des protections supplémentaires, il n'est jamais nuisible d'utiliser `QuoteNode` , sauf si vous souhaitez une interpolation. Cependant, comprendre la différence permet de comprendre où et pourquoi `Meta.quot` pourrait être un meilleur choix.

Ce long exercice est un exemple qui est clairement trop complexe pour apparaître dans une application raisonnable. Par conséquent, nous avons justifié la règle suivante, mentionnée précédemment:

- Si vous avez besoin ou souhaitez prendre en charge l'interpolation, utilisez `Meta.quot` ;
- Si vous ne pouvez pas ou ne voulez pas autoriser l'interpolation, utilisez `QuoteNode` .

Qu'en est-il d'Expr (: citation)?

`Expr(:quote, x)` est équivalent à `Meta.quot(x)` . Cependant, ce dernier est plus idiomatique et est préféré. Pour le code qui utilise beaucoup la métaprogrammation, une ligne `using Base.Meta` est souvent utilisée, ce qui permet de `Meta.quot` simplement `quot`

Les bits et bobs de métaprogrammation de π

Buts:

- Enseigner à l'aide d'exemples fonctionnels / utiles / non abstraits `@swap` (par exemple, `@swap` ou `@assert`) qui introduisent des concepts dans des contextes appropriés
- Préfère laisser le code illustrer / démontrer les concepts plutôt que des paragraphes d'explication
- Évitez de lier la «lecture requise» aux autres pages - cela interrompt la narration
- Présenter les choses dans un ordre raisonnable qui rendra l'apprentissage plus facile

Ressources:

julialang.org
wikibook (@Cormullion)
5 couches (Leah Hanson)
SO-Doc Quoting (@TotalVerb)
SO-Doc - Symboles qui ne sont pas des identifiants légaux (@TotalVerb)
SO: Qu'est-ce qu'un symbole dans Julia (@StefanKarpinski)
Fil de discussion (@ pi-) Métaprogrammation

La majeure partie du matériel provient de la chaîne discursive, la plus grande partie provient de fcard ... s'il vous plaît, pousse-moi si j'avais des attributions oubliées.

symbole

```
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
```

Passer des drapeaux en fonctions:

```
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
```

Un exemple de hashkey:

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

(Advanced) (@fcard) `:foo` aka `:(foo)` donne un symbole si `foo` est un identifiant valide, sinon une expression.

```
# 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
...
```

Fondamentalement, vous pouvez traiter les symboles comme des chaînes légères. Ce n'est pas ce qu'ils sont pour, mais vous pouvez le faire, alors pourquoi pas. La base de Julia elle-même le fait, `print_with_color(:red, "abc")` imprime un abc de couleur rouge.

Expr (AST)

(Presque) tout dans Julia est une expression, c'est-à-dire une instance de `Expr` , qui contiendra un AST .

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

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

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

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

# TRY: fieldnames(typeof(ast))

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

Imbrication `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))
```

`Expr` s utilisant `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
```

... donc `quote` est fonctionnellement le même mais fournit des informations de débogage supplémentaires.

(*) **ASTUCE** : utilisez `let` pour garder `x` dans le bloc

`quote` un `quote`

`Expr(:quote, x)` est utilisé pour représenter des guillemets entre guillemets.

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

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

`QuoteNode(x)` est similaire à `Expr(:quote, x)` mais empêche l'interpolation.

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

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

(Désambiguïser les différents mécanismes de citation dans la métaprogrammation de Julia

Est-ce que $ et : (...) sont en quelque sorte inversés?

`:(foo)` signifie "ne regarde pas la valeur, regarde l'expression" `$foo` signifie "change l'expression à sa valeur"

`:($(foo)) == foo . $(:(foo))` est une erreur. `$(...)` n'est pas une opération et ne fait rien par lui-même, c'est un "interpoler ça!" signe que la syntaxe de citation utilise. C'est-à-dire qu'il n'existe

que dans une citation.

Est-ce que `$ foo` le même que `eval( foo )` ?

Non! `$foo` est échangé pour la valeur de compilation `eval(foo)` signifie le faire à l'exécution

`eval` se produira dans l'interpolation de portée globale locale

`eval(:<expr>)` doit renvoyer la même chose que simplement `<expr>` (en supposant que `<expr>` soit une expression valide dans l'espace global actuel)

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

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

`macro` S

Prêt? :)

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

Faisons notre propre macro `@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` pour baisser un `Expr`

5 couches (Leah Hanson) <- explique comment Julia prend le code source comme une chaîne, il tokenizes dans un `Expr` -tree (AST), étend toutes les macros (encore AST), ***réduit*** (réduit AST), convertit ensuite en LLVM (et au-delà - en ce moment, nous n'avons pas besoin de nous inquiéter de ce qui se trouve au-delà!)

Q: `code_lowered` agit sur les fonctions. Est-il possible d'abaisser un `Expr` ? 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)
```

Si vous souhaitez uniquement développer des macros, vous pouvez utiliser **`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
```

... qui renvoie un AST non abaissé mais avec toutes les macros développées.

`esc()`

`esc(x)` renvoie un Expr qui dit "ne pas appliquer d'hygiène à ceci", c'est la même chose `Expr(:escape, x)` . L'hygiène est ce qui garde une macro autonome, et vous `esc` choses si vous voulez qu'elles fuient. par exemple

Exemple: `swap` macro pour illustrer `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
```

`$` nous permet d'échapper à la `quote` . Alors pourquoi ne pas simplement `$p` et `$q` ? c'est à dire

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

Parce que cela regarderait d'abord la portée de `macro` pour `p` , et trouverait un `p` local, c'est-à-dire le paramètre `p` (oui, si vous accédez ensuite à `p` sans `esc` , la macro considère le paramètre `p` comme une variable locale).

Donc, `$p = ...` est juste une assignation au `p` local. cela n'affecte pas la variable transmise dans le contexte d'appel.

Ok, alors qu'en est-il de:

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

Donc , `esc(p)` est « » fuite `p` dans le contexte d'appel. *"La chose qui a été passée dans la macro que* ***nous recevons comme*** `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
```

Comme vous pouvez le voir, `tmp` obtient le traitement d'hygiène `#10#tmp` , alors que `x` et `y` ne le font pas. Julia `gensym` un identifiant unique pour `tmp` , ce que vous pouvez faire manuellement avec `gensym` , à savoir:

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

Mais: **il y a un gotcha:**

```
julia> module Swap
       export @swap

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

julia> using Swap

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

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

Une autre chose que fait la macro hygiène de Julia, c'est que si la macro provient d'un autre module, les variables (qui n'ont pas été assignées dans la macro, comme `tmp` dans ce cas) sont globales au module actuel, donc `$p` devient `Swap.$p` , de même `$q` -> `Swap.$q` .

En général, si vous avez besoin d'une variable en dehors de la portée de la macro, vous devriez l'écouter. Vous devriez donc vous `esc(p)` et `esc(q)` qu'elles se trouvent sur le LHS ou le RHS d'une expression, voire sur elles-mêmes.

les gens ont déjà mentionné quelques fois le `gensym` sa et bientôt vous serez séduit par le côté sombre du défaut d'échapper à toute l'expression avec quelques `gensym` ici et là, mais ... Assurez-vous de comprendre comment fonctionne l'hygiène avant d'essayer d'être plus intelligent que ça! Ce n'est pas un algorithme particulièrement complexe, donc cela ne devrait pas prendre trop de temps, mais ne vous précipitez pas! N'utilisez pas ce pouvoir tant que vous n'en avez pas compris toutes les ramifications ... (@fcard)

Exemple: `until` macro

(@ 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) `|>` est controversé, cependant. Je suis surpris qu'une foule n'ait pas encore discuté. (peut-être que tout le monde en a assez). Il y a une recommandation d'avoir la plupart sinon la totalité de la macro soit juste un appel à une fonction, donc:

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

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

... est une alternative plus sûre.

Le défi macro simple de ## @ fcard

Tâche: permuter les opérandes, donc `swaps(1/2)` donne `2.00` ie `2/1`

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

Plus de défis macro à partir de @fcard ici

Interpolation et `assert` macro

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

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

Q: Pourquoi le dernier `$` ? R: Il interpole, c'est-à-dire qu'il force Julia à `eval` cette `string(ex)` lorsque l'exécution passe par l'invocation de cette macro. c.-à-d. si vous exécutez simplement ce code, cela ne forcera aucune évaluation. Mais dès que vous `assert(foo)` Julia ***invoquera*** cette macro en remplaçant son jeton / expr AST par ce qu'elle renvoie, et le `$` *mettra* en action.

Un hack amusant pour utiliser {} pour les blocs

(@fcard) Je ne pense pas qu'il y ait quelque chose de technique empêchant `{}` d'être utilisé comme des blocs, en fait on peut même caler la syntaxe résiduelle `{}` pour la faire fonctionner:

```
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
```

* (peu susceptible de continuer si / quand la syntaxe {} est réutilisée)

Donc, d'abord, Julia voit le jeton de macro, donc il va lire / analyser les jetons jusqu'à la `end` correspondante, et créer quoi? Un `Expr` avec `.head=:macro` ou quelque chose? Stocke-t-il `"a+1"` tant que chaîne ou le sépare-t-il en `:+(:a, 1)` ? Comment voir?

?

(@fcard) Dans ce cas, à cause de la portée lexicale, a n'est pas défini dans la `@M` de `@M` , il utilise donc la variable globale ... J'ai en fait oublié d'échapper à l'expression flipplin 'dans mon exemple idiot, mais *la même module "une* partie de cela s'applique toujours.

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

julia> a = 1
1

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

La raison en est que, si la macro est utilisée dans un module autre que celui dans lequel elle a été définie, toutes les variables non définies dans le code à développer sont traitées comme des globales du module de la macro.

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

AVANCÉE

@ 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
```

La macro de Scott:

```
"""
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 / non traité ...

afficher / vider une macro

(@ pi-) Supposons que je fasse juste la `macro m(); a+1; end` dans un nouveau REPL. Sans `a` définition. Comment puis-je le voir? Par exemple, existe-t-il un moyen de "vider" une macro? Sans l'exécuter réellement

(@fcard) Tout le code dans les macros est effectivement mis en fonctions, de sorte que vous ne pouvez voir que leur code abaissé ou inféré.

```
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!
```

Autres moyens d'obtenir la fonction d'une macro:

```
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)
```

^ semble que je puisse utiliser `code_typed` place

Comment comprendre `eval(Symbol("@M"))` ?

(@fcard) Actuellement, chaque macro est associée à une fonction. Si vous avez une macro appelée `M` , la fonction de la macro s'appelle `@M` . En général, vous pouvez obtenir une valeur de fonction avec par exemple `eval(:print)` mais avec une fonction de macro, vous devez faire `Symbol("@M")` , car juste `:@M` devient un `Expr(:macrocall, Symbol("@M"))` et l'évaluation qui provoque une macro-expansion.

Pourquoi ne pas afficher les paramètres de `code_typed` ?

(@pi-)

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

^ ici je vois un `::Any` paramètre, mais il ne semble pas être connecté avec le jeton `x` .

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

^ pareil ici; il n'y a rien de se connecter `io` avec le `::IO` donc sûrement cela ne peut pas être une décharge complète de la représentation AST de cette particulière `print` méthode ...?

(@fcard) `print(::IO, ::Char)` ne vous dit que la méthode utilisée, elle ne fait pas partie de l'AST. Il n'est même plus présent dans master:

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

(@ pi-) Je ne comprends pas ce que tu veux dire par là. Il semble que le dumping de l'AST pour le corps de cette méthode, non? Je pensais que `code_typed` donne l'AST pour une fonction. Mais il semble manquer la première étape, à savoir la mise en place de jetons pour les paramètres.

(@fcard) `code_typed` est censé afficher uniquement l'AST du corps, mais pour l'instant il donne l'AST complète de la méthode, sous la forme d'un `LambdaInfo` (0.5) ou `CodeInfo` (0.6), mais beaucoup d'informations sont omises lorsqu'il est imprimé sur le repl. Vous devrez inspecter le champ `LambdaInfo` par champ pour obtenir tous les détails. `dump` va inonder votre réplique, vous pouvez donc essayer:

```
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])
```

Ce qui donne toutes les valeurs des champs nommés de l'AST d'une méthode:

```
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)
```

Voir le lil ' `def = print(io::IO, c::Char)` ? Voilà! (aussi les `slotnames = [..., :io, :c]` part) Aussi oui, la différence de sortie est que je montrais les résultats sur master.

???

(@ Ismael-VC) tu veux dire comme ça? Envoi générique avec symboles

Vous pouvez le faire de cette façon:

```
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)
```

C'est pour l'algorithme d'Euler! Je me demande ce que @fcard pense de la répartition des symboles génériques! --- ^: ange:

Module Gotcha

```
@def m begin
  a+2
end

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

Plus précisément, ne fonctionne que dans le niveau supérieur du module dans lequel la macro a été définie.

```
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` empêche que cela se produise, mais le fait de ne jamais l'utiliser systématiquement va à l'encontre de la conception du langage. Une bonne défense consiste à ne pas utiliser et introduire de noms dans les macros, ce qui les rend difficiles à suivre pour un lecteur humain.

Python `dict` / JSON comme syntaxe pour les littéraux` Dict`.

introduction

Julia utilise la syntaxe suivante pour les dictionnaires:

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

Alors que Python et JSON ressemblent à ceci:

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

A **des fins d'illustration,** nous pourrions également utiliser cette syntaxe dans Julia et y ajouter de nouvelles sémantiques (la syntaxe `Dict` est la méthode idiomatique dans Julia, qui est recommandée).

Voyons d'abord quel *type* d'expression il s'agit:

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

Cela signifie que nous devons prendre l'expression `:cell1d` et la transformer ou renvoyer une nouvelle expression qui devrait ressembler à ceci:

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

Définition de macro

La macro suivante, bien que simple, permet de démontrer une telle génération et transformation de code:

```
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
```

Jetons un coup d'œil à l'expansion de la macro qui en résulte:

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

Usage

```
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
```

Le dernier exemple est exactement équivalent à:

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

Abus

```
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□}`
```

Notez que Julia a d'autres utilisations pour les deux `:` points `:` vous devrez donc envelopper les expressions littérales avec des parenthèses ou utiliser la fonction de `range` , par exemple.

Lire Métaprogrammation en ligne: https://riptutorial.com/fr/julia-lang/topic/1945/metaprogrammation

Chapitre 23: Modules

Syntaxe

- module module; ...; fin
- en utilisant le module
- Module d'importation

Examples

Wrap Code dans un module

Le mot-clé du `module` peut être utilisé pour démarrer un module, ce qui permet d'organiser le code et d'en placer les noms. Les modules peuvent définir une interface externe, généralement composée de symboles ed `export` . Pour prendre en charge cette interface externe, les modules peuvent avoir des fonctions internes non exportées et des types non destinés à un usage public.

Certains modules existent principalement pour envelopper un type et des fonctions associées. Ces modules, par convention, sont généralement nommés avec la forme plurielle du nom du type. Par exemple, si nous avons un module qui fournit un type de `Building` , nous pouvons appeler un tel module `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
```

Le module peut alors être utilisé avec l'instruction `using` :

```
julia> using Buildings

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

julia> height(ans)
```

```
830
```

Utilisation de modules pour organiser les packages

En règle générale, les packages se composent d'un ou de plusieurs modules. À mesure que les paquets grandissent, il peut être utile d'organiser le module principal du package en modules plus petits. Un idiome commun consiste à définir ces modules en tant que sous-modules du module principal:

```
module RootModule

module SubModule1

...

end

module SubModule2

...

end

end
```

Initialement, ni le module racine ni les sous-modules n'ont accès aux symboles exportés les uns des autres. Cependant, les importations relatives sont prises en charge pour résoudre ce problème:

```
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
```

Dans cet exemple, la valeur de `RootModule.z` est `30` .

Lire Modules en ligne: https://riptutorial.com/fr/julia-lang/topic/7368/modules

Chapitre 24: Normalisation de chaîne

Syntaxe

- normalize_string (s :: String, ...)

Paramètres

Paramètre	Détails
`casefold=true`	Pliez la chaîne dans un boîtier canonique basé sur la norme Unicode .
`stripmark=true`	Supprimez les marques diacritiques (c.-à-d. Les accents) des caractères de la chaîne d'entrée.

Examples

Comparaison de chaînes insensible à la casse

Les chaînes peuvent être comparées avec l' opérateur `==` dans Julia, mais ceci est sensible aux différences de casse. Par exemple, `"Hello"` et `"hello"` sont considérés comme des chaînes différentes.

```
julia> "Hello" == "Hello"
true

julia> "Hello" == "hello"
false
```

Pour comparer des chaînes de manière insensible à la casse, normalisez les chaînes en les pliant au préalable. Par exemple,

```
equals_ignore_case(s, t) =
    normalize_string(s, casefold=true) == normalize_string(t, casefold=true)
```

Cette approche gère également les Unicode non-ASCII correctement:

```
julia> equals_ignore_case("Hello", "hello")
true

julia> equals_ignore_case("Weierstraß", "WEIERSTRASS")
true
```

Notez qu'en allemand, la forme majuscule du caractère ß est SS.

Comparaison diacritique-insensible aux cordes

Parfois, on veut des chaînes comme `"resume"` et `"ré sumé "` pour comparer. Autrement dit, les graphèmes qui partagent un glyphe de base, mais peuvent différer en raison des ajouts à ces glyphes de base. Une telle comparaison peut être effectuée en éliminant les marques diacritiques.

```
equals_ignore_mark(s, t) =
    normalize_string(s, stripmark=true) == normalize_string(t, stripmark=true)
```

Cela permet à l'exemple ci-dessus de fonctionner correctement. En outre, il fonctionne bien même avec des caractères Unicode non-ASCII.

```
julia> equals_ignore_mark("resume", "ré sumé ")
true

julia> equals_ignore_mark("αβγ", "à β ỹ ")
true
```

Lire Normalisation de chaîne en ligne: https://riptutorial.com/fr/julia-lang/topic/7612/normalisation-de-chaine

Chapitre 25: Paquets

Syntaxe

- Pkg.add (package)
- Pkg.checkout (package, branch = "master")
- Pkg.clone (url)
- Pkg.dir (package)
- Pkg.pin (package, version)
- Pkg.rm (package)

Paramètres

Paramètre	Détails
`Pkg.add(` *`package`* `)`	Téléchargez et installez le paquet enregistré donné.
`Pkg.checkout(` *`package`* `,` *`branch`* `)`	Découvrez la branche donnée pour le paquet enregistré donné. *`branch`* est optionnel et par défaut `"master"` .
`Pkg.clone(` *`url`* `)`	Cloner le dépôt Git à l'URL donnée en tant que package.
`Pkg.dir(` *`package`* `)`	Obtenez l'emplacement sur le disque pour le package donné.
`Pkg.pin(` *`package`* `,` *`version`* `)`	Force le package à rester à la version donnée. *`version`* est facultative et utilise par défaut la version actuelle du package.
`Pkg.rm(` *`package`* `)`	Supprimez le package donné de la liste des packages requis.

Examples

Installer, utiliser et supprimer un paquet enregistré

Après avoir trouvé un package officiel Julia, il est facile de télécharger et d'installer le package. Tout d'abord, il est recommandé de rafraîchir la copie locale de METADATA:

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

Cela garantira que vous obtenez les dernières versions de tous les packages.

Supposons que le paquet que nous voulons installer s'appelle `Currencies.jl` . La commande à exécuter pour installer ce package serait:

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

Cette commande installe non seulement le paquet lui-même, mais aussi toutes ses dépendances.

Si l'installation est réussie, vous pouvez vérifier que le package fonctionne correctement :

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

Ensuite, pour utiliser le package, utilisez

```
julia> using Currencies
```

et procédez comme décrit dans la documentation du package, généralement liée à ou incluse dans son fichier README.md.

Pour désinstaller un package dont vous n'avez plus besoin, utilisez la fonction `Pkg.rm` :

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

Notez que cela ne supprime peut-être pas le répertoire du package. au lieu de cela, il ne fera que marquer le paquet comme n'étant plus requis. Souvent, cela va très bien - cela vous permettra de gagner du temps au cas où vous auriez encore besoin du paquet. Mais si nécessaire, pour supprimer le package physiquement, appelez la fonction `rm` , puis appelez `Pkg.resolve` :

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

julia> Pkg.resolve()
```

Découvrez une branche ou une version différente

Parfois, la dernière version balisée d'un paquet est boguée ou il manque certaines fonctionnalités requises. Les utilisateurs avancés peuvent souhaiter mettre à jour la dernière version de développement d'un package (parfois appelée "maître", nommée d'après le nom habituel d'une branche de développement dans Git). Les avantages de ceci incluent:

- Les développeurs contribuant à un package doivent contribuer à la dernière version de développement.
- La dernière version de développement peut avoir des fonctionnalités utiles, des corrections de bogues ou des améliorations de performances.
- Les utilisateurs signalant un bogue peuvent souhaiter vérifier si un bogue survient sur la dernière version de développement.

Cependant, l'exécution de la dernière version de développement présente de nombreux inconvénients:

- La dernière version de développement peut être mal testée et présenter de sérieux problèmes.
- La dernière version de développement peut changer fréquemment, brisant votre code.

Pour vérifier la dernière branche de développement d'un package nommé `JSON.jl` , par exemple,

utilisez

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

Pour extraire une branche ou un tag différent (non nommé "master"), utilisez

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

Toutefois, si la balise représente une version, il est généralement préférable d'utiliser

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

Notez qu'un littéral de version est utilisé ici, pas une chaîne simple. La version de `Pkg.pin` informe le gestionnaire de paquets de la contrainte de version, permettant au gestionnaire de paquets d'offrir des commentaires sur les problèmes qu'il pourrait causer.

Pour revenir à la dernière version balisée,

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

Installer un paquet non enregistré

Certains packages expérimentaux ne sont pas inclus dans le référentiel de packages METADATA. Ces packages peuvent être installés en clonant directement leurs référentiels Git. Notez qu'il peut y avoir des dépendances de paquets non enregistrés qui sont eux-mêmes non enregistrés; ces dépendances ne peuvent pas être résolues par le gestionnaire de packages et doivent être résolues manuellement. Par exemple, pour installer le package non enregistré `OhMyREPL.jl` :

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

Ensuite, comme d'habitude, utilisez `using` pour utiliser le package:

```
using OhMyREPL
```

Lire Paquets en ligne: https://riptutorial.com/fr/julia-lang/topic/5815/paquets

Chapitre 26: pour les boucles

Syntaxe

- pour i in iter; ...; fin
- tandis que cond; ...; fin
- Pause
- continuer
- @parallel (op) pour i in iter; ...; fin
- @parallel pour i in iter; ...; fin
- @goto label
- @label label

Remarques

Si cela rend le code plus court et plus facile à lire, envisagez d'utiliser des fonctions d'ordre supérieur, telles que `map` ou `filter` , plutôt que des boucles.

Examples

Fizz Buzz

Un cas d'utilisation courant pour une boucle `for` consiste à parcourir une plage ou une collection prédéfinie et à effectuer la même tâche pour tous ses éléments. Par exemple, nous combinons ici une boucle `for` avec une instruction `if` - `elseif` - `else` :

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

Ceci est la question d'interview classique de Fizz Buzz . La sortie (tronquée) est:

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

Trouver le plus petit facteur premier

Dans certaines situations, on peut vouloir revenir d'une fonction avant de terminer une boucle entière. La déclaration de `return` peut être utilisée pour cela.

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

Usage:

```
julia> primefactor(100)
2

julia> primefactor(97)
97
```

Les boucles peuvent également être terminées tôt avec l'instruction `break` , qui termine uniquement la boucle englobante au lieu de la fonction entière.

Itération multidimensionnelle

Dans Julia, une boucle for peut contenir une virgule (`,`) pour spécifier une itération sur plusieurs dimensions. Cela agit de la même manière que l'imbrication d'une boucle dans une autre, mais peut être plus compacte. Par exemple, la fonction ci-dessous génère des éléments du produit cartésien de deux itérables:

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

Usage:

```
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)
```

Toutefois, l'indexation sur des tableaux de toute dimension doit être effectuée avec `eachindex` , et

non avec une boucle multidimensionnelle (si possible):

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

Boucles de réduction et parallèles

Julia fournit des macros pour simplifier la distribution des calculs sur plusieurs machines ou travailleurs. Par exemple, le calcul suivant calcule la somme d'un certain nombre de carrés, éventuellement en parallèle.

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

Usage:

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

Pour plus d'informations sur ce sujet, consultez l' exemple sur `@parallel` dans la rubrique Processus parallèle.

Lire pour les boucles en ligne: https://riptutorial.com/fr/julia-lang/topic/4355/pour-les-boucles

Chapitre 27: Regexes

Syntaxe

- Regex ("[regex]")
- r "[regex]"
- match (aiguille, botte de foin)
- matchall (aiguille, botte de foin)
- chaque pièce (aiguille, meule de foin)
- ismatch (aiguille, botte de foin)

Paramètres

Paramètre	Détails
`needle`	le `Regex` à rechercher dans la `haystack`
`haystack`	le texte dans lequel chercher l' `needle`

Examples

Littéraux Regex

Julia prend en charge les expressions régulières [1] . La bibliothèque PCRE est utilisée comme implémentation regex. Les expressions rationnelles sont comme une mini-langue dans une langue. Étant donné que la plupart des langages et de nombreux éditeurs de texte prennent en charge les expressions rationnelles, la documentation et des exemples d'utilisation des expressions rationnelles en général ne sont pas abordés dans cet exemple.

Il est possible de construire une `Regex` partir d'une chaîne en utilisant le constructeur:

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

Mais pour plus de commodité et d'échapper plus facilement, la macro de chaîne `@r_str` peut être utilisée à la place:

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

[1] : Techniquement, Julia supporte les expressions rationnelles, qui sont distinctes et plus puissantes que ce que l'on appelle les expressions régulières dans la théorie des langues. Souvent, le terme "expression régulière" sera également utilisé pour désigner les expressions rationnelles.

Trouver des allumettes

Il existe quatre fonctions utiles principales pour les expressions régulières, qui prennent toutes des arguments dans `needle, haystack` ordre de `needle, haystack` . La terminologie "aiguille" et "botte de foin" vient de l'expression anglaise "trouver une aiguille dans une botte de foin". Dans le contexte des expressions rationnelles, le regex est l'aiguille et le texte est la botte de foin.

La fonction de `match` peut être utilisée pour trouver la première correspondance dans une chaîne:

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

La fonction `matchall` peut être utilisée pour rechercher toutes les correspondances d'une expression régulière dans une chaîne:

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

La fonction `ismatch` renvoie un booléen indiquant si une correspondance a été trouvée dans la chaîne:

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

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

La fonction `eachmatch` renvoie un itérateur sur les objets `RegexMatch` , utilisable avec les boucles `for` :

```
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
```

Groupes de capture

Les sous-chaînes capturées par les groupes de capture sont accessibles à partir des objets `RegexMatch` aide de la notation d'indexation.

Par exemple, la regex suivante analyse les numéros de téléphone nord-américains écrits au format `(555)-555-5555` :

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

et supposons que nous souhaitons extraire les numéros de téléphone d'un texte:

```
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"
```

En utilisant la fonction `matchall` , nous pouvons obtenir un tableau des sous-chaînes correspondantes:

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

Mais supposons que nous voulions accéder aux indicatifs régionaux (les trois premiers chiffres, entre crochets). Ensuite, nous pouvons utiliser l'itérateur `eachmatch` :

```
julia> for m in eachmatch(phone, text)
           println("Matched $(m.match) with area code $(m[1])")
       end
Matched (555)-505-1000 with area code 555
Matched (555)-999-9999 with area code 555
```

Notez ici que nous utilisons `m[1]` car le code de zone est le premier groupe de capture de notre expression régulière. Nous pouvons obtenir les trois composants du numéro de téléphone comme un tuple en utilisant une fonction:

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

Ensuite, nous pouvons appliquer une telle fonction à un `RegexMatch` particulier:

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

Ou nous pourrions `map` chaque match:

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

Lire Regexes en ligne: https://riptutorial.com/fr/julia-lang/topic/5890/regexes

Chapitre 28: REPL

Syntaxe

- julia>
- aider?>
- coquille>
- \[latex]

Remarques

D'autres packages peuvent définir leurs propres modes REPL en plus des modes par défaut. Par exemple, le package `Cxx` définit le mode shell `cxx>` pour une REPL C ++. Ces modes sont généralement accessibles avec leurs propres touches spéciales; voir la documentation du paquet pour plus de détails.

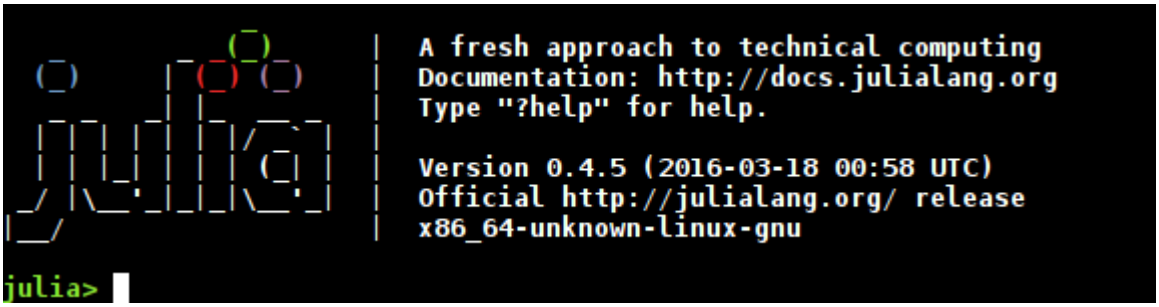
Examples

Lancer le REPL

Après avoir installé Julia , lancez la boucle REPL (read-eval-print-loop):

Sur les systèmes Unix

Ouvrez une fenêtre de terminal, puis tapez `julia` à l'invite, puis appuyez sur `Retour` . Vous devriez voir quelque chose comme ça:



Sous Windows

Trouvez le programme Julia dans votre menu Démarrer et cliquez dessus. Le REPL devrait être lancé.

Utiliser la REPL comme calculatrice

La Julia REPL est une excellente calculatrice. Nous pouvons commencer par quelques opérations

simples:

```
julia> 1 + 1
2

julia> 8 * 8
64

julia> 9 ^ 2
81
```

La variable `ans` contient le résultat du dernier calcul:

```
julia> 4 + 9
13

julia> ans + 9
22
```

Nous pouvons définir nos propres variables en utilisant l'affectation `=` opérateur:

```
julia> x = 10
10

julia> y = 20
20

julia> x + y
30
```

Julia a une multiplication implicite pour les littéraux numériques, ce qui rend certains calculs plus rapides à écrire:

```
julia> 10x
100

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

Si nous commettons une erreur et faisons quelque chose qui n'est pas autorisé, le Julia REPL émettra une erreur, souvent avec une astuce utile pour résoudre le problème:

```
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
```

Pour accéder ou modifier les commandes précédentes, utilisez la touche ↑ (Haut), qui permet de

passer au dernier élément de l'historique. Le `↓` se déplace à l'élément suivant de l'histoire. Les touches `←` et `→` peuvent être utilisées pour déplacer et apporter des modifications à une ligne.

Julia a des constantes mathématiques intégrées, y compris `e` et `pi` (ou `π`).

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

julia> pi
π = 3.1415926535897...

julia> 3π
9.42477796076938
```

Nous pouvons taper des caractères comme `π` rapidement en utilisant leurs codes LaTeX: appuyez sur `\` , puis sur `p` et `i` , puis appuyez sur la touche `Tab` pour remplacer le `\pi` que vous avez tapé avec `π` . Cela fonctionne pour d'autres lettres grecques et des symboles unicode supplémentaires.

Nous pouvons utiliser n'importe quelle fonction mathématique intégrée de Julia, allant de simple à assez puissante:

```
julia> cos(π)
-1.0

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

Les nombres complexes sont supportés en utilisant `im` comme une unité imaginaire:

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

Certaines fonctions ne renverront pas de résultat complexe à moins que vous ne leur donniez une entrée complexe, même si l'entrée est réelle:

```
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
```

Les opérations exactes sur les nombres rationnels sont possibles en utilisant l'opérateur `//` division rationnelle:

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

Voir le sujet arithmétique pour en savoir plus sur les types d'opérateurs arithmétiques supportés par Julia.

Faire face à la précision de la machine

Notez que les entiers de machine sont limités en taille et déborderont si le résultat est trop grand pour être stocké:

```
julia> 2^62
4611686018427387904

julia> 2^63
-9223372036854775808
```

Cela peut être évité en utilisant des entiers de précision arbitraire dans le calcul:

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

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

Les points flottants de la machine sont également limités en précision:

```
julia> 0.1 + 0.2
0.30000000000000004
```

Plus de précision (mais encore limitée) est possible en utilisant à nouveau `big` :

```
julia> big"0.1" + big"0.2"
3.000000000000000000000000000000000000000000000000000000000000000000000000000000000017e-01
```

Dans certains cas, l'arithmétique exacte peut être effectuée à l'aide de `Rational` :

```
julia> 1//10 + 2//10
3//10
```

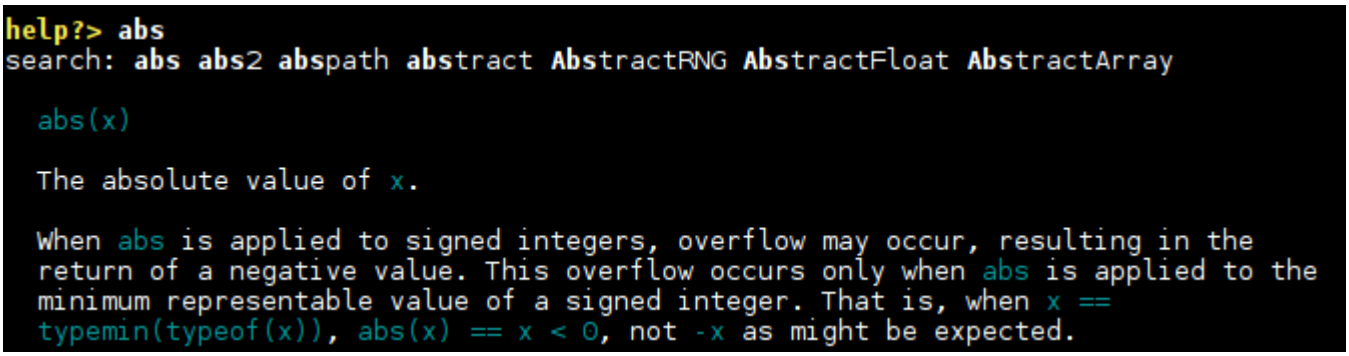
Utiliser les modes REPL

Il existe trois modes REPL intégrés dans Julia: le mode Julia, le mode d'aide et le mode shell.

Le mode d'aide

La Julia REPL est équipée d'un système d'aide intégré. Presse `?` à l'invite `julia>` pour accéder à l' `help?>`

À l'invite d'aide, tapez le nom d'une fonction ou d'un type pour obtenir de l'aide sur:



Même si vous n'épelez pas correctement la fonction, Julia peut suggérer certaines fonctions qui sont probablement ce que vous vouliez dire:

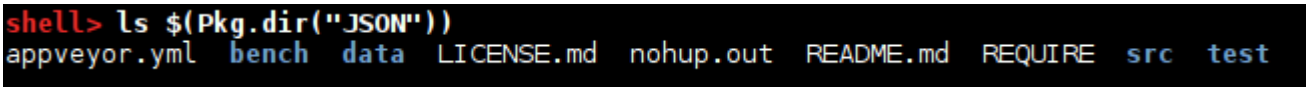
```
help?> printline
search:

Couldn't find printline
Perhaps you meant println, pipeline, @inline or print
  No documentation found.

  Binding printline does not exist.
```

Cette documentation fonctionne également pour d'autres modules, à condition qu'ils utilisent le système de documentation Julia.

```
julia> using Currencies

help?> @usingcurrencies
  Export each given currency symbol into the current namespace. The individual unit
  exported will be a full unit of the currency specified, not the smallest possible
  unit. For instance, @usingcurrencies EUR will export EUR, a currency unit worth
  1€, not a currency unit worth 0.01€.

  @usingcurrencies EUR, GBP, AUD
  7AUD  # 7.00 AUD

  There is no sane unit for certain currencies like XAU or XAG, so this macro does
  not work for those. Instead, define them manually:

  const XAU = Monetary(:XAU; precision=4)
```

Le mode shell

Reportez-vous à la section Utilisation de Shell depuis l'intérieur de REPL pour plus d'informations sur l'utilisation du mode shell de Julia, accessible en appuyant sur ; à l'invite. Ce mode shell prend en charge l'interpolation des données de la session Julia REPL, ce qui facilite l'appel des fonctions Julia et leur rendu dans des commandes shell:



Lire REPL en ligne: https://riptutorial.com/fr/julia-lang/topic/5739/repl

Chapitre 29: Shell Scripting et Piping

Syntaxe

- ; commande shell

Examples

Utilisation du shell depuis l'intérieur du REPL

De l'intérieur du shell Julia interactif (également appelé REPL), vous pouvez accéder au shell du système en tapant ; juste après l'invite:

```
shell>
```

A partir de là, vous pouvez taper n'importe quelle commande shell et elles seront lancées depuis la REPL:

```
shell> ls
Desktop     Documents   Pictures   Templates
Downloads   Music       Public     Videos
```

Pour quitter ce mode, tapez `backspace` lorsque l'invite est vide.

Écaillage du code de Julia

Le code Julia peut créer, manipuler et exécuter des littéraux de commande, qui s'exécutent dans l'environnement système du système d'exploitation. Ceci est puissant mais rend souvent les programmes moins portables.

Un littéral de commande peut être créé à l'aide du littéral `` . Les informations peuvent être interpolées à l'aide de la syntaxe `$` interpolation, comme pour les littéraux de chaîne. Les variables Julia transmises par des littéraux de commande ne doivent pas être échappées en premier; ils ne sont pas réellement transmis au shell, mais directement au noyau. Cependant, Julia affiche ces objets afin qu'ils apparaissent correctement échappés.

```
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(+)
```

Lire Shell Scripting et Piping en ligne: https://riptutorial.com/fr/julia-lang/topic/5420/shell-scripting-et-piping

Chapitre 30: sub2ind

Syntaxe

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

Paramètres

paramètre	détails
dims :: Tuple {Vararg {Entier}}	taille du tableau
I :: Entier ...	indices (scalaire) du tableau
I :: AbstractArray {T <: Integer, 1} ...	indices (vector) du tableau

Remarques

Le deuxième exemple montre que le résultat de `sub2ind` peut être très `sub2ind` dans certains cas spécifiques.

Examples

Convertir des indices en index linéaires

```
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
```

Fosses et chutes

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

On ne peut pas déterminer si un indice est dans la plage d'un tableau en comparant son index:

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

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

Lire sub2ind en ligne: https://riptutorial.com/fr/julia-lang/topic/1914/sub2ind

Chapitre 31: Tableaux

Syntaxe

- [1,2,3]
- [1 2 3]
- [1 2 3; 4 5 6; 7 8 9]
- Tableau (type, dims ...)
- ceux (type, dims ...)
- zéros (type, dims ...)
- trues (type, dims ...)
- falses (type, dims ...)
- pousser! (A, x)
- pop! (A)
- décaler (A, x)
- déplacer! (A)

Paramètres

Paramètres	Remarques
Pour	`push!(A, x)` , `unshift!(A, x)`
`A`	Le tableau à ajouter.
`x`	L'élément à ajouter au tableau.

Examples

Construction manuelle d'un tableau simple

On peut initialiser manuellement un tableau Julia en utilisant la syntaxe entre crochets:

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

La première ligne après la commande indique la taille du tableau que vous avez créé. Il montre également le type de ses éléments et sa dimensionnalité (dans ce cas `Int64` et `1` , respectivement). Pour un tableau à deux dimensions, vous pouvez utiliser des espaces et des points-virgules:

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

Pour créer un tableau non initialisé, vous pouvez utiliser la méthode `Array(type, dims...)` :

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

Les fonctions `zeros` , `ones` , `trues` , `falses` ont des méthodes qui se comportent exactement de la même façon, mais produisent des tableaux pleins de `0.0` , `1.0` , `True` ou `False` , respectivement.

Types de tableaux

Dans Julia, les tableaux ont des types paramétrés par deux variables: un type `T` et une dimensionalité `D` (`Array{T, D}`). Pour un tableau à 1 dimension d'entiers, le type est:

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

Si le tableau est une matrice à deux dimensions, `D` est égal à 2:

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

Le type d'élément peut également être des types abstraits:

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

Ici, `Any` (un type abstrait) est le type du tableau résultant.

Spécification des types lors de la création de tableaux

Lorsque nous créons un tableau de la manière décrite ci-dessus, Julia fera de son mieux pour en déduire le type approprié. Dans les exemples initiaux ci-dessus, nous avons entré des entrées qui ressemblaient à des entiers, et Julia a donc `Int64` type `Int64` par défaut. Parfois, cependant, nous pourrions vouloir être plus précis. Dans l'exemple suivant, nous spécifions que nous voulons que le type soit à la place `Int8` :

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

Nous pourrions même spécifier le type comme `Float64` , même si nous écrivons les entrées d'une manière qui pourrait être interprétée comme des entiers par défaut (par exemple, écrire `1` au lieu de `1.0`). par exemple

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

Tableaux de tableaux - Propriétés et construction

Dans Julia, vous pouvez avoir un tableau contenant d'autres objets de type Array. Considérez les exemples suivants d'initialisation de différents types de tableaux:

```
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
```

Considérons par exemple les différences entre C et D ici:

```
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)` produit un objet de type `Array{Float64,1}` . Comme la seule spécification pour les éléments de `C` est qu'ils sont des tableaux avec des éléments de type Float64, cela correspond à la définition de `C` Mais, pour `D` nous avons spécifié que les éléments doivent être des tableaux à deux dimensions. Donc, puisque `rand(3)` ne produit pas de tableau à deux dimensions, nous ne pouvons pas l'utiliser pour attribuer une valeur à un élément spécifique de `D`

Spécifier des dimensions spécifiques de tableaux dans un tableau

Bien que nous puissions spécifier qu'un tableau contiendra des éléments de type Array, et que nous pouvons spécifier que, par exemple, ces éléments doivent être des tableaux à deux dimensions, nous ne pouvons pas spécifier directement les dimensions de ces éléments. Par exemple, nous ne pouvons pas spécifier directement que nous voulons un tableau contenant 10 tableaux, chacun représentant 5,5. Nous pouvons voir cela à partir de la syntaxe de la fonction `Array()` utilisée pour construire un tableau:

> **Tableau {T} (dims)**
>
> construit un tableau dense non initialisé avec le type d'élément T. dims peut être un

tuple ou une série d'arguments entiers. La syntaxe Array (T, dims) est également disponible, mais obsolète.

Le type d'un tableau dans Julia englobe le nombre de dimensions, mais pas la taille de ces dimensions. Ainsi, il n'ya pas de place dans cette syntaxe pour spécifier les dimensions précises. Néanmoins, un effet similaire pourrait être obtenu en utilisant une compréhension Array:

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

Remarque: cette documentation reflète la réponse SO suivante

Initialiser un tableau vide

Nous pouvons utiliser le `[]` pour créer un tableau vide dans Julia. L'exemple le plus simple serait:

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

Les tableaux de type `Any` ne fonctionneront généralement pas aussi bien que ceux avec un type spécifié. Ainsi, par exemple, nous pouvons utiliser:

```
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}
```

Voir Initialiser un tableau vide de tuples dans Julia pour la source du dernier exemple.

Vecteurs

Les vecteurs sont des tableaux à une dimension et supportent principalement la même interface que leurs homologues multidimensionnels. Cependant, les vecteurs prennent également en charge des opérations supplémentaires.

Tout d'abord, notez que `Vector{T}` où `T` est un type signifie la même chose que `Array{T,1}` .

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

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

On lit `Array{Int64,1}` comme "tableau unidimensionnel d' `Int64` ".

Contrairement aux tableaux multidimensionnels, les vecteurs peuvent être redimensionnés. Des éléments peuvent être ajoutés ou supprimés à l'avant ou à l'arrière du vecteur. Ces opérations sont toutes des durées d'amortissement constantes .

```
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
```

Comme c'est la convention, chacune de ces fonctions `push!` , `pop!` , `unshift!` et `shift!` se termine par un point d'exclamation pour indiquer qu'ils modifient leur argument. Les fonctions `push!` et `unshift!` retourne le tableau, alors que `pop!` et `shift!` renvoyer l'élément supprimé

Enchaînement

Il est souvent utile de construire des matrices à partir de matrices plus petites.

Concaténation horizontale

Les matrices (et les vecteurs, qui sont traités comme des vecteurs de colonne) peuvent être concaténés horizontalement à l'aide de la fonction `hcat` .

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

Il existe une syntaxe pratique, en utilisant la notation entre crochets et les espaces:

```
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
```

Cette notation peut correspondre étroitement à la notation des matrices de blocs utilisées en algèbre linéaire:

```
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
```

Notez que vous ne pouvez pas concaténer horizontalement une seule matrice à l'aide de la syntaxe `[]` , car cela créerait plutôt un vecteur à un élément de matrices:

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

Concaténation verticale

La concaténation verticale est comme une concaténation horizontale, mais verticale. La fonction de concaténation verticale est `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
```

Alternativement, la notation entre crochets peut être utilisée avec des points-virgules ; comme délimiteur:

```
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
```

Les vecteurs peuvent également être concaténés verticalement; le résultat est un vecteur:

```
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
```

La concaténation horizontale et verticale peut être combinée:

```
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
```

Lire Tableaux en ligne: https://riptutorial.com/fr/julia-lang/topic/5437/tableaux

Chapitre 32: tandis que les boucles

Syntaxe

- tandis que cond; corps; fin
- Pause
- continuer

Remarques

Le `while` en boucle ne possède pas de valeur; Bien qu'il puisse être utilisé dans une position d'expression, son type est `Void` et la valeur obtenue ne sera `nothing` .

Examples

Séquence Collatz

Le `while` en boucle court son corps aussi longtemps que la condition est. Par exemple, le code suivant calcule et imprime la séquence Collatz à partir d'un nombre donné:

```
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
```

Usage:

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

Il est possible d'écrire une boucle récursive, et complexe `while` boucles, parfois la variante récursive est plus claire. Cependant, chez Julia, les boucles présentent des avantages distincts par rapport à la récursivité:

- Julia ne garantit pas l'élimination des appels de queue, la récursivité utilise donc de la mémoire supplémentaire et peut provoquer des erreurs de dépassement de capacité de la pile.
- De plus, pour la même raison, une boucle peut avoir un temps de traitement réduit et

s'exécuter plus rapidement.

Exécuter une fois avant de tester la condition

Parfois, on veut exécuter du code d'initialisation une fois avant de tester une condition. Dans certaines autres langues, ce genre de boucle a spécial `do` - `while` la syntaxe. Cependant, cette syntaxe peut être remplacée par une instruction `while` boucle et `break` , de sorte que Julia n'a pas de syntaxe `do` - `while` spécialisée. Au lieu de cela, on écrit:

```
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
```

Notez que dans certaines situations, de telles boucles pourraient être plus claires avec la récursivité:

```
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
```

Recherche en largeur

0.5.0

(Bien que cet exemple soit écrit en utilisant la syntaxe introduite dans la version v0.5, il peut également fonctionner avec peu de modifications sur les anciennes versions.)

Cette implémentation de la recherche en largeur (BFS) sur un graphique représenté avec des listes de contiguïté utilise les boucles `while` et la déclaration de `return` . La tâche que nous allons résoudre est la suivante: nous avons une séquence de personnes et une séquence d'amitiés (les amitiés sont mutuelles). Nous voulons déterminer le degré de connexion entre deux personnes. Autrement dit, si deux personnes sont amis, nous reviendrons à `1` ; si l'un est un ami d'un ami de l'autre, nous reviendrons à `2` , et ainsi de suite.

Tout d'abord, supposons que nous ayons déjà une liste d'adjacence: un `Dict` mapping `T` to `Array{T, 1}` , où les clés sont des personnes et les valeurs sont toutes les amies de cette personne. Ici, nous pouvons représenter des personnes avec le type `T` choisi. dans cet exemple,

nous utiliserons `Symbol` . Dans l'algorithme BFS, nous conservons une file d'attente de personnes pour «visiter» et marquer leur distance par rapport au nœud d'origine.

```
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
```

Maintenant, nous allons écrire une fonction pour construire une liste d'adjacence à partir d'une séquence de personnes, et une séquence de tuples `(person, person)` :

```
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
```

Nous pouvons maintenant définir la fonction d'origine:

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

Maintenant, testons notre fonction sur certaines données.

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

Jean est connecté en `0` étapes:

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

Jean et Cosette sont amis et ont donc un diplôme `1` :

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

Jean et Gavroche sont connectés indirectement via Cosette puis Marius, leur diplôme est donc `3` :

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

Javert et Marius ne sont connectés via aucune chaîne, une erreur est donc générée:

```
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
```

Lire tandis que les boucles en ligne: https://riptutorial.com/fr/julia-lang/topic/5565/tandis-que-les-boucles

Chapitre 33: Temps

Syntaxe

- à présent()
- Dates.today ()
- Dates.année (t)
- Dates.mois (t)
- Dates.day (t)
- Dates.heure (t)
- Dates.minute (t)
- Dates.second (t)
- Dates.milliseconde (t)
- Dates.format (t, s)

Examples

Heure actuelle

Pour obtenir la date et l'heure actuelles, utilisez la fonction `now` :

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

C'est l'heure locale, qui inclut le fuseau horaire configuré de la machine. Pour obtenir l'heure dans le fuseau horaire UTC (Coordinated Universal Time) , utilisez `now(Dates.UTC)` :

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

Pour obtenir la date actuelle, sans l'heure, utilisez `today()` :

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

La valeur de retour de `now` est un objet `DateTime` . Il y a des fonctions pour obtenir les composants individuels d'un `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
```

Il est possible de formater un `DateTime` utilisant une chaîne de format spécialement formatée:

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

Étant donné que la plupart des fonctions `Dates` sont exportées à partir du module `Base.Dates` , il est possible d'économiser de la saisie pour écrire

```
using Base.Dates
```

qui permet ensuite d'accéder aux fonctions qualifiées ci-dessus sans les `Dates.` qualification.

Lire Temps en ligne: https://riptutorial.com/fr/julia-lang/topic/5812/temps

Chapitre 34: Test d'unité

Syntaxe

- @test [expr]
- @test_throws [Exception] [expr]
- @testset "[nom]" commence; [tests] fin
- Pkg.test ([package])

Remarques

La documentation standard de la bibliothèque pour `Base.Test` couvre le matériel supplémentaire au-delà de celui montré dans ces exemples.

Examples

Test d'un package

Pour exécuter les tests unitaires pour un package, utilisez la fonction `Pkg.test` . Pour un paquet nommé `MyPackage` , la commande serait

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

Une sortie attendue serait similaire à

```
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
```

Bien évidemment, on ne peut pas s'attendre à ce qu'il corresponde exactement à ce qui précède, car différents packages utilisent des frameworks différents.

Cette commande exécute le fichier `test/runtests.jl` du `test/runtests.jl` dans un environnement propre.

On peut tester tous les paquets installés en même temps avec

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

mais cela prend généralement beaucoup de temps.

Écrire un test simple

Les tests unitaires sont déclarés dans le fichier `test/runtests.jl` dans un package. En règle générale, ce fichier commence

```
using MyModule
using Base.Test
```

L'unité de base de test est la macro `@test` . Cette macro est comme une assertion de toutes sortes. Toute expression booléenne peut être testée dans la macro `@test` :

```
@test 1 + 1 == 2
@test iseven(10)
@test 9 < 10 || 10 < 9
```

Nous pouvons essayer la macro `@test` dans le REPL:

```
julia> using Base.Test

julia> @test 1 + 1 == 2
Test Passed
  Expression: 1 + 1 == 2
   Evaluated: 2 == 2

julia> @test 1 + 1 == 3
Test Failed
  Expression: 1 + 1 == 3
   Evaluated: 2 == 3
ERROR: There was an error during testing
 in record(::Base.Test.FallbackTestSet, ::Base.Test.Fail) at ./test.jl:397
 in do_test(::Base.Test.Returned, ::Expr) at ./test.jl:281
```

La macro de test peut être utilisée à peu près partout, comme dans les boucles ou les fonctions:

```
# 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)
```

Ecrire un jeu de test

0.5.0

Dans la version v0.5, les ensembles de tests sont intégrés au module `Base.Test` bibliothèque standard et vous n'avez rien à faire de particulier (en plus d' `using Base.Test`) pour les utiliser.

0.4.0

Les jeux de tests ne font pas partie de la bibliothèque `Base.Test` de Julia v0.4. Au lieu de cela, vous devez `REQUIRE` le module `BaseTestNext` et ajouter à l' `using BaseTestNext` à votre fichier. Pour prendre en charge les versions 0.4 et 0.5, vous pouvez utiliser

```
if VERSION ≥ v"0.5.0-dev+7720"
    using Base.Test
else
    using BaseTestNext
    const Test = BaseTestNext
end
```

Il est utile de regrouper les `@test` associés dans un ensemble de tests. En plus d'une organisation de test plus claire, les ensembles de tests offrent de meilleurs résultats et une plus grande personnalisation.

Pour définir un ensemble de tests, `@test` tout nombre de `@test` s avec un bloc `@testset` :

```
@testset "+" begin
    @test 1 + 1 == 2
    @test 2 + 2 == 4
end

@testset "*" begin
    @test 1 * 1 == 1
    @test 2 * 2 == 4
end
```

L'exécution de ces jeux de tests imprime la sortie suivante:

```
Test Summary: | Pass  Total
  +           |    2      2

Test Summary: | Pass  Total
  *           |    2      2
```

Même si un ensemble de tests contient un test défaillant, l’ensemble de test sera exécuté jusqu’à la fin et les échecs seront enregistrés et signalés:

```
@testset "-" begin
    @test 1 - 1 == 0
    @test 2 - 2 == 1
    @test 3 - () == 3
    @test 4 - 4 == 0
end
```

L'exécution de cet ensemble de tests entraîne

```
-: 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.
    ...
```

Les ensembles de tests peuvent être imbriqués, permettant une organisation arbitrairement profonde

```
@testset "Int" begin
    @testset "+" begin
        @test 1 + 1 == 2
        @test 2 + 2 == 4
    end
    @testset "-" begin
        @test 1 - 1 == 0
    end
end
```

Si les tests réussissent, cela affichera uniquement les résultats pour le test le plus externe:

```
Test Summary: | Pass  Total
  Int         |    3      3
```

Cependant, si les tests échouent, un rapport détaillé sur le jeu de tests exact et le test à l'origine de la panne est signalé.

La macro `@testset` peut être utilisée avec une boucle `for` pour créer plusieurs ensembles de tests à la fois:

```
@testset for i in 1:5
    @test 2i == i + i
    @test i^2 == i * i
    @test i ÷ i == 1
end
```

qui rapporte

```
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
```

Une structure commune consiste à avoir des composants ou des types de test de jeux de tests externes. Dans ces ensembles de test externes, le test interne définit le comportement du test. Par exemple, supposons que nous ayons créé un type `UniversalSet` avec une instance singleton qui contient tout. Avant même d'implémenter le type, nous pouvons utiliser les principes de développement pilotés par les tests et implémenter les tests:

```
@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
```

Nous pouvons alors commencer à implémenter nos fonctionnalités jusqu'à ce qu'elles réussissent nos tests. La première étape consiste à définir le type:

```
immutable UniversalSet <: Base.AbstractSet end
```

Seulement deux de nos tests réussissent en ce moment. Nous pouvons mettre `in` œuvre `in` :

```
immutable UniversalSet <: Base.AbstractSet end
Base.in(x, ::UniversalSet) = true
```

Cela fait également passer certains de nos tests de sous-ensemble. Cependant, le `issubset` (`⊆`) ne fonctionne pas pour `UniversalSet` , car le repli essaie d'itérer sur les éléments, ce que nous ne pouvons pas faire. Nous pouvons simplement définir une spécialisation qui fait que `issubset` retourne `true` pour tout ensemble:

```
immutable UniversalSet <: Base.AbstractSet end
Base.in(x, ::UniversalSet) = true
Base.issubset(x::Base.AbstractSet, ::UniversalSet) = true
```

Et maintenant, tous nos tests réussissent!

Tester les exceptions

Les exceptions rencontrées lors de l'exécution d'un test échouent au test et si le test ne se trouve pas dans un ensemble de tests, fermez le moteur de test. Généralement, c'est une bonne chose, car dans la plupart des cas, les exceptions ne sont pas le résultat souhaité. Mais parfois, on veut tester spécifiquement qu'une certaine exception est soulevée. La macro `@test_throws` facilite cela.

```
julia> @test_throws BoundsError [1, 2, 3][4]
Test Passed
  Expression: ([1,2,3])[4]
      Thrown: BoundsError
```

Si la mauvaise exception est levée, `@test_throws` échouera toujours:

```
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
```

et si aucune exception n'est levée, `@test_throws` échouera également:

```
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
```

Test d'égalité approximative à virgule flottante

Quel est le problème avec ce qui suit?

```
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
```

L'erreur est causée par le fait qu'aucun des `0.1` , `0.2` et `0.3` n'est représenté dans l'ordinateur comme étant exactement ces valeurs - `1//10` , `2//10` et `3//10` . Au lieu de cela, ils sont approximés par des valeurs très proches. Mais comme on l'a vu dans l'échec du test ci-dessus, lors de l'ajout de deux approximations, le résultat peut être une approximation légèrement inférieure à celle possible. Il y a beaucoup plus à ce sujet qui ne peut pas être couvert ici.

Mais nous n'avons pas de chance! Pour vérifier que la combinaison de l'arrondi à un nombre à virgule flottante et de l'arithmétique à virgule flottante est *approximativement* correcte, même si elle n'est pas exacte, nous pouvons utiliser la fonction `isapprox` (qui correspond à l'opérateur `≈`). Nous pouvons donc réécrire notre test en tant que

```
julia> @test 0.1 + 0.2 ≈ 0.3
Test Passed
  Expression: 0.1 + 0.2 ≈ 0.3
   Evaluated: 0.30000000000000004 isapprox 0.3
```

Bien sûr, si notre code était complètement faux, le test détectera toujours que:

```
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
```

La fonction `isapprox` utilise des `isapprox` heuristiques basées sur la taille des nombres et la précision du type à virgule flottante pour déterminer la quantité d'erreur à tolérer. Ce n'est pas approprié pour toutes les situations, mais cela fonctionne dans la plupart des cas, et permet d'économiser beaucoup d'efforts pour implémenter sa propre version d' `isapprox` .

Lire Test d'unité en ligne: https://riptutorial.com/fr/julia-lang/topic/5632/test-d-unite

Chapitre 35: Traitement parallèle

Examples

pmap

`pmap` prend une fonction (que vous spécifiez) et l'applique à tous les éléments d'un tableau. Ce travail est réparti entre les travailleurs disponibles. `pmap` renvoie alors les résultats de cette fonction dans un autre tableau.

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

Si vous `pmap` plusieurs arguments, vous pouvez fournir plusieurs vecteurs à `pmap`

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

Comme avec `@parallel` , cependant, si la fonction donnée à `pmap` n'est pas dans la base Julia (c'est-à-dire définie par l'utilisateur ou définie dans un package), vous devez d'abord vous assurer que cette fonction est accessible à tous:

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

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

Voir aussi ce SO Q & A.

@parallèle

@parallel peut être utilisé pour paralléliser une boucle, en divisant les étapes de la boucle en différentes parties. Comme exemple très simple:

```
addprocs(3)

a = collect(1:10)

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

Pour un exemple légèrement plus complexe, considérez:

```
@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
```

Ainsi, nous voyons que si nous avions exécuté cette boucle sans `@parallel` 55 secondes au lieu de 27 auraient été nécessaires.

Nous pouvons également fournir un opérateur de réduction pour la macro `@parallel` . Supposons que nous ayons un tableau, nous voulons additionner chaque colonne du tableau et ensuite multiplier ces sommes les unes par les autres:

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

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

Il y a plusieurs choses importantes à garder à l'esprit lorsque vous utilisez `@parallel` pour éviter les comportements inattendus.

Premièrement: si vous souhaitez utiliser des fonctions de vos boucles qui ne sont pas dans la base Julia (par exemple, des fonctions que vous définissez dans votre script ou que vous importez depuis des packages), vous devez les rendre accessibles aux utilisateurs. Ainsi, par exemple, ce qui suit *ne* fonctionnerait *pas* :

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

Au lieu de cela, nous devrions utiliser:

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

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

Seconde Bien que chaque travailleur puisse accéder aux objets dans la portée du contrôleur, il *ne* pourra pas les modifier. Ainsi

```
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
```

Alors que si nous avions exécuté la boucle sans le @parallel, cela aurait pu modifier le tableau `a` .

POUR ADRESSER CECI, nous pouvons créer `a` objet de type `SharedArray` afin que chaque agent puisse y accéder et le modifier:

```
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 et @spawnat

Les macros `@spawn` et `@spawnat` sont deux des outils mis à disposition par Julia pour attribuer des tâches aux travailleurs. Voici un exemple:

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

julia>  From worker 2:  hello world
```

Ces deux macros évalueront une expression sur un processus de travail. La seule différence entre les deux est que `@spawnat` vous permet de choisir quel travailleur évaluera l'expression (dans l'exemple ci-dessus, le travailleur 2 est spécifié) alors qu'avec `@spawn` un travailleur sera automatiquement choisi, en fonction de la disponibilité.

Dans l'exemple ci-dessus, nous avons simplement demandé au travailleur 2 d'exécuter la fonction println. Il n'y avait rien d'intéressant à retourner ou à en récupérer. Souvent, cependant, l'expression que nous envoyons au travailleur donnera quelque chose que nous souhaitons récupérer. Notez que dans l'exemple ci-dessus, lorsque nous avons appelé `@spawnat` , avant que nous ayons obtenu l'impression de l'ouvrier 2, nous avons vu ce qui suit:

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

Cela indique que la macro `@spawnat` renvoie un objet de type `RemoteRef` . Cet objet contiendra à son tour les valeurs de retour de notre expression envoyée au travailleur. Si nous voulons récupérer ces valeurs, nous pouvons d'abord attribuer le `RemoteRef` que `@spawnat` renvoie à un objet, puis utiliser la fonction `fetch()` qui opère sur un objet de type `RemoteRef` , pour récupérer les résultats stockés dans une évaluation effectuée sur un travailleur.

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

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

julia> fetch(result)
7
```

La clé pour pouvoir utiliser efficacement `@spawn` est de comprendre la nature des expressions sur lesquelles il opère. Utiliser `@spawn` pour envoyer des commandes aux travailleurs est un peu plus compliqué que de simplement taper directement ce que vous tapez si vous exécutez un "interpréteur" sur l'un des travailleurs ou si vous exécutez du code en mode natif. Par exemple, supposons que nous voulions utiliser `@spawnat` pour assigner une valeur à une variable sur un agent. Nous pourrions essayer:

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

A-t-il fonctionné? Eh bien, nous allons voir en ayant 2 travailleurs essayer d'imprimer `a` .

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

julia>
```

Rien ne s'est passé. Pourquoi? Nous pouvons étudier cela plus en utilisant `fetch()` comme ci-dessus. `fetch()` peut être très utile car il récupère non seulement les résultats réussis, mais également les messages d'erreur. Sans cela, nous ne pourrions même pas savoir que quelque chose a mal tourné.

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

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

Le message d'erreur indique que `a` n'est pas défini sur worker 2. Mais pourquoi est-ce? La raison en est que nous devons envelopper notre opération d'affectation dans une expression que nous utilisons ensuite avec `@spawn` pour `@spawn` à l'ouvrier d'évaluer. Voici un exemple, avec une explication ci-dessous:

```
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
```

La syntaxe `:()` est ce que Julia utilise pour désigner des expressions . Nous utilisons ensuite la fonction `eval()` dans Julia, qui évalue une expression, et nous utilisons la macro `@spawnat` pour indiquer que l'expression doit être évaluée sur worker 2.

Nous pourrions également atteindre le même résultat que:

```
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
```

Cet exemple montre deux notions supplémentaires. Tout d'abord, nous voyons que nous pouvons également créer une expression à l'aide de la fonction `parse()` appelée sur une chaîne. Deuxièmement, nous voyons que nous pouvons utiliser des parenthèses lorsque nous appelons `@spawnat` , dans des situations où cela pourrait rendre notre syntaxe plus claire et plus facile à gérer.

Quand utiliser @parallel vs pmap

La documentation de Julia indique que

> pmap () est conçu pour le cas où chaque appel de fonction effectue une grande quantité de travail. En revanche, @parallel for peut gérer des situations où chaque itération est minuscule, en faisant simplement la somme de deux nombres.

Il y a plusieurs raisons à cela. Premièrement, `pmap` engendre des coûts de démarrage plus élevés en `pmap` emplois pour les travailleurs. Ainsi, si les tâches sont très petites, ces coûts de démarrage peuvent devenir inefficaces. Inversement, `pmap` accomplit un travail "plus intelligent" en répartissant les emplois entre les travailleurs. En particulier, il crée une file d'attente de travaux et envoie un nouveau travail à chaque travailleur chaque fois que ce travailleur est disponible. `@parallel` en revanche, divise tout le travail à faire parmi les travailleurs quand il est appelé. En tant que tel, si certains travailleurs prennent plus de temps que d'autres, vous pouvez vous retrouver avec une situation où la plupart de vos travailleurs ont fini et sont inactifs, tandis que quelques-uns restent actifs pour une durée excessive, terminant leur travail. Une telle situation est toutefois moins susceptible de se produire avec des emplois très petits et simples.

Ce qui suit illustre ceci: supposons que nous ayons deux travailleurs, dont l'un est lent et l'autre est deux fois plus rapide. Idéalement, nous voudrions donner au travailleur rapide deux fois plus de travail que le travailleur lent. (ou nous pourrions avoir des travaux rapides et lents, mais le principe est exactement le même). `pmap` parviendra, mais `@parallel` ne le fera pas.

Pour chaque test, nous initialisons les éléments suivants:

```
addprocs(2)

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

Maintenant, pour le test `@parallel` , nous `@parallel` les opérations suivantes:

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

Et récupérer l'impression:

```
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
```

C'est presque bon Les travailleurs ont "partagé" le travail de manière égale. Notez que chaque travailleur a effectué 6 tâches, même si le travailleur 2 est deux fois plus rapide que le travailleur 3. Il peut être touchant, mais il est inefficace.

Pour le test `pmap` , je lance ce qui suit:

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

et obtenir la sortie:

```
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
```

Maintenant, notez que le travailleur 2 a effectué 8 travaux et que le travailleur 3 a effectué 4 tâches. Ceci est exactement proportionnel à leur vitesse et à ce que nous souhaitons pour une efficacité optimale. `pmap` est un maître de tâche difficile - de chacun selon ses capacités.

@ async et @sync

Selon la documentation sous `?@async` , " `@async` une expression dans une tâche". Cela signifie que pour tout ce qui relève de sa portée, Julia lance cette tâche, mais passe ensuite à la prochaine étape du script sans attendre la fin de la tâche. Ainsi, par exemple, sans la macro, vous

obtiendrez:

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

Mais avec la macro, vous obtenez:

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

julia>
```

Julia permet donc au script de continuer (et à la macro `@time` de s'exécuter complètement) sans attendre que la tâche (dans ce cas, dormir pendant deux secondes) se termine.

La macro `@sync` , en revanche, "attendra que toutes les utilisations de `@async` , `@spawn` , `@spawnat` et `@parallel` `@spawn` dans une `@spawnat` `@parallel` soient terminées." (selon la documentation sous `?@sync`). Ainsi, on voit:

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

Dans cet exemple simple, il ne sert à rien d'inclure une seule instance de `@async` et `@sync` . Mais, où `@sync` peut être utile, vous `@async` appliquer `@async` à plusieurs opérations que vous souhaitez autoriser à démarrer sans attendre que chacune soit terminée.

Par exemple, supposons que nous ayons plusieurs employés et que nous souhaitons commencer chacun d'entre eux à travailler sur une tâche simultanément, puis à récupérer les résultats de ces tâches. Une tentative initiale (mais incorrecte) pourrait être:

```
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)
```

Le problème ici est que la boucle attend que chaque opération `remotecall_fetch()` se termine, c'est-à-dire que chaque processus termine son travail (dans ce cas, veille pendant 2 secondes) avant de continuer à lancer l'opération `remotecall_fetch()` . En termes de situation pratique, nous n'obtenons pas les avantages du parallélisme, car nos processus ne font pas leur travail (c.-à-d. Dormir) simultanément.

Nous pouvons corriger cela en utilisant une combinaison des macros `@async` et `@sync` :

```
@time begin
    a = cell(nworkers())
```

```
    @sync for (idx, pid) in enumerate(workers())
        @async a[idx] = remotecall_fetch(pid, sleep, 2)
    end
end
## 2.009416 seconds (274 allocations: 25.592 KB)
```

Maintenant, si nous comptons chaque étape de la boucle comme une opération distincte, nous voyons qu'il y a deux opérations distinctes précédées par la macro `@async` . La macro permet à chacun d'entre eux de démarrer, et le code à continuer (dans ce cas à l'étape suivante de la boucle) avant que chacun ne se termine. Mais, l'utilisation de la macro `@sync` , dont la portée englobe la boucle entière, signifie que nous ne permettrons pas au script de passer au-delà de cette boucle tant que toutes les opérations précédées de `@async` sont pas terminées.

Il est possible d'obtenir une compréhension encore plus claire du fonctionnement de ces macros en peaufinant l'exemple ci-dessus pour voir comment il change sous certaines modifications. Par exemple, supposons que nous ayons simplement le `@async` sans `@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)
```

Ici, la macro `@async` nous permet de continuer dans notre boucle avant même que chaque opération `remotecall_fetch()` ne se termine. Mais, pour le meilleur ou pour le pire, nous n'avons pas de macro `@sync` pour empêcher le code de continuer au-delà de cette boucle jusqu'à ce que toutes les opérations `remotecall_fetch()` terminées.

Néanmoins, chaque opération `remotecall_fetch()` est toujours en cours d'exécution en parallèle, même une fois que nous continuons. Nous pouvons voir cela parce que si nous attendons deux secondes, le tableau a, contenant les résultats, contiendra:

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

(L'élément "nothing" est le résultat d'une récupération réussie des résultats de la fonction sleep, qui ne renvoie aucune valeur)

Nous pouvons également voir que les deux opérations `remotecall_fetch()` démarrent essentiellement au même moment car les commandes d' `print` qui les précèdent s'exécutent également rapidement (sorties de ces commandes non affichées ici). Comparez cela avec l'exemple suivant où les commandes d' `print` s'exécutent à un intervalle de 2 secondes:

Si nous plaçons la macro `@async` sur toute la boucle (au lieu de simplement l'étape interne), notre script continuera à nouveau sans attendre la `remotecall_fetch()` opérations `remotecall_fetch()` .

Maintenant, cependant, nous autorisons uniquement le script à continuer au-delà de la boucle dans son ensemble. Nous ne permettons pas à chaque étape de la boucle de démarrer avant la précédente. En tant que tel, contrairement à l'exemple ci-dessus, deux secondes après le `#undef` du script après la boucle, le tableau de `results` contient toujours un élément comme `#undef` indiquant que la seconde opération `remotecall_fetch()` n'est toujours pas terminée.

```
@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
```

Et, sans surprise, si nous plaçons `@sync` et `@async` juste à côté l'un de l'autre, nous obtenons que chaque `remotecall_fetch()` s'exécute séquentiellement (plutôt que simultanément) mais nous ne continuons pas dans le code tant que chacun n'est pas terminé. En d'autres termes, ce serait essentiellement l'équivalent de si aucune macro n'était en place, tout comme `sleep(2)` se comporte essentiellement de manière identique à `@sync @async sleep(2)`

```
@time begin
    a = cell(nworkers())
    @sync @async for (idx, pid) in enumerate(workers())
        a[idx] = remotecall_fetch(pid, sleep, 2)
    end
end
# 4.019500 seconds (4.20 k allocations: 216.964 KB)
# Task (done) @0x0000000115e52a10
```

Notez également qu'il est possible d'avoir des opérations plus compliquées dans la portée de la macro `@async` . La documentation donne un exemple contenant une boucle entière dans le cadre de `@async` .

Rappelez-vous que l'aide pour les macros de synchronisation indique qu'elle "Attend que toutes les utilisations enfermées dynamiquement de `@async` , `@spawn` , `@spawnat` et `@parallel` soient complètes." Pour ce qui est considéré comme "complet", il est important de définir les tâches dans le cadre des macros `@sync` et `@async` . Considérons l'exemple ci-dessous, qui est une légère variation sur l'un des exemples donnés ci-dessus:

```
@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)
```

L'exemple précédent a mis environ 2 secondes à s'exécuter, indiquant que les deux tâches étaient exécutées en parallèle et que le script attendait que chacune exécute ses fonctions avant de continuer. Cet exemple, cependant, a une évaluation beaucoup moins longue. La raison en est que, pour les besoins de `@sync` l'opération `remotecall()` a "fini" une fois que le travail a été envoyé au travailleur. (Notez que le tableau résultant, a, ici, ne contient `RemoteRef` types d'objets `RemoteRef` , qui indiquent simplement qu'il se passe quelque chose avec un processus particulier qui pourrait en théorie être récupéré à un moment donné). En revanche, l'opération `remotecall_fetch()` n'a que "fini" lorsqu'elle reçoit le message de l'agent indiquant que sa tâche est terminée.

Ainsi, si vous cherchez des moyens de vous assurer que certaines opérations avec les travailleurs sont terminées avant de passer à votre script (comme cela est discuté dans cet article), il est nécessaire de bien réfléchir à ce qui compte comme "complet" et comment vous le ferez. mesurez puis opérationnalisez cela dans votre script.

Ajouter des travailleurs

Lorsque vous démarrez Julia pour la première fois, par défaut, il n'y aura qu'un seul processus en cours d'exécution et disponible pour donner du travail. Vous pouvez vérifier cela en utilisant:

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

Afin de tirer parti du traitement parallèle, vous devez d'abord ajouter des travailleurs supplémentaires qui seront alors disponibles pour effectuer le travail que vous leur assignez. Vous pouvez le faire dans votre script (ou à partir de l'interpréteur) en utilisant: `addprocs(n)` où `n` est le nombre de processus que vous souhaitez utiliser.

Alternativement, vous pouvez ajouter des processus lorsque vous démarrez Julia à partir de la ligne de commande en utilisant:

```
$ julia -p n
```

où `n` est le nombre de processus *supplémentaires* à ajouter. Donc, si on commence Julia avec

```
$ julia -p 2
```

Quand Julia commencera, nous aurons:

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

Lire Traitement parallèle en ligne: https://riptutorial.com/fr/julia-lang/topic/4542/traitement-parallele

Chapitre 36: Tuples

Syntaxe

- une,
- un B
- a, b = xs
- ()
- (une,)
- (un B)
- (un B...)
- Tuple {T, U, V}
- NTuple {N, T}
- Tuple {T, U, Vararg {V}}

Remarques

Les tuples ont de meilleures performances d'exécution que les tableaux pour deux raisons: leurs types sont plus précis et leur immuabilité leur permet d'être alloués sur la pile au lieu du tas. Cependant, cette saisie plus précise s'accompagne d'un temps système plus long au moment de la compilation et de plus de difficultés à atteindre la stabilité du type .

Examples

Introduction aux tuples

`Tuple` sont des collections ordonnées immuables d'objets distincts arbitraires, du même type ou de types différents. Généralement, les tuples sont construits en utilisant la syntaxe `(x, y)` .

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

Les objets individuels d'un tuple peuvent être récupérés à l'aide de la syntaxe d'indexation:

```
julia> tup[1]
1

julia> tup[2]
1.0

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

Ils implémentent l' interface itérable , et peuvent donc être itérés en utilisant `for` boucles `for` :

```
julia> for item in tup
```

```
            println(item)
        end
1
1.0
Hello, World!
```

Les tuples prennent également en charge diverses fonctions de collections génériques, telles que l' `reverse` ou la `length` :

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

julia> length(tup)
3
```

De plus, les n-uplets prennent en charge diverses opérations de collecte de niveau supérieur , y compris `any` , `all` , `all` `map` ou toutes les `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
```

Le tuple vide peut être construit en utilisant `()` :

```
julia> ()
()

julia> isempty(ans)
true
```

Cependant, pour construire un tuple d'un élément, une virgule est requise. C'est parce que les parenthèses (`(` et `)`) seraient autrement traitées comme des opérations de regroupement au lieu de construire un tuple.

```
julia> (1)
1

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

Par souci de cohérence, une virgule de fin est également autorisée pour les tuples comportant plusieurs éléments.

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

Types de tuple

Le `typeof` un tuple est un sous-type de `Tuple` :

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

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

Contrairement aux autres types de données, les types de `Tuple` sont covariants . Les autres types de données dans Julia sont généralement invariants. Ainsi,

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

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

C'est le cas parce que partout un `Tuple{Number, Number}` est accepté, tout comme un `Tuple{Int, Int}` , car il comporte également deux éléments, les deux étant des nombres. Ce n'est pas le cas pour un `Vector{Int}` par rapport à un `Vector{Number}` , car une fonction acceptant un `Vector{Number}` peut souhaiter stocker un point flottant (par exemple `1.0`) ou un nombre complexe (par exemple `1+3im`). un vecteur.

La covariance des types de tuple signifie que `Tuple{Number}` (contrairement à `Vector{Number}`) est en réalité un type abstrait:

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

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

Les sous-types concrets de `Tuple{Number}` incluent `Tuple{Int}` , `Tuple{Float64}` , `Tuple{Float64}` `Tuple{Rational{BigInt}}` , etc.

`Tuple` types de `Tuple` peuvent contenir une terminaison de `Vararg` comme dernier paramètre pour indiquer un nombre indéfini d'objets. Par exemple, `Tuple{Vararg{Int}}` est le type de tous les n-uplets contenant un nombre quelconque de s `Int` , éventuellement nul:

```
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
```

alors que `Tuple{String, Vararg{Int}}` accepte les tuples constitués d'une chaîne , suivis de tout nombre (éventuellement nul) de `Int` s.

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

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

Combiné à la co-variance, cela signifie que `Tuple{Vararg{Any}}` décrit un tuple. En effet, `Tuple{Vararg{Any}}` est une autre façon de dire `Tuple` :

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

`Vararg` accepte un second paramètre de type numérique indiquant combien de fois exactement son paramètre de premier type doit apparaître. (Par défaut, si non précisé, ce second paramètre de type est un TypeVar qui peut prendre toute valeur, ce qui est la raison pour laquelle un certain nombre de `Int` s sont acceptés dans le `Vararg` de ci - dessus.) `Tuple` types se terminant dans un spécifié `Vararg` sera automatiquement étendu à la nombre d'éléments requis:

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

La notation existe pour les tuples homogènes avec un `Vararg` spécifié: `NTuple{N, T}` . Dans cette notation, `N` désigne le nombre d'éléments dans le tuple et `T` désigne le type accepté. Par exemple,

```
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)
```

Notez que les `NTuple` au-delà d'une certaine taille sont simplement affichés comme `NTuple{N, T}` , au lieu de la forme `Tuple` développée, mais ils sont toujours du même type:

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

Envoi sur types de tuple

Étant donné que la liste des paramètres de la fonction Julia sont eux - mêmes tuples, l' envoi de différents types de tuples est souvent plus facile fait par la méthode des paramètres eux - mêmes, souvent avec l' utilisation libérale de la « splatting » `...` opérateur. Par exemple, considérez l'implémentation de `reverse` pour tuples, à partir de la `Base` :

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

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

Implémenter des méthodes sur des tuples de cette manière préserve la stabilité de type , ce qui est crucial pour les performances. Nous pouvons voir qu'il n'y a pas de surcharge à cette approche en utilisant la macro `@code_warntype` :

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

Body:
  begin
      SSAValue(1) = (Core.getfield)(t::Tuple{Int64,Int64,Int64},2)::Int64
      SSAValue(2) = (Core.getfield)(t::Tuple{Int64,Int64,Int64},3)::Int64
      return
(Core.tuple)(SSAValue(2),SSAValue(1),(Core.getfield)(t::Tuple{Int64,Int64,Int64},1)::Int64)::Tuple{Int6

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

Bien que difficile à lire, le code crée simplement un nouveau tuple avec les valeurs 3, 2 et 1 du tuple original, respectivement. Sur de nombreuses machines, cela se traduit par un code LLVM extrêmement efficace, composé de charges et de magasins.

```
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
}
```

Plusieurs valeurs de retour

Les tuples sont fréquemment utilisés pour plusieurs valeurs de retour. Une grande partie de la bibliothèque standard, y compris deux des fonctions de l' interface itérable (`next` and `done`), renvoie des tuples contenant deux valeurs liées mais distinctes.

Les parenthèses autour des tuples peuvent être omises dans certaines situations, ce qui facilite l'implémentation de plusieurs valeurs de retour. Par exemple, nous pouvons créer une fonction

pour renvoyer à la fois les racines carrées positives et négatives d'un nombre réel:

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

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

L'affectation de destruction peut être utilisée pour décompresser les valeurs de retour multiples. Pour stocker les racines carrées dans les variables `a` et `b` , il suffit d'écrire:

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

julia> a
3.0

julia> b
-3.0
```

Un autre exemple est les fonctions `divrem` et `fldmod` , qui effectuent une division entière (respectivement `divrem` ou `fldmod`) et une opération de reste en même temps:

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

julia> q
3

julia> r
1
```

Lire Tuples en ligne: https://riptutorial.com/fr/julia-lang/topic/6675/tuples

Chapitre 37: Type Stabilité

Introduction

L'instabilité de type se produit lorsque le type d' une variable peut changer à l'exécution et ne peut donc pas être déduit à la compilation. L'instabilité de type entraîne souvent des problèmes de performances. Il est donc important de pouvoir écrire et identifier un code de type stable.

Examples

Ecrire un code de type stable

```
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
```

La synchronisation des deux fonctions ci-dessus montre des différences majeures en termes d'allocations de temps et de mémoire.

```
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)
```

Cela est dû au code de type instable dans `sumofsins1` où le type de `r` doit être vérifié pour chaque itération.

Lire Type Stabilité en ligne: https://riptutorial.com/fr/julia-lang/topic/6084/type-stabilite

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec Julia Language	Andrew Piliser, becko, Community, Dawny33, Fengyang Wang, Kevin Montrose, prcastro
2	@goto et @label	Fengyang Wang
3	Arithmétique	Fengyang Wang
4	Combinateurs	Fengyang Wang
5	Comparaisons	Fengyang Wang
6	Compatibilité des versions croisées	Fengyang Wang
7	Compréhensions	2Cubed, Fengyang Wang, zwlayer
8	Conditionnels	Fengyang Wang, Michael Ohlrogge, prcastro
9	Contribution	Fengyang Wang
10	Cordes	Fengyang Wang, Michael Ohlrogge
11	Dictionnaires	B Roy Dawson
12	Enums	Fengyang Wang
13	Expressions	Michael Ohlrogge
14	Fermetures	Fengyang Wang
15	Fonctions d'ordre supérieur	Fengyang Wang, mnoronha
16	Iterables	Fengyang Wang, prcastro
17	JSON	4444, Fengyang Wang
18	Les fonctions	Fengyang Wang, Harrison Grodin, Michael Ohlrogge, Sebastialonso
19	Les types	Fengyang Wang, prcastro
20	Lire un DataFrame à partir d'un fichier	Pranav Bhat

21	Macros de chaîne	Fengyang Wang
22	Métaprogrammation	Fengyang Wang, Ismael Venegas Castelló, P i, prcastro
23	Modules	Fengyang Wang
24	Normalisation de chaîne	Fengyang Wang
25	Paquets	Fengyang Wang
26	pour les boucles	Fengyang Wang, Michael Ohlrogge
27	Regexes	Fengyang Wang
28	REPL	Fengyang Wang
29	Shell Scripting et Piping	2Cubed, Fengyang Wang, mnoronha, prcastro
30	sub2ind	Fengyang Wang, Gnimuc
31	Tableaux	Fengyang Wang, Michael Ohlrogge, prcastro
32	tandis que les boucles	Fengyang Wang
33	Temps	Fengyang Wang
34	Test d'unité	Fengyang Wang
35	Traitement parallèle	Fengyang Wang, Harrison Grodin, Michael Ohlrogge, prcastro
36	Tuples	Fengyang Wang
37	Type Stabilité	Abhijith, Fengyang Wang